Strategic Research Roadmap for European Web Security
FP7-ICT-2011.1.4, Project No. 318097
http://www.strews.eu/

Deliverable D1.1

Web-platform security guide: Security assessment of the Web ecosystem

Abstract

This deliverable reports on the broad web security assessment of STREWS. As part of this report, we provide a clear and understandable overview of the Web ecosystem, and discuss the vulnerability landscape, as well as of the underlying attacker models. In addition, we provide a catalog of best practices with existing countermeasures and mitigation techniques, to guide European industrial players to improve step-by-step the trustworthiness of their IT infrastructures. The report concludes with interesting challenges for securing the Web platform, opportunities for future research and trends in improving web security.

Deliverable details

Deliverable version: *v2.0*
Classification: *public*
Date of delivery: *30.09.2013*
Due on: *M12*
Editors: *Lieven Desmet and Frank Piessens*
Total pages: *168*

List of Contributors:
Philippe De Ryck, Lieven Desmet, Wouter Joosen, Jan Tobias Mühlberg, Frank Piessens, Martin Johns, Sebastian Lekies, Elwyn Davies, Stephen Farrell, Bert Bos, Thomas Roessler

Partners: ERCIM/W3C, SAP, TCD, KU Leuven

Document revision history

	Responsible	Date
Initial skeleton	KU Leuven	March 15, 2013
Initial revision	KU Leuven	July 5, 2013
Updated revision	KU Leuven	July 19, 2013
Early review W3C	W3C	August 12, 2013
Updated revision	KU Leuven	August 18, 2013
Early review TCD	TCD	August 18, 2013
Updated revision	KU Leuven	August 28, 2013
Integrated input W3C	W3C	September 4, 2013
Integrated input SAP	SAP	September 9, 2013
Integrated input TCD	TCD	September 16, 2013
Updated revision	KU Leuven	September 20, 2013
Pre-final revision	KU Leuven	September 25, 2013
Integrated input W3C	W3C	September 26, 2013
Internal review	TCD	September 27, 2013
Submitted version	KU Leuven	September 30, 2013
Threat landscape overview	KU Leuven	December 13, 2013
Updated executive summary	KU Leuven	December 17, 2013
Updated conclusion	KU Leuven	December 19, 2013
Internal review	W3C	December 19, 2013
Internal review	TCD	December 19, 2013
Final version	KU Leuven	December 20, 2013

Executive Summary

The Web security guide is the result of a broad security assessment of the current situation on the Web. It looks at the Web ecosystem and provides a **timely and comprehensive web security overview**. It was written by the **STREWS Consortium**, that brings together a **unique set of expertise in Europe** to grasp the complexity of the Web platform and its security characteristics. It is unique because it brings together strong peers in academic web security research in Europe, a large European software vendor, and principal actors in standardisation activities in W3C and IETF, the predominant specification developing organisations for the Web.

1. The first part gives a comprehensive overview of the current Web and the expected developments in the near future.

2. Based on the understanding of the Web ecosystem in the first part, the second part captures the breadth and complexity of the **Web security vulnerability landscape**.

   It describes the **Web assets** that are worth attacking and lists the capabilities attackers may have at their disposition and discusses the commonly-used **attacker models**.

3. In the third part, the **twenty most representative attack techniques** are discussed and analyzed, grouped in **seven high-level threat categories**.

   The guide presents and discusses the latest **state-of-the-art**, both from a **research perspective** as well as from a **standardization perspective**.

   Moreover, the guide provides a **catalogue of best practices** designed to mitigate the threats discussed, and to gradually improve the trustworthiness of web-enabled services.

4. Part four gives the **full Web security threat landscape** as an overview, indicates **upcoming challenges** resulting from the change of the web ecosystem and hints at some **interesting opportunities** for future research.

In this document, the complementary expertise of this one-of-a-kind consortium is assembled in a structured and reusable way. Moreover, the security guide is specifically geared towards **multiple audiences**:

Junior researchers The security guide provides junior researchers (i.e., Masters and PhD students) with insights into the complexity of the Web ecosystem, and presents a comprehensive overview of the web security landscape.
In addition, the guide bundles the key reference material as part of the state-of-the-art, and gives the necessary insights into the emerging research and standardization activities. Finally, the complementary intermezzo's provide additional context that will allow readers to get in touch with the complexity of the web security domain, and to be inspired by large scale security experiments and emerging technology in the field of web application security.

Senior researchers The security guide discusses emerging trends and mitigation techniques in web security, and enables senior researcher to understand how this complex ecosystem is evolving. Furthermore, a set of interesting challenges and opportunities for future research are presented in part IV of this document.

Industry players As part of the vulnerability landscape overview, a set of best practices is presented with existing countermeasures and emerging mitigation techniques. This catalogue of best practices, that can be implemented with a short to mid-term time horizon, enables European industrial players to gradually improve the trustworthiness of their IT infrastructures.

The Web platform security guide consists of four parts. In the following paragraphs, we briefly highlight the most important contributions and key takeaways for each part.

PART I: Foundations of the Web platform

In the first part of the guide, we briefly recap and discuss the foundations of the Web ecosystem. The goal of this first part is to provide the reader with a basic understanding of the Web ecosystem, needed to understand the security assessment.

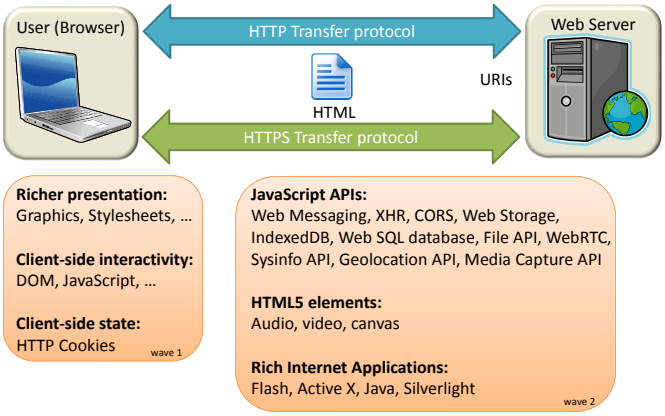
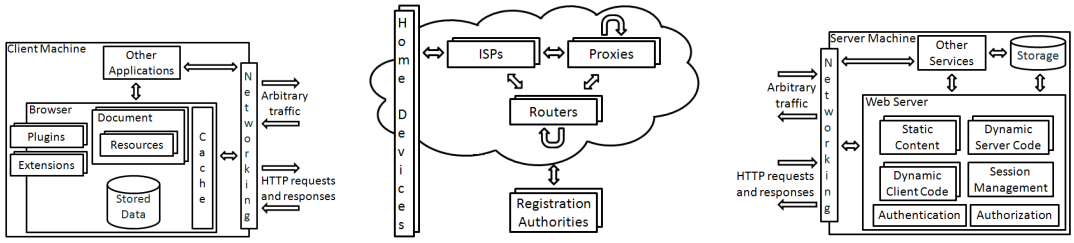


Figure 1: The Web ecosystem went through a series of technological waves.

Over the last 25 years, the Web ecosystem went through a series of technological waves (as depicted in Figure 1), enriching the platform to the current level where it provides an attractive alternative to stand-alone applications (or even replacing the operating system itself). Evolutions in the Web platform include richer presentation capabilities (e.g., graphics, style sheets and multimedia tags), client-side state (cookies and storage), client-side interactivity (JavaScript, the DOM and a rich set of JavaScript APIs), as well as rich Internet Applications (such as Flash, ActiveX and Silverlight).

The resulting ecosystem is represented in Figure 2, and is discussed in more detail in part I of this security guide. Important in this context is that, although the Web ecosystem has grown substantially over the last two decades, the basic security model of the Web still strongly relies on the Same-Origin Policy (SOP) from the mid 1990s. Major changes to this model face prohibitive deployment obstacles, as the currently-deployed legacy of web applications relies on the legacy model's properties.

This tension between ever-increasing complexity and limited built-in security fuels a continuous arms race between attackers and defenders, as will be illustrated by the wide variety of attacks discussed later in this document.



(a) Client Side (b) Network (c) Server Side

Figure 2: Perspectives on the contemporary Web platform

After reading part I, we expect all readers to share a common level of understanding of the Web platform. To serve the different audiences of the guide, the basic Web building blocks in part I are interleaved with some more advanced insights and additional pointers to aspects of the Web ecosystem, targeted to more advanced readers from industry and academia. For novice readers, the appendix discusses in more detail the underlying technologies of the Web, such as HTML, CSS, JavaScript and HTTP.

PART II: Threats to the Web platform

In part II, we identify the assets of the Web ecosystem, based on the model and concepts developed in part I, and enumerate the set of capabilities an attacker might have.

The assets are approached from an application-agnostic, technical point-of-view, but they can easily be correlated to actual business values once enriched with concrete application context. For instance, during the risk analysis of an E-Health web application, the *Server-side Content Storage* asset can be instantiated with electronic health records of the patients, and use their business value in the risk calculations.

The Web security platform guide identifies assets in the web infrastructure (i.e., the client and server machines), assets in the application (such as client-side application code and server-side application storage) and user-related assets (such as authentication credentials and personal information) (Figure 3).

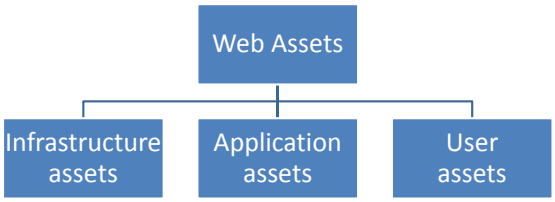


Figure 3: Assets in the web infrastructure, assets in the application, and user-related assets

For each asset, we discuss its importance and the attacker's incentives for compromising this asset, and analyze how an asset can be compromised. In order to structure the variety of ways available to compromise an asset, we use trees to explicitly define high-level threats against an asset. These threats are intermediate steps that an attacker has to take in order to compromise an asset, and are typically common across different assets.

For instance, if an attacker wants to tamper with an *Application Transaction* (e.g. forge a new wire transfer in an online bank application), the attacker can do this by first compromising the *Client-side Application Code* asset, as depicted in Figure 4. Similarly, there are various ways to compromise the client-side code, and an attacker might choose to intercept and manipulate the network traffic in order to control the client-side application code.

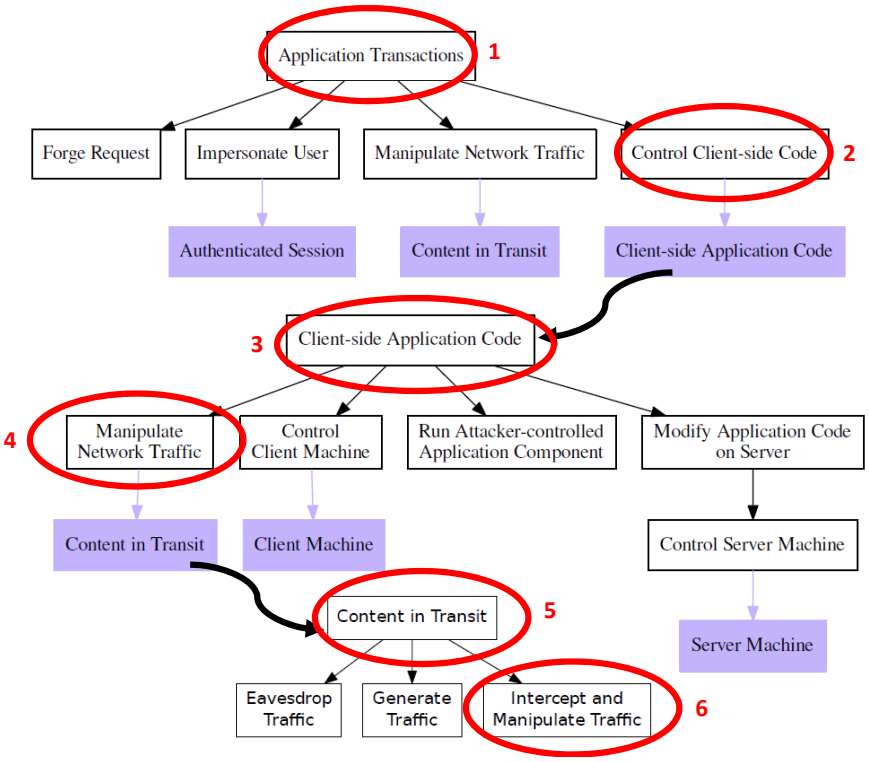


Figure 4: Walking the asset tree.

To be able to define an attacker's power in an unambiguous way, the platform guide enumerates the set of primitive and disjoint *attacker capabilities*. Essentially, attacker capabilities are the lowest level of capabilities available to an attacker. Examples of such fine-grained attacker capabilities are *Host content under a registered domain* or *Send a well-formed request to the application*.

The attacker capabilities can be composed to give a specific threat model, if desired. Within the security guide, we discuss threat models used in academic web security literature since 2008 (such as the *web attacker*, the *related-domain attacker* and the *passive network attacker*). We explicitly decompose the threat models into attacker capabilities, and compare them against each other (as illustrated in Table 1).

The main advantages of using attacker capabilities over threat models are that (1) they precisely define what technical capabilities an attacker has, and that (2) they can be more dynamically composed into new threat models that list the minimal set of capabilities needed to perform an attack.

PART III: Attacks on the Web platform

In the third part, the **web security vulnerability landscape** is constructed, by investigating how an attacker can execute the threats to compromise an asset. To do so, the threats identified in part II are grouped together in **seven high-level threat categories**:

1. Impersonating users
2. Forging requests
3. Attacking through the network

	Forum Poster	Web Attacker	Gadget Attacker	Related-domain Attacker	Passive Network Attacker	Active Network Attacker
Register Available Domain		⋆	⋆	⋆	⋆	⋆
Host Content under Registered Domain		⋆	⋆	⋆	⋆	⋆
Host Content under Existing Domain				⋆		
Register Valid Certificate for Domain Name		⋆	⋆	⋆	⋆	⋆
Respond to Legitimate Client Request		⋆	⋆	⋆	⋆	⋆
Send Well-formed Request to Application	⋆	⋆	⋆	⋆	⋆	⋆
Send Arbitrary Network Request to Server		⋆	⋆	⋆	⋆	⋆
Eavesdrop on Network Traffic					⋆	⋆
Generate Network Traffic						⋆
Intercept and Manipulate Network Traffic						⋆

Table 1: An overview of academic threat models, decomposed into fine-grained attacker capabilities.

4. Controlling the client-side context
5. Attacking the client-side infrastructure
6. Directly attacking the web application
7. Violating the user's privacy

For each of the seven high-level threats, the **most representative attack techniques** have been selected, and are reported in more detail. Selection of the representative subset of attack techniques is mainly driven by their prevalence, associated risk and potential impact, as indicated by the *OWASP top 10*, the *CWE/SANS Top 25 most dangerous programming errors* and relevant academic work, as presented in important security-related journals and conference proceedings. The guide aims to achieve completeness for the set of threats in part II, and to achieve a good coverage on the variety of attack techniques in part III.

Each of the **20 attack techniques** in the Web security platform guide is documented according to the following structure:

Problem description The problem description explains in detail the problem setting, the goal of the attack, the necessary attacker capabilities and how the assets get attacked. If multiple variations of the attack technique exist, the differences are briefly discussed. For more complex attack scenarios, an attack tree is provided to guide the reader through the different steps of the attack.

Mitigation techniques The most common mitigation techniques for the attack are presented and discussed. In this section, the reader gets an understanding of how the mitigation technique works, and insights in their effectiveness (or ineffectiveness).

State-of-practice In the state-of-practice section, insights and statistics on the prevalence of the attack, or the level of deployment of mitigation techniques are presented. This provides researchers and industry players a good understanding of the current security state of the

web ecosystem: Are available best practices being deployed? How widespread are known vulnerabilities? How often are these vulnerabilities being attacked?

Unfortunately, this highly-relevant material is not always available, or difficult to acquire. This platform guide aimed to include the publicly available statistics, which was possible for about half of the attack techniques presented.

Research and standardization activities The security guide summarizes the most important recent and ongoing research and standardization activities concerning this type of attack. This collection of key reference material captures most relevant evolutions and trends in academia (mainly geared towards an academic audience), and ongoing activities and new initiatives in standardization.

Best practices The guide provides a set of best practices to tackle this attack, now and in the near future. This catalogue of best practices should guide industry actors to gradually improve the security of their web-enabled services.

PART IV: Conclusion

In the conclusion of the Web platform security guide, we link together the assets from part II and the corresponding attacks from part III. This overview of the Web security threat landscape is depicted in Table 2, and can be interpreted in two ways:

- On the one hand, Table 2 illustrates the impact of a particular Web attack on the various assets of the Web platform. For instance, a successful *injection attack* can potentially impact 8 of 10 assets of the Web platform.
- On the other hand, Table 2 enumerates the list of attacks that need to be mitigated in a particular Web application in order to protect an asset. For instance, in order to fully protect *application transactions*, at least 17 attacks need to be mitigated.

The overview of the Web security threat landscape clearly illustrates the complexity of the Web ecosystem. To improve the end-to-end security, it is necessary to raise the bar on several (if not all) topics in parallel.

Finally, this document expresses the insights of the STREWS consortium on the way forward for web application security. We point out a set of interesting challenges for securing the Web platform, opportunities for future research and trends in improving Web security. Some important examples include:

Limited adoption of the best practices. There exists a remarkable mismatch between state-of-the-art mitigation techniques and best practices being available for almost all vulnerabilities, and we measured only a limited adoption of the best practices in the state-of-practice.

The question remains as to how web site owners can be incentivized to actually deploy best practices on their sites? Similarly, to track the adoption rate over time, it is important to have good metrics and measurements in place to be able to assess the state-of-practice of the Web ecosystem.

Trend towards server-driven browser enforcement. Significant areas of novel web security technology (both in research and standardization) follow the same pattern: The server issues a security policy, the policy is pushed towards the client as part of the web application, and the client is responsible for enforcing the policy correctly. Well-known examples in recent specifications are CSP, X-Frame-Options, HSTS, and Certificate Pinning.

In this context, the Content Security Policy (CSP) seems to be a very promising additional layer of defense, protecting against cross-site scripting and UI redressing.

	Server Machine	Client Machine	Server-side Content Storage	Client-side Content Storage	Content in Transit	Client-side Application Code	Authenticated Session	Application Transactions	Authentication Credentials	Personal Information
Session Hijacking							⋆			
Session Fixation							⋆			
Brute Force									⋆	
Stealing Authentication Credentials									⋆	
Cross-site Request Forgery								⋆		
Login Cross-site Request Forgery								⋆		
Clickjacking								⋆		
Eavesdrop on Network Traffic				⋆	⋆	⋆	⋆	⋆	⋆	⋆
SSL Stripping				⋆	⋆	⋆	⋆	⋆	⋆	⋆
Man-in-the-Middle Attack				⋆	⋆	⋆	⋆	⋆	⋆	⋆
Internal Attacks on TLS				⋆	⋆	⋆	⋆	⋆	⋆	⋆
Cross-site Scripting				⋆		⋆	⋆	⋆	⋆	⋆
Compromising JavaScript Inclusions				⋆		⋆	⋆	⋆	⋆	⋆
Malicious Browser Extensions		⋆		⋆		⋆	⋆	⋆	⋆	⋆
Drive-By Download		⋆		⋆		⋆	⋆	⋆	⋆	⋆
Attacking Local Infrastructure		⋆		⋆		⋆	⋆	⋆	⋆	⋆
Injection Attacks	⋆		⋆	⋆		⋆	⋆	⋆	⋆	⋆
Break Access Control	⋆		⋆	⋆		⋆	⋆	⋆	⋆	⋆
User Tracking/Fingerprinting							⋆	⋆	⋆	⋆
History Sniffing							⋆	⋆	⋆	⋆

Table 2: Overview of the Web security threat landscape: mapping assets from part II to attacks from part III.

Legacy building block as weak links. We clearly see the urge to fix some of the legacy building blocks of the Web model. For instance, passwords are still the primary authentication technique on the Web, and are almost always used in combination with bearer tokens (e.g., session management cookies and OAuth tokens).

Major changes to legacy building blocks of the Web model face prohibitive deployment obstacles, as the currently-deployed legacy of web applications relies on the legacy model's properties, and the adoption of best practices is rather slow.

Increasing need to compartmentalize web applications. As web applications are becoming larger, and contain more third-party components (e.g., third-party JavaScript inclusions), the secure containment or sandboxing of untrusted parts of the web application becomes crucial. Current state-of-the-art containment techniques still need to mature, both in terms of policy specification as well as enforcement techniques.

Shift from purely technical to user-centered. Web security is partially shifting from a purely technical topic to a user-centered topic. This is illustrated with the numerous phishing and social engineering attacks, and the web permission model relying more and more on decisions of the end-user. For sure, attackers will more and more target the user as the weakest link of the web infrastructure.

Moreover, UI security becomes a key factor in delivering a secure web ecosystem, especially with the rise of new web-capable devices, such as smartphones, tablets, etc. The web permissions model is extraordinarily complex, and hard to understand for the end-user. Key questions are how to involve (or not involve) users in security-related web decisions and how to communicate back security results to the user.

These topics have been identified during the assessment, and a representative selection will be tackled in more detail in the upcoming case study and the roadmapping activities of the STREWS project.

Contents

List of Intermezzos

List of Figures

Chapter 1

Introduction

Web technologies have evolved over the last ten years, leading to an interactive Web ecosystem, with numerous distributed components, cooperating with and depending on each other. Instead of simple web pages with optional interactive components, we are now using ubiquitous distributed applications with a client component. We no longer need to only protect the PC or the enterprise system from malware or Web-borne attacks: Our enterprise systems and personal information have moved to the Web. Web technologies have become the primary delivery platform for cloud-based software and services.

Software as a Service, executed in the web browser, has become a fully viable software delivery platform: Interactive applications that are executed within the browser have long become competitive to their native desktop-based counterparts. Even on smartphones that still have less computing power available than traditional computers, HTML5 + JavaScript has become the platform of choice for portable application development, either delivered just-in-time from the network, or sometimes executed from a local package.

As this development continues, the Web platform is acquiring numerous additional primitives that fundamentally change its nature. Standards work on these primitives is ongoing, and is occurring in parallel with their implementation across all major browser platforms.

At the same time, the underlying *security model* fundamentally remains the same model that was developed mostly ad-hoc during the mid 1990s. Major changes to this model face prohibitive deployment obstacles, as the currently-deployed legacy of web applications relies on the legacy model's properties.

In this Web platform security guide, we report on the broad security assessment that has been conducted in task T1.1 of the EC-FP7 project STREWS. In this task, we assessed the security of the Web ecosystem, focusing on the different components and their associated web technologies, both current and emerging, forming the foundations of every single web application.

As part of this report, we provide a clear and understandable overview of the web ecosystem (part I), and discuss the vulnerability landscape, as well as the underlying attacker models (part II). In addition, we provide a catalog of best practices with existing countermeasures and mitigation techniques, to guide European industrial players to improve step-by-step the trustworthiness of their IT infrastructures (part III).

The report concludes with interesting challenges for securing the Web platform, opportunities for future research and trends in improving web security (part IV). These topics have been identified during the assessment, and will be tackled in more detail in the upcoming activities of STREWS.

Structure of This Web Platform Security Guide

The Web platform security guide consists of four parts:

1. the model and the building blocks of the Web platform (part I)
2. the assets, the attacker goals and the attacker capabilities in the Web platform (part II)
3. the attacks, mitigation techniques and best practices (part III)
4. concluding remarks (part IV)

In the following paragraphs, we briefly summarize each of these parts.

PART I: The Web Platform

In this first part of the guide, the foundations of the Web ecosystem are briefly discussed. The goal of this first part is to provide the reader a basic understanding of the Web ecosystem, needed to understand the security assessment. In addition, the basic concepts are interleaved with some more advanced insights and additional pointers about the Web ecosystem, targeted to more advanced readers from industry and academia.

Chapter 2 provides an outline of the web model from a primarily technical point of view[1]. It dissects the basic anatomy of a web application and its content, and provides insights in the content distribution between the web server and the web client (i.e. browser).

Next, Chapter 3 and Chapter 4 describe in more detail the client-side browser architecture and the server-side application architecture. Finally, Chapter 5 discusses recent paradigms in developing and deploying web applications.

The Web ecosystem model, developed in this first part, is used as basis for the security assessment in parts II and III.

PART II: Threats to the Web Platform

In the second part of this guide, we identify the assets of the Web ecosystem, based on the model and concepts developed in part I, and enumerate the set of capabilities an attacker might have.

The assets are approached from an application-agnostic, technical point-of-view, but they can easily be correlated to actual business value once enriched with concrete application context. Chapter 6 enumerates assets in the web infrastructure (i.e. the client and server machines), assets in the application (such as client-side application code and server-side application storage) and user-related assets (such as authentication credentials and personal information).

For each asset, we discuss its importance and the attacker's incentives for compromising this asset, and analyze how an asset can be compromised. In order to structure the variety of ways to compromise an asset, we use attack trees to explicitly define high-level threats against an asset. These threats are intermediate steps an attacker is taking in order to compromise an asset, and are typically reoccurring for different assets.

Next, Chapter 7 lists the set of primitive and disjoint attacker capabilities, based on the web model of part I. Attacker capabilities can be combined, as typically is done in academic attacker models (such as the *web attacker* and the *network attacker*). As part of this chapter, the most important attacker models from literature are decomposed in attacker capabilities, and compared among each other.

PART III: Attacks on the Web Platform

In the third part of this guide, we investigate in more detail how an attacker can execute the identified threats to compromise an asset. To do so, the assessment starts from the seven high-level threats, identified in part II:

1. Impersonating users.

[1]For readers who are less familiar with the basic building blocks of the Web, we also provide more detail in appendix A for further reference.

2. Forging requests.
3. Attacking through the network.
4. Controlling the client-side context.
5. Attacking the client-side infrastructure.
6. Directly attacking the web application.
7. Violating the user's privacy.

For each of the seven threats, the most representative subset of attack techniques have been selected, and are reported in more detail. The guide aims to achieve completeness for the set of threats in part II, and to achieve a good coverage on the variety of attack techniques in part III. Selection of the representative subset of attack techniques is mainly driven by their prevalence, associated risk and potential impact, as indicated by the OWASP top 10 [261], the CWE/SANS Top 25 most dangerous programming errors [181] and relevant academic work, as presented in important security-related journals and conference proceedings.

In the web security guide, the description of an attack technique consists of a brief problem description and a representative set of mitigation techniques. Next, the guide briefly summarizes the most important recent and ongoing research and standardization activities concerning this type of attack. Insights and statistics on the prevalence of the attack, or the level of deployment of mitigation techniques are included when available, but are unfortunately not always easy to acquire. Finally, the guide provides a set of best practices to tackle this attack, now and in the near future.

PART IV: Conclusion

The final part of the report concludes with interesting challenges for securing the Web platform, opportunities for future research and trends in improving web security.

Part I

The Web Platform

Chapter 2

The Web Model

The Web platform is omnipresent in modern life, and has known a short but intense growth. It is also incredibly large and inherently complex, with the HTML 5 specification [33] alone amounting to 757 pages. Apart from the official specifications, multiple browser vendors have their own stake in the Web platform, and often have slightly differing implementations of the common specification [265], or even implement unspecified features.

In this chapter, we introduce the Web at a high level, shining a light on the way the Web has evolved, which technologies play an important role, and how they are related to each other. We do not aim to cover every detailed aspect of the Web, as that would lead us away from the goals of this document, but we do offer some insight into the details of several core technologies, such as HTML, cascaded style sheets (CSS) and JavaScript, in Appendix A. In the second part of this chapter, we focus on two essential evolutions, (1) the vast expansion of available client-side functionality, and (2) the strong push for more and better security on the Web, driven by the increasing role played by web applications.

2.1 The Web at a Glance

The World Wide Web started out as a distributed hypertext system, where documents hosted on networked computers contain references to other documents, hosted on different networked computers. These documents can be retrieved using a browser, dedicated client software for viewing hypertext documents, and following hyperlinks embedded within the text of these documents.

In order to make such a distributed hypertext system work, three fundamental agreements (standards) are necessary:

1. **Resource Identifiers** URIs (originally called URLs) provide universally dereferenceable identifiers for resources on the network.

2. **Transfer Protocol** A transfer protocol that is broadly (if not universally) implemented. HTTP (the Hypertext Transfer Protocol), at its most fundamental, provides a simple mechanism to retrieve a resource across the network, and to submit form data from a browser to a web server. HTTP is, in its most basic form, a simple request/response based client-server protocol: The client (Web browser) will send a request for a resource to the server that (a) identifies the resource, (b) identifies the media types of representations that the client is willing to consume in response (e.g., plain text or HTML; GIF or PNG), and (c) potentially is characterized as a form submission. The server responds with a resource representation that fulfills these constraints.

3. **Content Format** A format for content that is broadly (if not universally) implemented: HTML (the Hypertext Markup Language) is an originally simple and declarative markup

language that includes an anchor element which permits embedding hyperlinks to resources identified by URIs within the text. Based on plain text with embedded "tags", this markup language can be written in a simple text editor, and remarkably is often written in code editors to this day, 20 years later.

Notably, these three basic agreements are loosely coupled: While the URIs we use most frequently identify resources retrieved through HTTP, URIs can also be used to identify resources retrieved through other protocols (early on, FTP wasn't infrequently used to serve resources on the Web, and indeed, web browsers included implementations of FTP, Gopher, and a number of other protocols). URIs can identify, and HTTP can be used to retrieve resources (or, technically, their representations) in formats other than HTML – that can be images, PDF documents, scripts (i.e., executable program segments generally written in a interpretable scripting language such as JavaScript), or just about any other format that is represented in bits and bytes. Similarly, the concept of hyperlinks exists in formats beyond HTML: PDF, and even Word documents, might permit these. Nevertheless, the Web's basic fabric is built on universal support for URIs, HTTP, and HTML.

This basic fabric of the Web (from the early 1990s) is non-interactive, declarative, and (on the client side) stateless. While these properties enabled the World Wide Web in all their simplicity, they were insufficient to fulfill the demand for a rich application platform, as the Web has become today.

The first steps towards creating an actual application platform, were introduced in three major changes on top of the basic hypertext system described above.

1. **Richer presentation** Agreed upon graphics formats (early, GIF, then PNG and JPEG) enable the transmission of graphics for the Web. An agreed upon stylesheet format (CSS) permits the declarative description of styling information for HTML pages (and other markup languages – relatively loose coupling here, too), notably separating between content and presentation.

2. **Client-side interactivity** While it is entirely possible to use form submissions and hyperlinks as the basis for highly complex networked applications with very little client state (see [100] on the REST architectural style), the ability to execute code in a Turing-complete language on the Client proves convenient and highly powerful. Often motivated by the desire for more powerful and interactive presentation (the business logic would remain on the server), the Web acquired interfaces to a number of programming languages, notably Java and JavaScript (aka ECMAScript). The latter, intended to enable on-the-fly manipulation of HTML documents, has become the dominant programming language for the Web.

   Client-side interactivity requires an additional element of glue: In order to manipulate a document in place, code written in JavaScript needs an object model for that document. Today, the DOM (Document Object Model) is the standard model through which JavaScript code can refer to the elements and content of a web page.

3. **Client-side state** Early web authentication protocols attempted to preserve the basic statelessness of the Web: Instead of storing application state in the client, each HTTP request would be authenticated independently. HTTP Cookies were introduced in the early Web to provide simple client-side storage, either for simple user preferences (language, color, font, stylesheet, ...), or for session identifiers (long random numbers). Cookies are issued by the web server in an HTTP response, and stored by the browser, which also attaches it to future requests.

The quick *tour d'horizon* in this section describes a distributed hypertext system with a loosely coupled design, where rich presentation, client-side interactivity, and client-side state are a key factor to the success. These features and capabilities are governed by the Web's security model, which has grown organically since the mid 1990s with the introduction of these

new features, and has been fine-tuned as the technology matures. Two cornerstones of the Web's security model are the browser security policies, with the *same-origin policy* as the most imperative one, and heightened transport security, mainly achieved by deploying HTTP over TLS.

The same-origin policy is a browser security policy aimed at preventing application contexts from different origins from influencing each other. As HTML (and its implementation in browsers) matured, technologies like iframes, frames, pop-up windows permitted the simultaneous execution of several web applications that could to some degree communicate with each other, as well as with remote servers. The same-origin policy introduces a basic security boundary that prevents a resource from accessing another resource's context, unless it's origins (the *scheme, host, port* tuple) match. As a result, different web applications (web sites of competing services, for example) can coexist in the same web browser with a basic isolation and confidentiality guarantee against each other.

On another level, Transport Layer Security (TLS), earlier known as Secure Sockets Layer (SSL), offers entity authentication, confidentiality and integrity on the transport level. For the Web, this basically means that HTTP becomes HTTPS, which is in fact HTTP over a secure connection, preventing passive or active network-level attacks. Even though a TLS connection is initiated at the network level, its effects seep through in several important web concepts, introducing peculiarities that influence the Web's security model in subtle ways.

Looking at the evolution of the Web, we identify the expansion of client-side functionality as a major evolution, enabling highly interactive applications. A second essential evolution is the push towards more and better security, leading to tightened security policies and additional security measures. Both evolutions are covered in detail in the remainder of this chapter.

For readers who are less familiar with the Web's underlying technologies, such as HTML, URIs and HTTP, we suggest reading Appendix A, which covers the basics of the Web platform in detail, before continuing with the remainder of the document.

2.2 Expanding Client-side Functionality

During the course of the Web's evolution, the available client-side functionality has vastly expanded. The main expansion is twofold, with the introduction of rich content types, such as plugins and HTML5 media support on one hand, and the expansion of the available JavaScript APIs on the other hand, as the following section will explain in detail. These web technologies enable new application models leading to the Web as we know it today, with, for example, *mashups* that are composed of multiple interactive components.

2.2.1 Rich Client-side Content

One way to introduce rich client-side content is by including content in other formats than HTML, typically handled by a browser plugin. Principal examples are Flash® or Silverlight® objects, Java® applets and ActiveX® components, which can act as a standalone application within a web page. Examples of such applications are video players, in-browser games or in-browser remote desktop software. The runtime environments of these content types are bound by certain security limitations, which are configurable by the user. For example, the Flash permissions can be configured to deny or allow storing local data, give access to microphone or webcam, etc.

With the new HTML 5 standard [33], several HTML elements introduce rich content capabilities directly into HTML. Popular examples are the `audio` and `video` tags, which allow web pages to embed audio or video files. Another example is the `canvas` element, which is essentially a drawing canvas, controllable through JavaScript. Its uses can go from making simple drawings to mirroring the content of a video element, allowing subtle transformations or manipulations.

Additionally, separate specifications introduce alternative markup languages to the Web, such as MathML [49] for mathematical expressions, and SVG [64] for scalable vector graphics.

Another part of the rich client-side content evolution is the shift from static content on static pages, towards dynamic content, on dynamic pages. Initially, dynamic pages merely used JavaScript to manipulate the page's structure, and potentially fetch additional information from the server. Gradually, this has evolved towards fully dynamic applications, where pages are updated continuously, and where a web application can have a fully-fledged client-side component, which runs autonomously in the user's browser. A popular example of such an extensive client-side application are online office suites, including text editors and spreadsheets, that run entirely in the browser.

Such powerful client-side applications depend on numerous JavaScript APIs that expose client-side functionality to the web application. These JavaScript APIs are typically created and maintained by W3C working groups, and their specification defines a set of features, interfaces, implementation guidelines and the associated permission model (See Chapter 3). Below, we briefly discuss several relevant APIs, grouped by the functionality they offer.

DOM APIs The basic functionality exposed to JavaScript enables the creation and manipulation of the Document Object Model (DOM) of HTML documents within the browser [169, 170]. Typical operations are the inspection of HTML elements and attributes, the removal of certain elements, or the creation of new elements with their children. A recent update of the DOM API [255] consolidates several older APIs, offering a simpler API to access the DOM and handle events within the DOM.

Remote Communication Several remote communication APIs offer client-side scripts the capability to send data to remote hosts, and fetch remote resources. A well-known example is the `XMLHttpRequest` (XHR) object [15], which allows JavaScript to send HTTP requests asynchronously. The recent additions of the Cross-Origin Resource Sharing (CORS) specification [253] further governs the new capabilities of the XHR object. The Web Sockets specification [133] allows the creation of a socket-based communication channel, allowing arbitrary (binary) data to be transmitted. The Server-Sent Events specification [132] allows a script to keep a connection open, so it can be notified by the server in case of interesting events. Furthermore, the WebRTC API [31] enables real-time communication between browsers, allowing browsers to talk to each other[1].

Intra-Browser Communication Communication within the browser, for example between different JavaScript contexts, is enabled by the Web Messaging API [131], which allows text messages or JavaScript objects to be sent. Additionally, the browser exposes location variables that can be used to trigger navigation events in different contexts [265]. A recent addition are Web Workers [134], which are background contexts that allow asynchronous processing in the background. Workers can be shared among several documents of the same application, and communication with a worker is similar to the Web Messaging API.

Storage Several recent specifications enable the use of client-side storage mechanisms. The Web Storage specification [136] offers a lightweight API to store key-value pairs, and the Indexed Database specification [185] offers more advanced databases with records, which can be queried using indices. In addition to data-oriented storage mechanisms, several specifications offer parts of an API that allows the creation and use of a temporary, application-bound file system within the browser [205, 249, 248].

[1]The WebRTC API will be the subject of the first case study (as will be reported in STREWS Deliverable D1.2).

Device Access JavaScript has access to device information, ranging from information provided by local sensors [201, 247, 39], to actual device characteristics, such as networking information etc [163]. Additional APIs also enable JavaScript to send information to the device, such as system-level notifications [254], or triggering vibration events [159].

Intermezzo 1: Unsafe use of various browser features on popular web sites

Web browsers' access control policies, which regulate access to internal objects such as the DOM, cookies, XHR, etc., have evolved in an ad-hoc fashion, resulting in numerous incoherencies. The paper *On the Incoherencies in Web Browser Access Control Policies* investigates three major flaws: (1) Inconsistent principal labeling across resources (e.g., by domain, by origin, ...), (2) inappropriate handling of label changes (e.g., the selective effect of the *document.domain* property), and (3) disregard of the user principal (e.g., access to private data, such as the clipboard, without explicit user permission). In this intermezzo, we focus on the results of an empirical study of the use of potentially unsafe browser features presented in the cited paper, which give an overview of how widespread, or otherwise, new features have become.

The results in Table 2.1 show that cookies, a well-established concept, are used by the majority of the sites investigated, but that the use of the cookie-specific *HttpOnly* flag is rather limited. The limited use of cookies in combination with XMLHttpRequest indicates that fears of the *HttpOnly* flag interfering with XHR cookie usage may be unfounded. On the other hand, programmatic access to cookies is a commonly used feature, which may interact with the *HttpOnly* flag.

The study also looks at recently introduced technologies, such as *postMessage* [131] and *localStorage* [136]. Even though these technologies were brand new at the time of the study, 0.95% of sites already use the *postMessage* API, and 0.19% use the *localStorage* API.

Measurement & Criteria	Total instances (count)	Unique sites	
		Count	Percentage
document.cookie (read)	5656310	72587	81.36%
document.cookie (write)	2313359	68230	76.47%
document.cookie domain usage (read)	2032522	59631	66.83%
document.cookie domain usage (write)	1226800	41327	46.32%
Secure cookies over HTTP	259	62	0.07%
Non-secure cookies over HTTPS	15589	4893	5.48%
Use of "HttpOnly" cookies	33180	14474	16.22%
Frequency of duplicate cookies	159755	4955	5.55%
Use of XMLHttpRequest	19717	4631	5.2%
Cookie read in response of XMLHttpRequest	1261	265	0.30%
Cross-origin descendant navigation (reading descendant's location)	6043	61	0.07%
Cross-origin descendant navigation (changing descendant's location)	0	0	0.00%
Child navigation (parent navigating direct child)	22572	6874	7.7%
document.domain (read)	1253274	63602	71.29%
document.domain (write)	8640	1693	1.90%
Use of cookies after change of effective domain	295960	1569	1.76%
Use of XMLHttpRequest after change of effective domain	225	87	0.10%
Use of postMessage after change of effective domain	0	0	0.00%
Use of localStorage after change of effective domain	42	10	0.01%
Use of local storage	1227	169	0.19%
Use of session storage	0	0	0.00%
Use of fragment identifier for communication	5192	3386	3.80%
Use of postMessage	6523	845	0.95%
Use of postMessage (with no specified target)	0	0	0.00%
Use of XDomainRequest	527	125	0.14%
Presence of JavaScript within CSS	224266	4508	5.05%

Table 2.1: Usage of various browser features on popular web sites (February 2010). Analysis includes 89,222 sites [234].

The paper also presents results from a second empirical study, which analyzes the behavior of frames in popular web sites. Frame behavior is highly relevant for understanding attacks such as clickjacking[a], and their countermeasures. Table 2.2 presents the results of

the frame behavior study, showing that 40.8% of sites include at least one frame, and that on average, a page contains 3.2 *iframe* elements. Remarkably, overlapping iframes seem to be common, often even involving a transparent, cross-origin frame. Note that this legitimate behavior exhibits common characteristics with a potentially dangerous clickjacking attack (See Section 9.3).

Sites containing at least one <iframe>	36549 (40.8%)
Average number of <iframe>'s per site	3.2
Sites with at least one pair of overlapping frames	5544 (6.2%)
Sites with at least one pair of overlapping cross-origin frames	3786 (4.2%)
Sites with at least one pair of transparent overlapping frames	1616 (1.8%)
Sites with at least one pair of transparent overlapping cross-origin frames	1085 (1.2%)

Table 2.2: Summary of display layouts observed for the top 100,000 Alexa web sites (December 2009). 89,483 sites were rendered successfully and are included in this analysis [234].

The study and analysis conducted in this paper attributes to the understanding of how common and brand new features of the Web platform are used in popular web applications. Besides the results presented here, the paper goes into further detail on incoherencies in browsers' access control policies, following from a mishandling of the involved principals.

Source: Kapil Singh, Alexander Moshchuk, Helen J. Wang, Wenke Lee, On the Incoherencies in Web Browser Access Control Policies, Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10), pages 463-478, 16-19 May 2010 [234]

[a] A *clickjacked* page tricks a user into performing undesired actions by clicking on a concealed link. Clickjacking is possible because seemingly harmless features of HTML web pages can be employed to perform unexpected actions. See Section 9.3 for details of how this can happen.

2.2.2 Mashups: Interactive Client-side Web Applications

An application model explicitly enabled by the Web's nature and supporting web technologies is *mashups* that are web applications composed of multiple interactive components, often from different component providers. While components within a mashup can operate independently, the added value in a mashup is actually created by collaborations between components. Mashups use several underlying web technologies to achieve *composition*, *collaboration* and *communication*, as we will explain in the remainder of this section.

Composition within a mashup can be achieved by directly including remote script content, or by integrating the remote component in a separate frame within the integrating page. The trade-off between both approaches is flexibility versus security. The former approach offers great flexibility, but grants the included script full access to the integrating page's contents, while the latter approach isolates the remote content in a separate context, but offers little flexibility and limited communication options between components.

Enabling *collaboration* between isolated components is not trivial, initially it is only possible through workarounds over a less than totally reliable channel. Several proposals for creating flexible and secure mashup architectures [68] have been made, but the proposal to allow opt-in communication between different browsing contexts [27] has been adopted by the W3C as the way forward, and has been standardized as the Web Messaging specification [131].

Mashup components not only require local communication between components, but also need to be able to *communicate with the remote host of their provider* [68], to fetch additional content or process certain information. Depending on the way the remote content is integrated, this communication from JavaScript is hampered by the traditional browser security model, that

specifically prohibits cross-domain JavaScript-originated requests. In a response to workarounds using local or remote proxy-components [68], the Cross-Origin Resource Sharing (CORS) [253] specification has been conceived, essentially enabling client-side cross-origin communication, but giving the contacted server the necessary control to prevent misuse or abuse from unauthorized client contexts.

2.3 Improving the Security of the Web

The evolution from static content to dynamic web applications has not only impacted the Web as an application platform, but also reflects on the sensitivity of the data that is being provided and processed by web applications. A consequence of both the growth of the Web and the processing of increasingly sensitive data has caused a strong push towards more and better security, as shown by the tighter browser security models, the introduction of network-level security, such as TLS, and even the *secure-by-design* mentality in the standardization community, reflected in emerging web standards [72] (See Intermezzo 2).

One cornerstone of security in the web context are browser security policies, with the same-origin policy, the navigation policy and the cookie access policy as the oldest and most important examples (See Chapter 3). Alongside with the Web's evolution, these policies have evolved when necessary, attempting to find the right balance between usability and security. As the Web evolved, so did the browser security policies. One example is the frame navigation policy, which has seen evolutions from a *permissive policy*, where any frame can navigate any other frame, to a *child policy*, where a frame can only navigate its direct children, to a *descendant policy*, where a frame can navigate its descendants [25].

Intermezzo 2: A Security Analysis of HTML5 *and friends*

The web browser is arguably the most security-critical component in our information infrastructure, since it has become the channel for accessing information, conducting business, managing critical infrastructure, such as power networks, etc. In the recent years, the standards which govern the browser – and hence its security – have been undergoing a major transformation. In order to accommodate innovations in web applications and their business models, a raft of new standards is currently being developed (See image below), such as the new HTML5 standard, cross-origin communication standards such as CORS and XHR, standards for access to local data such as Geolocation, local storage and packaged stand-alone applications (widgets).

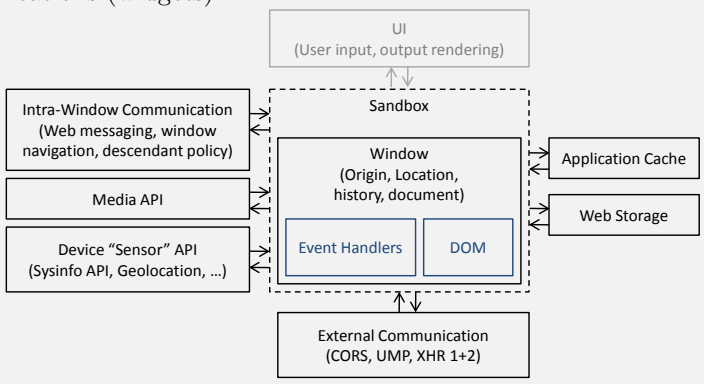


Figure 2.1: An abstract model of next-generation browser standards, showing how different specifications can be grouped together [71].

A study by the European Network and Information Security Agency (ENISA) of 13 mature, soon-to-be-implemented W3C specifications, yields the general observation that

the security quality of the specifications studied is reasonably high, especially given the number of introduced capabilities and quantity of specifications. However, ENISA identified 51 security threats and issues, which have been detailed in their HTML5 security report [71]. Most of the threats identify security-relevant capabilities in the individual specifications which are not well-defined or insecure. For instance, new attack vectors have been discovered to access sensitive information, to trigger form submission to adversaries, and to issue adversary-controlled cross-domain requests.

To conclude, the study shows that the application of the secure-by-design principle results in specifications with a reasonably high level of security, albeit with room for improvement on consistency and preciseness. Additionally, the study shows that the advent of HTML5 and the rich set of accompanying JavaScript APIs, not only provides a rich environment for web developers, but also dramatically increases the potential impact of malicious JavaScript running in your browser.

Source: Philippe De Ryck, Lieven Desmet, Pieter Philippaerts, Frank Piessens, A security analysis of next generation web standards, Technical Report, European Network and Information Security Agency (ENISA), 31 July 2011 [71]

Another sensitive area in the Web platform is the transport layer, where requests and responses are actually sent on the wire, often containing sensitive information and triggering sensitive operations at the server side. This is especially the case in ubiquitous wireless networks, with publicly accessible hotspots and a strong rise in mobile computing, with smartphones and tablets as the major facilitators. These evolutions have triggered a strong focus on network-level security, with TLS (and its predecessor SSL) as the prime candidate for securing HTTP traffic. The use of TLS prevents eavesdropping attacks on the requests and responses, and the built-in entity authentication is aimed at preventing man-in-the-middle attacks. Additionally, TLS has been subject to its own evolutions, with performance increases and extensions that offer additional security guarantees towards traditional web concepts, such as cookies [78].

Intermezzo 3: SSL/TLS Usage on the Web

In 2010, Qualys conducted the Internet SSL Survey to assess the state of practice in SSL usage on the Web. They scanned 119 million domain names, of which 92 million seemed active on port 80 or port 443. Of these 92 million domains, 33 million domains (36.52%) responded on port 443. Two out of three (22 million) actually talked SSL on this port, the remainder most probably used port 443 to tunnel SSH or VPN traffic through the firewall.

Of the 22 million active SSL domains, only in 720,000 cases (3.17%) was there a valid SSL certificate and a match between the domain name and the server name described in the server certificate. The huge mismatch can mainly be explained by the use of virtual hosting (i.e., the 22 million domain names with SSL active were represented by about two million IP addresses).

The usage of SSL strongly depends on the popularity of the site. For the one million most frequently visited sites examined in a subsequent experiment, 120,000 certificates were found. The two data sets were merged and the 867,361 entries account for about 25 to 50% of all the commercial certificates.

As part of the analysis, Qualys also investigated the chain of trust and the certificate authorities (CAs) being used. They qualified 70.05% of the SSL domains as trusted (about 607,000) , 27.56% as not trusted (of which 136,000 were expired, 96,000 were self-signed, and 43 had an unknown CA). In addition, 20,765 certificates (or 2.39% of the population) were classified as not trusted and suspicious.

With respect to the CAs, the analysis recorded 429 ultimately-trusted certificate issuers, leading to 78 trust anchors. Interestingly to know, this is only 50% of our trust base (in

Firefox 3.6.0), which has 155 trust anchors.

Finally, the study also investigated the use of older versions of the SSL/TLS protocol. It turned out that 49.85% of the domains still supported the insecure SSLv2 protocol, and virtually no servers supported TLS v1.1 and v1.2.

Source: Ivan Ristic (SSL Labs), Internet SSL Survey 2010, Black Hat USA 2010, 29 July 2010 [212]

Unfortunately, HTTP deployment over TLS is still rather limited on the modern Web, an issue that the future HTTP protocol will hopefully help to mitigate. HTTP/2.0 [30], currently under development by the IETF, addresses several problems with HTTP, without changing the actual semantics. HTTP/2.0, based on the SPDY protocol by Google [29], essentially changes how HTTP traffic is sent on the wire, reducing the page load time. HTTP/2.0 introduces new features such as multiplexed requests, prioritized requests, compressed headers and support for server-pushed content. On the security side, TLS has not been made mandatory, but the most efficient upgrade path from HTTP/1.1 to HTTP/2.0 is by deploying HTTP/2.0 over TLS. using the *Application Layer Protocol Negotiation* extension [107].

Chapter 3

Browser Architecture

The browser is the user's main window on to the internet, offering access to web sites and applications. This responsibility is fulfilled by a synergy of multiple components within the browser. For instance, the networking stack is responsible for retrieving the required resources, multiple parsers transform the raw web content into meaningful concepts, and the rendering engine makes sure the parsed content is visualized and displayed as intended.

The browser market is not limited to a single player, but has many competing mainstream browsers, supplemented by even more niche browsers. Notwithstanding the different views on browser architecture, all these browsers share a common view of the Web, with basic concepts such as browser windows, web documents, browsing contexts, browser caches, etc. Many differences between browsers come from major design decisions, such as the core architecture, underlying engines, additional features or a line of thought on user interface design, but several differences also lie in the detailed implementation of a concept. This text does not focus on a specific browser, nor does it zoom in on the differences between browsers, which have been extensively covered in other work [265].

All browsers do implement the general model of the Web, as described in the HTML 5 specification [33], and depicted in Figure 3.1. The model includes browsing contexts, documents, origins, cookies and script execution environments, which will be introduced briefly in this introduction. These concepts are fundamental in understanding the security implications in modern browsers, enabling users to have multiple web applications loaded in multiple tabs or windows, as well as the security implications within complex web applications combining content from multiple sources, nested documents and shared state between client and server in the form of *cookies*. Additionally, browsers now support content storage at the client-side, typically bound to the origin of the document.

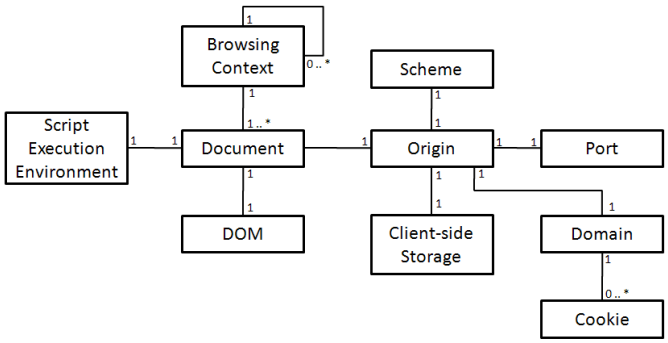


Figure 3.1: A domain model illustrating the internal representation of basic web concepts in the browser.

The main entity within a browser, a window or a tab in the browser can contain a resource from a web application, and when the user navigates to other resources, a history is built up. A window or tab has a *browsing context*, which loads *documents*. The currently loaded document is known as the *active document*, and the other documents are part of the history of this browsing context. The browsing context associated with a window or tab is known as the *top-level browsing context*, since it can contain nested browsing contexts, which will be explained later.

A document represents a resource from a web application, and it has a URI and origin associated with it. The internal structure of a document, which is a hierarchical tree of HTML elements, is represented using the *Document Object Model* (DOM). The DOM is also provided with a script-accessible API enabling inspection and modification of the tree. Documents also have an associated *script execution environment*, which holds the global state and code of the scripts within the given document.

Browsing contexts are not always top-level browsing contexts, and can be nested or associated with each other. Nesting is enabled by certain HTML elements, such as the *iframe* or *object* elements. Auxiliary browsing contexts are associated with another browsing context, without being nested. An auxiliary browsing context can be created by opening a popup window through JavaScript, where the opening browsing context is associated with the newly created browsing context.

The remainder of this chapter describes how browsers attempt to offer security guarantees, while enabling the wide range of functionality provided by the Web platform. The cornerstone within the browser are the security policies, with the same-origin policy as the most prominent one. Additionally, modern browsers typically support extensions and plugins, which are user-installable and greatly enhance the functionality of the browser. Finally, we explicitly cover several browser features specifically aimed at users, such as security indicators and alternative browsing modes.

3.1 Browser Security Policies

Modern browsers support different kinds of content, from static images to highly interactive scripts. Several security policies govern this content's behavior, regulating interactions, managing permissions and shielding sensitive information from unauthorized documents. We discuss four security policies: the *Same-Origin Policy*, the *navigation policy*, the browser's behavior with remote content inclusion, and the way access to certain resources is constrained by a document's origin.

3.1.1 Same-Origin Policy

The core security policy in a browser is the Same-Origin Policy (SOP), which originally aimed to regulate direct interactions between different browsing contexts, but has since then evolved towards regulating access between all kinds of content, such as contexts, images, video, etc. The basic function of the SOP is to prevent scripts loaded in one origin from programmatically accessing resources from other origins. For example, if you have a script loaded in a context with origin `http://www.example.com`, it is not allowed to access the DOM of a page loaded in a context with origin `http://www.secret.com`. The SOP originally started as a way to prevent access to the DOM, but has been gradually extended to other sensitive properties as well, with some exceptions for navigation (Section 3.1.2). Examples of other sensitive resources that are covered by a variation of the SOP are cookies, local storage facilities or the XMLHttpRequest (XHR) object. We cover several of these in Section 3.1.3. Currently, all modern browsers support the SOP fairly consistently, with some small exceptions [265].

In the previous chapter, we have already discussed the restrictions of the SOP on component collaboration in a mashup application. Due to the importance of the SOP within the browser, we give another example where the SOP is enforced to prevent leakage of information between

browsing contexts with different origins. The *canvas* element in HTML allows the web application to draw graphics, directly from the JavaScript code. Certain features of the canvas allow the script to draw arbitrary, cross-origin resources on the canvas (e.g., an image or video), making it possible to steal the contents of the video or image. To prevent such cross-origin leaking, the HTML 5 specification requires the browser to consider the canvas to be "tainted" with the origin of the image, effectively preventing any access from an origin other than that of the image.

One exception to the SOP is the *document.domain* JavaScript property, which allows a relaxation of the SOP between two applications that share the same parent domain. For example, the application at `www.example.com` and `login.example.com` can both set their *document.domain* property to `example.com`, overriding any future same-origin checks with this parent domain. This allows two sibling applications to cooperate freely. Even though both parties must explicitly opt-in to this feature, once they have opted-in, any other site within the same parent domain can "join" as well.

3.1.2 Context Navigation Policy

Navigation events occur frequently on the Web; examples include every time a user opens a page, follows a link, or when an automatic redirect happens. Triggering navigation events from within a document's context is straightforward, for example using JavaScript to modify the `document.location` property, or by automatically following a link. Navigation becomes more complicated when one context wants to navigate the window or frame from another context, possibly hosting a document from a different origin. Common examples are documents that want to navigate their child iframes, or opened popup windows. The decision as to whether to allow or deny such a navigation is not based on the same-origin policy, which would prohibit any navigation between contexts from different origins, but is determined by a separate navigation policy.

In modern browsers, top-level contexts can typically be navigated by anyone, since they have a visible address bar, which a user can inspect. Additionally, frame navigation is restricted by the *descendant policy* [27, 33], which states that a browsing context can only navigate its child frames. One exception to this rule is the permission for a context to navigate a non-child context, if the initiating context shares its origin with an ancestor of the non-child context. Allowing such navigation directly is more convenient, and does not compromise security, since they could be achieved indirectly, via injecting script code in the same-origin ancestor of the target frame [25].

3.1.3 Constraining Resources by Origin

In parallel with the Same-Origin Policy regulating access to a cross-origin browsing context, several other policies base their access rules on the origin, or a part thereof. Examples are access to local resources, such as storage facilities, use of the XHR object and access to cookies from JavaScript.

Several newly introduced standards offer an API to create, store and access local resources. Examples are the Web Storage API [136], the Indexed Database API [185], the in-browser filesystem APIs [205, 249, 248], etc. Common to these specifications is their partitioning of resources based on origin, effectively restricting data access to scripts running in the original origin. For example, the Indexed Database API offers a separate storage container to each origin, preventing the (unintentional) sharing of data between origins through this specific API.

Also the XHR object, which allows JavaScript code control over HTTP requests, is constrained by origin. Traditional XHR requests can only be sent to servers within the same origin as the origin of the document containing the script. Recent developments have lead to the XMLHttpRequest Level 2 specification [15], where cross-origin requests are enabled, with certain restrictions in place.

The security policy for cookies deviates from the notion of an origin, and is domain-based. Cookies are typically set for a domain, and only sent to the corresponding domain. A similar

situation for script-based cookie access exists: any application that resides on the domain of the cookies, or a valid subdomain, can access the cookies from JavaScript. For example, cookies set explicitly for `www.example.com` can be accessed by any resource in `www.example.com`, but also by resources under `dev.www.example.com`.

Another constrained resource within the browser are APIs offering access to sensitive features, such as determining the physical location of the machine running the browser [201], recording audio or video [44], etc. These APIs typically implement a permission system, allowing the user to grant or deny permission to these features, once or permanently. Whenever these permissions are stored, they are associated with the origin of the document that requested them. This means that in the future, the same permissions apply to any document within the same origin.

3.1.4 Security Model for Third-party Content Inclusion

Modern web applications include content from a wide variety of locations, often residing within a different origin. Common examples of such third-party content inclusions are images, stylesheets or JavaScript files, simply integrated by including an HTML tag with the appropriate URI. The inclusion of various JavaScript libraries is especially popular allowing the creation of highly responsive user interfaces, and enriching the web site with additional functionality, ranging from integration with social media sites, to context-sensitive advertisements and tools for web site analytics.

The caveat that applies to third-party content inclusion is the interaction with the security policies of the browser, mainly the Same-Origin Policy. Typically, the included content resides directly within the security context of the including document. For static content, this does not really matter, but when dynamic content, such as JavaScript, is included within the security context of the application, it gains access to all origin-restrained resources.

There are two commonly-used techniques to integrate third-party JavaScript into a web application: through *script inclusion* or via *iframe integration*. The former loads the script within the security context of the including application, resulting in a straightforward way to integrate components and enable interaction between components. The latter places the script in a separate iframe, within its own security context, effectively shielding sensitive resources, but making interaction a bit more complicated. We elaborate on both techniques below.

Script Inclusion

HTML script tags are used to execute JavaScript while a web page is loading. This JavaScript code can be located on a different server from the web page it is executing in. When executing, the browser will treat the code as if it originated from the same origin as the web page itself, without any restrictions of the Same-Origin Policy.

The included code executes in the same JavaScript context, has access to the code of the integrating web page and all of its data structures. All sensitive JavaScript operations available to the integrating web page are also available to the integrated code.

Iframe Integration

HTML iframe tags allow a web developer to include one document inside another. The integrated document is loaded in its own environment almost as if it were loaded in a separate browser window. The advantage of using an iframe in a web application is that the integrated component (coming from another origin) is isolated from the integrating web page via the Same-Origin Policy. However, the code running inside of the iframe still has access to all of the same sensitive JavaScript operations as the integrating web page, albeit limited to its own execution context (i.e., origin). For instance, a third-party component can use local storage APIs, but only has access to the local storage of its own origin.

The newly introduced HTML 5 `sandbox`[1] attribute [34] for the iframe element disables several security-sensitive features, supporting the safe inclusion of potentially untrusted content. Specific features can be allowed by setting the value of the attribute, such as enabling script execution with the "allow-scripts" keyword.

Intermezzo 4: JavaScript inclusions: assessing the state of practice

The majority of third-party JavaScript libraries are integrated through script inclusion, giving the external script provider full control over the client context of the integrating page. This can have detrimental consequences if such a script provider has malicious intentions, or gets compromised at some point in time. To better understand the consequences of third-party script inclusion and the relationship between script providers and integrators, the 10,000 most important sites have been investigated with regard to their script inclusion behavior [192].

The authors examined 3,300,000 pages of the top 10,000 web sites (according to Alexa), and analyzed 8,439,799 remote script inclusions. The results in the figure below show that 88.45% of the 10,000 web sites included at least one remote JavaScript library. Even more remarkably, some sites in the top Alexa list trust up to 295 unique remote hosts.

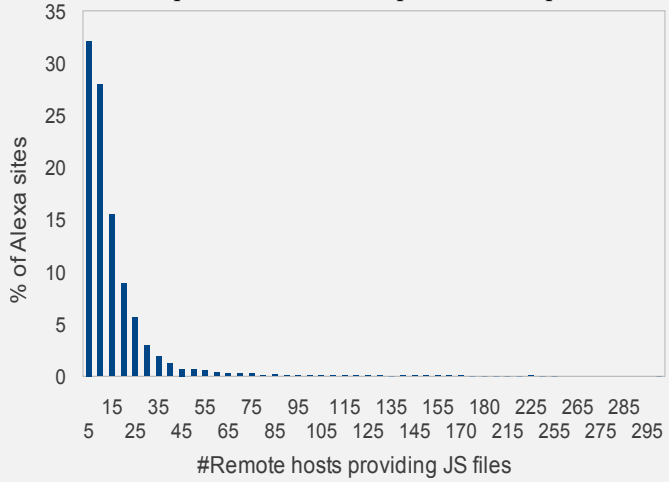


Figure 3.2: Relative frequency distribution of the percentage of top Alexa sites and the number of unique remote hosts from which they request JavaScript code

Assessment of the quality/security of both third-party script providers and integrators using a Quality of Maintenance metric shows that high-maintenance sites (in the sense of having a high score on the metric indicating a good quality product) tend to include scripts from high-maintenance providers, while low-maintenance sites often include from low-maintenance providers. However, the study also found that, for sites with high-maintenance scores, one out of four of their inclusions comes from providers with a low-maintenance score, which are potential "weak spots" in their security perimeter.

Finally, the paper introduces four new script inclusion attacks, based on anomalies discovered during the empirical studies.

1. **Cross-user and Cross-network Scripting** occurs when scripts are included from non-public resources, such as `localhost` or the `127.0.0.1` IP address, which re-

[1]The term *sandbox* is used to indicate that the operations of some set of components are confined to a notionally "safe" environment in which the effects of their operations are restricted to a controlled and limited area so that any damaging effects are not propagated into the wider environment of the components. The term is derived from the intentions of real-world children's play equipment; experience shows that sandboxes may leak! The *changeroot* capability of Linux and similar operating systems is an example of a sandbox that restricts the area of the machine filing system accessible to applications running in the sandbox.

solves to the client machine loading the web page, or private IP addresses, such as `192.168.2.2`. Such inclusions allow a local attacker to set up a service, either in the same network or on the same, shared machine, and compromise the client-side context of public web applications.

2. **Stale Domain-name-based Inclusions** occur when scripts are included from a no-longer-registered domain, which results in an error response. Such domains are up for grabs for an attacker, allowing him to register the domain and serve malicious JavaScript to sites that still include content from this domain.

3. **Stale IP-address-based Inclusions** applies the same technique as *stale domain-name-based inclusions*, but for IP addresses. While obtaining a specific IP address is more difficult than registering a domain name, it still remains an exploitable vulnerability, especially in a network with a dynamic address pool, distributed using DHCP.

4. In a **Typosquatting Cross-site Scripting** attack, the attacker attempts to exploit a developer error, where the URI of a script inclusion contains typographical errors ("typos"). By registering domain names similar to popular script inclusion domains, but with the same typo as in the inclusion, the attacker can serve malicious JavaScript when someone uses it. An example of this attack in the wild involved `worldofwaircraft.com` (instead of `worldofwarcraft.com`).

The large-scale evaluation of JavaScript inclusion behavior in the Alexa top 10,000 web sites sheds some light on the risks associated with trusting third-party library providers. The positive news is that high-maintenance sites caring about the quality of the included scripts are well-represented, but on the negative side, several top 10,000 sites, which are mainly low-maintenance, are very careless with including third-party scripts, leading to potential security vulnerabilities.

Source: Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, Giovanni Vigna, You are what you include: Large-scale evaluation of remote JavaScript inclusions, Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012), pages 736-747, Raleigh, NC, USA, 16-18 October 2012 [192]

3.2 Extending Browser Functionality

Generally speaking, the default browser functionality can be extended in three ways: adding plugins that handle arbitrary content, extending the browser functionality using extensions, and delegating content handling to a separate application. Browser plugins enable the browser to handle previously unknown content, enabling new or proprietary features in web applications. Popular examples are Adobe Flash and Microsoft Silverlight. Alternatively, browser extensions extend or alter the core functionality and behavior of the browser itself, and are not tied to a specific content type. Popular examples of extensions are NoScript or AdBlock. Finally, a browser can delegate content handling to a separate application, such as the well-known `mailto:` links, which invoke a mail client to compose and send an email. With the newly introduced content handlers, part of the HTML 5 specification [33], web applications can register themselves as the dedicated content handler for specific kinds of content. For example, a webmail client can register itself as the default handler for `mailto:` links.

These three mechanisms clearly extend the functionality of the browser, but also have their consequences. For instance, they significantly enlarge the attack surface, as demonstrated by regular discoveries of malicious or vulnerable plugins or extensions [128, 251]. In this section, we briefly discuss how plugins, extensions and content handlers work, how they are integrated

in the browser and what consequences are associated with their use.

3.2.1 Plugins for Arbitrary Content

Browser plugins are generally designated handlers for specific kinds of content. The most common example of a browser plugin is Adobe's Flash Player, responsible for processing and displaying Flash content within the browser. Other popular examples are Silverlight, Java or ActiveX. Browser plugins are associated with MIME types, and are automatically invoked when content from a specified MIME type is encountered. A browser plugin can provide arbitrary functionality, and is not limited to content rendering. An example of a non-content rendering plugin is the *Gnome Shell Integration* plugin, supporting the installation of additional Gnome functionality from the distribution web site.

Plugin content is typically embedded in a document, and the registered handler is triggered by the browser when this content is encountered. Most plugins allow communication between the document and the plugin using JavaScript, albeit with some restrictions, depending on the implementation. For example in the case of Flash, an interface can be exposed towards the document, and arbitrary JavaScript functions can be executed in the embedding page.

Plugins typically follow the cross-platform Netscape Plugin Application Programming Interface (NPAPI) plugin architecture, but can also be custom developed for a specific platform (e.g., ActiveX on Internet Explorer). Plugin content runs within the environment of the handler, where the security policies of the browser no longer reign. This effectively means that if the plugin does not restrict the behavior of plugin content, basic browser policies are easily circumvented. One example is Flash, which allows developers to specify a policy to enable cross-origin requests, regardless of SOP or CORS restrictions. A server can define the *crossdomain.xml* policy file, defining the origins from where remote requests are accepted. The Flash plugin is responsible for checking the file before carrying out the cross-origin request.

Plugins are a source for severe security problems, as illustrated by the numerous Java vulnerabilities in 2013 [251], eventually even leading to browser vendors recommending that Java should be disabled altogether. One potential source of vulnerabilities is the inability to deal with untrusted and potentially malicious input [265]. Therefore, close cooperation between browser vendors and plugin developers is crucial. In recent developments, the security of the Flash plugin has been greatly tightened. Flash is now effectively sandboxed on the OS level, preventing serious harm in case a vulnerability is found and exploited [154]. Alternatively, a new trend is emerging whereby plugin content is initially automatically disabled, but the user is then given the option to activate each piece of content separately, by a single click on a displayed *play* button [56] (See Figure 3.3).

3.2.2 Browser Extensions

Browser extensions extend the core functionality of the browser, and come in various flavors, from a simple toolbar to behavior-changing extensions. An extension is not associated with a MIME content type, but uses the exposed APIs to register hooks and react to events. Some popular examples of extensions are NoScript, which selectively disables JavaScript on web pages, AdBlock, which removes advertisements, or FireBug, a web development tool offering a debugger, giving a view on network traffic, etc.

Typical extensions for the Chrome and Firefox browsers are written in JavaScript[2], and are restricted by the API offered by the browser. On Firefox, extensions can access almost all browser internals, and can also access the file system or launch commands on the operating system. Chrome follows a more conservative approach, offering access to a select number of browser events, but preventing extensions from reaching outside the browser.

Browser extensions are very powerful, first of all because they can potentially access everything that happens within the browser. Additionally, Firefox extensions can also access other

[2]Native code is also supported, but discouraged, since it requires different versions for different platforms.

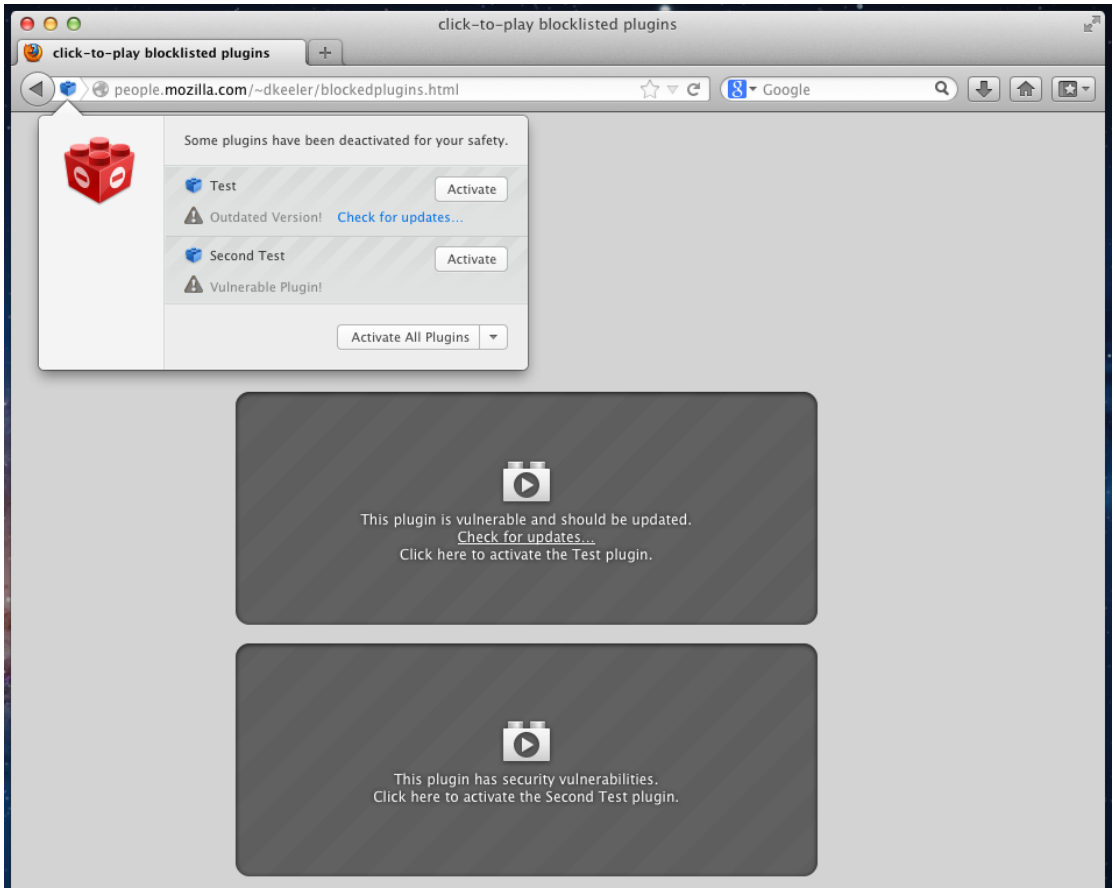


Figure 3.3: By requiring user interaction to explicitly enable running plugin content in a browser, automated background attacks against vulnerable plugins can easily be mitigated.

resources on the user's machine, making them even more powerful. Preventing every form of misuse is virtually impossible, even with the manual verification system employed by the Mozilla Add-Ons web site. Therefore, installing a browser extension effectively enlarges the attack surface, a risk accepted by the users. One example is a privacy violation by extensions that fail to correctly deal with private browsing mode (See Section 3.3), potentially exposing private information once the session is terminated [174].

3.2.3 Delegating Content Handling to Other Applications

Delegating content handling to other applications is a solution that has existed for a long time, and is mainly used by the `mailto:` links that invoke an email client. Whenever such a link is opened, the browser opens the system default or user-configured email client, using the parameters of the `mailto:` link.

HTML 5's *custom scheme and content handlers* take this delegation to a new level, enabling web applications to register themselves as the handler for a scheme or content type. Whenever the browser encounters a URI with a specific scheme, or a resource of a specific content type, the registered handler will be invoked. For example, a webmail application can register itself for the `mailto` scheme, allowing the user to directly compose a mail after clicking such a link. The possibilities are endless, with whitelisted schemes such as `irc`, `sms`, `magnet`, etc. In a process similar to these schemes, a web application can register itself as a handler for a specific content type, such as PDF documents or audio files.

3.3 User Features

In addition to the technical aspects and security policies in a browser, several user-targeted features are also relevant for security. Modern browsers use SSL/TLS security indicators, in an attempt to warn the user of suspicious or unsafe activities. Additionally, browsers have introduced private browsing modes, where a user can browse the Web without leaving a local trace. A last feature we cover in this section are the synchronization features, allowing users to share not only bookmarks and credentials across computers, but also physical resources such as printers etc.

SSL/TLS Security Indicators Traditionally, a browser displayed a small lock icon to indicate that a web site was loaded over a SSL/TLS-secured connection. As the features of SSL certificates changed, for example with extended validation certificates, the security indicators slightly changed as well. Currently, browsers use a combination of colors, lock icons and company names to indicate the level of security (See Figure 3.4). Unfortunately, these security indicators are often confusing and misunderstood [75, 225], therefore missing the targeted effect. This topic has once again become highly relevant with the rise of browsers on mobile devices with limited screen sizes.

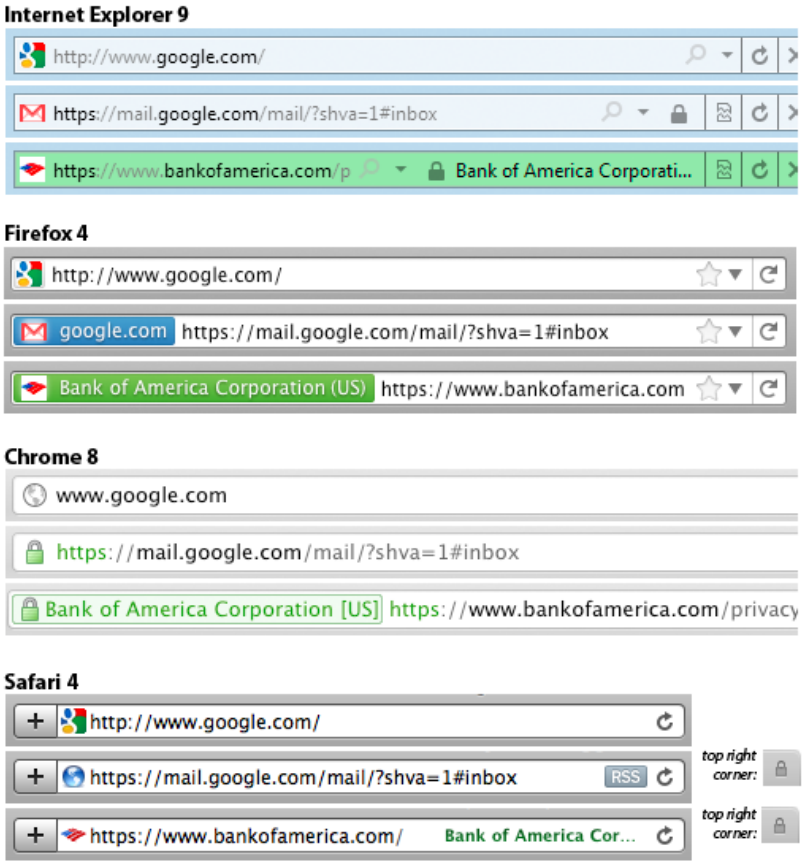


Figure 3.4: Browser vendors choose different ways to indicate the level of SSL/TLS validation the CA conducted, making it hard for users to grasp the precise meaning [225].

Private Browsing Modes When opening a window in *private browsing mode*, also known as *incognito mode* or *InPrivate*, the browser creates a window that allows the user to browse the

Web, without leaving a local trace after terminating the session. For example, all newly created cookies are removed, and no entries appear in the browser's history. The implementation details of the information that's available at the beginning differs per browser, but the general concept remains the same. Similarly, the offered guarantees, the behavior of extensions and permissions differ slightly between browsers.

Cross-Browser Synchronization Recently, browsers have started supporting synchronization services, allowing users to share bookmarks and stored credentials across multiple installations of a browser. Chrome takes this feature one step further, by also sharing devices connected to the machine of a running browser instance. For example, leaving a Chrome instance running on your workstation at work, allows you to print at work from your Chrome instance at home.

Chapter 4

Web Application Architecture

Even though every web application is different, they often have common characteristics, and even share the same conceptual components. Web application languages and development frameworks often offer some of these components out-of-the-box, requiring little or no effort from the application developer. More advanced systems, such as Content Management Systems (CMS), even provide a fully functional publishing system, lacking only the content.

This chapter identifies several common web application components, such as static content (images, stylesheets, ...), business logic (dynamic processing code), storage facilities, etc. Two aspects, session management and authentication, are fleshed out in detail, since they take a prominent role in the security of web applications.

4.1 Common Web Application Components

The model in Figure 4.1 shows several common components that can be found in almost any modern web application. The exact implementation of these components differs according to the type of application and the chosen deployment scheme (See Chapter 5), but the idea and functionality of each component remains the same. In this section, we briefly discuss each component's role, and in the next sections, we zoom in on the *Authentication* and *Session Management* components. The other components are fairly straightforward, and their internals are less relevant for the remainder of this document.

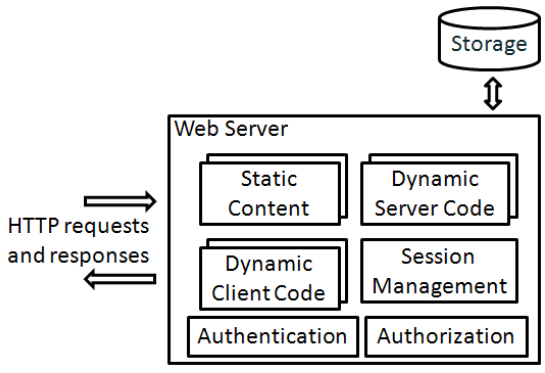


Figure 4.1: The common components that are part of any modern web application's architecture.

Static Content Almost every web application uses static content, such as company logos or other images, stylesheets, client libraries, etc. The static nature of this content makes it

inherently different from any dynamic content, which is user-specific and often depends on input from the request. Static content can often be cached and prefetched, but depending on the actual content, static content can still require authentication and be subject to authorization rules, making it unfit for caching.

Dynamic Client Code Complex and interactive web applications generate user-specific content, that needs to be processed or displayed at the client side. This content is considered dynamic client code, since it is dynamically generated at the server-side, but executed at the client. Dynamic client code typically contains sensitive information, and should not be cached

Dynamic Server Code The business logic of a web application is considered dynamic code, and is responsible for retrieving information and processing requests, often in close conjunction with backend storage mechanisms. Examples of business logic operations are the retrieval of contact information, sending an email or booking an appointment. The business logic of an application is typically closely coupled with authentication and authorization, and is almost always combined with static content.

Storage Dynamic web applications typically require backend data storage, to keep track of users and their data. The type of storage depends on the type of application, with database storage being the most popular choice, closely followed by filesystem storage, for example for temporary files or uploaded documents. The backend storage is typically only accessible through the business logic, which acts as the frontend.

Authentication Modern web applications offer access to large amounts of information, typically user-specific information, making authentication an important component within the web application. The authentication procedure allows a user to identify himself to the web application, enabling access to the protected features of the application. The stateless nature of HTTP (See Section A.4) couples authentication with session management, in order to maintain the authentication state across requests.

Authorization The functionality of a web application is typically shared among a large number of users, making authorization crucial to ensure that a user has the correct permissions to access certain features or specific data. Authorization checks need to be performed both on the available features, but also on the data used in these features, to prevent an authorized action on unauthorized data objects, such as making a wire transfer from an account that does not belong to the user. Naturally, authorization is closely coupled to authentication and session management.

Session Management By being able to tie multiple requests from the same client together as a session, web applications have the possibility to share state among these requests, such as the authentication status of the user, or a shopping cart with items to purchase. Session management is a crucial component for any dynamic web application, and is required by components such as the business logic, authentication or authorization. Several session management mechanisms are available to modern web application, as will be discussed below (Section 4.2).

4.2 Session Management

Session management allows a web application to link multiple requests from the same client together, enabling complex functionality. Once a session is established, the server keeps track of a session object, in which information can be stored. Typical examples are the authentication

state of the user, items in a shopping cart, partial transactions, etc. Usually, the session object is exposed by an API to the application's code, providing the necessary flexibility to developers.

Several techniques to achieve session management are available, of which cookie-based session management is undoubtedly the most popular. In the remainder of this section, we discuss cookie-based session management, parameter-based session management and session management using the authorization header.

4.2.1 Session Management with Cookies

When the server wants to initiate a session with the browser, it generates a new session identifier, associates it with the server-side session object and sends the session identifier to the browser using the *Set-Cookie* response header. From now on, the browser will include the session identifier using the *Cookie* request header, allowing the server to tie the request to the appropriate session, and locate the correct server-side session object.

The nature of the cookie mechanism makes cookie-based session management straightforward, applicable in almost all cases where cookies are supported. In cases where cookies are not supported, parameter-based session management is often used as a fallback.

The server-side session object is typically referred to using a *session identifier* (SID). When the client presents a valid SID, the server automatically assumes that the request belongs to this session. Therefore, a SID should be sufficiently long, random and unique, in order to avoid collisions, guessing attacks or brute forcing. Most web development languages and frameworks offer out-of-the-box support for session management, and developers are strongly encouraged to use them instead of rolling out their own session management mechanisms.

4.2.2 Session Management with URI Parameters

As an alternative to using cookies for session management, the session identifier can also be embedded as a parameter in the URI (See Section A.3). This enables the same session management techniques, with the delivery mechanism of the SID as the only difference.

Parameter-based session management has several disadvantages over cookie-based session management. The most noticeable disadvantage is that the session identifier needs to be present in the URI, requiring the server to rewrite every URI in the response to include the correct SID. This URI rewriting is especially problematic for web applications that generate URIs at the client side, for JavaScript-based retrieval of information. Additionally, embedding a parameter in the URI can cause unintentional leaking of the SID through the *Referer* header [73].

4.2.3 Session Management with the Authorization Header

Session management can also be implemented on top of the *Authorization* header. Since the browser sends the user's credentials on every request (See Section A.4), the server can keep track of the user sending the request, implicitly defining a session. This does not require the use of a session identifier, but uses the credentials of the user to identify the correct server-side session state.

The main disadvantage of this approach is that the user's credentials have to be sent on every request. Additionally, this approach does not support anonymous sessions when a user is not yet authenticated (e.g., a shopping cart on a web shop). A final, often used argument against use of the *Authorization* header is the lack of integration of the authentication prompt with the layout of the application.

4.3 Authentication

By completing the authentication process, a user can identify himself to the web application. This allows the application to provide access to user-specific data or features, providing a personalized

experience. A common example is a user authenticating to a web shop, to provide shipping details, provide payment information and complete the purchase. Many web applications allow unauthenticated access to some parts, but often require authentication at some point.

The most common way to authenticate is with a username and password, entered in a form and sent to the web application for verification. Upon first registration with a web application, a username and password can be chosen or are given. Future authentications depend on the knowledge of these credentials. The username and password are stored by the web application, typically in a database.

Another way of authentication uses a client certificate, which is used to create a secure SSL/TLS connection, where both client and server are authenticated with their certificates. The server verifies whether the client certificate is valid (e.g., has a valid signature by an expected CA), and can extract user-specific information from the certificate. Client certificates can be explicitly installed in the browser, or can be used in combination with electronic identity cards or other smartcards [67].

Alternatively to providing authentication, a web application can also depend on a third-party authentication provider to take care of the user authentication process. Examples of large authentication providers are OpenID, Google and Facebook. A web application simply directs the user to the authentication provider, who takes care of the whole authentication process. After a successful authentication, the application receives some evidence of the authentication, together with the user's details. The exact details depend on the implementation and user's configuration, but typically include a username, a name, an email address, etc.

As a consequence of many attacks on the authentication process and many breaches of backend databases, multi-factor authentication has been introduced. Multi-factor authentication no longer depends on a single secret, but requires multiple authentication factors to complete the process. Preferably, each factor uses a different channel, hardening the process against a single way of stealing credentials. A common example is two-factor authentication, with username and password authentication on one hand, and a token sent to your cellphone on the other hand. A relaxation of strict multi-factor authentication schemes is based on *trusted machines*, where a user can designate his computer as trusted, eliminating the need of the additional factors for authentications from that machine. The determination of a trusted or untrusted machine can be achieved by investigating additional client-side context information, for example using cookies or browser fingerprinting techniques (See Intermezzo 8).

Chapter 5

Deploying Web Applications

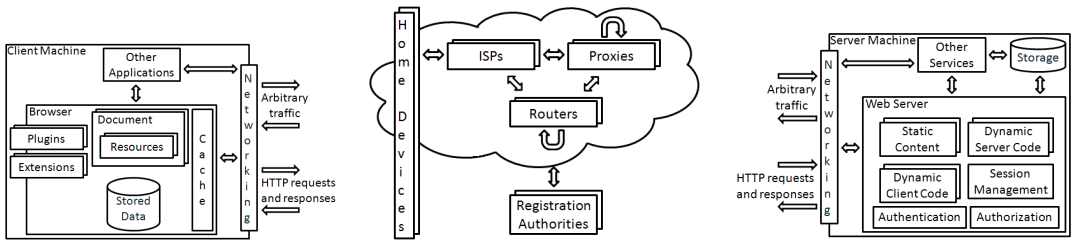
As described in the previous chapters, the Web is a distributed hypertext system, involving clients, servers and a network infrastructure. Deploying a web application requires knowledge of how each of these aspects work, and what's important to take into account. Furthermore, several evolutions in the client, server and network aspects have influenced the way in which web applications are deployed, and which properties need to be taken into account.

This chapter first introduces an overall model of the Web platform, highlighting the relevant aspects in each of the three key areas: client, server and network. The second part of this chapter describes several paradigm shifts in developing and deploying web applications, such as single-page applications, publicly available hotspots or fully customizable content management systems (CMS).

5.1 Model of the Web platform

The model of the Web platform in Figure 5.1 gives a high-level overview of the different components in the Web platform, which can be mapped to specific stakeholders in many different scenarios. For example, the client side of the model can be mapped to an end user, the network to a typical ISP network connected to the internet backbone, and the server side to a traditional web server, offering an interactive application. Similarly, the same components can be mapped to a scenario involving a powerful network attacker, able to perform man-in-the-middle attacks on the entire population of a nation.

In the remainder of this section, we discuss each aspect of the model of the Web platform in more detail. The goal of this section is to connect the sometimes abstract concepts of the previous chapter to a tangible and extensive model of the Web platform, completing the picture and providing enough insights to advance towards threats against the Web platform.



(a) Client Side (b) Network (c) Server Side

Figure 5.1: A model representation of the contemporary Web platform

5.1.1 Client Side

The client side covers the technology available to all users of the Web, regardless of their intent. Legitimate users use their machine and browser to visit web sites, conduct online shopping or manage their financial information.

At the client side, everything starts with the client machine, connected to a network, providing internet access to the user. The client machine is not only capable of running a web browser, but also runs numerous other applications, for example legitimate applications such as mail clients and word processors, or, in some cases, malicious software such as malware. These applications run isolated from each other, but the browser can choose to delegate the handling of certain content to a specific application (See Section 3.2.3).

The browser provides access to the World Wide Web, and has become the most-used application on any modern computer. Modern browsers incorporate a myriad of technologies, such as HTML 5, multimedia features and powerful JavaScript APIs (See Section 2.2). Additionally, browsers offer several mechanisms for extending their functionality, for example with plugins to handle arbitrary content, or using browser extensions to add functionality (See Section 3.2).

The browser is responsible for loading and rendering documents, which in turn can contain several external resources (See Section A.1). These resources are loaded over the network using HTTP, potentially being cached by the browser to allow a quick retrieval when future documents use the same resources. Documents themselves are also able to store data within the browser, using cookies (See Section A.5) or JavaScript APIs supporting data storage (See Section 2.2).

5.1.2 Server Side

Using the server side infrastructure, web applications, services and resources can be made available for users to visit, for developers to incorporate in their own applications, etc.

The server machine is capable of running numerous services, of which the web server and storage service are most relevant within the Web platform. The web server hosts web applications or resources, and is responsible for the server-end of the HTTP protocol (See Section A.4).

Most modern web applications contain a dynamic server-side part, where the server does the processing of requests, resulting in responses to the client. This dynamic code makes use of several common components, such as authentication, session management and access control or authorization (See Chapter 4). Typically, dynamic server-side applications are supported by a storage service running on the server side.

The web server responds to client requests, providing the browser with content and resources to process and render. Content sent to the browsers can be of static nature, such as images or videos, or can be dynamic, such as server-side generated HTML documents, stylesheets or interactive script code, resulting in a fully-functional client-side application.

5.1.3 Network

The network component is crucial in the Web platform model, since it connects the client side with the server side. The network infrastructure has endpoint components, such as home or company routers and access points, as well as a general networking infrastructure provided by ISPs. Additionally, several registration authorities play an important role as well, with the domain name registrars and certificate authorities being the most important ones.

The critical infrastructure of the network is provided and managed by ISPs all over the world, creating one connected network. Both client and server side are connected to this network, using local infrastructure components such as routers, cable or DSL modems, wireless access points, etc.

Within the large, connected network, several additional services, often referred to as *middleboxes*, can have an impact on the information flow through the network. Examples are *proxy*

servers, which relay traffic on behalf of a user, masquerading the user in the process, *web application firewalls* (WAF) which inspect web traffic and attempt to detect or stop web attacks, *load balancers* which distribute the HTTP request load for a web site over multiple machines, and *SSL terminators*, which are responsible for dealing with the SSL/TLS encryption of the connection with the client at the server end, and forwarding decrypted incoming traffic to an internal server. Most of these middleboxes are used by corporations and ISPs to protect their users and networks, but some are open to users on the Internet, with freely available proxy servers and the popular *Tor* network [79] as an example.

The registration authorities are technically not part of the network, but their support is needed to enable certain aspects of the network. The domain name registrar is responsible for registering domain names, such as *example.com* and pointing it to the correct IP address using the DNS system, which is not covered in this document, but for which relevant literature is available [145, 151]. The other registration authority are *Certificate Authorities*, which are responsible for creating and managing certificates, used for secure communication, such as HTTPS (See Section A.4.4).

5.2 Client-side Evolutions

Since the first web page, the client side environment of a web application has changed significantly. A first evolution was the introduction of JavaScript and CSS, enabling so-called *dynamic HTML*, resulting in interactive and dynamic web pages. Using the JavaScript APIs to interact with the DOM, pages could modify themselves, dynamically change the style of elements or validate forms. Several of these script-based features have been standardized in the most recent HTML standard, which for example offers several utilities to offer validation of form inputs.

Another JavaScript-driven evolution is the use of asynchronous network requests using the XMLHttpRequest object, enabling a new paradigm for client-side web applications. By being able to asynchronously make requests, applications can load data in the background, request partial updates or submit locally generated data to the server. This paradigm leads to *single-page applications*, where a whole web application is contained in a single page, fully controlled by JavaScript. The office suite offered by Google Docs is an excellent example of an elaborate single-page application. In support of this important trend, the newly introduced HTML5 History API offers the necessary functions to create entries in the browser's history, enabling single-page applications to behave exactly like multi-page applications, delivering a smooth user experience.

The growing JavaScript capabilities at the client side have sparked the development of JavaScript libraries that make development easy by offering easy access to the extensive client-side environment. Popular examples are, among others, *jQuery* [245] and its numerous plugins, and the *Prototype JavaScript Framework*. Alternatively, JavaScript templating systems and frameworks further support complex client-side applications, by allowing the definition of a template with data bindings, offering easy ways to create views for large volumes of data. Popular examples include *AngularJS* [114], *backbone.js* [13] and *Handlebars* [153].

The evolution towards client-side storage capabilities enables web applications to store data at the client-side. Overshooting its intended use, developers have discovered that the client-side storage facilities can be used to store application code as well, effectively transforming them into an additional cache mechanism. While this enables fast-loading applications, it also mixes data with code, a recipe well-known to lead to security problems. Compromise of a local cache of application code [171] can have serious consequences for the integrity of the client-side execution context.

The most recent paradigm evolution is the *appification of the Web*, where applications are packaged as an app which runs independently on the client-side, communicating with a remote API to retrieve and process data. The first popular platform for such *web apps* were smartphones, where web apps gained the capability to style and appear as a native smartphone app. Nowadays,

modern browsers also support the web app concept, and Google even distributes these apps through the Chrome Web Store. Additionally, the introduction of the application cache as part of HTML 5 [33], enables the caching, and hence the offline usage of such web apps.

5.3 Server-side Evolutions

In stark contrast to static server-side content, dynamic server-side content is actually able to process a request when it comes in, deciding how to handle it based on the presented parameters in the request. This essentially enables the processing of form data, executing search queries, authenticating users, etc.

Initially, this behavior was enabled by CGI (Common Gateway Interface), where the web server handles the HTTP connection, places the request parameters and context information in environment variables, which can in turn be processed by a server-side script, for example a Perl script. The web server is responsible for capturing the output of the script, and sending it as an HTTP response to the client.

CGI has been slowly replaced by server-side scripting languages, such as PHP or Ruby, which are designed for web development. These scripting languages typically run within the web server, although many still support a CGI-like mode of operation, where they run in a separate process. The main advantage of server-side scripting languages is the native support for some basic web concepts, such as headers and session management. Naturally, several frameworks have been built on top of these languages to offer more common features to web developers, allowing the rapid development of new web applications. Popular examples for PHP are *CodeIgniter* [87] and *CakePHP* [103], while for Python development there is the *mod_wsgi* [83] module and the *Django* [80] framework, and Ruby is dominated by *Rails* [127].

While server-side scripting languages offer basic support for common web concepts, they lack an underlying multi-tier architecture, often used in business applications to separate presentation logic, application processing and storage into different tiers. This gap is filled by multi-tier web development frameworks and associated application serves, of which the *Java Enterprise Edition* is the most prominent example, followed by *Microsoft's ASP.NET platform*. As an alternative to a multi-tier platform, where each tier has a specific language, a multi-tier language allows the development of an application in a single language, that is compiled into split code for each tier in the Web platform, such as presentation, logic and storage. A common example of a multi-tier language is *Google Web Toolkit*.

On an even higher level, Content Management Systems (CMS) offer a complete package containing all functionality needed to deploy a content-driven web application. After installing a CMS system, a user simply needs to add content through the user interface, after which the web application is operational. Typically, CMS systems have a modular architecture, where additional modules can be installed to add features and functionality. Popular examples of CMS systems are *Drupal*, *WordPress* and *Joomla!*.

A completely different server-side aspect is the underlying deployment infrastructure of a web application. The deployment model consisting of a single web server has quickly evolved into a load-balanced model, where multiple servers distribute the load of the many users visiting the application and SSL Terminators off-load the processor intensive authentication and encryption processing from the actual web servers. Recent evolution towards scalable infrastructures has resulted in a service-driven industry, where a deployment infrastructure or platform is offered as a service, so called *Infrastructure as a Service (IaaS)* and *Platform as a Service (PaaS)*. Customers of such a service provider simply rent the necessary amount of processing power, and can scale flexibly depending on their needs at any time. A large IaaS provider is *Amazon EC2*, where a customer can rent virtual computing power to run his own virtual machines, and a well-known PaaS provider is *Google App Engine*, a Google platform for developing and deploying web applications in the Google data center.

A second deployment evolution concerns static content, which is the same for numerous users,

but needs to be fetched by each user separately. Initially, this traffic pattern was optimized using web caches at strategic places, for example within an ISPs network. A more recent evolution comes through Content Delivery Networks (CDN), which exploit geographical advantages to serve content from a nearby data center, limiting the network travel distance for static content. Due to their distributed nature, CDNs are also very effective in mitigating DoS attacks (See Intermezzo 5).

Intermezzo 5: Content Delivery Networks in Practice

Akamai is one of the world's largest content delivery networks, renting out storage space to consumers that want to exploit geographical advantages, in order to increase the speed of their content distribution. Akamai operates more than 100,000 servers in more than 1,800 locations across nearly 1,000 networks.

Operating a CDN comes with several challenges, such as *content delivery cost minimization*, *end-user mis-location* and *network bottlenecks*. Minimizing the cost of content delivery to the end user is crucial, especially with the rising competition in the market of content delivery. Key to this challenge is the user assignment strategy, which is currently mainly driven by economic aspects, such as bandwidth or energy cost. The second challenge involves locating the appropriate content server for a user, which is currently done by determining the location of the DNS resolver used. However, recent studies show that in many cases, users do not use a nearby DNS resolver, leading to a selection of a less-appropriate content server. Finally, performance prediction is made difficult by the limited information about actual network conditions. Even worse, without sufficient information about the characteristics of network paths between the CDN servers and an end-user, CDNs might actually further stress current bottlenecks, or even create new ones.

Two key enablers for overcoming these challenges are *informed user-server assignment* and *in-network server allocation*. Fulfilling these enablers can be achieved by collaborations between CDNs and ISPs, where the latter are capable of providing the CDN with the appropriate information, creating room for CDN-ISP alliances, with the possibility for joint service deployment.

The *NetPaaS* prototype [105] incorporates the two key enablers, and effectively implements a system for CDN-ISP collaboration. The authors report on the evaluation of the prototype with traces from the largest commercial CDN and a large tier-1 ISP, as well as on the benefits fro CDNs, ISPs and end-users.

The main conclusion of this work is that the NetPaaS platform leads to a win-win situation with regards to the deployment and operation of servers within the network, and significantly improves end-user performance.

Source: Frank, Benjamin, et al. "Pushing CDN-ISP Collaboration to the Limit." ACM SIGCOMM CCR 43.3 (2013). [105]

5.4 Network Evolutions

One obvious evolution in the networking world is the development of new protocols, such as the Internet Protocol version 6 (IPv6) [74], which vastly expands the address space (from 4.3 billion to 3.4×10^{38} addresses), or the HTTP/2.0 protocol [30] (See Section 2.3), which improves the transmission of HTTP traffic on the wire. Next to the development of new protocols, new uses for existing protocols are also being deployed. For example, the widely-supported HTTP and DNS protocols, typically allowed to pass through most firewalls, are often used as a carrier for other protocols. One example is tunneling the *Secure Shell* (SSH) protocol over HTTP or DNS, effectively bypassing imposed network-level restrictions. In response to these tunneling

practices, firewalls are deploying *deep packet inspection*, allowing them to identify non-compliant traffic.

Another network evolution is sparked by the deployment model, combined with the increased connectivity of modern devices. End users connecting to the internet used to hook up their computer or device directly to a telephone line, establishing a dialed connection to an internet service provider. This slow connection slowly faded away with the coming of broadband over cable or DSL, offering high-speed, always-on internet connections, which are very attractive to connect multiple devices using the same connection.

This need prompted the quick evolution of home networks, where a wired or wireless router distributes the internet connection to the home network, putting all internet-capable devices in the home onto the internet. With the home network in place for computers, it could easily be used by numerous other devices, such as smartphones, tablets, printers, storage devices, etc.

The evolution in mobile devices, such as smartphones and small notebooks, has sparked the availability of publicly accessible wireless networks, so-called hotspots. Public places such as bars, coffee shops, railway stations and airports all offer their own publicly accessible network, shared among all visitors. Recently, several ISPs have transformed their endpoint devices also know as "home routers"), located in user's homes, into publicly accessible hotspots as well, further spreading the availability of public wifi networks. The use of these public networks is typically controlled by an authentication procedure, for example by subscribing to the provider's network service, or by identifying yourself as a customer of the party hosting the network, such as coffee shops or libraries.

Several technologies have been invented to alter or divert the data flow through the network, driven by security, anonymity and performance. Examples are the use of Virtual Private Networks (VPN), where specialized software enables a remote machine to become part of a local network located elsewhere, or onion routing techniques, such as Tor, where packets are sent through several layers of relaying hosts in order to hide the origin of the data. The previously mentioned CDNs also divert the flow of traffic in the network, by choosing geographically well-located hosts to retrieve content from. CDNs offer several performance and security benefits for legitimate customer,s but potentially introduce attack amplification capabilities for malicious users.

Another category of technologies focused on increasing security are monitoring services investigating the nature of a web site, potentially whitelisting or blacklisting sites based on their content. Blacklisting typically occurs when sites serve malware or virus-infected sites. Going further than monitoring are honeypot sites, which are fake web sites with realistic appearances, aiming to attract attackers in order to investigate their techniques, or lure them away from the actual sites.

Finally, many web applications have started to incorporate location-based services. The use of location-based services can be straightforward, such as showing a map of your current location, but can also serve as a form of access control, preventing users from certain countries from viewing particular content. For example, the BBC web site, and especially the integrated video player, offers a wholly different experience when accessed from inside the UK than from outside. In a response to location-based access control, several anonymous proxy services have appeared, allowing a user to tunnel his traffic to a proxy server located elsewhere in the world, potentially evading such location-based access control systems.

Part II

Threats to the Web Platform

Chapter 6

Assets

Assets are valuable resources within the entire web application model, spread over clients, networks and servers. Assets are important to an attacker, since obtaining or controlling them helps him achieve his goals, mainly driven by economic incentives [146]. Identifying these assets is therefore crucial for determining the focus of an attacker in the web application model.

We analyze the complete web application model from Part I, identifying important assets along the way. The assets identified are approached from a technical, application-agnostic point-of-view, but the intention is to derive an application-specific risk analysis, as the assets can be instantiated in different ways depending on the concrete deployment scenario. For each asset, we explain its importance and the attacker's incentives for compromising this asset, and analyze how an asset can be compromised, resulting in high-level threats towards the asset.

High-level attack tree diagrams are used to visualize the different approaches an attacker can take to compromise an asset. These trees start from the asset in the top node, and end with high-level threats in the leaf nodes. For example, to compromise the *server-side content storage* asset, an attacker might aim to use a *direct connection to the server-side storage*. Leaf nodes can also depend on the compromise of another asset, which is indicated in purple on the attack trees.

This chapter presents the identified assets, grouped into three categories. First, we discuss the infrastructure assets, in this case the client and server machines. Next, we cover several application-specific assets, such as storage facilities, application transactions and authenticated sessions. Finally, we address user-centered assets, such as personal information or authentication credentials.

6.1 Infrastructure Assets

Infrastructure assets are resources within the web model that enable web applications, independent of a concrete implementation or deployment of a web application. From the web application model, we identify two concrete infrastructure assets: the *server machine* and the *client machine*.

These assets are valuable to an attacker since their compromise enables the escalation of the attack towards application-specific or user-specific assets. A secondary incentive lies in the fact that these assets are also physical resources with processing power and network bandwidth. Controlling such resources, for example as part of a botnet, enhances the attacker's power and capabilities, such as carrying out denial-of-service attacks or brute force computations.

6.1.1 Server Machine

By compromising the server machine, an attacker gains full control over the target web application in all its aspects, including code and data. Additionally, the server machine itself is also a

valuable resource for running alternate services (e.g., illegal downloads), proxying traffic, masquerading other attacks, etc. Often, server machines host multiple web sites or even multiple virtual servers, all of which can be compromised by gaining control over the server machine. The attack tree in Figure 6.1 shows three ways of compromising the server machine through a vulnerable web application.

Abuse Server-side Application Privileges Web applications are often granted special system privileges, such as executing commands or reading system files, privileges needed for example to resize and convert uploaded images, or to store files on the local filesystem. Since these privileges are often executed based on specific input provided by a user of the web application, they are vulnerable to manipulation, leading to the execution of arbitrary commands (e.g., *command injection*) or the reading or manipulation of important system files (e.g., *path traversal*).

Run Attacker-controlled Server-side Component Modern web applications often integrate existing externally-provided functionality using third-party components, such as logging, identity management or image manipulation libraries. If an attacker succeeds into compromising one of these components, s/he can typically gain control over the entire web application. In addition to willful integration of potentially malicious components, a feature accepting user uploads can also be abused by an attacker to upload a malicious component, such as a *web shell*, potentially compromising the server machine.

Escape Server-side Sandbox/Environment Web applications running on a server typically depend on an underlying infrastructure, such as an HTTP server, an operating system etc. By providing carefully crafted requests or malicious input, an attacker might be able to escape the sandbox or environment of the web application. For example, a crafted network request could trigger a vulnerability in the pre-processing filters of the HTTP server, giving the attacker control over the process and potentially the entire server machine.

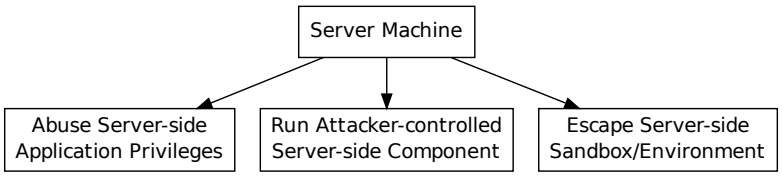


Figure 6.1: The *server machine* asset can be compromised in three different ways. This document focuses on entry points through a vulnerable web application.

6.1.2 Client Machine

The client machine is of high value to an attacker, since it offers full control over the interactions of the victim with any visited web application, from blogs to online banking services. The same goal can be achieved by controlling the user's browser, which requires control over the client machine. Therefore, we consider having control over the client machine as similar to having control over the browser. One additional advantage of having full control over a client machine, is the possibility to use it as a resource. Client machines often have a lot of processing power and network bandwidth, which an attacker can exert to expand a botnet, send spam, generate virtual money, etc. The attack tree in Figure 6.2 shows how the client machine can be compromised.

Run Attacker-controlled Client-side Component By controlling a component installed at the client-side, the attacker can easily gain control of the client machine. Typically, attackers attempt to trick users into installing a malicious client-side component, such as a fake Flash plugin or malicious browser extension. Additionally, an attacker can exploit a browser vulnerability to install malware on the client machine.

Escape Client-side Sandbox/Environment Similar as on the server-side, the client-side of a web application depends on the client infrastructure, such as an operating system, a browser, browser extensions, several plugins, etc. By exploiting vulnerabilities in any one of these, an attacker can easily escape the sandboxed environment of the browser, gaining control over the client machine.

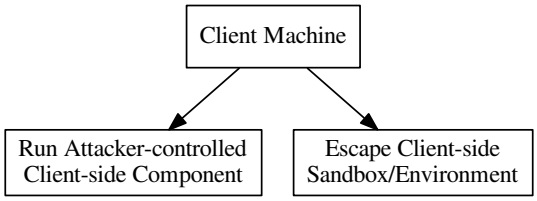


Figure 6.2: Compromising the *client machine* is typically achieved by tricking the user into installing malicious software, or by exploiting the existing infrastructure.

6.2 Application Assets

Application assets are valuable elements within a web application, either on the client side, on the server side or in transit. Application assets are typically related to accessing the application or its data, for example the *server-side data storage*, or an *authenticated session*. An important distinction between application assets and user assets (Section 6.3) is the scope of the asset. Application assets are bound to a single web application, while user assets are typically relevant across web applications.

Application assets are particularly interesting to an attacker, since they allow him to influence the core and operation of the application. For example, an attacker able to manipulate *application transactions* within an online banking system can potentially steal money from the victim's accounts, as illustrated by the numerous banking trojans of the past few years [81, 94]. A second important incentive, next to influencing the application's behavior, is stealing any of the stored information, sometimes even all of the stored information.

6.2.1 Server-side Content Storage

Having access to the server-side storage facilities allows an attacker to steal application data, or even manipulate application code or content. Attacks aimed at server-side content storage often result in the theft of large amounts of user data, such as usernames, passwords, email addresses, credit card information, customer records, patient files, etc., which can be directly monetized through underground market places. In addition to user data, server-side storage also contains application-specific information, such as management documents, industrial designs, etc., all highly valuable and attractive attacker targets. The attack tree in Figure 6.3 illustrates how this storage can be compromised.

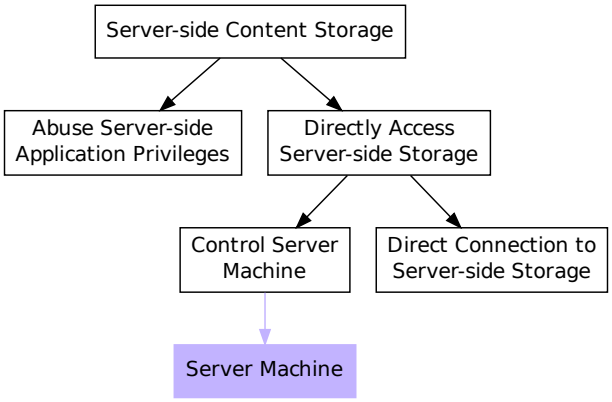


Figure 6.3: The *server-side content storage* asset can be compromised through the application, or by directly connecting to the storage facilities. A compromise of the *server machine* asset (in blue) also leads to direct access to server-side storage facilities.

Abuse Server-side Application Privileges The web application typically has access to the underlying storage, meaning that these access privileges can be abused by an attacker. Whenever the application relies on user input to construct a command to read or manipulate data, a potential injection vulnerability awaits. A common example is *SQL Injection*, where the attacker is able to inject SQL code into a query, resulting in unauthorized access to the database, either as a read, write or delete operation.

Directly Access Server-side Storage Apart from going through the web application, an attacker can also attempt to directly access the server-side storage. One possibility is to gain control of the server machine, and escalate the attack from there. The second possibility is to directly connect to the storage server or service, using valid authentication credentials, if any are required. Credentials can be obtained by brute force or guessing attacks, or by stealing them from the web application or server machine.

6.2.2 Client-side Content Storage

Recently introduced capabilities in the browser allow a web application to store data at the client-side, within the browser. Data stored at the client-side can be a cached version of server-side data, client-side application code or user-specific data calculated at the client-side. By gaining access to this data, an attacker can obtain personal user information and application-specific information. The attack tree in Figure 6.4 shows how client-side content storage can be compromised.

Directly Access Client-side Storage Directly accessing the data stored at the client side is the most straightforward approach. This can be achieved by taking control over the client machine or browser, as covered in the infrastructure assets (Section 6.1), or by running attacker-controlled code within the application's context. This essentially gives an attacker control over the client-side application, allowing him to directly access the storage facilities used by the web application.

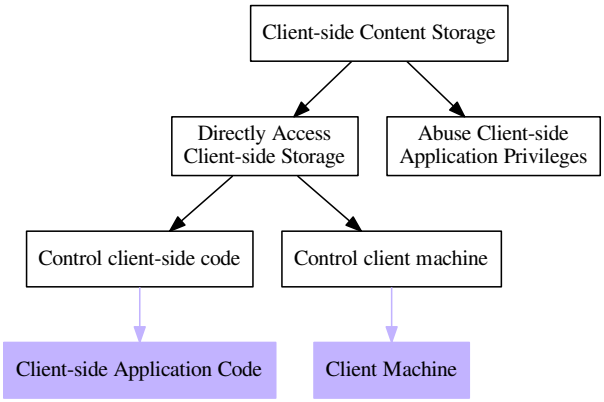


Figure 6.4: The *client-side content storage* asset can be compromised by controlling the client machine, controlling a component within the application or by abusing the application's legitimate access privileges.

Abuse Client-side Application Privileges A second way to access client-side storage is by abusing the application's privileges, in a similar way as an attack on the server-side. If the client-side application depends on untrusted input to construct commands towards the storage facility, an attacker might be able to carry out an injection attack. Such attacks can be initiated through crafted URIs, attacker-provided application data or through local communication within the browser, for example between two contexts or workers.

6.2.3 Content in Transit

Content in transit can either be application data or application code, where the former is interesting for an attacker to eavesdrop on, and the latter is interesting to manipulate and forge. Being able to read application data can not only result in the theft of personal information, but can also lead to an escalation of the attack. Similarly, being able to manipulate requests to the server or responses to the client, an attacker can interfere with the web application's operations. The attack tree in Figure 6.5 shows three different ways to compromise content in transit.

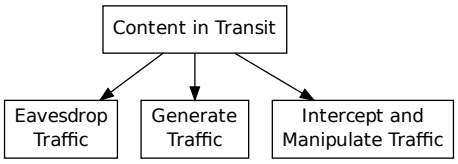


Figure 6.5: Compromising *content in transit* allows an attacker to steal information or escalate the attack, by reading, generating or intercepting network traffic.

Eavesdrop on Traffic Eavesdropping on a network is fairly straightforward nowadays, with many publicly available wireless networks and hotspots. An attacker on the same wireless

network can easily pick up any traffic, especially on unprotected networks. Eavesdropping can not only lead to the theft of personal information or sensitive data, but can also lead to an escalated attack. For example, if an attacker manages to steal usernames and passwords, he can fully compromise the user's account, and maybe many more.

Generate Traffic Under traffic generation, we consider any network traffic generated by the attacker, without impacting existing traffic. For example, if the victim's browser sends out a request for a resource, an attacker might generate a response that appears to come from the target server before the actual response reaches the browser, tricking the browser into accepting the malicious response. Such tactics can lead to the full compromise of the client-side context, since an attacker can manipulate included resources, such as scripts, plugin content, etc.

Intercept and Manipulate Traffic A more advanced attack involves intercepting and manipulating traffic, where the attacker acts as a *man-in-the-middle* (MitM). This not only allows an attacker to send forged responses, but offers full control over all the network traffic of a user. Depending on the security measures deployed by an application, a MitM attack can be detectable by a careful user, or completely stealthy for even the most security-conscious users.

6.2.4 Client-side Application Code

Compromising the client-side application code gives an attacker control over the client-side context of the web application, enable many other attacks, such as accessing *client-side content storage*, compromising *application transactions*, etc. Delivering and running client-side application code involves a lot of infrastructure, resulting in multiple ways to compromise this asset, as shown by the attack tree in Figure 6.6.

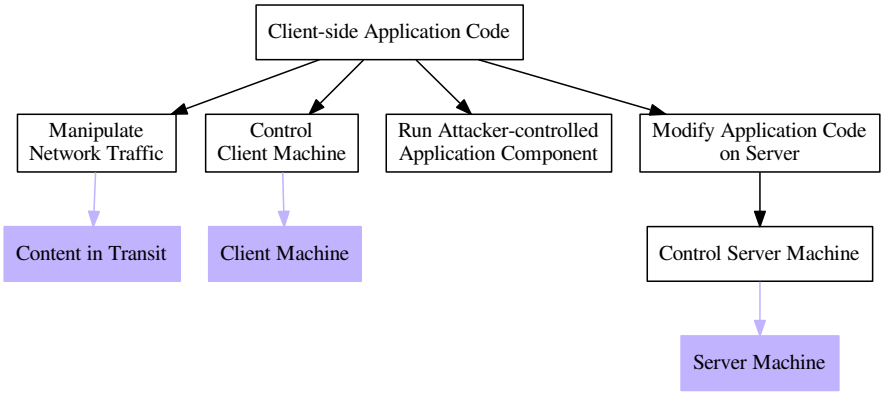


Figure 6.6: Compromising the *client-side application code* asset can be achieved by compromising other assets, or by controlling a component that's integrated with the application.

Several ways to compromise client-side application code have already been covered. For example, by manipulating network traffic, an attacker can inject malicious code into the client-side context, gaining control over the client-side application. When an attacker controls the client machine, he gains the capability to interfere with any client-side web application, within any browser. Finally, when an attacker gains control over the server machine, modifying the code that will be sent to the client is straightforward, allowing further escalation of the attack.

Run Attacker-controlled Application Component A previously uncovered way to compromise client-side application code is through an attacker-controlled component within the application. One way is for an attacker to trick web developers to include malicious components willingly, for example by masquerading to them as a legitimate, useful library. Another way is by focusing on existing third-party libraries and scripts, which are often included in numerous web applications (See Intermezzo 4). By compromising such a component or its hosting infrastructure, an attacker can gain control over the client-side context of the applications that use this component.

6.2.5 Authenticated Session

An authenticated session is valuable for an attacker, since it allows him to access the target application, posing as a legitimate user. It not only offers access to features requiring authentication, but also leads to a potential full compromise of a user's account. The attack tree in Figure 6.7 covers multiple ways to obtain an authenticated session.

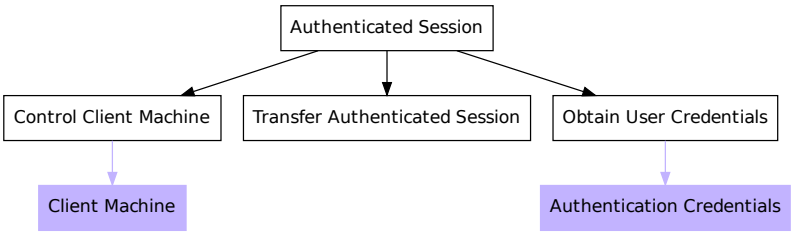


Figure 6.7: Having an *authenticated session* allows the attacker to access the target application, posing as a user, giving him the same level of access as the impersonated user.

Two obvious ways to compromise the authenticated session asset are by controlling the *client machine*, giving the attacker full control over every aspect that happens at the client, and by obtaining the victim's *authentication credentials*, allowing the attacker to establish his own authenticated session, in the name of the victim.

Transfer Authenticated Session Another way to compromise an authenticated session is by transferring an authenticated session, established by the user, to an attacker-controlled machine. The feasibility of this attack depends on the security parameters configured by the application, but for typical cookie-based session management mechanisms, it suffices to obtain the cookie with the identifier associated with the user's session. This cookie can be stolen from an application's client-side context or from the network.

6.2.6 Application Transactions

Application transactions cover the actual operations of a web application. Being able to compromise this asset allows an attacker to perform unauthorized actions in the name of the victim, often with serious consequences. A compromise of this asset means that a transaction or operation is executed within the web application, through the appropriate channels, making it hard to distinguish the malicious operations from legitimate user operations. The attack tree in Figure 6.8 shows several ways to compromise this asset.

Manipulating application transactions can be achieved by compromising other assets. Examples are impersonating a user to the web application by obtaining an *authenticated session* in the user's name. Another example is by manipulating *content in transit*, changing a user's

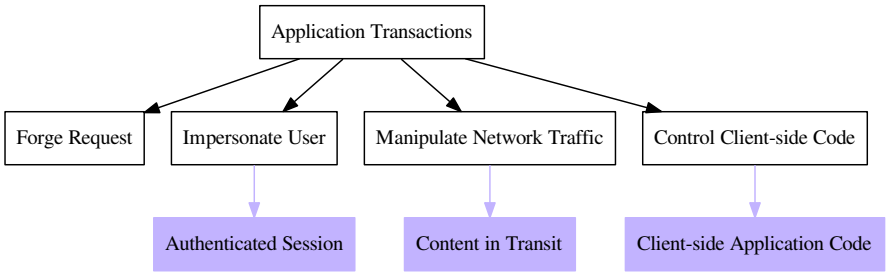


Figure 6.8: Compromising the *application transactions* allows an attacker to influence the operation of a web application, while looking legitimate. Compromising this asset can be done through other assets, or directly by forging requests.

request, or injecting additional requests associated with the user's session. Finally, an attacker controlling the client-side context can easily modify the application's behavior, including the application transactions that are initiated from the client-side.

Forge Request One way, not previously covered, to compromise application transactions is by forging requests. A forged request is a request that appears legitimate to the web application, and is generated by confusing the user or browser. Typically, forged requests are possible due to certain design decisions in the way the Web works, and push certain useful and legitimate features beyond their designed intentions, with, as the most common examples, Cross-site Request Forgery (CSRF) and clickjacking.

6.3 User Assets

User assets represent valuable data that belongs to a person, such as *authentication credentials* or *personal information*. The difference between user assets and application assets is the scope of the data. User assets involve information that is bound to a specific user, but might be valuable across different applications. Application assets on the other hand, focus on data that is relevant within one application, but less or not relevant across applications.

Precisely this cross-application nature of user assets makes them valuable for an attacker. Especially authentication credentials, which are often shared across multiple applications, that provide the gateway to impersonating the user, offering access to valuable services within numerous online applications. Common attacks on user assets involve credit card fraud, sending spam, etc., or in the worst case, identity theft in the physical world.

6.3.1 Authentication Credentials

Authentication credentials are very valuable for an attacker, since it allows him to establish an *authenticated session* with the target application, in the user's name. Additionally, most users re-use their credentials in multiple places, giving the attacker access to these applications as well. The attack tree in Figure 6.9 illustrates how an attacker might obtain authentication credentials.

Trick User into Revealing Information An attacker can try to confuse the user, tricking him into revealing his authentication credentials for the target application. This attack is known as phishing, and is very prevalent on the Web.

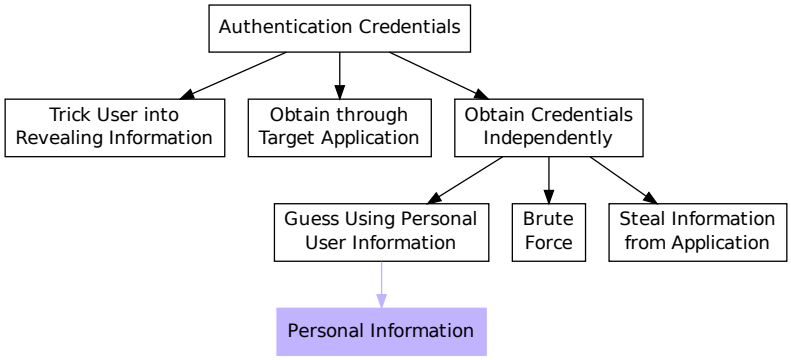


Figure 6.9: *Authentication credentials* give an attacker access to the target application, in the user's name.

Steal Credentials from Target Application An attacker able to compromise the target application, can often retrieve authentication credentials for numerous users as well. One common example is by compromising the *server-side content storage*, revealing all user information to the attacker.

Obtain Credentials Independently Alternatively, an attacker can use several ways to obtain authentication credentials, without involving the user or target application. One way is to simply guess a predictable password, using *personal information* known about the user. Instead of educated guessing, an attacker can also apply brute force ("dictionary attack") to obtain a password for a known username. Finally, stealing user information from other applications can result in a valid set of authentication credentials for the target application, due to re-use of authentication credentials by the victims.

6.3.2 Personal Information

Web applications contain a wide variety of personal information about users, apart from *authentication credentials*. This information is valuable for an attacker to track users, to deploy very targeted attacks, or to guess a user's password, either directly or through some easy-to-answer security questions. The attack tree in Figure 6.10 shows several ways to obtain personal information about a user.

If an attacker succeeds in running attacker-controlled code within the browser, s/he might be able to extract information from the client-side context, using readily available APIs or side-channel attacks. Being able to run attacker-controlled code within the target application's context opens even more options for obtaining personal information about the user.

Harvest Public Information Many applications expose information about their users in a public way, such as social networking or messaging sites. Harvesting this public information gives an attacker insights into the user's life, allowing for very targeted attacks.

Steal Information from Application By compromising assets of a web application, such as the *server-side content storage*, an attacker can obtain all kinds of user information.

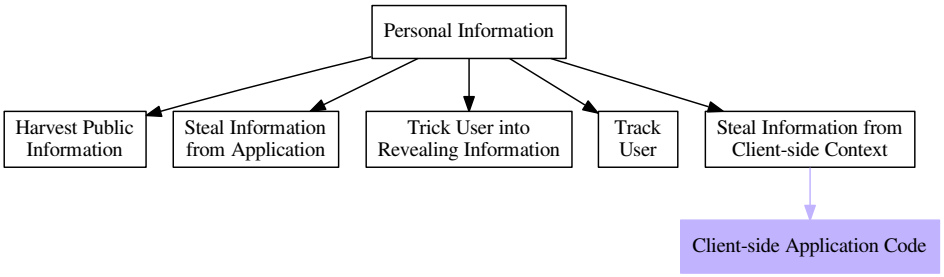


Figure 6.10: *Personal information* about a user enables tracking, identification or targeted attacks directed at the user.

Trick User into Revealing Information As with tricking the user into revealing *authentication credentials*, an attacker can trick the user into revealing personal or sensitive information, such as credit card numbers or a social security number.

Track User An attacker can deploy tracking techniques, which keep track of the sites visited by a user, to compile a user profile. While this may not lead to a personal identification of the user, it gives a profile of the visited sites, the user's interests and other aspects of the user's life.

6.4 Mapping Threats to Assets

The individual attack trees presented in this chapter give a per-asset breakdown of the threats that can lead towards the compromise of the asset. Figure 6.11 shows an overview tree containing all assets and their associated threats. In addition to the overview, the tree also shows relations between assets, where a compromise of one asset can threaten the integrity of other assets, potentially allowing an escalation of the attack. This dependency relationship between assets is also crystallized in Table 6.1.

The identified threats are a high-level description of the approach of an attacker towards compromising an asset. In Part III, we analyze each threat, starting from specific attacker capabilities as defined in the next chapter, resulting in the description of concrete attack techniques and potential countermeasures. The selection of these attack techniques and countermeasures is based on their prevalence, associated risk and potential impact, as indicated by the OWASP top 10 [261], the CWE/SANS Top 25 most dangerous programming errors [181] and relevant academic work, as presented in important security-related journals and conference proceedings. Part III is divided into seven chapters, each covering a specific set of threats, targeting a single or related assets. We briefly cover the sets of threats, leaving further details for Part III.

Impersonating users becomes possible if an attacker controls an *Authenticated Session*, or if an attacker is able to obtain *Authentication Credentials*. Specifically, the following threats can lead to the compromise of these assets:

- Transfer Authenticated Session
- Trick User into Revealing Information
- Steal Credentials from Target Application
- Obtain Credentials Independently

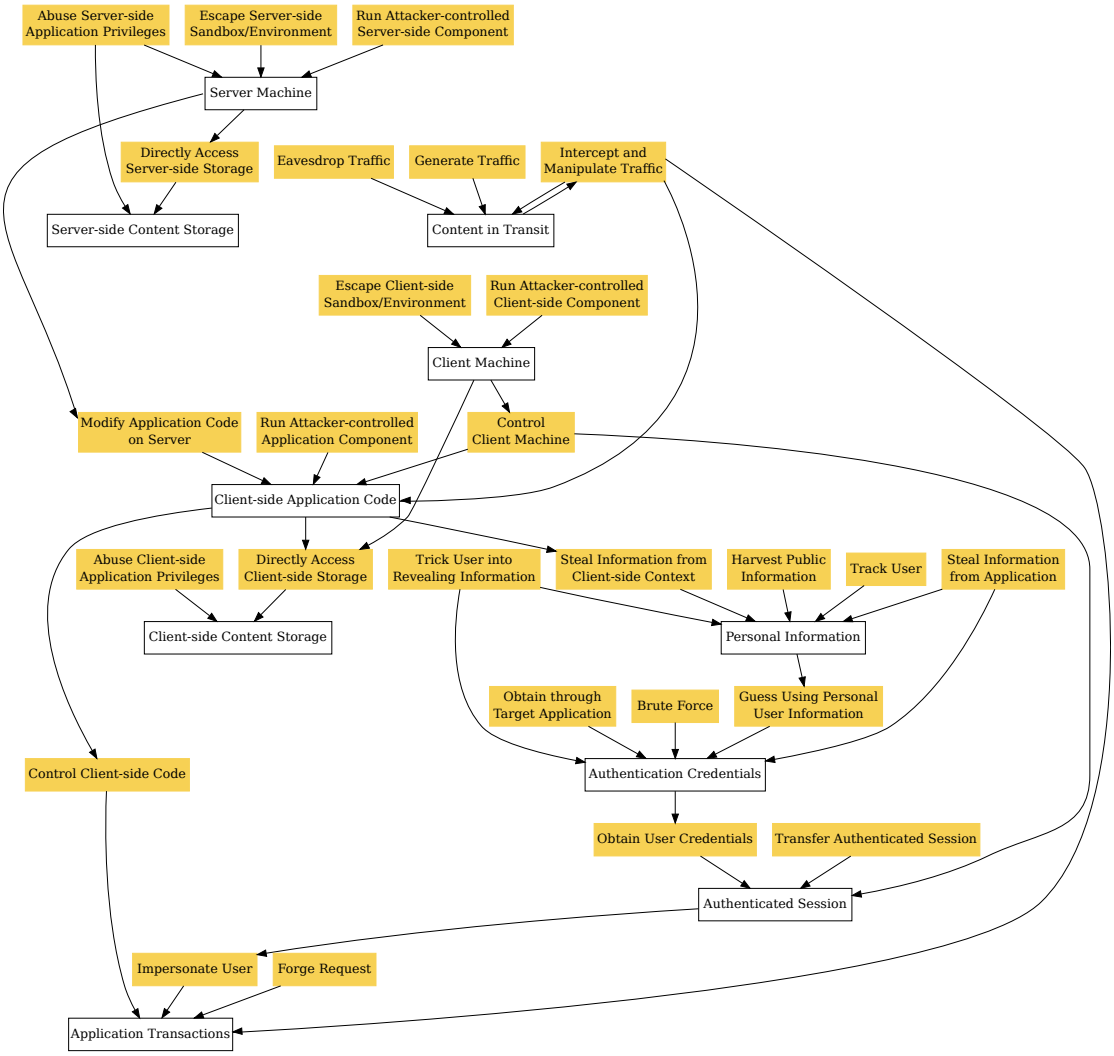


Figure 6.11: An overview tree showing the assets (white boxes) and their associated threats (yellow boxes). Additionally, the tree shows the relations and dependencies between assets, which can result in an escalation of an attack.

Forging requests enables an attacker to compromise *Application Transactions*, potentially modifying or adding transactions that appear legitimate. This asset's attack tree has one threat that is not covered by related assets:

- Forge Request

By **Attacking content in transit**, an adversary might be able to eavesdrop on *content in transit*, or even manipulate and intercept it. The attack tree identifies the following threats:

- Eavesdrop Traffic
- Generate Traffic
- Intercept and Manipulate Traffic

Controlling the client-side application context can be achieved by compromising the *client-side application code*, which in turns allows an escalation towards several other assets,

	Server Machine	Client Machine	Server-side content storage	Client-side content storage	Content in Transit	Client-side application code	Application Transactions	Authenticated session	Authentication credentials	Personal information
Server Machine			⋆			⋆				
Client Machine				⋆		⋆		⋆		
Server-side content Storage										
Client-side Content Storage										
Content in Transit						⋆	⋆			
Client-side Application Code										
Application Transactions										
Authenticated Session							⋆			
Authentication Credentials								⋆		
Personal Information									⋆	

Table 6.1: Asset matrix showing the relation between different assets. Compromising an asset in the first column will allow the escalation of the attack towards the assets marked with an x in that row.

such as the *application transactions* or the *client-side content storage*. The specific threats that can lead to the compromise of this asset are:

- Directly Access Client-side Storage
- Abuse Client-side Application Privileges
- Run Attacker-controlled Application Component

By **attacking the client-side infrastructure**, an attacker can gain control over the *client machine*, allowing further escalation of the attack. Two threats are identified in the attack trees:

- Run Attacker-controlled Client-side Component
- Escape Client-side Sandbox/Environment

Attacking the server-side application covers attacks on both the *server machine* and the *server-side content storage*, which are strongly connected, server-side components. More specifically, the following threats can lead to the compromise of these assets:

- Abuse Server-side Application Privileges
- Run Attacker-controlled Server-side Component
- Escape Server-side Sandbox/Environment
- Directly Access Server-side Storage

A final group of threats is focused at **violating the user's privacy**, where the user's *personal information* is the main asset. Specific threats to the user's personal information are:

- Harvest Public Information

- Steal Information from Application
- Trick User into Revealing Information
- Track User

Chapter 7

Attacker Capabilities

An attacker is only able to successfully compromise a certain asset if all conditions are right. The most basic of these conditions are the attacker capabilities, which he can leverage to execute a threat, eventually leading to the compromise of the asset. Traditionally, an attacker's capabilities are expressed using academic *threat models*, which precisely define what an attacker can and cannot do. Unfortunately, these threat models entail multiple capabilities, are highly tailored to a specific problem statement and solution, and different works have slightly different definitions, making general reasoning within the Web platform difficult.

To be able to define an attacker's power in an unambiguous way, we introduce fine-grained *attacker capabilities*, which can be composed to give a specific threat model, if desired. After introducing each attacker capability, we explicitly discuss the mapping of capabilities to the following, relevant academic threat models [7, 26, 27, 42, 144]:

- Passive Network Attacker
- Active Network Attacker
- Forum Poster
- Web Attacker
- Gadget Attacker
- Related-domain Attacker

7.1 Concrete Attacker Capabilities

An attacker capability is an action that can be performed by an attacker in the web model. Typically, these actions are legitimate operations within a certain context, and are not necessarily considered harmful. By using his capabilities in a certain way, an attacker can however take advantage of certain features of the Web platform, escalating his power and capabilities, until the compromise of one or more assets is achieved, as illustrated in the attack trees in Part III.

For example, the capability to *host content under a registered domain* is harmless in itself, and used by many web developers across the Web. However, by hosting malware under this attacker-controlled domain and tricking the user into installing this malware, an attacker can escalate his capabilities towards controlling a *client machine*, one of the assets defined in the previous chapter.

Essentially, attacker capabilities are the lowest level of capabilities available to an attacker. Naturally, the exact set of capabilities an attacker possesses depends on his position in the Web ecosystem. An attacker that merely hosts a web site under his own domain will not have the

capability to eavesdrop on a user's local network traffic, but an attacker sitting next to the user, using the same wireless network might.

In the remainder of this section, we will cover each of the identified attacker capabilities, carefully expressing their power and indicating their position in the web model, presented in Chapter 5.

7.1.1 Register an Available Domain

Any internet user, including an attacker, is able to register a currently-unregistered domain. The procedure typically requires the payment of an annual service fee to a registrar, who is licensed to hand out sub-domains for the specific parent domain (e.g., registering *example.org.uk* would be authorized by the licensee for the *org.uk* domain). Getting control of an already-registered domain is not possible, unless by court-order or by snatching it away after its previous owner has let it expire.

This capability is situated in the *Registration Authorities*, as modeled in the Web platform, but is not specific to either the client or the server side.

7.1.2 Host Content under a Registered Domain

Hosting content under a registered domain is a basic capability for web developers, and thus also available to any web user with malicious intentions. Setting up hosted content is fairly straightforward, especially using hosting services from a provider, in exchange for a small service fee. Well-organized attackers issue such payments using stolen credit cards or entire identities, making it difficult to track and prevent such transactions.

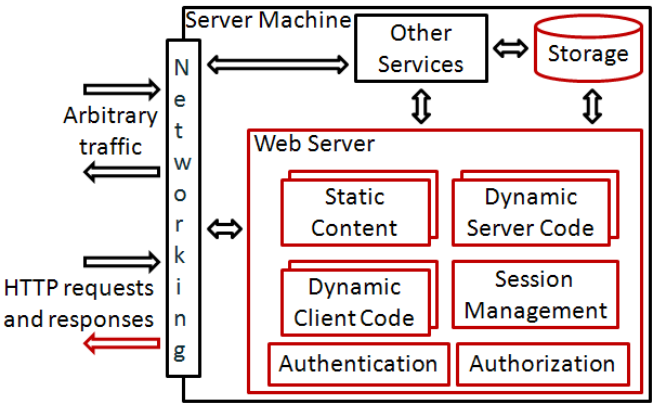


Figure 7.1: By placing his own content under rented hosting space, an attacker has control over the server-side context of a web application, including the dynamic client code, which is transferred to and executed in the user's browser.

Hosting web content gives an attacker full control over a server-side context, including code that will be sent to the client, where it is executed in the browser. Having a victim visit the attacker controlled content is easily achieved, for example by posting links to social networking sites, offering interesting content, etc. Once a victim visits the attacker content, the attacker also gains control over a client-side context within his origin. Figure 7.1 shows the compromised server-side components in an example attack.

7.1.3 Host Content under an Existing Domain

As an alternative to hosting content in his own domain, an attacker might also be able to host content in a domain he does not control. For example, several providers allow users to build and

host applications in subdomains of their main registered domain, such as `myapp.example.com`. Similarly, many providers, such as local ISPs or even Google, offer a small amount of web space to their users, typically located in subfolders of a specific domain, such as *users.someisp.net/user1*.

As with hosting web content under a registered domain (See Figure 7.1), the attacker has full control over a server-side context, and can serve content to be loaded in a victim's browser. However, depending on the deployment scheme, an attacker now not only gains control of his own client-side context, but potentially also of others within the same origin as well. For applications hosted within a related domain, an attacker only gains some access to some interesting features, such as domain cookies or the *document.domain* property. For applications hosted within the same origin, under different paths, an attacker gains full access to other client-side contexts.

7.1.4 Register a Valid SSL Certificate for a Domain Name

Setting up secure connections using HTTPS requires a valid certificate, otherwise the browser will generate disconcerting warnings to the user. An attacker, or anyone for that matter, can apply for a certificate for a given domain name, as long as the identify verification checks succeed. Passing simple ownership validation is straightforward, and is for example used when attackers register a domain name that resembles the target domain, thereby obtaining a valid certificate for the fraudulent domain, hoping to impersonate the target domain to victims. Obtaining certificates for non-attacker controlled domains should be impossible, but depending on the verification process and/or gullibility of the administrators, an attacker might succeed in obtaining a certificate for a domain that is not controlled by the attacker.

As with registering a domain name, this capability is situated with the *Registration Authorities*. Note however that once an attacker obtains a valid certificate for a domain, he is able to impersonate a legitimate server for this domain over a secure connection, corresponding to the capability *intercept and manipulate network traffic* (Section 7.1.10).

7.1.5 Respond to a Legitimate Client Request

Whenever an attacker controls a server, either his own or a compromised server, he has the possibility to respond to legitimate client requests. This is considered a separate capability, because an attacker is not necessarily bound by the HTTP specification or underlying web serving software, and can respond with arbitrarily constructed responses, attempting to exploit vulnerabilities at the client-side.

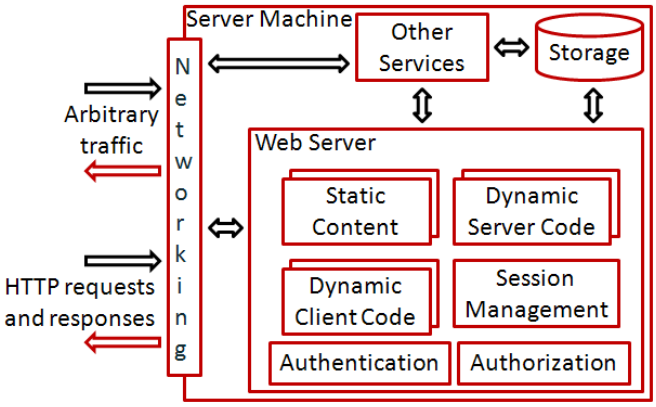


Figure 7.2: An attacker that has taken control of a legitimate application's server can send arbitrary responses, potentially impacting any resource inspecting or processing the response, which include intermediate network components and client-side components.

This capability is situated at the server-side, and has an impact on the responses sent from the server to the client. Figure 7.2 illustrates how an example attack originates in the web model.

7.1.6 Send a Well-formed Request to an Application

A user on the Web sends well-formed requests to applications, and receives responses in return. This is the basic behavior of the Web, and is also available to attackers. Depending on the application, the authentication and authorization infrastructure, an attacker might have access to only public resources, or resources deep within the application. Sending a well-formed request means that the request is sent by a standards-compliant client, such as a browser, and can not deviate from the implemented protocols.

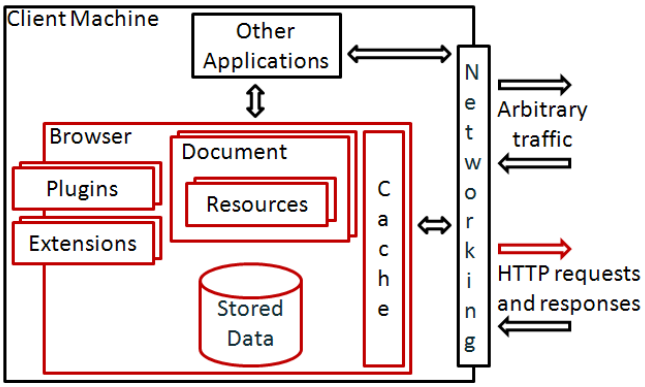


Figure 7.3: An attacker can send carefully crafted but legitimate HTTP requests to a web application from his own machine, potentially compromising the server-side application logic.

This capability is situated at the client-side, and can be used by an attacker from his own client machine, or from a compromised client machine. Figure 7.3 illustrates an example in the web model.

7.1.7 Send an Arbitrary Network Request to a Server

Apart from sending well-formed requests, any user controlling the network stack of his machine can also send arbitrary generated requests, which can be in any format and do not have to be standards-compliant. This allows an attacker to exploit parsing vulnerabilities, or connect to other services than HTTP(S), such as a SQL server. Typically, these requests cannot be generated from within a browser, and require full control over the client machine.

This capability is situated at the client side, and impacts the network traffic sent by the client. Figure 7.4 illustrates an example attack. Note that the requests are also impacted, since an attacker can also produce network traffic that very closely resembles legitimate HTTP requests.

7.1.8 Eavesdrop on Network Traffic

In certain circumstances, an attacker might be able to eavesdrop on network traffic from legitimate users. One common case is an attacker that sits on the same wireless network, who is able to receive all transmitted data. Alternatively, attackers on a wired network may also be able to see a certain fraction of network traffic. By eavesdropping on the network, an attacker can gather valuable information, which can be used to escalate an attack against a user or application.

This capability is located in the middle of the web model, where the network functionality is situated. Figure 7.5 illustrates a straightforward attack using a rogue access point. Note that

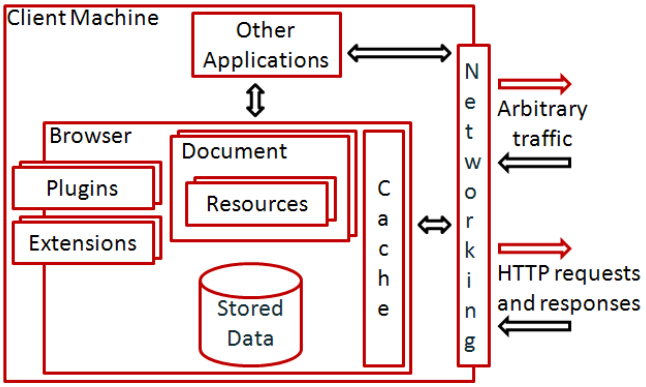


Figure 7.4: An attacker can send carefully crafted network requests to a web server from his own machine, potentially compromising the server-side application logic.

eavesdropping can be done in a completely passive way, preventing detection by the network infrastructure or monitoring software.

7.1.9 Generate Network Traffic

In addition to eavesdropping on traffic, a network attacker can also generate new traffic with spoofed parameters. For example, if an attacker sees a browser making a request for a certain resource, he can generate a response that seems to come from the target server, and send it before the actual response reaches the browser. This allows an attacker to provide malicious content to legitimate client applications, potentially compromising the application or even the client machine.

This capability is also situated in the network component of the web model (Figure 7.5). Generating traffic cannot be done in a stealthy way, and can be detected by monitoring software in the network.

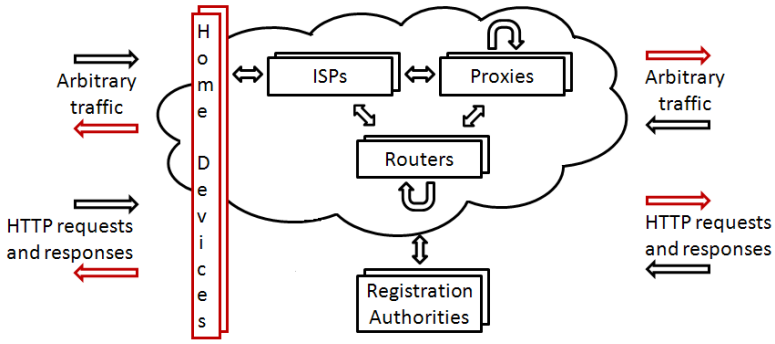


Figure 7.5: An attacker that sets up a rogue wireless access point can lure users into connecting through it, giving him full network capabilities. This includes eavesdropping on traffic, generating arbitrary requests and responses as well as performing a full MitM attack..

7.1.10 Intercept and Manipulate Network Traffic

A third network-based capability is to intercept and manipulate network traffic, conducting a full Man-in-the-Middle (MitM) attack. An attacker sitting in the path between a user and a server

	Forum Poster	Web Attacker	Gadget Attacker	Related-domain Attacker	Passive Network Attacker	Active Network Attacker
Register Available Domain		⋆	⋆	⋆	⋆	⋆
Host Content under Registered Domain		⋆	⋆	⋆	⋆	⋆
Host Content under Existing Domain				⋆		
Register Valid Certificate for Domain Name		⋆	⋆	⋆	⋆	⋆
Respond to Legitimate Client Request		⋆	⋆	⋆	⋆	⋆
Send Well-formed Request to Application	⋆	⋆	⋆	⋆	⋆	⋆
Send Arbitrary Network Request to Server		⋆	⋆	⋆	⋆	⋆
Eavesdrop on Network Traffic					⋆	⋆
Generate Network Traffic						⋆
Intercept and Manipulate Network Traffic						⋆

Table 7.1: An overview of academic threat models, decomposed into fine-grained attacker capabilities.

can intercept and inspect every request and response. Additionally, he can modify them to his wishes, potentially compromising the client-side or server-side context of the web application.

Again, this capability is situated in the network component of the web model (Figure 7.5). Carrying out a MitM attack is far from stealthy and can be detected by monitoring software. Additionally, whenever HTTPS is used, an attacker has to obtain an appropriate certificate for the client and/or server, in order to prevent alarm bells from going off.

7.2 Capabilities in Academic Models

In academic literature, it is common to define a *threat model*, representing the exact capabilities of the attackers that are considered to be in scope for a certain attack or countermeasure. In this section, we analyze these academic threat models in the light of the list of proposed attacker capabilities, showing that each of these models can be expressed with a combination of one or more attacker capabilities. Each academic threat model is discussed in detail, and an overview of all threat models and their mapping to specific capabilities is given in Table 7.1.

7.2.1 Forum Poster

A *forum poster* [26] is the weakest attacker model, representing a user of an existing web application, who does not register domains or host application content. A forum poster typically uses a web application, and potentially posts active content to the application, within the provided features. Additionally, a forum poster remains standards-compliant, and can not create HTTP(S) requests other than those he can trigger from his browser.

The forum poster possesses the following capability:

- Send a well-formed request to application

7.2.2 Web Attacker

The *web attacker* [7, 26, 27, 42] is the most common threat model encountered in papers, and represents a typical attacker who is able to register domains, obtain valid certificates for these domains, host content, use other web applications to post content to, etc. Since none of these capabilities requires a special physical location or any other typical attacker properties, we consider every other attacker model, except for the forum poster, to possess these capabilities as well, following the example of many threat models in academic literature.

The web attacker possesses the following capabilities:

- Register an available domain
- Host content under a registered domain
- Register a valid SSL certificate for a domain name
- Respond to a legitimate client request
- Send a well-formed request to an application
- Send an arbitrary network request to a server

7.2.3 Gadget Attacker

A *gadget attacker* [7, 27] is a special case of a web attacker, where the attacker hosts a component that is willfully integrated into the target application. Popular examples are a JavaScript library, such as JQuery, or a widget, such as Google Maps. The gadget attacker is extremely relevant in the context of code isolation for mashups or complex, composed sites.

A gadget attacker possesses no explicit additional capabilities over a web attacker, but has a level of integration with the target application, as discussed in Section 3.1.4.

7.2.4 Related-domain Attacker

A *related-domain attacker* [42] is a special case of a web attacker, where the attacker is able to host content in a related domain of the target application. Two cases are either a domain that is a sibling or child domain of the target application, or within the same domain but in a different path. This attacker model is relevant because it interferes with some security features in the Web platform, where separation between applications is violated somehow.

A related-domain attacker possesses the following additional capabilities over a web attacker:

- Host content under an existing domain

7.2.5 Passive Network Attacker

A *passive network attacker* [144] is considered to be an attacker that is able to passively eavesdrop on network traffic, but not able to manipulate or spoof traffic. Typically, a passive network attacker is expected to learn all unencrypted information. Additionally, a passive network attacker can also act as a web attacker, for which no specific requirements are posed.

A passive network attacker possesses the following additional capabilities over a web attacker:

- Eavesdrop on network traffic

7.2.6 Active Network Attacker

A *active network attacker* [7, 26, 144] is considered to launch active attacks on a network, for example by controlling a DNS server, spoofing network frames, offering a rogue access point, etc. An active network attacker has the ability to read, control and block the contents of all unencrypted network traffic. An active network attacker is typically not considered to be able to present valid certificates for HTTPS sites that are not under his control.

An active network attacker possesses the following additional capabilities over a web attacker:

- Eavesdrop on network traffic
- Generate network traffic
- Intercept and manipulate network traffic

Intermezzo 6: Attackers beyond the Web

Attackers and their associated threat models are not limited to using the Web, and have a wide variety of technologies to their disposal. Two such prevalent and important attackers are attackers combining the Web with other communication channels, and pervasive attackers, capable of monitoring or compromising systems on a large scale.

The first type of attackers combines traditional web capabilities with the use of other communication channels, such as emails, instant messages, text messages, phone calls, etc. Well-known attacks carried out over these channels are *phishing attacks*, where an attacker masquerades as a legitimate and trustworthy entity in an out-of-band message, tricking the user into visiting a – supposedly trustworthy – site and entering account details or personal information. Phishing techniques have improved in the past few years, with more detail being spent on the phishing messages and he phishing websites. Additionally, phishing attacks are often based on harvested personal information, ranging from a little (targeting banks within the victim's country) to an astonishing amount, with so-called *spear phishing* (masquerading as a company's internal department, accurately addressing a victim in order to compromise the victim's account or a company asset). Even the modus operandi has changed, with attackers now calling their victims to give more credibility to the phishing attack.

A second type of attackers is able to use traditional attack techniques in a pervasive manner. One example is the capability of *eavesdrop on network traffic*, for which the actual power depends on the medium, with a vast difference between a local wireless network or an international network backbone. Similarly, *intercepting and manipulating network traffic* can occur on a small scale in a local network, or on a large scale for an entire nation, potentially even using fraudulently issued certificates. Pervasive attackers also influence security protocols and standards, which are typically designed to withstand a specific threat model, and might crumble under pervasive attacks. For example, attacks on secure TLS connection are based on DNS redirection or man-in-the-middle, but are foiled by a missing or invalid server certificate (if noticed by the user). A pervasive attacker controlling both the DNS traffic or DNS systems, and able to issue legitimate – but fraudulent – certificates easily breaks the guarantees offered by TLS. While the pervasive attacker may have been hypothetical in the past, the recent revelations of state-sponsored surveillance have made the pervasive attacker very real, and omnipresent [244].

7.3 Analyzing Threats to the Web Platform

The attacker capabilities introduced in this chapter are the basic building blocks determining an attacker's power. These capabilities are the starting point for attacks targeted towards the

assets defined in the previous chapter, using one of the threats to compromise an asset.

In Part III, we link assets, threats and capabilities, analyzing known attacks targeting the Web platform, as well as their countermeasures. The threats, that are represented as the leaf nodes in the asset attack trees, are grouped into logical sets, a structure followed by the different chapters of Part III. Each chapter covers one or more attack techniques that can be used to meet this threat, compromising the asset. These attacks are fleshed out using an attack tree, that ends with leaf nodes representing attacker capabilities, clearly indicating the required attacker power to carry out such an attack.

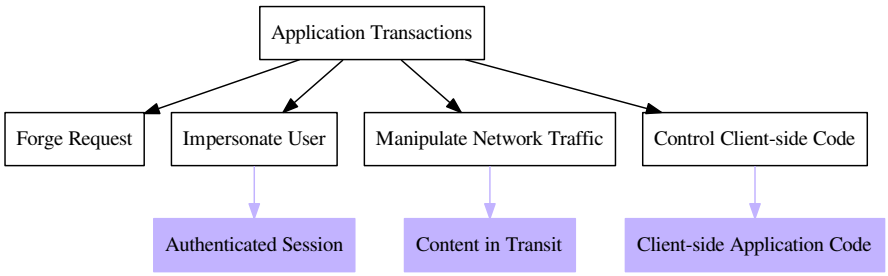


Figure 7.6: The attack tree of the *Application Transaction* asset shows that one of the threats is to *forge requests* to the application.

We illustrate this process by an example, using the threat *forge requests*, which is a way to compromise the *application transactions* asset, as shown in Figure 7.6. Chapter 9 in Part III covers several ways to forge requests, such as *Cross-Site Request Forgery* (CSRF) and *clickjacking*. Each of these attacks is fully covered in their respective sections, which includes a detailed attack tree, as shown in Figure 7.7 and Figure 7.8. These attack trees explain conceptually how the attack can be carried out, and the leaf nodes indicate the required attacker capabilities.

Part III is focused on covering important attacks on the Web platform, based on an extensive literature review. However, the methodology introduced here also supports future research, both on the offensive and defensive side. On the offensive side, the attack trees support an analysis of either new combinations of attacker capabilities, or the addition of fresh capabilities, which may have their impact on certain threats. On the other hand, research towards countermeasures can take advantage of the attack trees to figure out the impact of disabling a certain attacker capability, or determining the effect of ensuring that a certain asset can no longer be easily compromised.

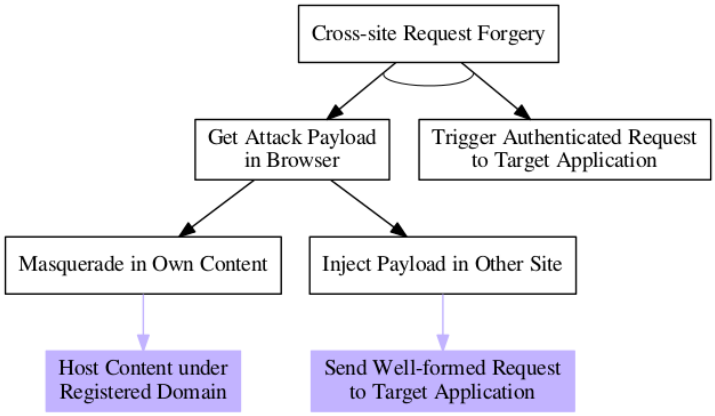


Figure 7.7: The attack tree for a *CSRF* attack, as covered in Part III, shows the required attacker capabilities to carry out the attack, which enables an attacker to *forge requests* to the application.

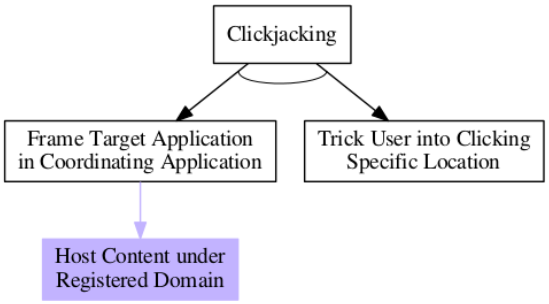


Figure 7.8: The attack tree for a *clickjacking* attack, as covered in Part III, shows the required attacker capabilities to carry out the attack, which enables an attacker to *forge requests* to the application.

Part III

Attacks on the Web Platform

Chapter 8

Impersonating Users

The threat to *impersonate users* envisages the possibility that the attacker will compromise the *application transactions* asset, by performing operations in the victim's name. Impersonating a user implicitly means that the attacker gains control over a user's *authenticated session*, which is also defined as an asset in Chapter 6. Gaining access to an authenticated session can happen in various ways, such as compromising the client machine, transferring an existing authenticated session or establishing a new session with the user's credentials. This chapter focuses on ways to impersonate a user by gaining access to the *authenticated session*, other than by compromising the *client machine*, which is covered in Chapter 12. Attacks that are clearly privacy-related, such as user tracking or identity theft are also not covered here, but are addressed in Chapter 14.

The core problem behind user impersonation is that many applications neither deploy unforgeable authentication mechanisms, nor do they sufficiently protect an authenticated session. This allows an attacker to impersonate a user through an authenticated session, often by simply obtaining a single piece of information, such as a session identifier for an authenticated session, or a user's credentials.

This chapter covers two ways to transfer an existing authenticated session from the victim's browser to the attacker's browser: *session hijacking* and *session fixation*. Additionally, we cover two ways to obtain a new authenticated session, either by *stealing authentication credentials* or by *brute-forcing authentication credentials*.

In addition to stealing or brute-forcing, an attacker can also employ other techniques to obtain a user's credentials, which are not covered in detail in this document. Example techniques are abusing an application's *password reset* functionality, setting up a faux-legitimate application, hoping that users will register with credentials they also use for other applications, or setting up a phishing web site, tricking users into revealing sensitive information, such as authentication credentials.

8.1 Session Hijacking

A successful session hijacking attack gives an attacker full control over an authenticated user's session, allowing him to perform every action that is available to the user. Session hijacking attacks are highly dangerous, and security-sensitive systems attempt to counter them by using additional out-of-band authentication and authorization approaches, such as USB tokens or smartcards[1]

[1] *Smartcard* here refers, typically, to an inductively powered contactless card or equivalent powered through a suitable interface device, and containing a silicon chip that acts as a secure repository for a suitable secret and provides computation to apply built-in cryptographic algorithms to some suitable combination of input and secret to authenticate the possessor of the card. This may be something akin to bank "chip and pin" debit/credit cards or mobile telephone SIM devices. They generally use the same basic chips.

Problem Description

The goal of a session hijacking attack is to transfer the user's authenticated session to a different machine or browser, enabling the attacker to impersonate the user. In order to succeed, it is essential that the user has an established session with the target application, but not that the user has already authenticated himself. Since the session-state, such as the authentication status, is typically stored at the server-side, it will be accessible for the attacker as well, through the stolen (previously unauthorized) session.

Technically, when a session is created, the server assigns it a random and unique session identifier. This identifier is communicated to the client, and needs to be presented with every request, allowing the server to look up the correct session state, and process the request accordingly. In a session hijacking attack, an attacker succeeds in stealing the session identifier, which he can subsequently use to send requests to the server (Illustrated in Figure 8.1).

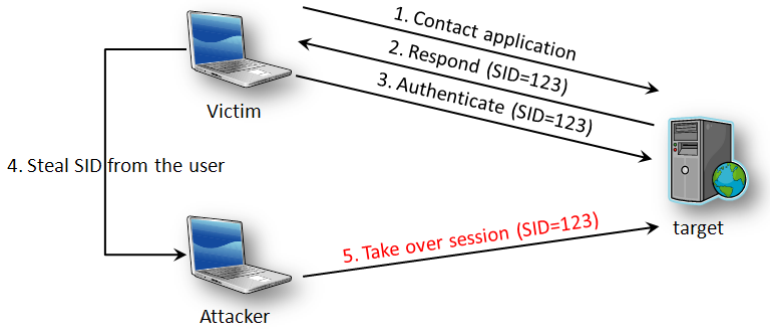


Figure 8.1: In a *session hijacking* attack, an attacker steals the session identifier of the user (step 4), resulting in a complete compromise of the user's session.

The most common session management mechanism on the Web is cookie-based, where the session identifier is stored in a cookie within the browser. Depending on the security parameters of the cookie, an attacker has several ways to obtain the session identifier, as illustrated in the attack tree in Figure 8.2. One example is through the `document.cookie` property, exposed to the JavaScript environment. Another way is by directly accessing the cookie store from a compromised browser, for example by installing a malicious extension. A third alternative is by eavesdropping on the network traffic, and snatching the session identifier from the response, or any subsequent request, as illustrated by point-and-click tools such as Firesheep [46]. Finally, a weak or predictable session identifier can be guessed or obtained through a brute force attack.

An alternative session management mechanism is based on URI parameters, including a session identifier as a parameter in the URI in every request to the application. This mechanism is often used as a fallback mechanism for browsers that do not support cookies, or refuse to store them. Technically, little changes for a session hijacking attack, other than the means to obtain the session identifier. An attacker can still access it from JavaScript or eavesdrop on the network to extract it. Additionally, an attacker can attempt to trigger a request to an attacker-controlled resource, hoping that a `Referer` header will be included, since it contains the full URI, including the parameter with the session identifier.

In essence, a session hijacking attack is possible because the session identifier, which acts as a bearer token for an authenticated session, is easily obtained and transferable between browsers. Making the session identifier accessible through JavaScript or by eavesdropping on the network is a sub-optimal decision, which enables a highly dangerous and harmful attack.

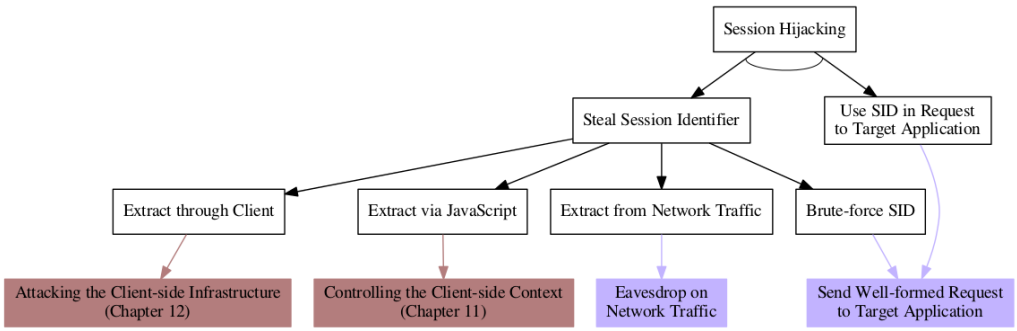


Figure 8.2: In a session hijacking attack, the attacker first steals the session identifier from the victim's browser, and subsequently attaches it to his own requests.

Mitigation Techniques

A traditional mitigation technique for session hijacking is *IP address binding*, where the server binds a user's session to a specific IP address. Subsequent requests within this session need to come from the same IP address, and any requests coming from another IP address will be discarded. While this mitigation technique works well in scenarios where every machine has a unique, unchanging public IP address, it is ineffective when the same public IP address is shared among multiple machines, or when the public IP address changes during a session. Precisely these two cases have become ubiquitous in modern network infrastructure, with NATed home and company networks, publicly accessible, shared wireless networks and mobile networks.

Another approach focuses on preventing the theft of the session identifier, typically stored and transmitted in a cookie. The *HttpOnly* and *Secure* cookie attributes can be used to respectively prevent a cookie from being accessible through JavaScript, and prevent a cookie sent over HTTPS from being used (and thus leaked) on a non-HTTPS connection. Correctly applying both attributes to cookies holding a session identifier effectively thwarts script-based session hijacking attacks, as well as session hijacking attacks through eavesdropping on network traffic.

State of Practice Unfortunately, many sites still use unprotected cookies to store session identifiers, leaving users vulnerable to session hijacking attacks. On the bright side, the adoption of the *HttpOnly* and *Secure* attributes is gaining ground, starting to be turned on by default [11].

Another practice that is being deployed by major sites is to operate split session management between HTTP- and HTTPS-accessible parts of the site. For example, a web shop can run its catalog inspection and shopping cart filling operations over HTTP, and use HTTPS for sensitive operations, such as logging in, checking out the cart, payments or account modifications. Technically, they use two different session cookies, one for HTTP usage and one for HTTPS usage, where the latter is declared *HttpOnly* and *Secure*. While this leaves the user vulnerable to a session hijacking attack on the HTTP part, it effectively protects the HTTPS part, where sensitive operations are conducted.

Research and Standardization Activities

One long-lived line of research focuses on providing protection against session hijacking attacks from within a web application, without specific infrastructure support at the client side. *SessionSafe* [148] focuses on script-based session hijacking attacks, and proposes to limit the session identifier accessibility, preventing theft by scripts. Additionally, SessionSafe also protects

against two other script-based attacks, browser hijacking and background cross-site scripting (XSS) propagation. *SessionLock* [5] introduces a secret value in the client-side context, which is in turn used to add an integrity check to outgoing requests. This effectively protects against eavesdropping attacks, since the secret value is never transmitted in the clear, but does not protect against script-based session hijacking, since the secret value is stored in the JavaScript context. Other techniques follow a similar approach, but base their security measures on the user's password, a shared secret between the user and the web application. *BetterAuth* [150] revisits the entire authentication process, offering secure authentication and a secure subsequent session. *Hardened Stateless Session Cookies* [190] uses unique cookie values, calculated using hashing functions based on the user's password, effectively preventing the generation of new requests within an authenticated session. A final mitigation technique, *GlassTube* [124], ensures integrity on the data transfer between client and server, and can be deployed both within an application or as a modification of the client-side environment, for example as a browser plugin.

SessionShield [194] is a client-side proxy that mitigates script-based session hijacking attacks by ensuring all session cookies are marked *HttpOnly* before they reach the browser. Determining which cookies are session cookies at the client side, in an application-agnostic way, is achieved by applying sensible heuristics, including an entropy test. *Serene* [73] implements SessionShield as a browser extension for Firefox, and extends it to support parameter-based session management techniques as well.

Best Practices

The best practice for preventing session hijacking attacks is to use strong, random session identifiers [233], and deploy the application over HTTPS (See best practices in Chapter 10), using the *HttpOnly* and *Secure* attribute for all cookies not needed by JavaScript, especially the cookies containing a session identifier.

Additionally, web development frameworks and application servers that offer easy-to-use session management mechanisms should deploy these protections by default, and discourage their users from turning them off.

8.2 Session Fixation

A session fixation differs in approach from a session hijacking attack, but serves the same purpose, gaining access to an authenticated user's session. As with a session hijacking attack, a successful session fixation attacker can perform all actions available to a legitimate user, in the victim's name.

Problem Description

The goal of a session fixation attack is to ensure that the victim uses a session controlled by the attacker, before the authentication procedure is completed. Once the victim authenticates within the fixated session, the attacker can take over the session, now having access to an authenticated session between the victim and the application, a similar result to that obtained by a session hijacking attack.

Technically, an attacker obtains a valid session identifier for the application, either by visiting the target application himself, or by crafting a session identifier. In the next step, the attacker has to fixate the session identifier in the victim's browser, which depends on the session management mechanism used by the application. Once the session is fixated and the user visits the application, the user will be working within the attacker's session. This means that the authentication state at the server side will be stored within this session as well, allowing the attacker to take over the session later on (Illustrated in Figure 8.3).

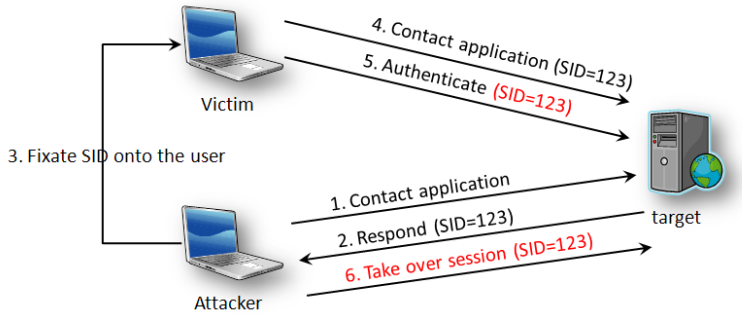


Figure 8.3: In a *session fixation* attack, an attacker fixates his own session identifier into the browser of the user (step 4), causing the user to authenticate in the attacker's session.

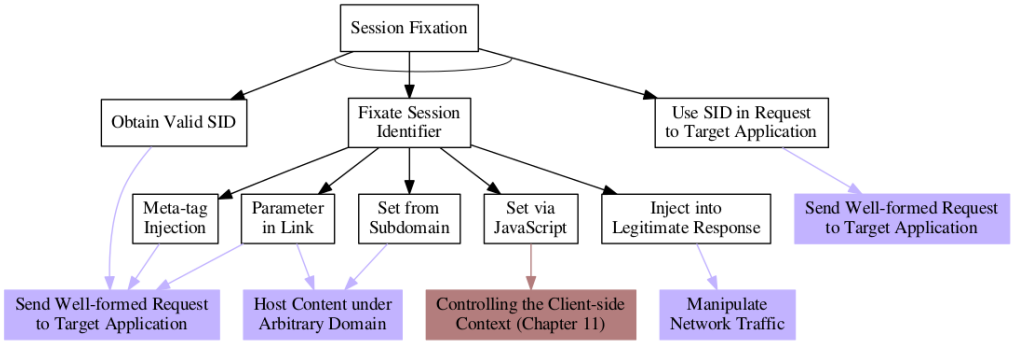


Figure 8.4: In a session fixation attack, the attacker forces the victim's browser to use his session identifier, which can be achieved in numerous ways.

Fixating the session identifier is the crucial part of a session fixation attack. As shown in the attack tree in Figure 8.4, the fixation in cookie-based session management systems can happen in several different ways [73], for example by setting the `document.cookie` property from JavaScript, by using an injection attack to insert `meta` elements that mimic header operations into the page's content, or by manipulating network traffic. Fixating a session identifier in parameter-based session management systems is straightforward. All it takes is tricking the user into visiting a URI which contains the fixated session identifier as a parameter in the query string.

In essence, session fixation attacks are possible because the session identifier acts as a bearer token for a session between a user and an application, combined with the fact that sessions are easily transferable between browsers. Session fixation and session hijacking attacks both exist for the same reasons, but use a different attack vector to obtain an authenticated session.

Mitigation Techniques

Due to the multiple attack vectors that can lead to a session fixation attack, plugging them all is difficult. Nonetheless, protecting session cookies with the *Secure* and *HttpOnly* attributes makes the attack more difficult, since it prevents an attacker from easily overwriting an already existing session cookie. These protections can however be bypassed, for example by overflowing the cookie jar with meaningless cookies, causing the browser to purge the oldest ones (i.e., the

session cookie), allowing the attacker to fixate a new session identifier.

An effective mitigation technique for fixation attacks consists of renewing the session identifier after a privilege change within the application. For example, by issuing a new session identifier after user authentication, an application ensures that the authentication information is not associated with the fixated session identifier, preventing the attacker from taking over the authenticated session. Renewing the session identifier is the server's responsibility, and is often supported by the web programming language or web framework. No explicit client support is required, since the server typically just overrides the already-existing session cookie using a *Set-Cookie* header.

Research and Standardization Activities

Session fixation has long remained unnoticed in the research community, but recent work proposes mitigation techniques for session fixation at the client and server side. Server-side protection against session fixation attacks can be achieved by renewing the session identifier after a privilege change. Integrating such protection into new applications is straightforward, but reliably protecting legacy applications without changing the code is challenging. Depending on the available frameworks at the server-side, renewing the session identifier can be integrated in the framework's session management mechanism, or be offered as a server-side reverse proxy solution [149].

Unfortunately, updating all web applications to prevent session fixation attacks is a long and tedious task, while in the meantime the users remain vulnerable to these attacks. A client-side protection mechanism against session fixation attacks, called *Serene*, offers protection to a user, without requiring a change or modification at the server-side [73]. *Serene* is a browser add-on that detects cookie and parameter-based session identifiers in requests and responses, and offers additional protection for these identifiers. *Serene* can prevent fixation attacks initiated by script or meta-tag injection attacks, as well as parameter-based session fixation attacks.

A variant of a session fixation attack can be carried out by a *related domain attacker* [42], who controls an application hosted under the same registered domain as the target application (e.g., *example.com*). By setting a cookie that applies to all sibling domains, an attacker can easily fixate a session identifier. This kind of attack can be addressed by *Origin Cookies* [42], which is a proposal to allow cookies to be limited to one domain, preventing manipulation from sibling sites.

Best Practices

The best practice for protecting against session fixation attacks is to renew the session identifier on every privilege change within the application. This effectively ensures that the new privilege level is never accessible using the pre-change session identifier, thus preventing session fixation attacks. Popular web frameworks support the renewal of the session identifier, however, this API call needs to be explicitly invoked [232].

8.3 Brute-forcing Authentication

Brute-forcing the authentication system of a web application is an easy, low-complexity way of obtaining user credentials, which in turn can be used to impersonate the user to the web application. Obtaining a victim's credentials is worse than a session hijacking or session fixation attack, since it is not limited to a single authenticated session. The credentials can be used at any future time, and multiple times, at least until the victim becomes aware of the problem and changes the credentials. In a worst case scenario, the same credentials are even valid for multiple web applications. In some circumstances the attacker may gain authority to change the

credentials and lock out the victim: Depending on the application concerned this may or may not be worthwhile for the attacker.

Problem Description

The goal of a successful brute-forcing attack on the authentication mechanism is to gain access as an authenticated user, without knowing the credentials associated with this account in advance (See Figure 8.5). An attacker can attempt to brute force the password belonging to a known user account, or can try to brute force both username and password. Once an attacker has brute-forced the authentication credentials, he is likely to have unrestricted access to the compromised account.

Carrying out a brute force attack is straightforward. An attacker simply enters a combination of username and password into the authentication form, and submits them to the application. The application either accepts the credentials, or refuses them, after which the attacker tries the next combination. This process can be highly automated and distributed, making the attack very effective against weak credentials.

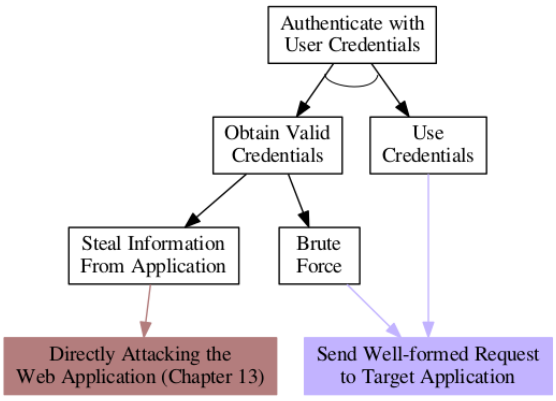


Figure 8.5: By obtaining valid user credentials, an attacker can easily establish an authenticated session in the user's name. This tree covers two ways to obtain valid credentials: Brute-forcing the credentials and stealing the credentials from the application.

A major enabling factor of brute force attacks are weak passwords, which are too short, based on the username, existing words, etc. By using dictionaries of existing words and frequently occurring passwords, an attacker can focus the brute force attack, increasing the level of success.

Mitigation Techniques

Traditionally, brute force attacks are mitigated by enforcing constraints on the number of authentication attempts for an account. A common constraint is to lock the account, effectively refusing further authentication attempts after a limit has been reached. This technique does cause a significant burden when a legitimate user reaches this limit, and can certainly backfire when an attacker willfully generates numerous authentication attempts for all known accounts, effectively causing an application-wide user lockout.

Alternatively to completely refusing any authentication attempts, an application can integrate a delay in the authentication process, which gradually increases as the number of unsuccessful authentication attempts increases. This causes little discomfort for a legitimate user,

who has to wait a bit longer before the authentication is complete, but causes enough delay to prevent automated brute-forcing, with dozens of attempts per minute.

Additional authentication steps can also be deployed, to discourage automating the authentication process. One example is the use of CAPTCHAs®, a small puzzle that needs to be solved before the form can be submitted. Since a CAPTCHA is supposedly easy for humans to solve, but hard for computers, it can be used to ensure that only humans can submit the form. It also has the effect of reducing the potential repetition rate of authentication attempts.

A recent evolution towards securing the authentication process is the use of multi-factor authentication. In a multi-factor authentication process, the application no longer depends on a single piece of knowledge, such as a set of credentials, but requires additional factors, such as a token sent to a user's phone by text message, a token generated by a dedicated device [88], a smart card, biometric information, etc. Multi-factor authentication is an effective mitigation technique for brute force attacks, since one of these authentication factors is typically an out-of-band device, beyond the control of an attacker. One caveat with a phone as a second factor in the authentication process arises with phone theft, where a smart phone typically provides both the browser, with potentially stored credentials, and the out-of-band device.

State of Practice In practice, the use of credentials remains a major factor in web application security breaches. Apart from insecure server-side storage (See Section 8.4), users exhibit little responsibility when dealing with credentials, choosing insecure passwords, sharing or insecurely storing credentials, etc. Numerous reports of hacked and stolen accounts illustrate these issues [156, 161, 162, 208].

Another trend on the rise is the use of multi-factor authentication, often in combination with a (smart)phone. Google offers such a system, where a user is required to provide a token, generated on a smartphone, whenever signing in from an unfamiliar or unknown computer. Google not only offers this service for its own applications, but also enables other, unrelated web applications, to use Google as a third-party authentication service.

Research and Standardization Activities

Credentials depending on a password have been around since the introduction of shared computers, and have been the main way of authentication in web applications since the advent of the World Wide Web. The use of passwords has been well studied, with studies suggesting that password strength makes little difference when combined with lockout strategies [102], at the same time indicating that a bulk brute force attack on all accounts may be dangerous when combined with short passwords. Other studies focus on user behavior, investigating the usability of system-assigned passwords [228] or the effect of password strength meters [250], concluding that system-assigned passwords are best kept pronounceable, and strength meters are best kept stringent.

Other lines of research investigate alternative authentication mechanisms. For example, the previously discussed CAPTCHA was originally proposed as a research concept [256], and has been broken by deploying legions of CAPTCHA-breaking people [189]. The possibilities and benefits of multi-factor authentication have also been researched, resulting in the proposal of practical systems [4] or alternative authentication methods, for example with graphical passwords [219], using an external device [61] or with biometrics [37].

Best Practices

The leading best practice is to enforce sane password policies, which allow users to choose a password to their liking, without enforcing complex and hard to grasp composition rules that require the inclusion of uppercase and lowercase characters, digits, punctuation signs, etc. Good practice is to enforce a minimum password length, combined with a password strength meter plus a clear explanation about strong passwords and their advantages [250].

Second, the implementation of the authentication process should take care not to reveal any useful information to an attacker. For example, the result of an authentication attempt with a correct username and incorrect password, and an attempt with an incorrect username and incorrect password, should be identical. Failure to do so may result in a brute force attack on the list of valid user accounts, effectively leaking important internal system information. Additionally, the application should keep track of the failed authentication attempts for an account, and delay the authentication process accordingly, significantly slowing down any automated authentication processes.

Finally, serious consideration should be given to deploying multi-factor authentication using an out-of-band authentication mechanism, effectively preventing brute force attacks on a set of credentials. Consideration could also be given to requiring an extra level of authentication for certain sensitive operations, thereby altering the application from one that uses two security levels to three or even more. Certain banking applications have adopted this enhancement making operations that are known to be the targets of attackers (e.g., establishing a new payee account, which might be owned by a fraudster, and making a first payment to that account require the use of out-of-band authentication, whereas inspection of accounts, internal transfers and payments to previously created payees only require conventional credentials.) Viable solutions depend on the available deployment infrastructure, and include phone tokens, either by text message or via an app on a smartphone, or dedicated hardware tokens. Alternatively, the authentication process can be outsourced to an authentication provider, who typically offers multi-factor authentication as a standard option.

8.4 Stealing Authentication Credentials

Stealing authentication credentials is a more subtle way of obtaining a user's credentials for a web application. By stealing the authentication credentials of the users of a web application, an attacker can impersonate any user to the web application, and due to sloppy password re-use, the attacker can probably impersonate a significant of those users to other web applications as well.

Problem Description

The goal of stealing authentication credentials is to obtain valid credentials, allowing an attacker to establish an authenticated session with the target application, in the victim's name . An attacker can attempt to access the target application's database, which typically contains the user credentials, potentially giving the attacker access to every user account in the system (See Figure 8.5). How an attacker can gain unauthorized access to the *server-side content storage* is covered in Section 13.1. The remainder of this section however focuses on mitigating the theft of credentials when the server-side content storage is compromised.

Mitigation Techniques

A common alternative to storing passwords in plain text in the database, is to store a one-way hash value of the password. This means that the actual password is never stored, and upon authentication, the one-way hash value of the user-submitted password is compared to the one-wash hash value stored at the server side. While this effectively hides the password from prying eyes, these one-way hash values can easily be compromised using rainbow tables, which are large tables containing pre-computed one-way hash values for all passwords with a certain length.

An improvement over storing the one-way hash value is the use of salts. For each password that is stored, a random string is generated, which is then added to the password (i.e., *salting* the password), after which a one-way hash value is computed. The salt is stored in plain text along with the password. This approach defeats the use of pre-computed tables, since each

password has a different salt, and the salt is used to make the password long enough to make pre-computing unfeasible.

State of Practice In practice, many systems store passwords in plain text, and will offer users the possibility of being reminded of their original password when requested, clearly a suboptimal approach to password management. Similarly, many systems store their passwords as a one-way hash value, without salting applied. Password recovery is typically email-based, sometimes combined with personal questions [41].

Unfortunately, storage of sensitive information, including credentials, often happens in an insecure fashion, causing problems in the event of data breaches, which occur all too frequently [230]. These practices have inspired the introduction of a *security breach notification law*, requiring organizations to disclose a data breach to customers. These laws have been enacted by most of the U.S. states [32], and have been implemented in the European Union *Directive on Privacy and Electronic Communications* [57].

Research and Standardization Activities

Research in this area focuses on different facets of stealing authentication credentials, such as preventing unauthorized access to server-side content storage facilities (See Section 13.1), deploying client-side tools that enable the use of unique passwords per application, or investigating currently deployed password-recovery mechanisms [41].

Currently, several client-side tools are available to store a user's passwords [167, 214], no longer requiring the user to the remember all accounts and associated passwords. Such tools enable the use of unique, application-specific passwords, limiting the harmful effect of credential theft at the server side. Implementing a safe browser extension for managing passwords is a non-trivial task, as illustrated by research [214] and disclosed vulnerabilities [155].

Best Practices

A first step in preventing theft of authentication credentials is to prevent unauthorized access to server-side storage mechanisms (See Section 13.1).

Second, it is highly recommended that passwords are stored securely, generating a cryptographically random salt for each password, with a significant length. It is recommended to calculate the one-way hash value of the value *salt concatenated with password*, which should be stored in the form *salt and one-way-hash value* [239].

Third, password recovery should be handled carefully, using an out-of-band channel such as email or text message to send a randomly generated token. A successful password reset should be only allowed once, when the user presents the generated token in a limited time frame [98].

Finally, the multi-factor authentication systems discussed in the previous section typically use an out-of-band device, which increases the resistance against attacks that attempt to steal credentials through the user's machine, for example through key-stroke monitors or compromised applications. Alternate solutions to target this kind of credential theft only request part of the user's credentials, preventing the theft of the full set of credentials. One example is to request a different subset of the characters in a password, using CAPTCHA-like techniques to conceal which characters are being requested.

Chapter 9

Forging Requests

The threat to *forge requests* is one way to compromise the *application transactions* asset, where an attacker manipulates the legitimate transactions within an application. In this particular case, an attacker forges a request from the victim's browser to the target application, effectively executing an operation within the user's authenticated session with the application, without the user intending to do so.

The core problem behind forging requests is the fact that a target application often can not distinguish between legitimate requests, made by the user, and forged requests, made without the user's consent. Due to the way the Web platform works, it is nearly impossible to determine whether a request is legitimate, without taking some additional measures, as will be explained as we discuss the different attacks that can result in forged requests.

In this chapter, we cover several known ways to forge requests to an application, of which the most common and best-known attack is *Cross-Site Request Forgery* (CSRF), where the attacker tricks the user's browser into sending a request to the target application. A special variant of a CSRF attack is *login CSRF*, where an attacker tricks the user's browser into authentication to the target application with the attacker's credentials, potentially stealing sensitive information from the user. A third attack is *clickjacking*, where the attacker tricks the user into clicking on an element, triggering a forged action in the target application.

One less common attack, which is not covered in this document, involves abusing the redirect mechanism in browsers, tricking the browser into forwarding a request to the target application [7]. One problematic vulnerability in this respect is *open redirects* in server-side web application code, which are capabilities that allow an attacker to determine where a request should be redirected to without the code checking that the redirection target is relevant and legitimate, essentially a form of injection attack on the web application code.

9.1 Cross-Site Request Forgery

A Cross-Site Request Forgery (CSRF) attack enables an attacker to forge requests to the target application from a legitimate user's browser. Successful CSRF attacks can trigger many actions in vulnerable applications, such as modifying account settings, or stealing money through an online banking system [266].

Problem Description

The goal of a successful CSRF attack is to forge a request from a victim's browser to the target application, triggering state-changing effects in the target application. For a CSRF attack to succeed, it is essential that the user is already authenticated to the target application, since the user will not see the forged request, nor authenticate himself specifically for it to succeed.

The forged request has the same structure as a legitimate application request, and is therefore indistinguishable from a legitimate request.

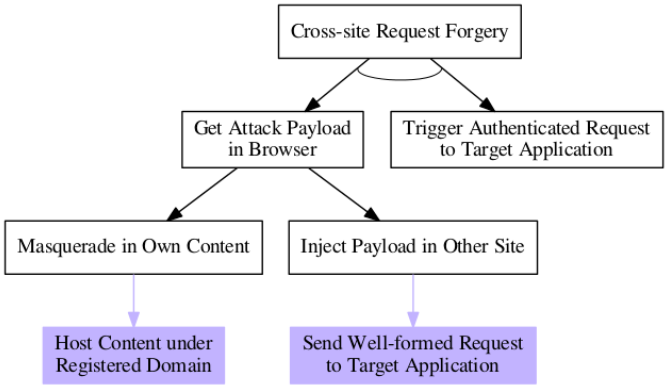


Figure 9.1: A CSRF attack tricks the victim's browser into forging a request to the target application.

The attack tree in Figure 9.1 shows how a CSRF attack is conducted. The first step is to trick the victim's browser into making a request to the target application, which is a straightforward task in any browser. Requests to other sites happen frequently, for instance when loading an external resource such as an image, a stylesheet or a document to load in a frame, but also when submitting form data to a cross-origin URI, as shown in Listing 9.1. Getting the payload to make the cross-site request into the victim's browser is also not very complicated. An attacker can simply host an innocent looking page with useful information, with the CSRF attack hidden in the background, or he can include the payload in an unrelated, legitimate site. An example of the latter is for instance a web forum, allowing users to post images or other content.

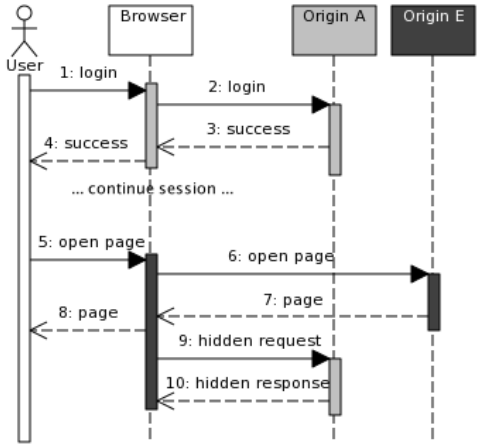


Figure 9.2: In the CSRF attack depicted here, the attacker triggers a request from origin E to origin A (step 9), to which the browser attaches the cookies from the existing session. If origin A does not have CSRF protection, this request will be executed as if it was generated by the user.

A CSRF attack can only be successful if the forged request happens within a previously

authenticated session between the victim's browser and the target application. Unfortunately, the design of current session management mechanisms in the browser attaches session information to any outgoing request, fostering the prevalence of CSRF attacks. For example, the browser attaches the relevant cookies for the domain, scheme and path to each outgoing request, both for requests internal to the application and cross-site or cross-application requests (Illustrated in Figure 9.2). Additionally, many applications prefer long-living sessions, sticking around as long as the browser remains open, regardless of whether an application is currently active in a browser tab. This essentially means that if a user had an authenticated session with the target application in the lifetime of the browser, it is likely that forged requests within an authenticated session can be made.

```
1  document.getElementById("somediv").innerHTML += "<iframe id='attackframe'
2            style='height: 0px; width: 0px;'></iframe>";
3  var f = document.getElementById("attackframe");

4  var code =   "<form id='attackform' action='http://admin.example.com/
5            createAccount.php' method='POST'>";
6  code +=      "<input type='hidden' name='username' value='attacker'>";
7  code +=      "<input type='hidden' name='password' value='12345678'>";
8  code +=      "<input type='hidden' name='action' value='create'>";
9  code +=    "</form>";

10 f.contentDocument.body.innerHTML = code;
11 f.contentDocument.getElementById("attackform").submit();
```

Listing 9.1: A CSRF attack carried out by JavaScript code that creates a hidden `iframe` containing a `form`, which is then automatically submitted to the target application.

In essence, the problem of a CSRF attack is the lack of intent, leaving the server in the dark as to whether a request was made intentionally by legitimate application code, or was forged by an attacker. The fact that browsers handle same-origin and cross-origin requests identically, and web applications now heavily depend on this behavior, enables CSRF attacks and hampers effective countermeasures.

Mitigation Techniques

During the early years of occurrence of CSRF, several simple mitigation techniques have been proposed, but proven ineffective at protecting against CSRF attacks. One suggestion is to only carry out state-changing operations using POST requests, as actually mandated by the HTTP specification [99], assuming that forging POST requests is not feasible. Unfortunately, this is not the case, as shown by the code example in Listing 9.1, rendering this advice useless in protecting against CSRF.

A second mitigation technique enforces referrer checking at the server side. State-changing requests should only be accepted if the value of the `Referer` header contains a trusted site, and rejected otherwise. Referrer checking would effectively mitigate CSRF attacks, but it is not reliably added to requests, making it unsatisfactory as a basis for countermeasures. The `Referer` header is often missing due to privacy concerns, since it tells the target application which resource at which URI triggered the request. Browsers do not add the header when an HTTPS resource, which is considered sensitive, refers to an HTTP resource. Additionally, browser settings, privacy proxies or extensions [1, 2] and referrer-anonymizing services [195] enable the stripping of automatically added `Referer` headers. This mitigation technique did inspire additional research, leading to the `Origin` header, covered below.

Token-based approaches are an effective countermeasure against CSRF attacks [45]. A token-based approach adds a unique token to the code triggering state-changing operations, for example using a hidden form field. When the browser submits the request leading to the action, the token is included automatically, and verified by the server. Token-based approaches hamper the first step in the attack tree, since the attacker cannot include a valid token in his payload, causing the request to be rejected. Key to the success of this mitigation technique is keeping the token for an action out of the attacker's reach. This requires the tokens to be unique, or at least bound to a specific user. Additionally, the tokens embedded in the page need to be protected by the same-origin policy, preventing theft by an attacker-controlled context, loaded in the same browser.

Another effective mitigation of CSRF attacks involves explicit user approval of state-changing operations. By requiring additional, unforgeable user interaction, the attacker is unable to complete the CSRF attack in the background, affecting the second step in the attack tree. Examples are explicit re-authentication for sensitive operations, or the use of an out-of-band device to generate security tokens, as employed by many European online banking systems. The risk associated with this mitigation technique is a shift in attack from CSRF to clickjacking, which is covered in Section 9.3.

State of Practice CSRF is prevalent in modern web sites, and is ranked in both the OWASP Top 10 project [261], listing the 10 most critical web application security risks, and the CWE/SANS Top 25 Most Dangerous Programming Errors [181]. Both small and large scale projects are affected, with for example CSRF vulnerabilities in online banking systems [266], Gmail [130] and eBay [160].

Current practices for mitigating CSRF attacks are focused on token-based approaches, either custom-built for the application, or deployed as part of a web framework. Popular frameworks, such as Ruby on Rails, CodeIgniter and several others. Alternatively, server-side libraries or APIs offer CSRF protection as well, such as the community-supported OWASP ESAPI API and CSRFGuard. Sites being built using a content management system (CMS) – instead of being built from scratch – can benefit from built-in CSRF support as well. For example Drupal, Django and WordPress offer token-based CSRF protection, with Drupal even extending its support to optional customized modules.

Research and Standardization Activities

As an improvement on the `Referer` header, the `Origin` header provides the server with information about the origin of a request, without the strong privacy-invasive nature of the `Referer` header [26]. Unfortunately, the specification [23] only states that the `Origin` header *may* be added, but does not require user agents to do so, potentially causing the same problems as with the `Referer` header. The `Origin` header is however mandatory when using Cross-Origin Resource Sharing (CORS) [253], an API that enables the sharing of resources across origins.

Further research on token-based approaches, which often struggle with Web 2.0 applications, has yielded several improvements over traditional tokens, enabling complex client-side scripting and cross-origin requests between cooperating sites. jCSRF [199], a server-side proxy solution, transparently adds security tokens to client-side resources and verifies the validity of incoming requests. Alternatively, double-submit techniques require the browser to submit a token out of reach for the attacker, but not included automatically by the browser [172].

One of the observed problems in CSRF attacks is the cross-origin accessibility of web application resources, allowing the attacker to target any resource by making a request from a different origin. Several techniques propose to mitigate CSRF by fixing the set of entry points to known safe resources, thus eliminating a CSRF attack on a sensitive resource. These entry points can be enforced purely at the server side [53], or in combination with a browser-based mechanism [62].

CSRF attacks are prevalent in many web applications, but often require specific developer attention to remedy the security flaws. Unfortunately, many applications are no longer updated, or developers do not know or care about CSRF vulnerabilities, leaving users vulnerable in the end. As a response, several client-side solutions attempt to solve CSRF at the client side, for all sites at once. Examples are RequestRodeo [152], the first client-side mitigation technique in the form of a proxy, followed by browser extensions CsFire [69, 70], RequestPolicy [220], DeRef [108] and NoScript ABE [177]. While these client-side solutions have registered some success among enthusiasts, their main disadvantage is the need for compatibility with all sites, often resulting in false positives.

Best Practices

Best practices for protecting against CSRF attacks are twofold, combining token-based approaches with selective unforgeable user involvement. Ideally, web developers protect should against CSRF attacks by using built-in protection mechanisms for state-changing operations. Alternatively, custom token-based approaches can be integrated as well, taking precautions to prevent token compromise. For legacy applications, the use of a transparent server-side solution, such as CSRFGuard, can enable CSRF protection without having to fiddle with the application code.

As a second line of defense, it is recommended that user involvement should be required for truly sensitive operations, especially when they have direct financial consequences, or can lead to the compromise of a user account. For example, it is not unreasonable to require explicit user involvement or re-authentication when changing the password or making a wire transfer in an online banking system.

9.2 Login CSRF

A Login CSRF attack is a special variant of a CSRF attack, where the attacker secretly authenticates the victim with an attacker-chosen account. When the user unknowingly uses the targeted application, any entered information belongs to the attacker-chosen account, and can potentially leak to the attacker. A common example is a search engine keeping a history for authenticated users.

Problem Description

The goal of a login CSRF attack is to authenticate the victim to the target application using the attacker's credentials, thereby establishing an authenticated session in the victim's browser, associated with the attacker's credentials in the target application. Such a forged authentication can have serious consequences, for example when search engines store a query history or payment sites that require disclosure of credit card information [26].

The process of carrying out a login CSRF attack is identical to a CSRF attack (Section 9.1), albeit with slightly different consequences. The state-changing action that is executed in the background is the submission of an authentication form, and instead of executing an action in an authenticated session, the attack will establish an authenticated session. Establishing the session typically happens by setting a cookie using the `Set-Cookie` header.

A traditional login CSRF attack submits the authentication form of the target application using the attacker's credentials. In the modern Web, more and more applications have started using third-party authentication providers such as Google single sign-on, Facebook authentication or OpenID. These authentication providers typically provide the target application with an *assertion*, containing the necessary info to confirm a successful authentication, as well as the user's identity. Using a login CSRF attack, an attacker can submit his own assertion to the

target application from the victim's browser, effectively establishing an authenticated session tied to the attacker's credentials.

Mitigation Techniques

Login CSRF is a variant of a CSRF attack, so the same mitigation techniques can be applied. Most mitigation techniques are easier to apply, since requests leading to authentication are unlikely to happen across multiple origins. One difference that might affect certain mitigation techniques is the potential lack of an established session before completing the authentication form. For example token-based approaches that store information in the user's session might not work, but establishing a temporary session or using application-wide tokens easily overcome these problems.

Research and Standardization Activities

As with the mitigation techniques, research on CSRF also applies to login CSRF attacks. The `Origin` header [26] can be used to restrict authentication requests to the origin of the application, effectively preventing cross-origin requests leading to authentication. A similar approach is proposed by the client-side countermeasure DeRef [108].

A formal analysis of web site authentication flows [18] has shown that applications using third-party authentication based on OAuth suffer from the login CSRF problem, which the authors named a *social login CSRF*. OAuth is a protocol enabling third-party clients limited access to an API, such as used in Facebook authentication [93]. The OAuth specification recommends using a generated nonce, strongly bound to the user's session, which would prevent a social login CSRF attack, if followed by the implementations of the protocol.

Best Practices

The main mitigation of a login CSRF attack is preventing unintentional authentication requests. For applications with traditional login forms, this can be achieved by using traditional CSRF protection techniques. Session-bound tokens can be used when a session between browser and server exists before authentication. Applications using third-party authentication services have to ensure that the assertion produced by the authentication provider is tied to a user's session, preventing the attacker from using his assertion in the victim's session. Typically, this is supported by including a session-bound nonce in the assertion.

9.3 Clickjacking

A clickjacking attack, also known as a UI redressing[1] attack, in one form redresses or "re-decorates" a target application so that the user believes he is interacting with a (typically, non-sensitive) part of the application, but actually tricking the user into clicking on a specific location that actually activates some other (more sensitive) function, with the result that the click is actually sent to the target application to perform the more sensitive operation. Many forms of UI redressing are possible, from transparent overlays to very precise positioning of elements, or even fake cursors stealing the user's attention. The result of the attack is a legitimate click on an element of the target application although not the one that the user thought he had clicked, potentially approving security-sensitive actions.

[1] The term *redress* as used in this context is somewhat confusing because it doesn't conform to the normal English language definitions of "redress" as in "making amends" or "rectifying a problem". It should more properly be written *re-dress* as it signifies that an existing item on the browser display has been given a new set of clothes, typically some of the Emperor's (invisible) New Clothes. However, unlike the case of the Emperor, these new clothes can be made to have a real effect.

Problem Description

The goal of a clickjacking attack is to forge a request from the victim's browser to the target application, by making the user unintentionally click an element on a page of the target application. An attacker typically achieves this using misdirection, by redressing the UI of the page to hide the real element that will be clicked by the user (illustrated by Figure 9.3).

Figure 9.3: The essence of a clickjacking attack is tricking the user into clicking on a specific location, under which an element of the target application is positioned. This example shows a clickjacking attack on Twitter's account deletion page [217].

The attack tree in Figure 9.4 shows that a clickjacking attack requires a coordinating application, which is under control of the attacker and will coordinate the attack. The coordinating application masquerades the target application and tricks the user into clicking on a specific location. The coordinating application makes sure that the click is not directed at the coordinating application, but at the target application. A common way to achieve this, is by hosting the target application in a transparent `iframe`, and positioning it exactly under the mouse pointer (illustrated in Figure 9.3). Note that while the attacker loads the target application in a frame, the same-origin policy prevents script-based access to the frame's contents. This also means that a clickjacking attack cannot be stopped by countermeasures depending on same-origin policy protection, such as CSRF tokens, since the forged request in a clickjacking attack will be generated by the target application itself.

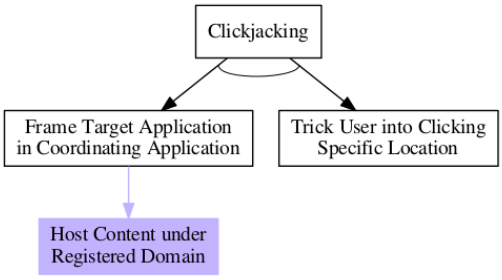


Figure 9.4: A clickjacking attack tricks the user into clicking an invisible or hidden element on a page of the target application.

In essence, a clickjacking attack generates an unintentional request by misdirecting the click of a user. Since a well-conducted clickjacking attack is impossible to observe, users are not to

blame. Once again, a clickjacking attack is inherent to how the Web works, exacerbated by the overlapping window model generally used for browser displays.

Mitigation Techniques

Traditional mitigation of clickjacking attacks happens through *framebusting code*. Framebusting code is targeted at detecting the unpermitted framing of an application, and subsequently breaking out of the frame by moving the application to a top-level frame, as shown by the example in Listing 9.2). Simple framebusting code is often easily evaded, but more advanced framebusting code has been proposed [217]. The downside of framebusting is its strict on or off mode, either allowing all kinds of framing or no framing at all, not even by trusted applications. In the modern Web, with mashups and composed applications, this might be problematic.

```
1 if(top != self) top.location.replace(location);
```

Listing 9.2: A simple approach aimed at detecting unpermitted framing, and breaking out of the frame by moving the application to the top-level frame. While this countermeasure is often used, it is easily evaded [217].

A second popular mitigation technique is the regulation of framing through the `X-Frame-Options` header [216]. By adding this header to the response, an application can indicate that framing is denied, allowed within its origin, or allowed by an explicitly listed set of URIs.

Alternatively, clickjacking can be combated from the client-side as well. The popular security add-on NoScript [176] includes the ClearClick module, which does a screenshot-based comparison of the area to be clicked with the actually clicked element. When a difference between both is detected, the user is warned and explicitly asked to confirm the action, before the request is sent.

State of Practice By design, all web applications are vulnerable to clickjacking attacks, but the attack receives little attention compared to higher-risk attacks. Users of major, well-known web applications such as Facebook, ... etc. have fallen victim to clickjacking attacks, typically indicated by spam messages making their way through the application.

Many web applications deploy some form of framebusting code, of which several variants are known to be vulnerable to evasion [217]. Additionally, many applications have a different front-end for normal browsers and mobile browsers, often only implementing framebusting in their normal version [218]. Similarly, the `X-Frame-Options` header is used by a small number of sites, offering only limited protection to users.

As a last resort, users have been able to protect themselves using the NoScript browser add-on [176], which includes the ClearClick module, which detects potential clickjacking attacks, and explicitly asks for user confirmation before continuing.

Research and Standardization Activities

A first line of research investigates new kinds of clickjacking attacks, as well as their potentially devastating effects. With the recent popularity of smartphones and mobile browsers, clickjacking has evolved to *tap-jacking* [218], with a series of new tricks to hijack taps on a smartphone or mobile device. Alternatively, advanced research into the consequences of a clickjacking attack has shown that clickjacking allows an attacker to obtain webcam access, steal private user data or break the user's anonymity [138].

Another line of research focuses on the defensive side, improving existing countermeasures and developing new countermeasures. As discussed before, framebusting code is often ineffective and easily evaded, but carefully constructing robust framebusting code can withstand evasion,

or fail in safe ways [217]. Recent work has also focused on a browser-supported solution that addresses the root of clickjacking attacks, i.e., user misdirection. *InContext* [138] incorporates several measures that ensure that a user's click is genuine, for example by comparing screenshots at the time of the click, similar to ClearClick, or by highlighting the area of the cursor to prevent attacks involving a fake cursor.

InContext also serves as inspiration for the new standardization efforts by W3C to ensure UI integrity in web applications, effectively preventing clickjacking attacks [178]. The specification proposes several new directives to include in the Content Security Policy (covered in Section 11.1), giving the developer control over several heuristics to determine the genuineness of a click. Similar to other directives in the Content Security Policy, the browser will enforce these heuristics for user clicks, blocking and reporting any violations.

Best Practices

Ideally, applications employ both effective framebusting code where possible (illustrated in Listing 9.3), combined with the `X-Frame-Options` header, configured as tightly as possible. Introducing additional user interactions, such as an explicit confirmation dialog, will certainly make clickjacking attacks more difficult, but will not always be sufficient to eradicate them [138].

```
1 if( self == top ) {
2   document.documentElement.style.display = 'block' ;
3 } else {
4   top.location = self.location ;
5 }
```

Listing 9.3: Combining this framebusting code with content that is hidden by default, offers clickjacking protection that can not be evaded, and fails securely when the detection is hampered somehow.

In the near future, the newly standardized User Interface Safety Directives for CSP will become available [178], giving more fine-grained control to determine whether a click is genuine.

Chapter 10

Attacking through the Network

With the Web being a distributed platform, the *content in transit* asset plays a central role, not only covering a user's personal information, but all network data relevant to the web application. It is therefore not surprising that the threats to content in transit, *eavesdropping*, *generating traffic* and *intercepting and manipulating traffic*, not only compromise the content in transit, but also allow escalation aimed at the compromise of additional assets.

The core problem of protecting the *content in transit* asset, is the open infrastructure of the Web, with, for example, wireless access points making eavesdropping child's play. The focus for overcoming this problem does not lie on making the infrastructure less open, but rather on securing the information that is transmitted over the network, instead of simply transmitting even the most sensitive information in plain text.

In this chapter, we cover these threats in order of increasing complexity, starting from an eavesdropping attacker. Then, we consider active, man-in-the-middle attackers who can carry out SSL stripping attacks, or attempt to execute direct man-in-the-middle attacks on a secure connection. Finally, we cover recently discovered sophisticated attacks that attempt to break the guarantees of a secure connection, after it has been established.

This chapter focuses on the security of the network infrastructure and its mitigation techniques, and does not cover the consequences of compromising the *content in transit* asset. Consequences such as privacy violations or identity theft belong in Chapter 14, where user privacy is covered.

As a final remark, we would like to note that at the time of writing, network security is a "trending topic", and a fast-moving area, due to the high-profile news stories following from the Snowden revelations [244]. The text is written based on the assumptions that were believed to be true at the time of writing.

10.1 Eavesdropping

Eavesdropping is a passive attack on the network infrastructure, allowing an attacker to remain largely undetectable. Targeted eavesdropping attacks focus on a specific user or application, and typically exploit components in the local network infrastructure. Pervasive eavesdropping is very similar, but on a larger scale, targeted at collecting large-scale information, without specifically focusing on a single user or application.

By eavesdropping, an attacker can collect sensitive information on both the user, as well as the target application. In an HTTP context, this information can lead to further escalation, such as stealing a session cookie to perform a session hijacking attack.

Problem Description

With an eavesdropping attack, an attacker is able to listen in on other users' network traffic, such as DNS queries, HTTP requests and responses, etc. By eavesdropping on their network traffic, an attacker is not only able to learn sensitive, personal information, such as credit card info, financial means, usernames, passwords, contents of email messages, etc., but can also listen in on important web metadata, such as session identifiers or supposedly secret cookies. Obtaining any of this information can cause direct harm, or might enable an attacker to escalate the attack, for example through *session hijacking* (See Section 8.1).

The way of executing an eavesdropping attack depends on the network under attack. For example, eavesdropping on airborne signals, such as wifi, radio or cellular, typically only requires an antenna in the proximity of the network. Eavesdropping on a switched, wired network requires some interference, for example by running an ARP spoofing attack. Eavesdropping can also occur at intermediaries within the network infrastructure, for example at an ISP, a proxy server or a Tor [79] exit node. Even higher up in the network (See Intermezzo 6), an attacker can eavesdrop on backbone traffic, with submarine taps on fiber-optic cables [14] as an extreme example.

Technically, running an eavesdropping attack is fairly straightforward. The browser add-on *Firesheep* [46] enables a user to eavesdrop on a wifi network, abusing obtained session identifiers to perform a session hijacking attack with one point-and-click operation. Alternatively, software tools such as Subterfuge [246] and dedicated devices such as the *Pineapple* [122] make collecting sensitive information a straightforward task. Eavesdropping on a wired, switched network is also possible with a wide variety of freely available tools, such as *Ettercap* [89] or *dsniff* [236].

Essentially, eavesdropping attacks will always be possible, especially with the evolution towards wireless networks. However, the real problem with eavesdropping is the huge amount of information that is transmitted in the clear, making the physical access to the network signals the only barrier to overcome.

Mitigation Techniques

The main approach to protect against eavesdropping attacks is to deploy security protocols, effectively protecting the data being sent over the network. The use of network-specific data link-layer security protocols, such as WPA [260] and EAP [3] can effectively help in mitigating a local eavesdropping attacker, but does not protect against eavesdropping beyond the local network.

An approach offering end-to-end security is using HTTP deployed over TLS, where TLS is aimed at offering confidentiality, integrity and entity authentication, effectively eliminating the usefulness of the data in transit to an eavesdropping attacker (See Section A.4.4 for more details about TLS). Note that TLS can also be deployed with self-signed certificates, where no Certificate Authority has vetted the server's identity. While this deployment scheme offers no protection against man-in-the-middle attacks, covered in the next section, it can offer protection against passive eavesdropping attacks.

State of Practice While TLS effectively tackles network attacks, it is not yet widely deployed, although adoption is growing (See Intermezzo 3). Additionally, older and less secure versions of TLS that are deployed, are rarely upgraded to the latest version, leaving a trail of inadequate legacy implementations across the Web. While the specific reasons for the slow adoption of TLS are hard to pinpoint, potential candidates are its (antiquated [164]) reputation imputing significant performance impact, the difficulty of managing and deploying certificates, potential interference or incompatibility of the encrypted traffic with middleboxes, such as proxies and caches, and general ignorance from web application operators. Additionally, TLS is often used incorrectly, which may be attributed to relatively hard-to-use APIs and incorrectly configured trust-roots.

As an indication of the current deployment state of TLS, a monitoring site[1] reports that approximately 30% of the top 10-million web sites are using TLS with certificates issued by a recognized CA.

Research and Standardization Activities

The TLS protocol itself is undergoing constant revision and its security is the topic of ongoing cryptographic research. Recent examples are the discovery of timing attacks against the cipher block chaining (CBC) mode of operation [10], allowing the extraction of cookies from the encrypted stream, and the identification of weaknesses in the RC4 algorithm [9], supporting the common belief that RC4 should be considered broken. Reacting to these results and other similar research output, the TLS working group within the IETF is currently considering new cipher suites, for example based on the *ChaCha20* cipher [165].

In addition to new ciphers, TLS deployment is also an important factor to consider. Older versions of TLS remain in use for a considerable period, even after the introduction of newer versions with better security features. For example the attacks mentioned in the previous paragraph have already been countered via authenticated encryption cipher suites that were defined as part of TLS 1.2 [77], but so far version 1.2 has not seen very widespread deployment. However, in response to these attacks, deployment roadmaps for TLS 1.2 have been accelerated by browser vendors and the open-source security community.

Another standardization proposal focuses on a new Best Current Practice (BCP) for the use of TLS on the Web [229], aiming to aid web application developers and administrators in the use of TLS. One example strategy that is advocated is the achievement of *perfect forward secrecy* (PFS), which guarantees that previously recorded TLS sessions can not be deciphered by learning the server's private key at a later point in time. While PFS has previously been little deployed due to an impact on performance and requiring less commonly used cipher suites, it has come into the spotlight again, due to private keys leaking out, for example if someone hacks a web server, or if server units are decommissioned inappropriately.

Finally, a large effort is being spent on the development of the successor to HTTP, HTTP/2.0 based on Google's SPDY protocol. While initially SPDY was proposed to always run over TLS, that position was somewhat watered down in HTTP/2.0, mainly due to interference with middleboxes, such as HTTP proxies that need to be able to see some HTTP metadata (e.g., headers) to function. Nonetheless, HTTP/2.0 will be far more likely to be deployed running over TLS, since the *application layer protocol negotiation* TLS extension [107] offers the most efficient upgrade path, with the fewest additional network round trips. Additionally, discussions to allow clients and servers to make use of TLS even for HTTP (i.e. non-HTTPS) URIs when using HTTP/2.0, are ongoing.

Best Practices

The best practice to protect against eavesdropping attacks is to deploy TLS everywhere (See the best practices of Section 10.2 for additional information), achieving confidentiality for all data and metadata sent between the user's browser and the web application. Additionally, by selecting cipher suites offering perfect forward secrecy, the encrypted data is even protected against an attacker who learns the private key in the future. The IETF's Best Current Practice document exclusively covers the best practices with regard to TLS deployment [229].

10.2 Overturning a Secure Connection

Securing the network connection is the typical approach to providing protection against network-level attacks. However, the interactions between insecure and secure connections on the Web,

[1] `http://w3techs.com/technologies/overview/ssl_certificate/all`

respectively over HTTP and HTTPS, can cause subtle vulnerabilities, allowing attackers to downgrade the setup of a secure connection, without the user being aware of the attack. Such a degradation gives the attacker access to personal user information, such as authentication credentials, and allows further tampering with the user's actions within the web application.

Problem Description

In 2009, Moxie Marlinspike argued [179] that users visiting a secure web application probably will not type the `https://` part of the URI manually, meaning that the initial request will be made over HTTP. Typically, web applications then redirect the user towards the correct HTTPS URI, causing a transition from HTTP to HTTPS. Exactly this transition can be exploited by an attacker that sits between the victim and the target application, causing the downgrade of the connection from HTTP to HTTPS, which is called an *SSL Stripping* attack. This downgrading allows the attacker to steal the victim's sensitive information, such as authentication credentials or cookies.

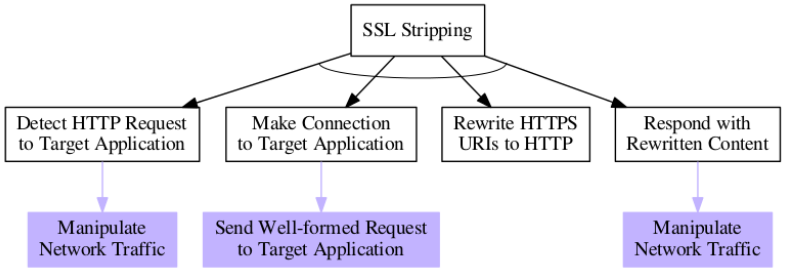


Figure 10.1: An SSL stripping attack allows an man-in-the-middle attacker to downgrade the secure connection by rewriting all HTTPS URIs to HTTP URIs, thereby gaining access to the user's sensitive information.

Downgrading from HTTPS to HTTP is straightforward, as shown in the attack tree in Figure 10.1. An attacker that already has a man-in-the-middle (MitM) position in the network, detects the user contacting the target application on an HTTP URI, which causes a redirect to an HTTPS URI. The attacker translates every HTTPS URI towards the victim's browser into an HTTP URI, preventing the secure connection from being established. However, because the browser never expects a secure connection to be initiated, no warnings are generated. The only noticeable effect is the missing lock icon, somewhere in the browser's user interface.

SSLstrip [180] is an automated tool to conduct SSL stripping attacks. SSLstrip intercepts the HTTP traffic from the client, and impersonates the client to initiate HTTPS communication with the server, serving the page's contents over HTTP to the victim's browser. Every HTTPS URI that is encountered is rewritten with an HTTP counterpart, and a database with mappings from HTTP URIs to HTTPS URIs is stored, allowing SSLstrip to restore the URI when sending it to the target application. Figure 10.2 shows the flow of an SSL stripping attack.

In essence, an SSL stripping attack is possible because the browser does not know when to expect a secure connection, and thus gladly accepts an insecure connection.

Mitigation Techniques

For long, the main mitigation for SSL stripping attacks has put the burden on the user, who should detect the presence of the lock icon to indicate a secure connection. While this may work

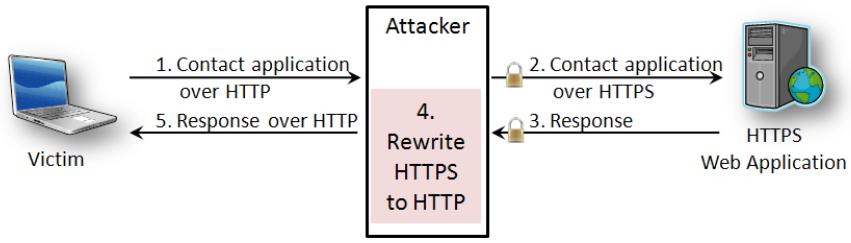


Figure 10.2: In an SSL stripping attack, a man-in-the-middle attacker intercepts an HTTP request to the target application, initiates an HTTPS request from his own machine and relays the content between both connections, effectively downgrading the victim's connection to an insecure HTTP connection.

for experienced security professionals, it is definitely not an appropriate mitigation technique for the common web user. Additionally, using stored links, that immediately refer to the secure part of the web application, instead of relying on the HTTP redirect, also mitigate SSL stripping attacks. For example, by bookmarking the correct link, a user can prevent the SSL stripping attack from taking place.

One technological solution is provided by the HTTPS Everywhere [86] browser add-on, which forces the use of HTTPS on sites that support it. By forcing the use of HTTPS, SSL stripping attacks are effectively mitigated, since a direct HTTPS connection will be made. Additionally, HTTPS Everywhere prevents sites from downgrading to an HTTP connection after authentication, a commonly used bad practice, leading to session hijacking attacks (See Section 8.1).

Research and Standardization Activities

The research proposal *HProxy* [196] prevents SSL stripping attacks by leveraging the browser's history to compose a security profile for each site, and validating any future connections to the stored security profiles. This approach effectively detects and prevents SSL stripping attacks without server-side support and without relying on third-party services.

An alternative proposal is *ForceHTTPS* [144], allowing web sites to opt-in to strict rules regarding TLS usage, such as strict TLS error processing, preventing network attacks against HTTPS connections. Specifically, ForceHTTPS protects against SSL stripping attacks by preventing the browser from connecting to a ForceHTTPS-protected site over HTTP, and redirecting it to HTTPS instead.

The idea behind ForceHTTPS has been taken towards standardization as *HTTP Strict Transport Security (HSTS)* [137], allowing a server to require that browsers supporting HSTS can only connect over HTTPS, effectively thwarting any SSL stripping attacks. A server can enable HSTS protection by including a *Strict-Transport-Security* response header, declaring the desired lifetime for the HSTS protection. One caveat to HSTS being implemented as a response header, is the first contact with a site, when it is unknown whether an HSTS policy applies or not. This issue has been addressed by modern browsers including a predefined list of HSTS-enabled sites, effectively avoiding an initial HTTP connection.

Best Practices

Best practices to prevent downgrading attacks on HTTPS connections are twofold: ensure that the web site is only reachable over HTTPS, and deploy a long-lived HSTS policy, instructing browsers to connect over HTTPS.

10.3 Man-in-the-Middle Attacks

Performing a man-in-the-middle attacks not only gives the attacker full access to all transmitted content, but also allows tampering with the user's actions within the web application. TLS-secured connections are designed to withstand man-in-the-middle attacks, but flaws in the supporting systems may allow for subtle attacks to be carried out anyway. These flaws are caused by misplacement of trust in certain parties, or by placing the decision-making burden on the user.

Problem Description

A man-in-the-middle attack (MitM) is an active network attack, where the attacker inserts himself in the network, between the victim and the targeted web application. This position not only allows the attacker to inspect all traffic that is sent between the victim and the target application, but also allows modification of the traffic. Such a compromise gives the attacker full control over the user's actions, with potentially disastrous effects. Note that there are also "legitimate" use cases for performing a MitM attack, such as ISPs injecting advertisements into HTTP responses, or corporations deploying a web content filter, responsible for filtering unwanted or harmful content.

Actually becoming a man in the middle in the network can be achieved at many levels. An attacker can physically place a machine in the network path, forcing the data to flow through this machine, or can manipulate the network's parameters, to act as a gateway on the logical level, for example through ARP poisoning attacks. Once an attacker has become a MitM, inspecting and manipulating traffic is straightforward.

Traditionally, TLS is deployed to prevent eavesdropping and MitM network attacks, since it offers confidentiality, integrity and entity authentication. The confidentiality and integrity effectively prevent an attacker from modifying any network traffic, while the entity authentication property ensures that the involved parties are who they claim they are, thereby preventing a MitM attack within the TLS connection. Even though TLS is designed to counter MitM attacks, in reality, they remain possible for several reasons.

First, the entity authentication in TLS is based on private/public key pairs, of which the public key is verified by a Certificate Authority (CA), which is part of the Public Key Infrastructure (PKI) (See Section A.4.4 for more details about TLS). Unfortunately, any CA in the Web's PKI can issue a certificate for any web site, since no tightly bound name constraints are offered, in spite of the technology being available [58]. With approximately 57 trusted root CAs in a modern browser, any web site is vulnerable to an attack with fraudulent but verified certificates being issued.

Second, whenever an invalid certificate is encountered by a browser, the burden of the security decision is placed on the user. Regardless of whether the invalid certificate is caused by an expired expiration date, or a complete mismatch with the targeted web site, browsers show scary warnings, asking the user to decide whether to trust the site or not. Since users also encounter these warnings for legitimate sites, a simplistic MitM using an invalid certificate has some chances of success.

A third degradation of the CA system in TLS comes from the deliberate MitM devices, deployed by enterprises and large organizations with the goal of filtering inbound and outbound web traffic. Reasons to deploy such filtering mechanisms go from offering protection, for example with a web application firewall (WAF), to preventing employees from accessing sites that are deemed inappropriate, such as social networking applications. The problem with such devices is that in order to perform a MitM function on secure connections, they either have to install their own certificate on a user's machine, or they have to obtain a valid certificate for every TLS-protected web site on the Web. The former is a configuration hassle, which only works if you control all the client-side devices as well, and the latter seems impossible. Unfortunately,

some CAs have been found to be collaborating with the vendors of deliberate MitM devices, thereby harming the trust placed in the system. [2]

Finally, the trust placed in CAs is easily abused when a CA is compromised. For example, the hacking of DigiNotar [202] resulted in the issuing of fraudulent certificates, allowing MitM attacks on secure connections to Yahoo, Mozilla, WordPress and the Tor project. The trusted roles of CAs can even be further compromised by government coercion to issue fraudulent certificates. This strategy is believed to be common practice in non-democratic countries [224], but recent revelations show that this practice is widely deployed by secret agencies across the world [182].

The essence of the problem with MitM attacks, especially against TLS connections, is the misplaced trust in the system on the one hand, and the burden of the security decision on the user on the other hand. Clearly, blindly trusting every root CA in the world has been proven to be a bad idea, and typical Web users are not capable of making technical decisions about trusting a certificate or not.

Mitigation Techniques

Mitigation techniques against MitM attacks on TLS focus on determining the trustworthiness of the presented certificate. Certificate Transparency (CT) [168] aims to maintain a public, write-only log (based on a Merkle hash-tree) of issued certificates so that either user agents or auditors can detect fraudulent certificates. This would require a user agent to query the log during the TLS handshake, and auditors can query the log offline, to check for certificates being unexpectedly issued for one of their sites.

A second approach is based on detecting discrepancies between the currently presented certificate, and previously seen certificates. While this approach requires the first connection to be secure, it effectively enables the detection of unexpected future updates. This approach is implemented in the browser add-on, and proposals to achieve this at the HTTP, TLS or other layers have been made [91, 137].

Alternatively, Google has taken the approach of hardcoding the certificate fingerprints of Google-related TLS certificates, allowing Google Chrome to detect a potential MitM attack, even with a fraudulent certificate issued by a CA. Naturally, controlling both the services and the distribution platform is a key to the success of this approach.

Research and Standardization Activities

Several proposals for alternate schemes to verify certificates have been made and evaluated [116], but the standardization work on Certificate Transparency seems to be most likely to gain widespread support, which is required for it to become an effective mitigation technique. In addition to CT-like approaches, the IETF's DNS-based Authentication of Named Entities (DANE) working group has explored linking of the public key infrastructure to DNS, leveraging the security of DNSSEC, thereby avoiding the name constraint problem that enables MitM attacks. DANE however, is perhaps less suited for the Web due to the current lack of deployment of DNSSEC and a corresponding lack of a well-defined transition path from today's PKI to a DANE based PKI. DANE may however, be valuable for other protocols such as SMTP/TLS.

State of Practice

Best practices for avoiding MitM attacks on TLS-secured connections are hard to give, since preventing such attacks is entirely the goal of TLS. Depending on the future technology relating to certificate validation, the publication of DANE TLSA records through DNSSEC and choosing a CA that supports Certificate Transparency, are likely to be good practices. As a side note,

[2] For example: "Clarifying The Trustwave CA Policy Update," Feb. 2012, `http://blog.spiderlabs.com/2012/02/clarifying-the-trustwave-ca-policy-update.html` and "Public announcements concerning the security advisory," Feb. 2013, `http://turktrust.com.tr/en/kamuoyu-aciklamasi-en.html`

self-signed TLS certificates pose a certain risk, since anyone can generate a valid self-signed certificate for any given domain. However, self-signed certificates can be used in one of the previously described systems, which detect changes in the certificate anyway.

Browsers can provide better TLS error handling, clearly indicating which errors are severe, and which are benign. Fortunately, improving the TLS experience on the user side seems to be an ongoing effort for browser vendors.

10.4 Internal Threats to TLS and HTTP/TLS

Once a secure connection has been initiated, without a man-in-the-middle being present, the transmitted content should be secure. However, sophisticated attacks on the TLS and HTTP over TLS protocols have been able to extract data from a secure connection, or to inject data into the stream. Luckily, these attacks are largely mitigated in upgraded versions of the TLS protocol. Nonetheless, actively scrutinizing and repairing security protocols remains essential for network-level security.

Problem Description

As one of today's major Internet security protocols, TLS is widely studied and scrutinized, leading to the discovery of new attacks and associated mitigation techniques. Recent history indicates that these attacks are anticipated well in advance, mitigated by the introduction of new cipher suites, which are in turn rolled out in a TLS update. Unfortunately, adoption of these new versions is slow, potentially due to the anticipated attacks as being perceived as theoretical. Naturally, the discovery of a practical attack sparks a rush to adopt the mitigation techniques already proposed.

A lesser-known feature of HTTP and TLS protocols is client authentication, where HTTP offers the *Authorization* header (See Section A.4.2), and TLS offers the ability to use client certificates. These features have some non-trivial use, for example TLS's client authentication being used in WebDAV [112], making them a relevant part of the Web's infrastructure. These features have been around for quite some time, and over the years, some problems and vulnerabilities have been discovered, sparking new mitigation techniques and additional research.

In essence, establishing a secure connection on the Web remains an active research topic, which will be illustrated by the remainder of this section, and requires deployments to keep up-to-date with the latest protocol versions.

Mitigation Techniques

Important use cases of TLS client authentication are WebDAV [112] deployments, and server-to-server communication. A vulnerability in the TLS renegotiation procedure allowed the injection of plaintext into the channel, but confidentiality was never threatened. This attack has been quickly mitigated [209], and has seen reasonably good deployment since its release.

Additionally, a number of recently demonstrated attacks violate the security guarantees offered by the TLS protocol. TLS 1.2 [77] effectively mitigates these attacks, and as a result, the adoption of the new version is accelerating. For completeness, we describe several attacks below.

Lucky-13 The *Lucky-13* attack [10] is a side-channel attack against the *MAC-then-encrypt* scheme used in TLS for cipher-block-chaining (CBC) cipher suites. Since Lucky-13 is a network timing attack, it requires the attacker to be nearby in the network and to issue tens of thousands requests to the target application, in order to extract a plain text value. One target is sensitive cookie values, such as the session identifier, which may lead to an escalation of the attack.

BEAST The *Browser Exploit Against SSL/TLS (BEAST)* [84] attack uses active scripting in the browser, in conjunction with a colluding intermediary, to exploit a CBC vulnerability, allowing the decryption of network traffic. BEAST is the first practical implementation of a previously-known vulnerability, and has already been mitigated in TLS 1.1.

CRIME and BREACH The CRIME[213] and the recent BREACH[111] attack exploit compression at the HTTP or TLS layers, allowing an attacker to guess plain texts based on compression ratios. These attacks can be prevented by turning off compression within TLS, the default configuration, although turning off compression in HTTP might be less practical.

RC4 Attacks Recent attacks against the use of RC4[9] target sensitive plain text in fixed positions within the cipher text. Since cookies are often transmitted in this way, or can be easily forced to be, this attack impacts real-life deployment scenarios.

Research and Standardization Activities

Recent standardization activities focus on the development of TLS 1.3 within the IETF's TLS working group. TLS 1.3 aims to counter all known TLS attacks, and will use more modern cipher suites, deprecating the vulnerable ones. Similarly, the development of HTTP/2.0 will counter attacks abusing header compression, such as the CRIME attack [213].

Recently, HTTP authentication schemes are receiving renewed attention, with proposals aiming to overcome the problems with HTTP Basic and Digest authentication [106], such as lack of control over the user interface, lack of a logout function, and the clear text or hashed transmission of the user credentials. One proposal is *HTTP Origin Bound Authentication (HOBA)* [95], which aims to provide a digital signature challenge-response mechanism to perform HTTP-based authentication.

Best Practices

The best practice to prevent internal threats to a secure channel is to closely follow new developments in the field, deploying the necessary updates when they are released. Selecting a good set of cipher suites and choosing a trustworthy CA are good practice.

Chapter 11

Controlling the Client-side Context

By compromising the *client-side application code* asset, an attacker essentially gains control over the client-side context, allowing further escalation aimed at the *client-side content storage* and *application transactions* assets. The main threat targeting the compromise of *client-side application code* is the ability to *run an attacker-controlled application component* within the application's security context. Applications that heavily depend on *client-side content storage*, also risk being compromised through the *directly accessed client-side storage* threat and the *abuse client-side application privileges* threat.

The core problem behind an attacker controlling the client-side context is that any attacker-controlled code typically runs within the same security context as the application's code, giving the attacker code full access to the application's client-side data, resources, APIs, permissions, etc. Furthermore, from within the client-side context of the application, an attacker can easily construct requests that appear to be legitimate requests coming from the application, but are in fact attacker-constructed requests, thereby heavily compromising the *application transactions* asset. The main culprit leading to this kind of attack is a lack of code integrity validation, making it impossible to distinguish between legitimate application code and code manipulated or injected by an attacker.

In this chapter, we cover several ways to insert attacker-controlled code into the client-side execution context, giving the attacker full control over the client-side context. The best known attack is *Cross-site Scripting* (XSS), where the attacker inserts malicious code into legitimate application pages, which is in turn executed within the application's context. Another attack vector is to *compromise JavaScript inclusions*, either remotely at the server, while in transit or affecting a locally cached version.

This chapter does not focus on the escalations that can follow from controlling the client-side context, such as exploiting browser vulnerabilities to install malware, or specifically targeting vulnerable browser extensions, with the goal of taking over their privileged execution context. These kind of attacks are covered in Chapter 12.

11.1 Cross-Site Scripting (XSS)

With cross-site scripting, an attacker is able to execute attacker-controlled code within the application's execution context, where the attacker-controlled code has the same privileges as the application code. This exposes all client-side application data, resources and APIs to the attacker, and gives the attacker the possibility to manipulate and generate legitimate application requests towards the server-side application code.

Problem Description

The goal of a cross-site scripting (XSS) attack is to execute attacker-controlled code in the client-side application context within the victim's browser. In a successful XSS attack, the attacker typically provides JavaScript code instead of static content, making the target application unknowingly deliver the payload to the victim's browser, where it's executed. Since the browser is unaware that part of the document being rendered is actually injected as code, it is not capable of preventing the execution of the payload.

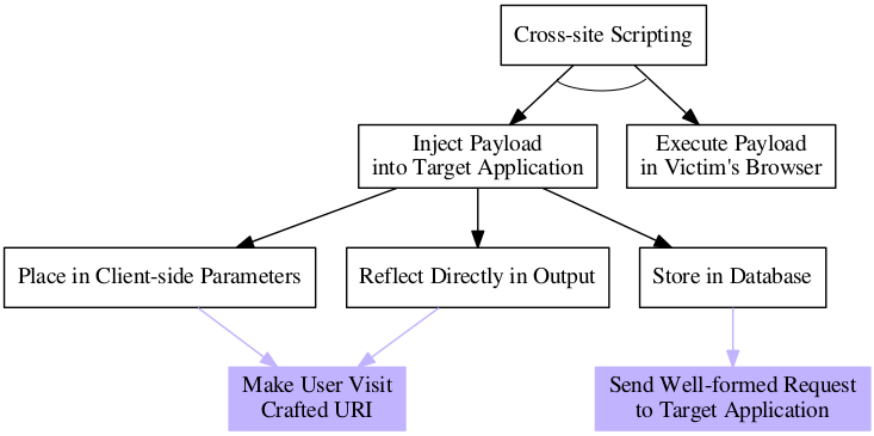


Figure 11.1: A cross-site scripting attack tricks the target application into executing an attacker-controlled payload in the victim's browser, granting the payload full access to the target application's client-side context.

The attack tree in Figure 11.1 shows how an XSS attack is conducted and illustrates the different ways to inject the payload into the target application, which are often used to classify XSS attacks. A first way is by placing the payload in parameters that are processed by client-side scripts, such as a URI parameter or fragment identifier. Whenever the client-side script processes this parameter, for example to determine the preferred language, it will unknowingly execute the payload. This type of XSS attack is known as *DOM-based XSS* or *XSS type 0*. A second way to get the payload accepted by the client-side is to inject it into a parameter that is used by the server-side code, and then reflected back to the client as part of the result page (Figure 11.2). A typical example is a search field, of which the contents are part of the response, for instance in the page title. This type is known as *reflected XSS* or *XSS type 1*. Finally, an attacker can also place the payload in data that is stored by the application, and used to create responses to dynamic requests. A common example here is a forum post or comment to a blog post, where the attacker includes script code in his post. Whenever the application sends the forum posts or blog comments to a user, the attacker's code will be inserted as well. This type of XSS is known as *stored XSS* or *XSS type 2* (illustrated in Figure 11.3).

```
http://example.com/search.php?q=%3Cscript%3Ealert(%22XSSed!%22)%3C%2Fscript%3E
```

Figure 11.2: When the vulnerable web application processes this URI, the source of the response will include `<script>alert("XSSed!")</script>`, leading to a reflected cross-site scripting attack.

In essence, the problem of an XSS attack is the failure of the target application to recognize the insertion of code, thus allowing the payload to be executed. The combination of the facts that code can be placed anywhere in a document, and that browsers attempt to correct syntactically incorrect documents rather than rejecting or ignoring them helps the easy exploitation of injection vulnerabilities.

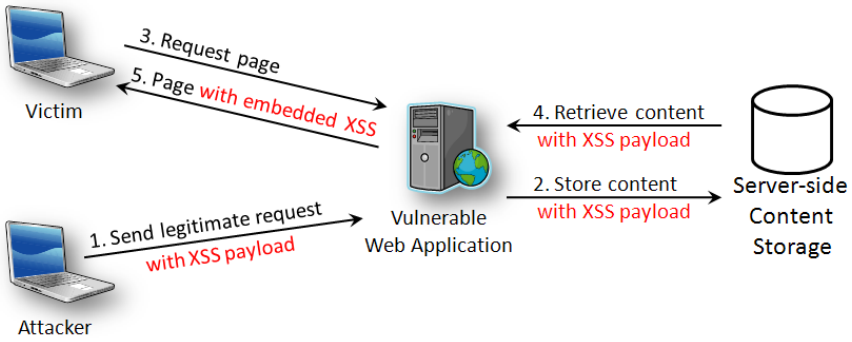


Figure 11.3: In a stored cross-site scripting attack, the attacker injects script code into the application's server-side content storage, which is then unknowingly served to victim users, visiting legitimate pages of the application.

Mitigation Techniques

The traditional mitigation technique used against cross-site scripting attacks depends on sanitizing input and output, preventing any dangerous input from reaching the final output. Traditional sanitization techniques attempted to simply replace or remove dangerous characters such as `< > & " '`, but modern sanitization techniques take the context of the output into account.

Modern web applications generate output with different contexts, with different output formats and injection vectors. Some example contexts are HTML elements, HTML element attributes, CSS code, JavaScript code, etc. Several publicly available libraries provide context-sensitive content encoding, and effectively mitigate XSS attacks. Popular examples for Java applications are the OWASP Java Encoder Project [140], which offers several context-specific sanitization operations, and OWASP's Java XML Templates [139], which offers automatic context-aware encoding. Alternatively, HTML Purifier [263] offers automatic sanitization for PHP applications, and even ensures that the output is standards-compliant HTML.

Alternative strategies to detect and protect against XSS attacks rely on penetration testing (colloquially referred to as *pentesting*) and static analysis [227, 97], or limiting the capabilities of potentially malicious scripts using confined environments such as the Secure ECMAScript library [187] and JavaScript sandboxing approaches [243, 186, 252, 6, 141].

State of Practice Injection vulnerabilities leading to XSS attacks are prevalent in both new and legacy web applications, and are highly ranked in both the OWASP Top Ten [261] and the CWE/SANS Most dangerous programming errors [181]. Cross-site scripting attacks are often only the first step in a more complicated attack, involving underlying infrastructure or higher-privilege accounts. The consequences of escalating an XSS attack are aptly demonstrated by exploitation frameworks, such as the Browser Exploitation Framework *BeEF* [8] or Metasploit [206].

Currently, almost every newly developed web application sanitizes its inputs and outputs, in an attempt to avoid injection vulnerabilities altogether. Almost all programming languages and development frameworks offer library support for sanitization. Unfortunately, sanitization

libraries are not always context-sensitive, and many applications apply sanitization procedures wrongly or inconsistently [223]. Additionally, a few context-sensitive sanitization libraries are available, as discussed above as an aspect of mitigation techniques.

Since injection vulnerabilities remain widespread, several attempts have been made to stop them from within the browser, independent of any application-specific mitigation techniques. Examples of in-browser mitigation techniques are XSS filters [28, 215], or the popular security add-on NoScript [176]. The newly introduced Content Security Policy (CSP) [238] is starting to be adopted by several major sites, including Facebook and Twitter.

Additionally, applications often apply code-based isolation techniques to prevent the damage that can be done by untrusted or injected scripts. Examples of currently available isolation techniques are HTML 5 sandboxes [34], or browser-based sanitization procedures for dynamic script code, such as Internet Explorer's `toStaticHTML` [51].

Research and Standardization Activities

Automatic context-sensitive sanitization of web content is an active research topic, aiming to avoid injection vulnerabilities leading to cross-site scripting attacks. For example, ScriptGard [223] focuses on the detection of incorrect use of sanitization libraries (e.g., context-mismatched sanitization or inconsistent multiple sanitizations), and is capable of detecting and repairing incorrect placement of sanitizers. Other work focuses on achieving correct, context-sensitive sanitization, using a type-qualifier mechanism to be applied on existing web templating frameworks [221].

Another research trajectory is the discovery and detection of potential injection vulnerabilities. Kudzu [222] achieves this using symbolic execution of JavaScript. Gatekeeper [119] on the other hand allows site administrators to express and enforce security and reliability policies for JavaScript programs, and was successfully applied to automatically analyze JavaScript widgets, with very few false positives and no false negatives.

Research on the offensive side looks into the post-XSS world, where the injected payload no longer relies on JavaScript, but uses other powerful client-side features such as advanced HTML features, Cascading Style Sheets (CSS), SVG images and font files [129, 264]. Using these techniques in the right way can enable an attacker to steal a user's password, exfiltrate security tokens (e.g., CSRF tokens embedded in the page), redirect forms, etc.

Even with the most advanced mitigation techniques, both newly created and legacy applications remain vulnerable to XSS attacks. As a second line of defense, Mozilla proposed Content Security Policy (CSP) [237]. CSP allows a developer or administrator to strictly define the sources of trusted content, such as scripts, stylesheets, images, etc., preventing the inclusion of malicious scripts from untrusted sources. Additionally, CSP prevents the execution of harmful inline content by default. When deploying CSP, a reporting-only mode is available. This mode will report any violations of the policy to the developer, without actually blocking any content. This allows to dry-run a policy before actually deploying it towards users.

CSP's restrictions on dangerous inline content effectively renders injected script code harmless, since it will not be executed. Additionally, the list of trusted sources further limits an attacker when including a remote script file. Currently, CSP is being adopted by the major browsers, and is on the standardization track of W3C [238]. One downside of CSP is its impact on an application's code, since the application is no longer allowed to use inline code, or dangerous features such as `eval`. For newly developed applications, this is manageable, but legacy applications require some effort to be made compatible [257].

Another line of research focuses on limiting the capabilities available to a specific JavaScript context, limiting the potential damage of an injection attack. Techniques allowing the enforcement of fine-grained behavior control are the use of security wrappers in JavaScript [175, 200], policy-controlled client-side sandboxes [6, 141, 188, 243] and the use enhanced browser support [186, 252].

Best Practices

The best defense against XSS attacks is to apply proper input and output sanitization. Sanitization is (or, at least, can be) context-sensitive, so one should use either sanitization libraries that automatically determine the correct context, or use the appropriate sanitization function for the output context at hand.

When filtering input or output, use a whitelist approach, where only the valid patterns are whitelisted, instead of a blacklist approach, which has to be an exhaustive list of prohibited patterns. Additionally, avoid writing custom sanitization libraries, which is an error-prone process, especially due to the different encoding options, browser quirks and obscure web features [265].

If possible, use a tight Content Security Policy on your site, fully preventing dangerous inline content and strictly limited the sources of external content. Even when not directly deploying CSP, it might be useful to adapt your code to support CSP in a subsequent incarnation (e.g., do not use inline scripts, `eval` etc.).

Intermezzo 7: The Content Security Policy (CSP)

Cross-site Scripting (XSS) is one of the most prevalent security problems of the Web. It is listed at the second place in the OWASP Top Ten list of the most critical web application security vulnerabilities. Even though the basic problem has been known since at least 2000, XSS still occurs frequently, even on high-profile web sites and mature applications. The primary defense against XSS is secure coding on the server-side through careful and context-aware sanitization of attacker provided data. However, the apparent difficulties in mastering the problem on the server-side have led to investigations of client-side mitigation techniques.

Content Security Policy (CSP) A very promising approach in this area is the Content Security Policy (CSP) mechanism [238] , currently under active development and already implemented by the Chrome and Firefox web browsers. A web application can set a policy that specifies the characteristics of JavaScript code which is allowed to be executed. CSP policies are added to a web document through an HTTP header or a meta-tag. More specifically, a CSP policy can:

1. Disallow the mixing of HTML mark-up and JavaScript syntax in a single document (i.e., forbidding inline JavaScript, such as event handlers in element attributes).

2. Prevent the runtime transformation of string-data into executable JavaScript via functions such as eval().

3. Provide a list of web hosts, from which script code can be retrieved.

If used in combination, these three capabilities lead to an effective thwarting of the vast majority of XSS attacks: The forbidding of inline scripts renders direct injection of script code into HTML documents impossible. Furthermore, the prevention of interpreting string data as code removes the danger of DOM-based XSS. And, finally, only allowing code from whitelisted hosts to run reduces the adversary's capabilities to load custom attack code from external web locations. Compromising the code on a whitelisted host remains a potential attack vector, even with a tight CSP policy in place.

In summary, strict CSP policies enforce a simple yet highly effective protection approach: Clean separation of HTML-markup and JavaScript code in connection with forbidding string-to-code transformations via eval().

Mitigating Information Exfiltration Besides directly stopping the execution of injected JavaScript, CSP also provides the means to mitigate the effects of malicious

JavaScript, that is executed even in the presence of a CSP. To do, a CSP can used to prevent HTTP communication with untrusted web hosts. In a CSP, for all network-aware HTML-tags and JavaScript-APIs a whitelist of trusted hosts can be specified. HTTP requests to hosts that are not contained on the list are stopped by the browser.

As long as this list is well maintained and does not contain wildcards, this effectively thwarts the adversary's ability to leak sensitive user information, that his JavaScript might have obtained, back to his server.

Policy Example An exemplary site's policy could be as follows:

```
1 Content-Security-Policy: default-src 'self';
2     img-src images.example.com, *.flickr.com;
3     object-src media.example.com;
4     script-src trusted.example.com;
```

Listing 11.1: A sample CSP policy, limiting content inclusion to the web application's own domain, except for the specified exceptions for images, objects and scripts.

Outlook The future of CSP appears to be promising. The mechanism has been implemented in major web browsers, with recent versions of Firefox (since version 4.0) and Chrome (since version 13) already supporting it. Furthermore, CSP is currently an active standardization activity in the W3C [238].

However, using CSP comes with a price: Many common practices in current JavaScript usage, especially with respect to inline scripts and the use of `eval()`, will have to be altered. Making an existing site CSP compliant requires significant changes in the codebase, such as getting rid of inline JavaScript, such as event handlers in HTML attributes, and string-to-code transformations, which are provided by `eval()` and similar functions.

11.2 Compromising JavaScript Inclusions

Compromising an included JavaScript file allows an attacker to execute attacker-controlled code within the execution context of the application, giving him the same level of privilege as the application code itself. The capabilities gained by such an attack are access to all local application-specific data, resources and APIs, as well as the ability to manipulate and create application transactions by sending seemingly-legitimate requests to the server-side application.

Problem Description

The goal of compromising the inclusion of a piece of JavaScript is to gain control over the client-side context. Since included JavaScript code typically runs within the security context of the including page, it suffices for the attacker to compromise any of the included scripts. One example that achieves this goal is to compromise a popular JavaScript library, hosted on a third-party server and included by many web applications.

The attack tree in Figure 11.4 shows several options that will lead to compromised included scripts; these can come from third-party providers, from the application's own server or from a local storage facility. The attacker either compromises the locally or remotely stored script file, or can attempt to manipulate the network traffic when the browser makes a request for a remote script. The former typically requires the compromise of a remote host or the local storage facility, while the latter requires networking capabilities.

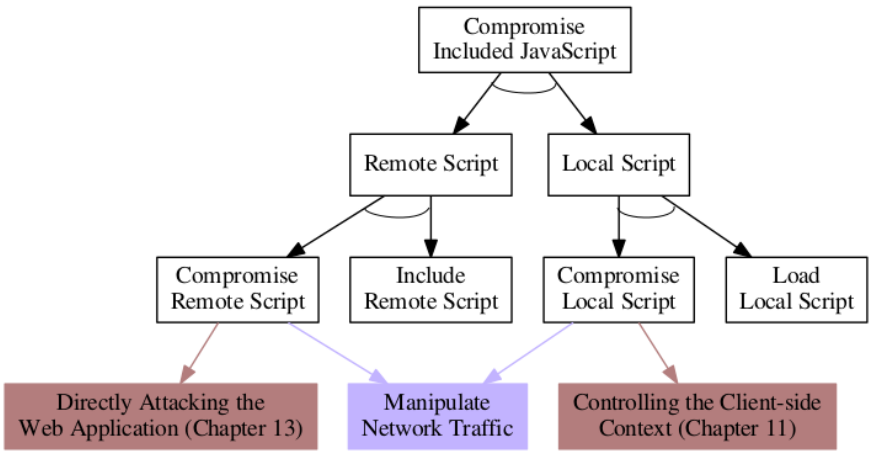


Figure 11.4: Compromising an included JavaScript, either locally or remotely, gives an attacker full control over the client-side application's security context.

Technically, compromising script files located on a remote host requires control over the remote *server machine* or *server-side content storage*, a topic covered in Chapter 13. Script files loaded over an insecure connection can be manipulated by an attacker who sits on the same network, or controls the entire network. Such situations are legion, with publicly accessible hotspots and freely available wifi networks as the prime example. Manipulation of script files on the network is particularly dangerous for secured web applications, deployed over HTTPS, since this violates the security guarantees offered by the secure deployment. A final attack approach attempts to compromise locally stored scripts, which are cached by the application using the browser's storage facilities, such as the WebStorage API [136]. The compromise of such a script allows an attacker to inject custom code, make an XSS persistent on the client-side, or to extend an XSS vulnerability towards multiple users in a shared browser environment. In essence, the inclusion of compromised JavaScript is a problem of code integrity, where the browser lacks the ability to verify that the included code corresponds to the expected code and delivers precisely the required functionality without malicious additions.

Mitigation Techniques

The key to mitigating the compromise of script files stored on the server lies in protecting the server and its application against potential adversaries, as covered in Chapter 13. For applications that require a high degree of control over these script files, it can be useful to copy the third-party script files to their own environment, where they can be optimally protected. Naturally, the copy is only a snapshot of the third-party code, and should be kept up to date with new versions as they are released.

Mitigating the manipulation of script files on the network can be achieved by deploying the application over HTTPS, and only including remote files received over secure connections. *Mixed content* web pages, where a secure page includes content over insecure HTTP connections, should be avoided at all times, since they would allow total compromise of the secure application context.

State of Practice The line of research focusing on the exploration of current practices offers valuable information on the state of practice. A study of the JavaScript inclusion behavior of the top 10,000 Alexa sites [192] reveals that 88.45% of these sites include at least one remote JavaScript library, and some sites trust as many as 295 remote hosts. The study also attempts to characterize the security of both including web applications and the third-party script providers, showing that about 12% of web applications with a high security classification include content from at least one provider with a low security classification.

Another study of the client-side caching of script code on the top 500,000 Alexa sites [171] shows that 386 web applications use local storage facilities to cache JavaScript, HTML and CSS code, as well as 68 entries of remote URIs, used to fetch resources during execution. Additionally, a mitigation technique for attacks on locally cached code is proposed, based on cryptographic checksums of the stored code.

Unfortunately, many sites use *mixed content*, violating the security guarantees of HTTPS pages. A recent study shows that 26% of the TLS-protected Alexa Top 100,000 Web sites included JavaScript over HTTP [55]. A positive evolution is that browsers are all deciding to effectively block the loading of insecure content on secure pages. This feature has been introduced in Internet Explorer 9, Firefox 23 and Chrome 14.

Research and Standardization Activities

One line of research focuses on the exploration of current practices, thereby discovering potential problem areas and security issues (See *State of Practice* above).

Another line of research focuses on controlled execution of third-party JavaScript, giving the integrating page control over the set of features available to external scripts. Common approaches towards fine-grained behavior control are the use of security wrappers in JavaScript [175, 200], policy-controlled client-side sandboxes [6, 141, 188, 243] and the use enhanced browser support [186, 252].

Best Practices

Best practices to prevent the compromise of included scripts focus on limiting the number of trusted third-party hosts, as well as securing the remote host containing the scripts, potentially copying third-party scripts to a controlled server. Additionally, deploying your application over HTTPS and ensuring that scripts are only included over HTTPS connections can thwart a network adversary.

When storing any code fragments at the client-side, the application needs to ensure its integrity before loading it into its execution environment. This can be achieved by using checksums, which are stored and computed independently from the potentially untrusted code.

In the near future, we expect sandboxing technologies, such as Google Caja [188], Secure ECMAScript [187], JSand [6], etc., to improve in performance and become more developer-friendly, making it a viable candidate for isolating potentially untrusted scripts in a fine-grained, controlled manner.

Chapter 12

Attacking the Client-side Infrastructure

The holy grail of client-side assets is the *client machine*, which, when compromised, gives an attacker full control over a user's activities, as well as the *client-side content storage*, *client-side application code* and *application transactions* assets. Compromise of the client machine can be achieved by *running attacker-control client-side components*, of which drive-by downloads leading to the installation of malware are a common example. By *escaping the client-side sandbox or environment*, an attacker can not only gain unauthorized access to client-side resources, but can also escalate the attack towards other assets in the client infrastructure, such as home routers or network-enabled printers.

Compromise of the *client machine* has far-reaching consequences, since it means that the trusted platform for many applications has been compromised. A common consequence of compromising this asset is the incorporation of the client machine in a botnet, infamous for their involvement in large-scale denial-of-service attacks. Apart from the more anonymous participation in botnets, a compromised client machine also leads to targeted attacks, such as banking trojans that mislead the user when visiting his online banking application, leading to the actual theft of financial means.

This chapter focuses on two important compromises of the client machine, either through malicious browser extensions or through native code. Malicious browser extensions, either by nature or vulnerable legitimate extensions, not only give an attacker full control within the browser, but often allow access to the entire client machine. Another vector is by executing native code on the client machine, which can be achieved by exploiting vulnerabilities in the browser, or in one of the many browser plugins, such as Flash, Java or Acrobat. A third part of this chapter focuses on possibilities to escalate the attack towards local infrastructure, including many of the network-enabled devices that can be found in a modern home.

This chapter does not focus on attacks on the client machine that do not require the cooperation of a web application. Examples are the distribution of malware through malicious email attachments, or the exploitation of OS level flaws over the internet, without using a web application as the attack vector.

12.1 Malicious Browser Extensions

Browser extensions run within the browser, and have significantly more privileges than traditional web code. Attackers who are able to compromise legitimate extensions, or trick users into installing malicious extensions, potentially gain the power to inspect and manipulate all web applications running within the compromised browsers, and might even be able to compromise the host system of the victim.

Problem Description

The goal of controlling a browser extension is to have attacker-controlled, privileged code running within the browser. This potentially gives an attacker access to the browser's internal state, and can be an enabling factor allowing escalation of the attack towards full compromise of the client machine. Since browser extensions run on a higher privilege level, outside of the traditional browser security policies, and are able to inspect and manipulate multiple sites, they are an attractive, high-powered target for attackers.

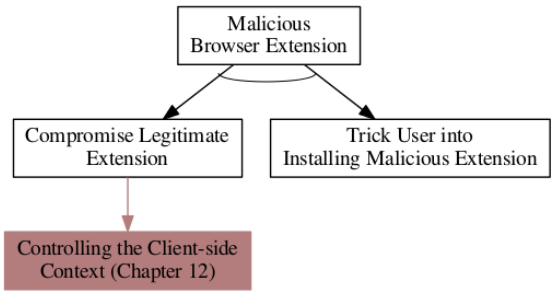


Figure 12.1: An attacker can gain elevated privileges within the browser by compromising a legitimate extension, or tricking the user into installing a malicious extension.

As illustrated in the attack tree in Figure 12.1, an attacker can either attempt to compromise a legitimate, already-installed extension, or try to trick the user into installing a malicious extension. The former attack technique shows several similarities to running attacker-controlled code in the client-side context, where an attacker can attempt to insert code into the extension, or compromise resources on the network. Tricking the user into installing a malicious extension is a form of social engineering, which can be made very convincing.

Technically, inserting code into the extension becomes possible when an extension processes untrusted input, for example when inspecting a page loaded in the browser. Handling such input carelessly results in a script injection attack vector, allowing the attacker to execute arbitrary code within the extension's context.

Tricking the user into installing a malicious extension can be done in various ways. The simplest way is to simply provide the extension on a web site, hoping to trick the user into installing it manually. Another approach is to offer the extension through the official download channels, such as the browser vendor's extension store. Finally, a powerful attacker can spoof the entire extension store, allowing him to have a malicious extension masquerade as a legitimate, popular extension.

The essence of compromising an extension is privileged extension code that processes untrusted page contents, potentially including unprivileged code, which can lead to an injection attack. The core problem of malicious extensions is social engineering, where users can be tricked into installing an extension from potentially untrusted sources.

Mitigation Techniques

The techniques for mitigating compromised extensions or preventing the installation of malicious extensions are rather limited, and are part of the browser's architecture and the extension store's platform. Generally speaking, the compromise of a legitimate extension can be addressed by a thorough extension architecture, where code is strictly separated and APIs are restricted by permissions. Preventing the installation of malicious extensions is not trivial, and can be

addressed by preventing installations through unofficial channels, and performing code reviews on the extensions published through the official channel. Each of these techniques is covered in more detail in the *State of Practice*.

State of Practice The state of practice in protecting an extension against compromise, or protecting the users from installing malicious extensions, is defined by the currently available browsers. We cover two modern browsers, Mozilla Firefox and Google Chrome, both of which have extensive support for extensions, and offer a large number of extensions through an official channel.

Firefox's architecture supports privileged extensions, which have access to a large set of browser-provided APIs, offering numerous services, as well as access to the browser's internals and operating system resources, such as reading/writing files, spawning new processes, etc. In Firefox, extensions can define core components, which can be exposed through an API, as well as scripts that interact directly with web content. In the recently introduced JetPack model, extensions can also choose to adopt a more modular model, offering some isolation and restrictions. Concretely, extensions in Firefox are subject to very few limitations, and can easily share their functionality among core components and scripts interacting with web content.

Chrome's architecture is based around the principles of least privilege, isolated worlds and permissions. Extensions have a core component, which runs separately from content scripts, which interact with actual web content. Communication between both contexts is available through the Web Messaging API [131]. Additionally, extensions all have a distinct namespace, and are isolated from each other. The browser APIs offered by Chrome are more limited than the Firefox APIs, especially for reaching out of the browser sandbox, into the OS. Furthermore, Chrome extensions have to explicitly request a set of permissions for a determined set of web sites (wildcards are allowed) upon installation. Without the necessary permissions, several APIs become inaccessible, preventing an extension from escalating its power within the browser.

Mozilla's extension platform, called *Mozilla Add-Ons*, is the official channel to offer extensions to users. Published extensions are guaranteed to have undergone a review by an editor, who is tasked with checking the functionality and behavior of the extension. Firefox also supports the installation of *unofficial* extensions from arbitrary sites, but not without the explicit approval of the user.

Chrome's extension platform, called the *Chrome Web Store*, is Chrome's official channel for distributing extensions. Chrome does not perform any reviews, making the Web Store a reputation-based system, where users are expected to file abuse reports in case of a misbehaving extension. Chrome does not support *unofficial* extensions, except from local folders in developer mode.

Finally, Chrome also disables extensions by default in private browsing mode, called *incognito mode*, since they might be a risk for a user's privacy. They can however be explicitly enabled in private browsing mode if desired.

Research and Standardization Activities

Thorough research of the Firefox extension system, as well as the gap between required and granted permissions of Firefox extensions, has lead to the proposal of an extensions system based on the principles of least-privilege, isolated worlds and permission systems [24], a model that has been adopted as the Google Chrome extension system. Follow-up research [48] investigates the actual Google Chrome extension security architecture in detail, concluding that even with these restrictions in place, many extensions can be compromised by an attacker. As a result, additional defenses have been proposed and deployed, such as the default enforcement of Content Security Policy [238] on Chrome extensions.

Another line of research uses formal systems to verify security properties, often finding and fixing security vulnerabilities in the process. Most research focuses on existing extension systems and extensions, using for example information flow analysis to determine whether extensions

suffer from privilege escalation [17, 76], or employing type systems to check whether extensions violate the properties of private browsing mode [174]. Other research concludes that the current systems grant too many privileges to an extension, and therefore propose a new extension security model, underpinned by a verification methodology to check an extension's safety [120]. The feasibility of this new extension model has been demonstrated by implementing extensions for popular browsers, including Firefox, Chrome and Internet Explorer.

Best Practices

A best practice for any web user is to limit the number of extensions to the minimum, and uninstall or disable those that are not or infrequently needed. Additionally, when using a form of private browsing mode, it can be useful to disable extensions, since they can potentially compromise the private nature of the browsing mode [174]. In a corporate environment, it makes sense to prevent the installation of extensions altogether.

12.2 Drive-By Download

Rendering a simple web page involves a lot of client-side components, all of which can contain subtle vulnerabilities. An attacker looking to exploit such a vulnerability, can carefully craft specific web content to exploit this vulnerability. By putting the exploit software online, and tricking the user into visiting it, the attacker can gain ground on the client machine, allowing him to install malware and further compromise the machine.

Problem Description

In a drive-by download attack, also known as a drive-by exploit attack, a user's computer becomes infected with malware, delivered through the Web platform. By exploiting a vulnerability at the client side, for example in the browser, a rendering engine or a browser plugin, an attacker can install malware on the user's machine, giving him full control over the client machine and all the user's actions. When exploiting a specific vulnerability, a drive-by download attack can be executed in a way that is completely undetectable to the user. Alternatively, an attacker can attempt to trick the user into installing malicious software, for example by having it masquerade as an antivirus package.

The attack tree in Figure 12.2 shows the different steps involved in a drive-by download attack. First, the attacker puts the code that will trigger the process on a web server, either a compromised server or his own server. When this code is loaded in the victim's browser, it will contact a redirection service that will guide the user towards an appropriate exploit server, depending on the detected operating system, browser version and available plugins. If the exploit server succeeds into exploiting the targeted vulnerability, the actual malware will be downloaded from a malware server, and installed on the user's machine. Once installed, the malware can typically be controlled using a *command and control* system.

Technically, exploiting a vulnerability at the client side can be as simple as serving a specially crafted image, aiming to abuse a vulnerability in the rendering engine [52], or exploiting a vulnerability in a plugin [251]. Another exploit tactic is a *heap spraying attack*, where malicious code is injected into memory, and an application is tricked into executing the malicious code [109].

In essence, drive-by download attacks are a distributed variant of traditional native code attacks, such as buffer overflow attacks. The exploited code is responsible for processing web content, which comes from various sources, making it easy to insert malicious content somewhere along the way, thereby exploiting the vulnerable client-side code.

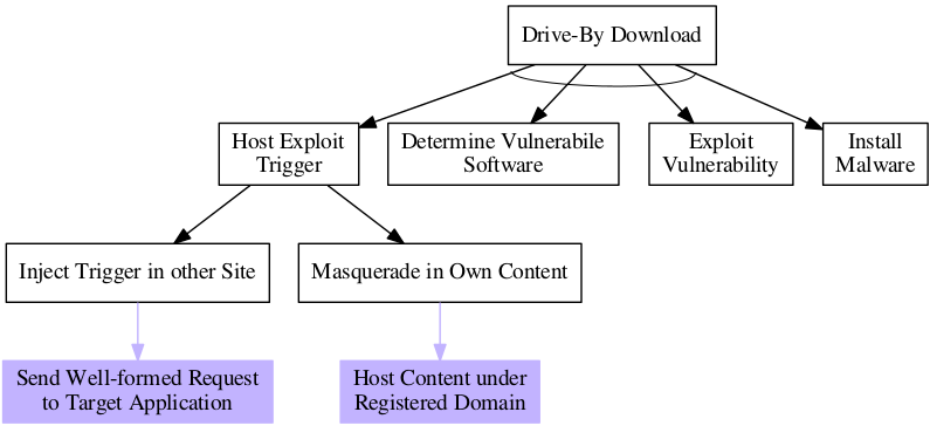


Figure 12.2: Exploiting a native-code vulnerability in client software responsible for processing web content, allows an attacker to install malware on the user's machine.

Mitigation Techniques

One technical mitigation technique is to isolate the plugin execution environment in a process-level sandbox, making it significantly harder to gain full system-level access with a successful exploit against a plugin. This mitigation technique has currently been deployed by all major browsers to sandbox the Flash player, which has a rocky security history on the Web.

A non-technical mitigation technique is to frequently update the installed plugins, in order to benefit from security updates. A significant improvement [82] in this area are the automatic update processes, either deployed by the plugins themselves, or by integrating them into the browser. An example of the former is the Java plugin, which installs its own auto-update mechanism, and an example of the latter is Google Chrome in combination with the Flash plugin, which are bundled and automatically updated when necessary. Automatic updates help mitigate newly developed exploits, often based on the latest security update [226].

Finally, traditional mitigation techniques against native code attacks, such as address space layout randomization (ASLR) or data execution prevention (DEP), also help preventing drive-by downloads.

State of Practice Drive-by download attacks are still on the rise, and are considered an important threat to user's security on the Web. An emerging trend are the shift to single URIs that distribute the malicious software, instead of using an entire underlying botnet infrastructure [90]. This shift in distribution mechanism makes lawful takedowns more difficult, as URIs are not that easily blocked, and quickly changed after takedown.

Research and Standardization Activities

The detection and analysis of web-based malware is an expanding research field, covering the detection and prevention of drive-by download attacks, heap spraying attacks, or the underlying economic models of the malware industry. One approach focuses on static detection of malware [60, 166], while others rely on feature extraction and classification [59, 211]. Other research focuses on supportive tasks for detection, such as de-cloaking malware [158] or automated collection and replay of malware scenarios [54].

A portion of the research on malware focuses on the specific problem of heap spraying, for example by introducing a new way to execute a heap spraying attack with HTML5 [191]. Runtime monitoring infrastructures are able to detect heap spraying attacks [207], and a modification of the JavaScript engine can completely prevent heap spraying attacks [109].

A final category of research focuses on the underlying economic models of the malware industry. The underground malware economy has evolved quickly, with *pay-per-install* services being offered as a commodity. An extensive study [47] investigates the different families of malware, repacking strategies to avoid detection and the targeting of specific countries. Another study examining the underground economy of fake antivirus software [241] reveals that three large-scale businesses earned a combined revenue of $130 million. Fake antivirus businesses actively monitor refund requests of their customers, attempting to manipulate the system and evade detection.

Best Practices

The best practice for web users is to reduce the number of plugins installed to its absolute minimum, and keep all client software up to date, including operating system, drivers, browsers, plugins, etc. Additionally, the use of browser's *click-to-play* features can reduce the attack surface significantly. In corporate environments, the software on client machines should be controlled, and be kept up to date as much as possible.

Web developers should ensure that their applications are well-protected, especially against injection attacks, preventing the leverage of their web site as a malware distribution platform. Third-party libraries and their providers should be selected carefully, as a compromise of a library provider can also lead to a compromise of all depending web applications.

12.3 Attacking the Local Infrastructure

Assets within the local infrastructure, such as intranet applications, are typically inaccessible from the Web, placing them out of reach for a potential attacker. However, by compromising the client-side context within a user's browser, an attacker might be able to launch a web attack on such an intranet application, since the user's browser is able to connect to it. One example is an attacker carrying out a CSRF attack on an intranet application, essentially making a request in the user's name.

The impact of these scenarios may seem far fetched, since intranet applications traditionally only occur in large network environments, such as governmental, corporate or academic networks, but were very uncommon for the home user. Nowadays, with numerous network-enabled devices, which typically have an administration interface, every home has several intranet web applications available on its network. Examples are router administration interfaces, printers, media centers, and even home automation systems controlling lights, shutters and heating.

Problem Description

The goal of escalating an attack aiming at the local infrastructure is to compromise assets, otherwise inaccessible from the Web, such as intranet web sites, network-enabled printers, etc. Essentially, by attacking the local infrastructure directly from within the client-side context of a traditional web application, an attacker is able to circumvent any perimeter security barriers, such as firewalls or intrusion prevention systems.

Attacks on the local infrastructure from within the client-side context of a web application are typically an instantiation of traditional web capabilities or web attacks, focused on a network-accessible interface of the local asset. Examples are XSS or CSRF attacks towards device's management interfaces, JavaScript-based port scanning attacks [126], etc.

Technically, there is little difference between an attack geared towards a publicly available application and an attack targeting an intranet web application or interface. The attacker ensures that the payload of the attack is executed in the browser of the targeted user, which results in for example requests being sent to the intranet web application under attack. The main difference is the restricted accessibility of the target application, which uses for example a host and domain name that only resolves on the intranet, or an IP address in the range of private networks. In neither cases is there a limitation for the client machine's browser to initiate a connection.

In essence, attacking an intranet application from the client-side context of a public web application is not really different from attacking another public web application. The power of traditional web attacks is however exacerbated by the fact that intranet applications typically assume a safer execution context, since they are believed to be inaccessible to unauthorized users.

Mitigation Techniques

While this problem has been well known since 2006, few specific mitigation techniques preventing escalation from a public client-side context towards a non-public web application exist. Suggestions for preventing client-side contexts of a public web application initiating requests to IP addresses in the private range have been made, but the idea found no traction [118].

Naturally, intranet web applications can protect themselves against specific web attacks by applying the mitigation techniques described in other places of this document.

State of Practice In practice, numerous devices are vulnerable to common web attacks, allowing an attacker to take control or even damage intranet devices. One potentially harmful attack is called *pharming*, where an attacker takes control of the router's DNS settings, rerouting traffic at his discretion [117]. Other sources disclose cross-site scripting vulnerabilities in common devices, such as routers, photo frames, IP cameras, either in the web interfaces [40] or through log files [115]. Finally, a study of multi-function printers yields a concerning list of vulnerable devices [12].

Local infrastructure can also be attacked directly, since many misconfigured devices on local networks are directly accessible from the internet. The infamous search engine Shodan [231] specializes in finding such devices, including traffic lights, heating systems, security cameras, NAS devices, etc. Some of these publicly accessible systems are highly critical, and even display a banner about life-threatening situations upon connecting [113].

Research and Standardization Activities

An active research topic of the past few years is the security of embedded web applications [40, 115], such as routers, printers, etc. Unfortunately, these studies have to conclude that consumer-grade devices are highly vulnerable to a myriad of web attacks, such as cross-site scripting and cross-site request forgery, bypassing authentication, etc.

One of the proposed solutions, *SiteFirewall*, proposes to restrict a web application to accessing external resources, unless whitelisted [40]. Applying such a policy to the web application of an embedded device prevents a successful attack from loading additional resources, or leaking any of the obtained information, such as authentication credentials, session cookies, etc.

Another proposal, *WebDroid*, is an open-source web framework aimed at designing secure embedded web interfaces [115]. *WebDroid* follows the secure-by-default principle, and enables additional security mechanisms, such as Content Security Policy [238] and framebusting [217] by default.

Best Practices

No mitigation techniques attempt to prevent the escalation of an attack towards intranet web applications, which essentially means that intranet resources are responsible for their own security features. Since the attacks geared towards intranet applications are essentially web attacks, it suffices to carefully apply the mitigation techniques and best practices for protecting against these attacks.

As a consumer, consciously choosing vendors with a respected reputation in the field of security might be a good shot at preventing these attacks.

Chapter 13

Directly Attacking the Web Application

By directly attacking the web application, the attacker aims to compromise the *server machine* or *server-side content storage* assets, which are closely tied together, and are the core assets for running a web application. Compromise of either one of these assets not only severely compromises the integrity of the server-side web application, but also enables the escalation of the attack towards other assets. For example, a compromise of the *server machine* or *server-side content storage* can easily lead to the serving of malicious client-side code, essentially compromising the *client-side application code* asset and associated assets, such as *application transactions* and *authenticated sessions*.

The server-side content storage suffers from two threats, either an attacker *directly accessing the server-side storage*, or an attacker attempting to subvert legitimate application access, by *abuse server-side application privileges*. Compromising the server machine can be done by executing *an attacker-controlled server-side component*, either by uploading malicious code, or by compromising an external component or library. Other threats to the server machine are *escaping the sandbox or environment*, which essentially allows an attacker to execute commands or extract files from the machine, outside of the web application's boundaries.

In this chapter, we go into detail on server-side injection attacks, which is a class of attacks that re-occurs on many levels, allowing an attacker to abuse the application's privileges. Examples of injection attacks are *SQL injection* or *command injection*, respectively allowing an attacker to tamper with the server-side database storage or allowing an attacker to execute arbitrary commands. The second part of this chapter focuses on access control within the web application, where vulnerabilities give attackers unauthorized access to sensitive data or powerful operations, potentially leading to administrator privileges within a web application.

This chapter focuses on complex attacks that interact with a web application, and therefore does not go into detail on more straightforward attacks, that occur in isolation from the web application's operation. Examples are establishing a direct connection to a database server, and brute-forcing the authentication credentials, or tricking the developer into directly integrating a malicious library or component.

13.1 Server-side Injection Attacks

Web applications frequently integrate additional technologies and execution environments to enhance their functionality. SQL Databases and system level command execution are two important representatives of such technologies. Often these technologies offer a string-based API to the web application. So, for example, in order to store or retrieve data from a database, a web application constructs an SQL query and sends it to the database in the form of a string.

Often these strings are made of hardcoded components and user input. If there is no validation, filter or encoding function in place to check the user input for certain control characters, an attacker is able to change the semantics of the string-encoded commands/statements by sending a specially crafted input to the application.

Problem Description

The goal of a server-side injection attack is to divert the server-side control flow in such a way that attacker-provided commands are executed in the context of the server-side application (See Figure 13.1). The root cause of this problem is that applications often process user-controllable inputs in such a way that certain control characters are not escaped properly. This fact, allows an attacker to change the semantics of the server-side code. Thereby, many flavors of such injection attacks exist depending on the context of the injection. Examples for well-known injection attacks are SQL injection, Command injection and File inclusion. Furthermore, there are many other flavors of injection attacks, such as LDAP injection, XPATH injection, SSI injection, "No SQL" injection, etc.

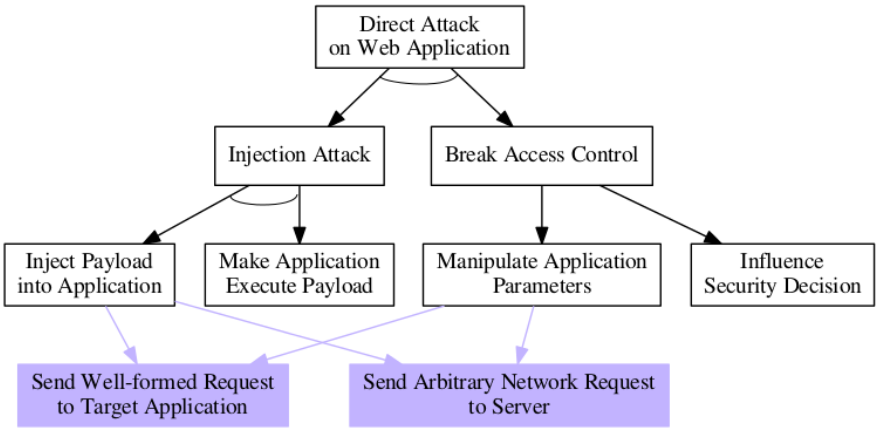


Figure 13.1: Direct attacks on the web application require an attacker to send carefully crafted requests to the application, which do not necessarily have to be standards-compliant. Two common examples of direct attacks are *injection attacks* and *access control attacks*.

According to the OWASP Top 10, injection flaws are the most serious vulnerabilities on the Web. The consequences of a successful injection attack range from information disclosure to the complete take over of a vulnerable host. In order to showcase the serious nature of such vulnerabilities, we give three examples:

SQL Injection: An SQL injection vulnerability is present whenever SQL statements are constructed from strings, whereas user-provided strings are not encoded properly. This allows an adversary to change the semantics of a given statement by altering the user-input. Listing 13.1 demonstrates such a vulnerability. In this example, the user input ($\$\_GET['article\_id']$) is not encoded/validated at all. Hence, instead of presenting a numerical identifier for the id field, an attacker could enter the following malicious code to change the semantics of the statement: *"1 UNION SELECT username, password FROM user"*. Which will lead to the following statement being send and executed by the database management system: *SELECT * FROM article WHERE id = 1 UNION SELECT username, password FROM user*. So instead

of only returning an article from the article table this statement also returns all usernames and passwords stored in the user table. Depending on the inner workings of the web application, this data might then be passed back to the attacker.

```
<?php
  [...]

  $query = "SELECT * FROM article WHERE id = ".$_GET['article_id'];
  mysql_query($query);

  [...]
?>
```

Listing 13.1: An example of SQL Injection vulnerability

File Inclusion: File inclusion is another form of injection attack. Here, a web application includes and/or executes a certain file depending on user-provided input. By altering the input an attacker is able to force the application to include a file that contains sensitive information or to execute a file that contains malicious code.

Listing 13.2 shows a code snippet with a typical file inclusion vulnerability. In this example, the web application includes different pages depending on the information provided in the GET parameter named 'page'. Although, this kind of functionality offers a flexible way to extend a web application, it is problematic as it misses carrying out proper validation of the input. So, instead of requesting the inclusion of a legitimate file called 'guestbook.php', the attacker could request that the file '../../../../../etc/passwd' should be included. As a consequence, the web server would include the contents of this file ('/etc/passwd') into the HTTP response and send it back to the attacker.

```
<?php
   if (isset( $_GET['page'] ) ){
      include( './pages/'.$_GET['page']);
   }
?>
```

Listing 13.2: File inclusion vulnerability

In general, there are two forms of file inclusion - Local File Inclusion (LFI) and Remote File Inclusion (RFI). While LFI enables an attacker to include and/or execute files residing on the server hosting the vulnerable script, RFI allows an attacker to include and execute files on any host reachable from the vulnerable host.

Command Injection Similar to SQL injection, the goal of a command injection attack is to execute arbitrary commands. However, in this attack the adversary does not aim at executing SQL commands, but system level commands. Listing 13.3 shows a typical (and simplified) command injection vulnerability. The web application receives some user input (via $\$\_GET['page']$) and displays the content of the corresponding file via the "cat" shell command. However, this web application again misses to properly handle the user input. Hence, an attacker is able to inject additional commands into the string that is executed by the system function. So for example, by injecting the following value into the "page" GET parameter, the attacker is able to get a directory listing of the current directory: *page.html; ls* (by injecting this string, the following

command is executed: *cat page.html; ls*). In this way, the attacker is able to execute arbitrary commands in the name of the user that started the web server.

```
1 <?php
2    system('cat ./pages/'.$_GET['page']);
3 ?>
```

Listing 13.3: Command injection vulnerability

Mitigation Techniques

The root cause of almost every injection vulnerability is the missing separation of code fragments and user-provided data. By using strings for data transmission between different technologies, the information of the origin of a certain character is lost. Hence, when executing string-encoded commands/statements, it is not possible to tell whether a sequence of characters was coming from the user (potentially malicious) or from the application (benign). In order to mitigate injection flaws, most of the techniques, thus, focus on enforcing such a separation. They do so by either filtering, validating or encoding user input or by replacing the insecure string APIs with functionality that preserves a clean separation of data and code. Furthermore, various techniques exist that can be used to limit the consequences of a successful attack.

Validation Validation refers to the process of validating the legitimate nature of user input, before processing it any further. There are many different ways in which such a validation can be conducted. For example, an application should ensure that user input is strongly typed (use integers for numerical identifiers, etc), resides within certain length and value boundaries and contains only legitimate characters (no control characters, etc.). If a value does not pass the validation routines it must not be processed any further by the application, and error messages must be generated carefully, in order to prevent injection attacks in the contents of the error message. By doing so, attackers cannot infer with the semantics of a given command and thus are not able to inject code tokens into the application.

Filtering Filtering refers to the process of removing forbidden/malicious characters or character sequences from a given input string. So, for example, certain control characters or commands can be removed from a given string in such a way that the attacker is not able to alter the semantics of a string-encoded statement/command. Although filtering can defuse malicious payload in certain situations, it is a tedious process, often even introducing new injection vulnerabilities. Often filters can be bypassed when either new, previously unknown features/keywords are being added or when developers overlook certain edge-cases. Simply rejecting non-compliant input is more safe than trying to fix non-compliant content.

Encoding Encoding in this context refers to the process of transforming the user input character by character into a special format that is then further processed by the application. By transforming all control characters into another encoding scheme the resulting encoded string cannot infer with the semantics of the statement/command. In most of the web application programming frameworks standard-encoding functions can be found. However, encoding always has to be used in a context sensitive fashion. Sometimes, it is difficult to determine the exact context, as each injection could refer to the multiple contexts at once.

Clean separation of data and code As described earlier, the root cause of injection attacks is the missing separation of data and code. With string-based APIs it is possible but very difficult to ensure such a clean separation (see above). However, enforcing such a separation is essential

to mitigate injection attacks. One way of doing better is to replace string-based APIs with more expressive APIs that do preserve this separation across technology boundaries. One example for such an alternative API are Prepared Statements to counter SQL injection attacks.

Instead of mixing up user input and code tokens before communicating with the database, Prepared Statements split up this process into two steps. In the first step, the developer defines the syntax of a statement and communicates this information to the database. In a second step, the user input is transmitted to the database. Due to the information transmitted in step one, the database will not interpret the user input as code tokens. Hence, no matter how the user input is composed, the database will always only execute the legitimate statement.

Principle of Least Privilege Besides the mitigation techniques described, there are a lot of other measures that are able to severely limit the consequences of a successful attack. One important fact here is that commands are always injected through a legitimate application. This means that the injected commands are executed in the name of this vulnerable application. Hence, when exploiting a server-side injection vulnerability, the attacker is always as mighty as the application he is running the attack through. Therefore, it is a good practice to apply the principle of least privilege. So, an application should never be executed with root privileges, but only with the privileges needed to function correctly.

State of Practice In modern web applications, the most prevalent server-side injection attack remains SQL injection, even though adequate mitigation techniques exist. At the time of this writing, the Web Hacking Incident Database [230] contains 1313 reports of SQL injection attacks. Additionally, automated tools such as *sqlmap* [66] and *havij* [143] have specialized in the exploitation of SQL injection attacks. Other injection attacks are also relevant, as indicated by its first place in the OWASP top ten [261], but concrete numbers on the prevalence are not available.

Research and Standardization Activities

The main area of research on server-side injection attacks focuses on SQL injection, with a large body of work investigating attacks and defenses. A comprehensive classification of attacks and countermeasures [123] not only covers relevant research in this area, but also compares the different techniques and countermeasures to each other. The classification identifies several trends in the results, such as many techniques addressing problems resulting from attacks that hide using alternate encodings, and that the preventive techniques seem to offer better protection, presumably due to their close integration with good coding practices.

Best Practices

As injection vulnerabilities are context- and application-specific, it is difficult to give a general advice on the usage of a certain technique or technology for mitigation purposes. Nevertheless, in general, a developer should avoid the uncontrolled usage of user inputs within string-based interfaces. In order to explain best practices in more detail, we will again focus on the example injection vulnerabilities presented above:

SQL injection The best practice to avoid SQL injection attacks is to use Prepared Statements. Listing 13.4 shows an example usage of this mitigation technique. As described in the previous sections, Prepared Statements enforce a clean separation of data and code. In a first step the query syntax is defined via the $\$dbh- > prepare()$ function. After that, the user input is handed over to the database and the query is executed. In this fashion, the attacker is not able to influence the query's syntax via the user input. Hence, the attack is successfully mitigated. One important point here, is that user input should never be used in the first step (when

constructing the query syntax). Otherwise, this could re-enable the attacker to conduct an SQL injection attack.

```php
<?php
  // step 1:
  $stmt = $dbh->prepare("SELECT * FROM article WHERE id = ?");

  // step 2:
  $stmt->bindParam(1, $_GET["article_id"]);

  // execution
  $stmt->execute();
?>
```

Listing 13.4: Prepared Statement

File inclusion The best practice for file inclusion is to only include white-listed resources into an application. Listing 13.5 shows such a solution. In this example, the user input is validated against a whitelist. Only if a legitimate entry exists within the whitelist, the application includes the corresponding file. In this way, the attacker is by no means able to force the application to include arbitrary local or remote files.

```php
<?php
  $whitelist = array("index" =>"index.php",
            "contact" => "contact.php");

  if array_key_exists($_GET["page"], $whitelist) {
    include($whitelist[$_GET["page"]]);
  }
?>
```

Listing 13.5: Secure file inclusion

Command Injection In order to avoid Command Injection vulnerabilities, developers should avoid the usage of string-based APIs such as the *system()* command. Instead, developers should focus on existing APIs for their language. So, for example, instead of using system level command execution to read the content of a file via the *cat* command, a developer should use the corresponding programming language APIs for processing and reading files. If a certain functionality is only available via system commands, developers should carefully validate the user input. Ideally, this happens in a whitelist-based fashion.

13.2 Breaking Access Control

Sensitive data and operations lie at the core of many web applications. Ensuring that only authorized parties have access to this data and these sensitive operations is paramount. Therefore, web applications deploy an access control policy, which defines the rules and conditions to access sensitive operations or resources. Unfortunately, access control vulnerabilities are common, caused by incomplete policies, vulnerable implementations, or misplaced trust in user-supplied inputs.

Problem Description

The goal of breaking a web application's access control mechanisms is to gain unauthorized access to protected information. Unauthorized access can happen horizontally or vertically. A horizontal breach is a user gaining access to other user's data, without escalating the privilege level within the application. In a vertical breach on the other hand, a non-privileged user gains access to privileged operations, for example a user of a web application that becomes an administrator.

Access control mechanisms can be subverted in multiple ways, depending on the access control mechanism in place, and its implementation specifics. Common examples are access control mechanisms that base their security decision on untrusted input, which can be manipulated by the attacker (See Figure 13.1). Failure to protect all access paths to a resource may also lead to unauthorized access to certain files, especially problematic for sensitive system files.

A technical example of an access control mechanism that depends on untrusted input is a security decision that checks for an administrator flag, which is stored and transmitted as a cookie. An attacker can easily modify his own cookie to reflect the administrator flag ,giving him unauthorized access to administrator features. Another common case is an access control mechanism that uses certain URI parameters as input, allowing an attacker to specify a different resource or action than intended, bypassing the access control mechanism [16]. Another example of manipulating trusted input is a *path traversal* attack, where an attacker succeeds into navigating the server-side directory structure, inadvertently gaining access to files that are supposed to be protected.

In essence, access control mechanisms can be broken or bypassed by manipulating the inputs used in an unexpected way. These inputs can define the resource being accessed, or can be actual parameters used in the security decision for determining whether a user can access a resource or not.

Mitigation Techniques

Vulnerabilities in the access control mechanism are typically mitigated at design or implementation time, by implementing a centralized authorization policy, which is applied to all resources of a web application. Attention points for implementing the authorization policy are the different access paths for a resource, for example through a controller of a web application, or directly by constructing the correct URI for the resource. Additionally, access to resources through an identifier in a URI, such as an account number, should be checked against the authenticated user, in order to prevent an authorized user from accessing other resources, simply by changing the identifier in the URI.

An access control policy typically depends on several inputs to make a security-related decision. These inputs should be kept server-side as much as possible. For example, the privilege level of a user can be stored in the server-side session, and should not be stored using a client-side mechanism, such as cookies. Additionally, all inputs used to make security-relevant decisions should be carefully vetted for their origin, and if they are potentially untrusted, their validity should be verified.

State of Practice Access control vulnerabilities often occur in web applications. The OWASP top 10 project [261] identifies the ten most critical risks in web applications, and categorizes *Insecure Direct Object References* as number four, and *Missing Function Level Access Control* as number seven. The former indicates vulnerabilities where an attacker might tamper with identifiers or parameters, gaining unauthorized access to supposedly inaccessible resources. The latter indicates an inconsistently applied access control policy, leaving certain actions unprotected.

Rich web deployment environments typically offer built-in support for enabling access control. One common example is the JavaEE security model, a role-based declarative model based

on container-managed security, where resources are protected by roles that are assigned to users [197].

Research and Standardization Activities

Several research activities approach access control vulnerabilities by application of information flow techniques to detect unauthorized access. One approach is to use static analysis mechanisms to infer and enforce an implicit access control policy [242]. An analysis of the different roles within an application, and the accessible resources per node, allows determination of which resources are inadequately protected. *FixMeUp* [235] is another static analysis approach that starts from the high-level description of the access control policy, which defines the sensitive operations, as well as the required access control checks. Based on this information, *FixMeUp* is able to find missing access control checks, and propose candidate repairs. Finally, *Nemesis* [65] uses dynamic information flow tracking to track the user's authentication information throughout the application. By combining this authentication information with the given access control policy, *Nemesis* can prevent unauthorized access to sensitive resources and data.

Specifically focusing on parameter tampering vulnerabilities, *PAPAS* [16] applies a black-box scanning technique, analyzing the output for changes, indicating a potential parameter tampering vulnerability. *PAPAS* uses a crawler to investigate a running web application for parameter tampering vulnerabilities, and does not require source code to operate.

The standardization activities surrounding Cross-Origin Resource Sharing (CORS) [253] are also related to access control, albeit to relax certain origin-based restrictions in the browser using a cross-origin access control policy, rather than a tool to prevent direct, unauthorized access to application resources. CORS enables a web application to share a resource with another origin, allowing the client-side context to request and process a cross-origin resource. CORS policies are distributed on a per-resource basis, and are added to the response. For sensitive actions or resources, the browser first checks whether it can access the resource, before actually sending the request.

Best Practices

The best practice for preventing unauthorized access to server-side resources is to deploy a centralized authorization policy, consistently applied across all resources of a web application, covering all access paths to sensitive resources or data. When developing web applications on top of server-side frameworks with built-in support for access control policies, these features should be used to implement the desired, coarse-grained policy. Additionally, applying a fine-grained policy to specific sensitive functions in the application's code is a good defense-in-depth strategy.

For resources that might be shared with other applications, running in a different origin, the best approach is to deploy a consistent, carefully verified CORS policy. Be aware however that relaxing certain limitations using CORS might give other origins access to previously inaccessible information, which may lead to unwanted information leaks.

Chapter 14

Violating the User's Privacy

A user's *personal information* is a valuable asset within the Web ecosystem, and is therefore an attractive attacker target. Common threats towards the user's personal information, and thus the user's privacy, are the *harvesting* or *stealing* of personal information, as well as *tricking* the user into revealing personal information, for example through well-known phishing attacks. Finally, *tracking* is a common threat towards user's privacy, since it enables the leaking of information about visited sites. Note that the concept of an *attacker* in privacy-related issues not necessarily carries a criminal stigma, as illustrated by the major sites and corporations harvesting information to sustain their business model.

Within these threats to the user's privacy, we can identify two categories of issues. One category exists because users choose to disclose personal information to web applications, often run by large corporations aiming to make money based on this information. A very common example is the *harvesting* of publicly available data from social networking sites. The second category of issues consists of privacy violations that are out of the user's control, such as the unwanted tracking across different web sites, using commodity web technologies, such as cookies or browser fingerprints.

This chapter focuses on a number of attacks against the user's privacy that are enabled by the Web's technical architecture, and are thus directly related to the Web's technologies. More specifically, we cover two concrete attacks that reveal information about user's online behavior and browsing profiles. *User tracking* employs techniques to recognize the (anonymized) user when visiting a site, while *history sniffing* is aimed at discovering the user's browsing profile using the browser history.

As this chapter focuses on privacy issues related to web technology, we do not go into detail on out-of-band attack such as phishing or social engineering, which target the user's gullibility. Similarly, we do not focus on the non-technical aspects of user privacy on the Web, which is mainly covered by privacy policies and legislation, an entirely different topic.

14.1 User Tracking

User tracking enables an adversary to determine the user's browsing interests and behavior, enabling a so-called customized user experience. This customization is typically noticed as user-targeted advertising, with the goal of increasing the click rate, and thus profits earned by advertising.

Problem Description

User tracking – an attack on which assorted advertising business models are built, and which has been at the forefront of recent policy and regulatory debates – typically involves (technically)

lightweight collaboration from web sites that the user visits.

Modern web sites involve content from many different sources, typically imported by loading scripts or iframes from other origins: A site might show advertising; might use scripts served from elsewhere; might show a map from a large service provider; might involve widgets that interact with big social network sites.

In the simplest form of user tracking, these resources are served with a cookie that uniquely identifies the user; information gleaned from Referer headers can then be used by these "third parties" to reconstruct the user's browsing history, and other information about the user's behavior online.

We can distinguish between trackers that place state data on the client, and those that don't.

- The simplest tracking approach places state on the client that is then read back in order to link different transactions to each other. That can happen through Cookies (as is the case in the canonical example), or it can happen through more involved techniques. These include the use of client- side HTTP caching or eTags. Attackers that have the ability to execute JavaScript code on the client (e.g., many advertisers) can rely on other client-side storage mechanisms (such as IndexedDB), usually intended to enable web applications to enable offline operation, or quick retrieval of data that is stored on the client [183].
- Fingerprinting-based trackers do not place state on the client, but are instead based on the observation that the configuration of a given browser instance often carries sufficient entropy to permit unique identification of that browser instance. Information like the supported content types, fonts, geographic location, browser version, OS version, screen size, and other characteristics that can be discovered either on the wire or through appropriate JavaScript code, create an almost-unique browser fingerprint [85, 104].

Mitigation Techniques

As mentioned above, online tracking has been a political and policy hotbutton issue in ongoing struggles around web privacy. Several distinct approaches have been proposed to address the topic:

- *Deleting client-side state.* One of the simplest anti- tracking measures available to users is to delete client side state – cookies, flash locally stored objects, and other information. However, fingerprinting-based techniques, and parties that collect personal identifiers (such as user logins with social sites, email addresses, ...) can enable trackers to associate different identifiers with each other, even when client-side data is deleted.
- *Opt-out cookies.* Various industry self-regulation groups offer cookie-based opt-out mechanisms. It is often unclear what limitations on data collection these opt-out mechanisms actually entail. Further, cookie-based opt-out mechanisms often require users to opt out of each individual tracking scheme. Web sites that are intended to provide users with a simplified approach have been criticized as difficult to use [173].
- *Do Not Track.* The premise here is to enable user preference expression about online tracking (mediated by the browser), self- regulation, and existing regulatory contexts in order to create an ecosystem in which undesired tracking is kept under control, but certain desirable forms of tracking can continue. This model is based on the notion of largely voluntary compliance by all parties in the ecosystem, and needs to rely on the regulatory context for enforcement. The implementation consists of specifications that enable the expression of tracking-related user preferences through HTTP header and JavaScript API, and on a specification that defines the meaning of that preference [101, 43].
- *Anti-tracking technology* is typically deployed on the client side (e.g., through browser plugins). We typically see a combination of two approaches: On the one hand, anti- tracking tools implement systematic limitations on client-side state (such as third party cookie

blocking [184], and comparable restrictions on other client-side state). These restrictions apply across web sites, and effectively change the features of the Web platform that are available to developers. On the other hand, anti-tracking tools work with heuristics that attempt to recognize and block known tracking sites and tools. Known tools include various advert blocking tools (AdBlock Plus [198] is perhaps the most prominent), and dedicated anti-tracking tools like Ghostery [92].

In summary, widespread web tracking is a reality of today's Web. Tracking (either through actively placed identifiers, or through fingerprinting) is difficult to prevent through technical means. Even the most effective technical anti-tracking mechanisms are subject to an arms race between those who wish to track, and those who wish to prevent tracking – not unlike computer viruses.

At the same time, the business opportunities opened up by web tracking data make effective self-regulatory approaches to address the issue difficult.

Research and Standardization Activities

The most prominent ongoing standardization activity around online tracking is clearly W3C's effort around *Do Not Track*. As described above, this work aims at standardizing user preference expression in the tracking space, and at achieving a certain degree of agreement on what it means to comply with that user preference expression. The standardization work remains contentious at the time of writing this document.

The debate about whether web fingerprinting is a tractable problem or a lost cause is ongoing within the web security research and standards community. On the one hand, avoiding additional fingerprinting vectors is often a goal in the development of new web technologies. On the other hand, attempts to produce hard-to-single out web clients have not yet led to a usable browser that would be part of a sufficiently large anonymity set of browsers with the same fingerprint. Within the standards community, fingerprinting is a recurring theme. For example, the HTML 5 specification now specifically identifies features that can be used to identify the user [35].

Additional research in industry and academia aims at accommodating tracking- based business models such as targeted advertising while significantly reducing the privacy impact of these business models. Proposals often focus on privacy-by-design approaches in which tracking and targeting happens through dedicated mechanisms (such as Google's recent proposal of an "AdID") that can be controlled and customized more tightly by users than conventional tracking mechanisms. These approaches often assume wide-spread deployment of additional technology in web browsers in order to become effective; none of the research proposals have reached that stage as yet [121, 210, 21].

Best Practices

The best practice to prevent state-based user tracking is to keep a clean browser profile, using one of the many available extensions. Regularly cleaning the cookie store, especially cookies from unknown domains, might help as well. Unfortunately, no mitigation technique against fingerprint-based is currently known.

14.2 History Sniffing

A history sniffing attack abuses certain browser features to extract information, such as previously visited sites. Traditional history sniffing attacks use a CSS feature, while more advanced techniques are using timing-based attacks. By extracting a user's history, web applications can determine the user's interests, or whether the user visits any competitor's sites.

Problem Description

History sniffing techniques attempt to identify a user's previous browsing history *without* the collaboration of the other web sites.

Ever since the early days of the Web, visited links were styled differently from links that the user had not visited yet; it's a feature that was continued in CSS. Intended as a convenience for the user, this design choice implies that the presentation of a web page to the user depends on information that should not be accessible to that page's origin: Whether or not the user has visited some other site before.

Early versions of history sniffing exploited the fact that JavaScript code could investigate the styling of an element: If a link was colored purple, JavaScript code would be able to detect that fact, and could conclude that the web site pointed at by the link had been visited before by the user [20].

More recently, techniques such as timing attacks against the web browsers' cache have been used to discover whether a user has previously visited a site, across origins [240].

All known history sniffing attacks have several characteristics in common: They constitute a leak of user behavior across different origins, without the collaboration of these different web sites. They further enable the attacker to inquire whether the user's browser has (recently) accessed a specific web site. While efficient attacks can permit the attacker to inquire about a large number of domain names within a short time, the attacks seen to date have not been able to simply extract the entire browsing history.

History sniffing attacks continue to be found on the Web, and have at times been broadly deployed [147]. More recently, they have been the subject of enforcement actions by the United States Federal Trade Commission [96].

Mitigation Techniques

The main mitigation technique against the original, CSS-based history leak is broadly implemented, but not standardized. [20] The approach is based on subtle special case handling of the `:visited` CSS selector, with the goal being, on the one hand, to preserve the ability of web site authors to style visited links, while, on the other, mitigating against an attacker's ability to access that style information:

- JavaScript code that attempts to read styling information will always see style information as if a link is unvisited.
- More generally, when the `:visited` selector is used, browser implementations will pretend that a link is not visited. A visited link's impact on page styling is highly localized; the effect is further limited to very few (mostly color-related) CSS properties.

As a result, the original CSS history sniffing vulnerability is considered closed.

However, as discussed above, some graphics operations and web site retrieval from the browser cache are subject to cross-origin timing attacks, that remain an ongoing concern.

Users who wish to mitigate against history stealing attack should either browse in a private mode that does not expose their browsing history, or should disable visited link styling in their browser – as is, for example, possible through the `layout.css.visited_links_enabled` setting in Firefox' configuration.

Research and Standardization Activities

Timing attacks in general are an ongoing area of research for the Web platform, as they permit the exfiltration of data across origins [240].

While the main mitigation technique against CSS-based is broadly deployed, there are currently no plans to standardize this approach.

Best Practices

Best practices to prevent history sniffing attacks are hard to give, since they are either already mitigated within a modern browser, or active research, for which no real mitigation techniques exist. Similar as to user tracking, keeping a clean browsing profile may be a nuisance, but can help mitigate these attacks, albeit marginally.

Intermezzo 8: Tracking and Fingerprinting

Million of web sites worldwide attract billions of users on a daily basis. In this Web ecosystem, it is very lucrative for advertising companies to be able to track users and their online habits. By profiling end-users, they can customize advertisements to their audience, and increase the charges made for the advertisements being clicked.

Traditionally, tracking companies used third-party cookies to track the user's surfing behavior over multiple sites. On each site that takes part in the tracking campaign, a small JavaScript under the control of the tracking company gets loaded, and calls back to the tracking company by sending HTTP requests. To identify the end-user, a so-called third-party cookie is issued by the tracking company, and accompanies each call back request. The name third-party cookie reflects the fact that the cookie is issued and sent to a third-party domain, which differs from the domain shown in the address bar.

As more and more browsers support the blocking of third-party cookies, other techniques have been investigated to track users in a much more pervasive way. The most recent technique being deployed is browser fingerprinting, in which a stable snapshot is made of the client-side configuration.

To fingerprint a browser instance, up to 126 different characteristics of the client configuration can be assessed. This includes browser customizations (such as the set of installed browser plugins), browser-level user configurations (such as the cookie configuration and Flash settings), the browser family and version (calculated in various ways), the operating system and applications, and the underlying hardware and network configuration.

In particular, Nikiforakis *et al* revealed that (1) Flash is heavily used to retrieve more sensitive information for the user, that (2) the variety of browser implementations and versions allow very accurate fingerprinting of the local configuration (e.g., based on the JavaScript properties (and their order) of `navigator` and `screen`), and (3) the system fonts and the order in which they are listed in JavaScript and Flash give away information about the operating system configuration and the installed applications. Moreover, their research concludes that browser extensions that spoof the identity of the browser all fail to completely hide the browser's real identity.

Curious to see how unique your browser configuration is? Check it out at `https://panopticlick.eff.org/`

Source: *Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, Giovanni Vigna, Cookieless monster: Exploring the ecosystem of web-based device fingerprinting, IEEE Security and Privacy, San Francisco, 19-22 May 2013 [193]*

Part IV

Conclusion

Chapter 15

Conclusion

In this Web platform security guide, we reported on the broad security assessment that has been conducted in task T1.1 of the EC-FP7 project STREWS. In this task, we assessed the security of current and emerging web technology, forming the foundations of every single web application.

As part of this report, we provided a clear and understandable overview of the web ecosystem (part I), and discussed the vulnerability landscape, as well as of the underlying attacker models (part II).

In addition, we provided a catalog of best practices with existing countermeasures and mitigation techniques, to guide European industrial players to improve step-by-step the trustworthiness of their IT infrastructures (part III).

In next two sections, we take a step back to oversee the full Web security landscape. In Section 15.1, we link back the assets of part II and the attacks from part III in a unified Web security threat landscape. Section 15.2 concludes this Web security platform guide with a set of interesting challenges for securing the Web platform, opportunities for future research and trends in improving Web security.

15.1 Web Security Threat Landscape

In the first three parts of this document, we systematically assessed the security of the current Web ecosystem, based on a set of relevant assets within the web application model, covering the infrastructure, the web applications and the user-centered assets. For each asset, we identified the relevant threats using high-level attack trees, clearly indicating the various approaches an attacker can take to compromise the asset. An attacker aims to compromise an asset using one of the defined threats, which is executed by a concrete attack, for which the attacker requires a certain set of capabilities. We have expanded these threats into concrete attacks, based on the body of work currently known in the academic world, the standardization bodies, and the developer-centered Web security communities. Figure 15.1 shows a combined view of the results of this process, showing the assets, the threats that can lead to their compromise, and the concrete attacks embodying these threats.

The combined tree aggregates a lot of information and relations, supporting several ways of reading it, leading to several conclusions about the security of the Web platform. A first and important observation is the wide scope, clearly indicating that Web security is not limited to securing individual assets, but requires a broad, integrated approach. For example, aiming to secure the *Application Transactions* asset involves mitigating the closely related threat of *Forging Requests*. Additionally, it also requires mitigation of the threat to *Impersonate Users* and to *Intercept and Manipulate Traffic*. Using the tree from the viewpoint of an asset enables the identification of potential escalation scenarios, following from the compromise of another asset.

Additionally, the tree enables the construction of a list of dependencies, supporting decisions of which assets to secure, and which threats to mitigate.

Another use case of the combined view is assessing the risk or danger of a specific attack or threat. Starting at an attack, the tree allows the identification of potential assets that can be compromised, and potential escalations that can follow from there. For example, assume that a web application is vulnerable to *Cross-Site Scripting*, clearly a threat against the *Client-side Application Code*. Using the tree, we are able to discover that the attack is not limited to this asset, but supports the escalation towards unauthorized access to *Client-side Content Storage*, the compromise of *Application Transactions*, as well as the theft of users' *Personal Information*.

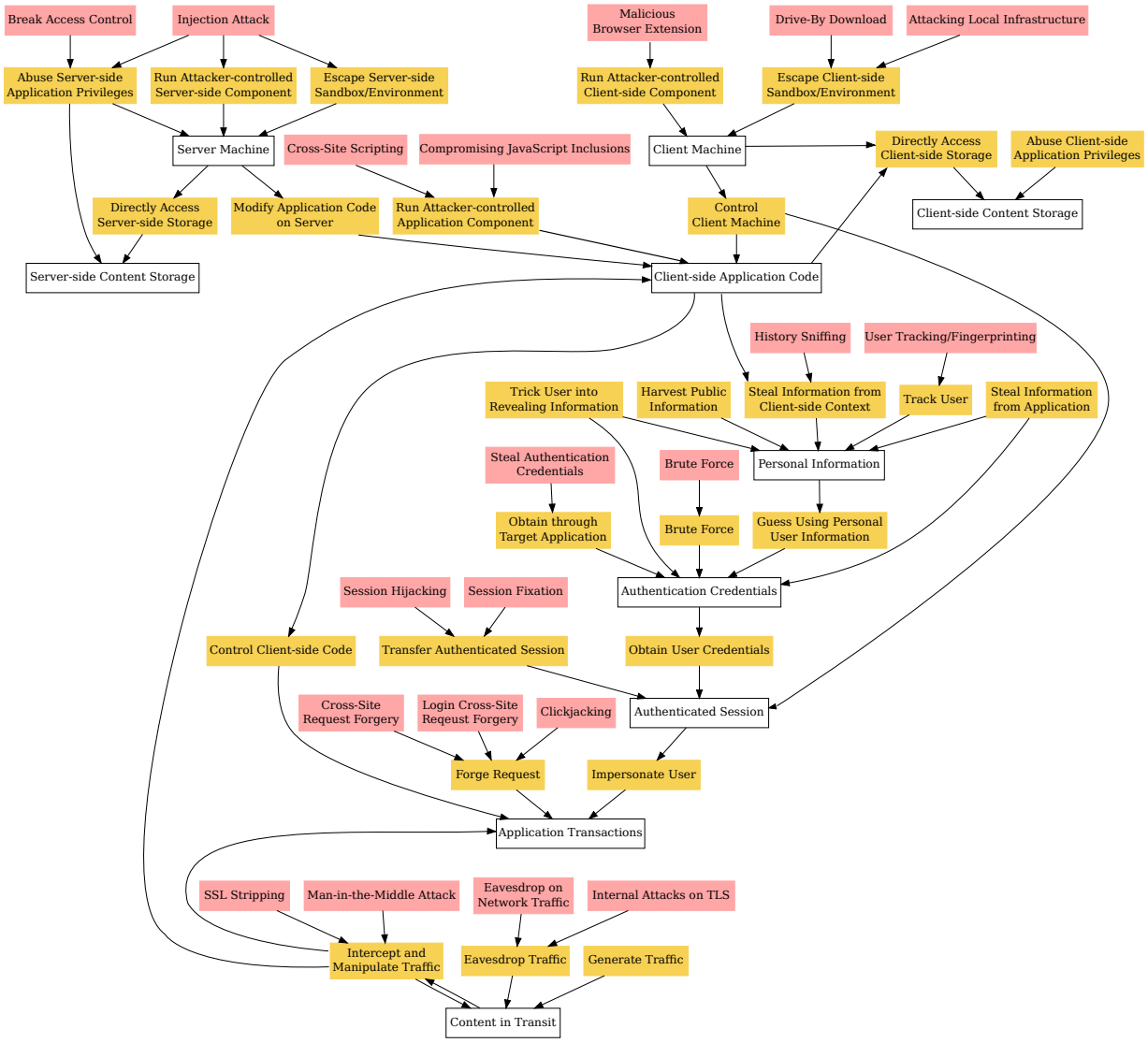


Figure 15.1: A combined view aggregating the results from the security assessment, showing assets (white), their associated threats (yellow) and the concrete attacks embodying a specific threat. The combined view not only shows the threats for each individual asset, but also identifies interesting relations and dependencies, allowing for a potential escalation of an attack.

The combined view of assets, threats and attacks supports several approaches towards assessing the security within the Web platform, tailored to specific situations. The combined tree

however does not provide the details that we presented in the separate low-level attack trees in Part III, such as the separate steps in an attack, or the specific attacker capabilities needed for each attack. Due to the inherently complex and intermingled nature of the Web, the combined view would become a labyrinth of dependencies, thereby clouding the use cases we discussed above. For example, a *Session Hijacking* attack has several attack vectors that can be used, each depending on different capabilities or related assets. One way is for an attacker to compromise the *Client Machine*, another to gain control over *Client-side Application Code*. Additionally, the attacker can eavesdrop on *Content in Transit*, or *brute force* the session identifier.

The overview of the Web security threat landscape is also abstracted in Table 15.1, which maps the assets from part II and the attacks from part III. The table can be read twofold:

- On the one hand, Table 15.1 illustrates the impact of a particular Web attack on the various assets of the Web platform. For instance, a successful *injection attack* can potentially impact 8 of 10 assets of the Web platform.
- On the other hand, Table 15.1 enumerates the list of attacks that need to be mitigated in a particular Web application in order to protect an asset. For instance, in order to fully protect *application transactions*, at least 17 attacks need to be mitigated.

	Server Machine	Client Machine	Server-side Content Storage	Client-side Content Storage	Content in Transit	Client-side Application Code	Authenticated Session	Application Transactions	Authentication Credentials	Personal Information
Session Hijacking							★			
Session Fixation							★			
Brute Force									★	
Stealing Authentication Credentials									★	
Cross-site Request Forgery								★		
Login Cross-site Request Forgery								★		
Clickjacking								★		
Eavesdrop on Network Traffic				★	★	★	★	★	★	★
SSL Stripping				★	★	★	★	★	★	★
Man-in-the-Middle Attack				★	★	★	★	★	★	★
Internal Attacks on TLS				★	★	★	★	★	★	★
Cross-site Scripting				★		★	★	★	★	★
Compromising JavaScript Inclusions				★		★	★	★	★	★
Malicious Browser Extensions		★		★		★	★	★	★	★
Drive-By Download		★		★		★	★	★	★	★
Attacking Local Infrastructure		★		★		★	★	★	★	★
Injection Attacks	★		★	★		★	★	★	★	★
Break Access Control	★		★	★		★	★	★	★	★
User Tracking/Fingerprinting							★	★	★	★
History Sniffing							★	★	★	★

Table 15.1: Overview of the Web security threat landscape: mapping assets from part II to attacks from part III.

The Web security threat landscape in Table 15.1 clearly illustrates the complexity of the Web ecosystem. To improve the end-to-end security, it is necessary to raise the bar on several (if not all) topics in parallel.

To conclude, the security assessment in this document gives a broad but accurate view on the security of the current Web, including the existing countermeasures, mitigation techniques and best practices. The view presented in the combined tree supports high-level risk assessment activities, and clearly shows the tight relations between different security aspects within the Web platform. An interesting future line of work on the Web security assessment is to to create a systematic process of assessing Web security, focusing on a phased deployment of security measures, starting with the most basic and high-risk assets, and working towards the more isolated, lower-risk assets.

15.2 Future Challenges, Trends and Opportunities

To grasp the complexity of the Web platform and its security characteristics, the STREWS consortium brings together a unique set of expertise in Europe. In particular, the consortium comprises strong peers in academic web security research in Europe, a large European software vendor, and principal stakeholders in standardization activities in W3C and IETF.

Given the complementary expertise of this one-of-a-kind consortium, we have been brainstorming on the way forward for Web application security during one of our face-to-face meetings. Based on this brainstorm session, and the hands-on expertise collected during the construction of this Web platform security guide, we identified a set of challenges for securing the Web platform, some opportunities for future research and trends in improving web security. We would like to briefly share them here, and elaborate them in more detail during the roadmapping activities of STREWS.

Challenges

1. We clearly see the urge to fix some of the legacy building blocks of the web model. For instance, passwords are still the primary authentication technique on the web, and are almost always used in combination with bearer tokens (e.g., session management cookies and OAuth tokens).

   Major changes to legacy building blocks of the Web model face prohibitive deployment obstacles, as the currently-deployed legacy of web applications relies on the legacy model's properties, and the adoption of best practices is rather slow.

2. In order to advance, it is important to re-unify the specification and implementation fronts. For example, the WHATWG fork of the W3C HTML5 working group, the BLINK fork of WebKit, and the numerous disagreements on various topics (such as the use of URL/URI/IRI, the use of unicode in domain names and URIs) might confuse people or slow down the necessary security innovation.

   The various forks could potentially lead to a situation where security technology is only partially picked, or adopted at varying speeds, or even worse continue to work on potentially conflicting variations of the same technology.

3. Web security is partially shifting from a purely technical topic to a user-centered topic. This is illustrated with the numerous phishing and social engineering attacks, and the web permission model relying more and more on decisions of the end-user. For sure, attackers will more and more target the user as the weakest link of the web infrastructure.

4. Although the Content Security Policy (CSP) is a very promising security technology, the risk exists that CSP will be seen as the universal web security solution. Pushing all the

complexity of web security into CSP extensions might simply kill the adoption potential of the technology.

5. There exists a remarkable mismatch between state-of-the-art mitigation techniques and best practices being available for almost all vulnerabilities, and we measured only a limited adoption of the best practices in the state-of-practice.

   The question remains as to how web site owners can be incentivized to actually deploy best practices on their site? Similarly, to track the adoption rate over time, it is important to have good metrics and measurements in place to be able to assess the state-of-practice of the Web ecosystem.

6. From time to time, the moving semantic model of the web breaks the (security of) legacy applications. Interesting examples are the introduction of the application cache, or the interference of XHR/CORS with other web features, such as the traditional same-origin restrictions of XHR, or the *null* origin of an HTML sandbox [71].

7. Updates to the foundations on which the Web ecosystem is built, often impact certain aspects within the Web ecosystem, regardless of all attempts of achieving a cleanly-separated, layered approach. Recent updates requiring additional attention are the potential interference of the ongoing deployment of DNSSEC, or the IPv6 rollout, for example in code or policies that explicitly deal with bare IP addresses.

Opportunities for Future Research

1. There is certainly need for improved techniques to increase the user's privacy. For instance, few or no effective countermeasures exist to protect an end-user against tracking and browser fingerprinting.

2. As web applications are becoming larger, and contain more third-party components (e.g., third-party JavaScript inclusions), the secure containment or sandboxing of untrusted parts of the web application becomes crucial. Current state-of-the-art containment techniques still need to mature, both in terms of policy specification as well as enforcement techniques.

   Within STREWS, we will dedicate our second case study to *web security architecture*, and will tackle some of the challenges in compartmentalize web applications based on their trust level.

3. UI security becomes a key factor in delivering a secure web ecosystem, especially with the rise of new web-capable devices, such as smartphones, tablets, etc. The web permissions model is extraordinarily complex, and hard to understand for the end-user. Key questions are how to involve (or not involve) users in security-related web decisions and how to communicate back security results to the user.

   This is also one of the aspects we are currently tackling in the first cast study of STREWS on real-time web communication (WebRTC).

Trends

1. Significant areas of novel web security technology follow the same pattern: The server issues a security policy, the policy is pushed towards the client as part of the web application, and the client is responsible for enforcing the policy correctly. Well-known examples in recent specifications are CSP, X-Frame-Options, HSTS, and Certificate Pinning.
   We observe similar patterns in the research space, for instance the server-driven JSand technology [6], developed as part of EC-FP7 project WebSand – Server-driven Outbound Web-application Sandboxing.

2. The Content Security Policy (CSP) seems to be a very promising additional layer of defense, protecting against cross-site scripting and UI redressing. In addition, CSP is an interesting enabler for more advanced security architectures, such as the architecture underneath the office document reader inside Chrome OS [258].

3. In light of the recent revelations of Snowden [244], the security community is investigating a new attacker model, covering the pervasive monitoring and wide-scale man-in-the-middle capabilities of high-ranking agencies. Additional effects are the reinforced interest in security protocols that offer strong guarantees, such as perfect forward secrecy (PFS), as well as TLS-everywhere proposals and alternatives for the currently-deployed CA architecture. Finally, *algorithm agility* is key to guarantee the long lifetime of security-related protocols in the Web.

   This is one of the topics we have put on the agenda for the first STREWS workshop. This will be a joint W3C and IAB workshop, called STRINT (*Strengthening the Internet Against Pervasive Monitoring*) and it will take place in London, from 28 February to 1 March 2014.

Appendix

Appendix A

Basics of the Web Platform

Content is the main reason for the existence of the Web. Web content resides on remote servers, and ends up displayed in browsers on our desktops, laptops, tablets, mobile phones, televisions, adverising devices and soon in wearable devices, such as Google Glass. This chapter starts with the basic building block of web content, web pages, and follows through with the most important evolution of the past few years, interactivity. At the end of the chapter, we shed some light on how URIs contribute to locating web content, which is the starting point for the next chapter, where we discuss how web content is actually loaded in the browser.

A.1 Defining Web Content

Web pages are a basic building block of the Web, since they exist on the server, are transferred over the network and displayed by your browser. The first web pages were very static, only containing some (largely unformatted) content, a stark contrast compared to contemporary web pages, which are highly dynamic and sophisticatedly styled (at least in the more aesthetically pleasing cases). Two core technologies behind the structure of web pages are HTML and CSS: the former adds structure, while the latter adds style. This section covers both, and the next session adds interactivity to the mix.

A.1.1 HTML

HTML, HyperText Markup Language in full, is the main language used to describe the structure and semantic content of a web page, which is typically processed and displayed on your screen by a web browser. HTML is only intended to offer structure and meaning to the content, and leaves styling and formatting to other technologies, such as Cascading Style Sheets (CSS).

The first version of HTML was created by Tim Berners-Lee in 1989 as a prototype in a system to share information among researchers. Several standardization efforts have lead to the extensive HTML 4 and 4.01 standard [203, 204] in 1997, but the highly competitive browsers of that time had already introduced individual custom elements or interpretations, causing major incompatibilities, both with the standard and between browsers. All this variation makes HTML parsing a daunting task, riddled with quirks and pitfalls [265]. With the recently introduced HTML 5 standard [33], hopes are high that the cooperation between browser vendors and standardization committees will lead to more interoperability. Currently, the Web Hypertext Application Technology Working Group (WHATWG) maintains the HTML standard as a living document, which represents what browser vendors do, while the W3C maintains snapshots of the standard, which guarantees interoperability between implementations [38, 135].

An HTML document is composed of hierarchically structured elements, defined by tags. It is based on an ISO standard for text markup languages called Standard Generalized Markup

Language (SGML) [142]. Typical elements have an opening and closing tag (Listing A.1, lines 5 and 9), with their children or content in between. Empty elements can be self-closing, by contracting the opening and closing tag (Listing A.1, line 7). Additionally, opening tags can contain parameters using the *name=value* syntax (Listing A.1, line 6). The code example also shows the typical structure of an HTML document, with the top-level `<html>` element, followed by a document head (`<head>`) and body (`<body>`). The former contains document meta information, such as the title, keywords or additional resources, while the latter contains the actual content.

```
1  <html>
2    <head>
3      <title>HTML Code Example</title>
4    </head>
5    <body>
6      <div id="maincontainer">
7        <img src="screenshot.png" />
8      </div>
9    </body>
10 </html>
```

Listing A.1: HTML documents contain hierarchically structured elements.

The HTML elements relevant for the remainder of this document are briefly explained below. A detailed view of each element can be found in the HTML specification [33], supplemented by a discussion of parsing details and quirks [265].

`p` The `p` element represents a paragraph, adding basic structure to text documents. The element can contain both child elements and text content.

`a` The `a` element typically represents a hyperlink, pointing to content located elsewhere on the page or in a different document. The content of the element is displayed by the browser, and the destination of the link is specified in the `href` attribute. The `target` attribute specifies whether the browser should replace the current document when following the link, or open a new tab or window.

`form` The `form` element groups form-associated elements, which typically contain values that need to be sent to the web application. A common example is the authentication form, where a username and password are entered by the user and sent to the web application. The attributes of the `form` element control its behavior, such as the remote location to send data to (`action`), the HTTP method to use (`method`) or the desired encoding (`encoding`). The form-associated elements responsible for obtaining user-specified input are typically `input` elements.

`input` The `input` element represents a data field, and is typically used as a form-associated element. The `type` attribute specifies how the data field is represented to the user (e.g., a checkbox, radio button or text field). `input` elements can be hidden from the user, in which case they contain an application-provided value that needs to be submitted as part of the form. A new feature introduced in HTML 5 offers syntactical validation of the input, allowing early feedback to the user in case of erroneous input.

`iframe` An `iframe` element represents a nested context within the current document, in which additional documents can be loaded. The use of the `iframe` element allows multiple documents to be loaded within the parent document, while keeping them separate, thus avoiding naming conflicts, enabling separate reloads or independent navigation

within each document. The document to be loaded in the `iframe` can be specified using the `src` attribute.

`object` An `object` element allows the inclusion of an external resource that, depending on the type, is handled as an image, document or external resource with the operation delegated to a plugin. In the first case, the content is simply added to the document as an image. In the second case, the new document is loaded in a separate context (similar to the `iframe` element), and in the last case the content is given to the appropriate plugin to handle, for example the Flash player. The resource to be loaded can be specified using the `data` attribute.

`script` The `script` element allows the inclusion of dynamic scripts, adding interactivity to the document (See Section A.2). The dynamic script can be specified as the contents of the `script` element, or as an external resource using the `src` attribute. The `type` attribute defines the language of the script, and the loading and execution can be influenced using the `async` and `defer` attributes.

`meta` The `meta` element allows the expression of three kinds of metadata, that do not fit elsewhere. The first kind is document-level metadata, such as keywords for cataloging the document, and is specified using the `name` attribute. The second kind is HTTP metadata (See Section A.4), specified using the `http-equiv` attribute. The third kind defines the encoding to use when serializing an HTML document, for example to store it on file or send it over the network, and is specified using the `charset` attribute.

A.1.2 CSS

Cascading Style Sheets (CSS) are documents witten in a language used to describe the presentation semantics of HTML documents. A style sheet describes how the structured elements of HTML need to be rendered (i.e., displayed on the screen or other rendering surface) by the browser, in some cases up to the pixel-level if desired. CSS was first developed in 1996 by W3C, with a second level appearing in 1998. A third, expanded level is currently still under development, but already has a wide variety of new functionality, enabling complex graphical applications.

```
1  body {
2    background-color: white;
3  }

4  #menu {
5    background-color: gray;
6    color: #000066;
7  }

8  .titles {
9    font-family: "Times New Roman", Times, serif;
10   font-style: italic;
11 }
```

Listing A.2: CSS style sheets apply the defined style to HTML elements that correspond to the selector.

CSS offers a variety of features, such as positioning content, controlling spacing between elements, assigning fonts and colors, modifying the cursor appearance, etc. The syntax for

defining features is straightforward, using the *name:value;* format (See Listing A.2). CSS also supports a separate style for different target platforms, such as printing, or the screen of a mobile device.

Typically, CSS code is shared by all parts of a document, or even the entire web application. The recommended way to incorporate style sheets is by creating separate files, which can be included using the `link` element. An inferior alternative is to embed the CSS code directly in the HTML file, using a `style` element[1]. CSS code can be applied to specific HTML elements by defining a selector that filters out the desired elements. The most basic selector is the name of an HTML element ((Listing A.2, line 1)). Additionally, CSS has built-in support for the `id` (Listing A.2, line 4) and `class` attributes (Listing A.2, line 8), which respectively identify an element with a specific id, or all elements belonging to a class. Selectors can also be restricted to a certain context, e.g., all elements within a given other element. Recently, CSS selectors have become very powerful [50], even allowing selections based on a specific value of any attribute of any element.

A.2 Client-side Interactivity

The first *scripting engine*, introduced in 1995 in the browser Netscape Navigator, enabled the automation of basic client-side tasks, such as manipulate HTML documents, display dialogs and manipulate open browser windows. The language was rebranded to *JavaScript*, and became extremely popular. Further development of the language is driven by the European Computer Manufacturers Association (ECMA), leading to an official specification of *ECMAscript version 5*, with several new features, many focused on isolation and security. In addition to the language standardization, W3C is involved in the standardization of several browser-supplied APIs, which define the extensive functionality available to JavaScript within the browser.

A.2.1 JavaScript in the Browser

The recommended way to integrate JavaScript in an HTML document is by including a `script` element which refers to the external file containing the JavaScript code. Alternatively, the JavaScript code can be directly integrated in the HTML document by adding it as the content of the `script` element, albeit with a lowered separation between content and code. A third, but discouraged way, is to attach JavaScript directly to HTML elements, for instance as event handling code using the `on*` (e.g., `onfocus`) attributes.

Within the browser, a separate JavaScript context is assigned to each HTML document. Each context has its own global scope and function space, and all scripts loaded within the associated HTML document will execute in this context, regardless of their source. Browser-supplied APIs offer the required functionality to navigate, communicate with or access other JavaScript contexts, albeit constrained by several browser-based security policies (See Chapter 3).

JavaScript's execution model in the browser is based on an event loop, which runs the scripts in a page in a synchronous fashion. An additional task of the event loop is to run asynchronous callbacks that have been triggered. The latter enables calls to browser APIs, for example to send a network request, without having to wait for the result, which can take some time to complete. Using the newly introduced Web Workers [134], JavaScript code can be launched in a separate OS-level thread. Workers have no access to thread-unsafe data, making concurrency problems a hard thing to achieve. Data exchanges between the main context and a Worker is possible using event-based message passing.

[1]The third option is to attach CSS code directly to an HTML element using the `style` attribute. This completely violates the intended separation between content and presentation, and should be avoided.

A.2.2 JavaScript Data Format

With the capability to make additional HTTP requests from JavaScript with the *XMLHttpRequest object*, the need for a simple format to exchange data led to the introduction of the JavaScript Object Notation (JSON). JSON is a human and machine readable language, based on a subset of the JavaScript language (See Listing A.3 for a code example), currently standardized by the IETF [110]. While JSON was originally used in combination with JavaScript, it is now a language-independent format, used by many services on the web to return data to client requests.

```
{
    "title": "Nineteen Eighty-Four",
    "year": 1949,
    "language": "English",
    "authors": [
        {
            "firstname": "George",
            "lastname": "Orwell"
        }
    ]
}
```

Listing A.3: The JSON format is based on a subset of JavaScript, and is commonly used to describe data objects in HTTP requests and responses.

In a nutshell, the JSON format supports six basic types: strings, numbers, booleans, arrays, objects and null. Arrays, denoted with square brackets, can contain a list of entries, while Objects, denoted with curly brackets, contain properties, which have a key and a value, separated by a colon. Since JSON is constructed with a subset of JavaScript, it can be processed by the `eval` function within JavaScript, a common practice that leads to numerous injection vulnerabilities, with reported cases where attackers can run arbitrary JavaScript code by manipulating a JSON response. Nowadays, modern browsers all offer dedicated API calls to parse JSON, without the risk of inadvertently running attacker-controlled code in the client-side context.

The popularity and widespread support for JSON has sparked a new evolution towards a secure object format, with support for signing and encryption [19] (JOSE). Use cases for the secure object format are commonly used security tokens, for example in authentication protocols between multiple providers [125], or the representation of sensitive data, such as browser-generated keys in the newly introduced Web Cryptography API [63].

A.3 Identifying Resources on the Web

Resources on the Web can be uniquely referred to by a Uniform Resource Identifier (URI), the typical web address you see or type in the address bar of your browser. URIs have had a long history, and are swamped with peculiarities that need to be taken into account [265]. In this section, we cover the basics of URIs, and focus on the parts relevant for the remainder of this document.

A.3.1 Anatomy of a URI

The URI shown in Figure A.1 consists of several parts, of which the underlined parts are optional. The role of each part in the URI is discussed below.

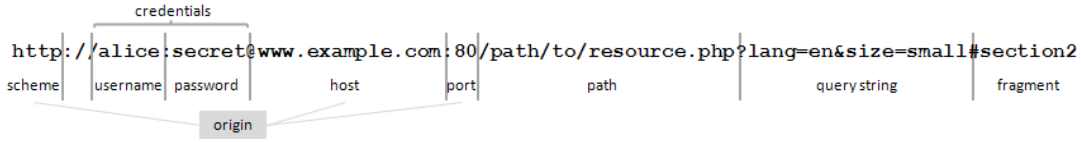


Figure A.1: The decomposition of a URI, used to locate resources on the Web.

Scheme — The scheme identifies the protocol used to fetch the resource. Typical options are `http` and `https`, although numerous other protocols are supported by modern browsers.

Credentials — The credentials consist of a username and password, separated by a colon[2]. These credentials are used to fetch the resource, in a protocol-specific way. In case no credentials are given, the resource is fetched anonymously, which may fail in case credentials are required. In this case, the browser typically prompts the user to enter valid credentials or cancel the request.

Host — The host identifies the remote server from which the resource is to be fetched. This has to be a valid IPv4 or IPv6 address, or a DNS name that can be resolved to an IP address. Browsers tend to be lenient when parsing the host part of a URI [265], for example by allowing octal, decimal or hexadecimal IP address notations, or special characters in DNS names.

Port — The port identifies the port to connect to on the remote server. If no specific port is given, the protocol default is used, i.e., port 80 for HTTP and port 443 for HTTPS.

Path — The path must conform to a UNIX-style path, identifying a resource located on the server. In the traditional model, the path actually refers to a file on the file system, located in some folder, but advanced web application models may not depend on the file system anymore, and translate the path to something meaningful in their content or, indeed, their context as the path need not refer directly to a content element.

Query string — The query string is an optional place to pass parameters, separated by an & or ;, to the resource being fetched. In the example above, the parameters could indicate the preferred language to be English and the size to be small, although the semantics of these parameters are completely application-dependent.

Fragment — The fragment is an optional part of the URI, intended to hold a value meant for client-side processing, which should not be sent to the server (most servers will reject URIs containing fragment identifiers). Traditionally, the fragment holds the name of an anchor tag somewhere in the page, allowing the browser to scroll to the location indicated by the fragment. Since no strict rules about the fragment exist, it has also been used to store client-side secrets, or even to enable messaging between contexts [27].

In addition to the traditional protocol-related schemes, such as `http` and `https`, modern browsers also support internal schemes [157], such as `data`, `javascript` or `chrome`. A `data` URI can hold arbitrary data, such as a base64-encoded image (See Figure A.2), a `javascript` URI contains JavaScript code to be executed when the URI is loaded and `chrome` URIs are used to refer to internal settings and collected data within the Chrome browser.

[2]RFC 3986 [36] deprecates the inclusion of credentials in the URI, since "the passing of authentication information in clear text has proven to be a security risk in almost every case where it has been used".

```

AAUAAAAFCAYAAACNbyblAAAAHElEQVQI12P4//8/w38GIA
XDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg==
```

Figure A.2: A URI with the `data:` scheme, here used to hold the base64-encoded contents of a PNG image.

A.3.2 URIs in the Browser

The most common use of URIs in the browser is through the address bar, where a user enters a URI and checks whether the loaded document corresponds to the expected URI. URIs are also used to load additional content within a page, such as images and scripts, or whenever a link is followed.

When a URI is used within a document, a relative URI can be used, instead of an absolute URI. Relative URIs only contain (part of) the parts starting from the path. The browser automatically resolves the URI using the currently loaded document as the base. For example, the relative URI `test/image.png` in a document with URI `http://www.example.com/images/gallery.html` would resolve to `http://www.example.com/images/test/image.png`.

As well as indicating the location and name of a resource to be loaded, URIs also serve a second function within the browser. Many security policies and permission systems are based on parts of the URI, as will be explained elsewhere in this document. Two concepts derived from the URI are relevant to remember:

Domain The domain corresponds to the host part of the URI. A domain can have subdomains, which reside on a lower hierarchical level than the parent domain. For example, `internal.test.example.com` is a subdomain of `test.example.com`, which is in turn a subdomain of `example.com`. The `com` part is known as a *Top-Level Domain* (TLD), which cannot exist by itself[3].

Origin The origin is defined by the triple *(scheme, host, port)*. The origin is used within the browser to restrict access to certain client-side resources, or to make security decisions in general. Since deriving the origin from a URI can be tricky, especially with all the possible appearances of a URI, it has been standardized in an RFC [23].

A.4 Loading Web Content

To access resources on the Web, identified by a URI, the browser sends a request to the appropriate server, asking to process some information or to return a specific resource. The server attempts to fulfill the request, and returns the results in a response to the client. This request-response protocol is known as the *HyperText Transfer Protocol (HTTP)* [99]. Listing A.4 shows a typical HTTP request and response.

[3]All TLDs are defined in a publicly available list, maintained by IANA. This list should be respected by all browsers when determining the TLD of a URI.

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:19.0) Gecko/20100101 Firefox/19.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive


HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 131
Connection: close

<html>
<head>
  <title>An Example Page</title>
</head>
<body>
  Hello World, this is a very simple HTML document.
</body>
</html>
```

Listing A.4: An example of a typical HTTP request and response.

HTTP is an application-layer protocol built on top of TCP/IP, and is the de facto standard used for exchanging information between browsers and servers. Since HTTP works on the application layer, it is based on URIs and hostnames, but lower in the protocol stack, IP addresses are used. The translation between hostnames and IP addresses is supported by the Domain Name System (DNS), which we do not elaborate on here. Alternatively, HTTP can use IP addresses directly if the URI contains an IP address instead of a hostname.

HTTP requests and responses follow a certain pattern, but have many configurable fields. In this chapter, we cover the relevant configurable fields, such as the different HTTP methods, request and response headers, and response codes. The variation enabled by these configurable fields enables a lot of important web technology. For example, since HTTP is a stateless protocol, there is no relation or required order between subsequent requests. Any need for relations or order between requests needs to be maintained by the browser and/or servers, on top of HTTP. This behavior is enabled by HTTP request and response headers, as we will explain later.

Finally, we also discuss the lack of security in HTTP, and elaborate on how these problems are addressed by using HTTPS, a deployment of HTTP on top of a SSL/TLS-secured connection.

A.4.1 HTTP Methods

The *method* is a mandatory field of an HTTP request, and determines the type of action that needs to be performed by the server. The two most common actions are to send a resource to the client (GET) and to accept the information sent by the client (POST). Next to these two, several others are defined in the specification (HEAD, OPTIONS, PUT, DELETE, TRACE, CONNECT), and custom methods are allowed as well.

GET Method

The GET method is used by the client to retrieve a resource from the server, specified by the URI in the request. These resources are not restricted to static resources, such as HTML documents and images, but can also be dynamically generated by the server. Examples of dynamic resources are a list of user-specific contacts, the contents of a user's shopping cart, etc.

The HTTP specification [99] states that GET requests should be both *safe* and *idempotent*. The former means that a GET request should only result in the retrieval of a resource, not in any other action with consequences on the server-side. Idempotence means that no matter how many times a GET request is sent, the side-effects of these requests remain the same as for a single request. Unfortunately, many applications violate the correct semantics of a GET request, resulting in security problems later on [69]. A very common example of an unsafe GET request is a logout action, which clearly has more effects than retrieving a resource.

POST Method

A POST request is used to submit data to the server. In a web application context, this typically means the result of a form submission, such as an authentication form, a message to post to a forum, etc. The resource that handles the request is identified by the URI, and the submitted data is typically embedded as the payload. Listing A.5 shows a POST request generated by submitting an authentication form.

```
1  https://secure.example.com/

2  POST / HTTP/1.1
3  Host: secure.example.com
4  User-Agent: Mozilla/5.0 (X11; Linux i686; rv:19.0) Gecko/20100101 Firefox/19.0
5  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
6  Accept-Language: en-US,en;q=0.5
7  Accept-Encoding: gzip, deflate
8  Referer: https://secure.example.com/
9  Connection: keep-alive
10 Content-Type: application/x-www-form-urlencoded
11 Content-Length: 55
12    _action=login&_timezone=2&_url=&_user=test&_pass=secret
```

Listing A.5: An example of a POST request.

Since POST requests are intended to change the application state at the server-side, the client needs to handle them with consideration. For example, browsers will typically require user confirmation before re-sending a POST request. Following redirects with POST requests is also restricted, as discussed in Section A.4.3.

Other Methods

Other than GET and POST, the HTTP specification mentions six other methods. They are not that commonly used, but they are briefly covered below.

HEAD — The HEAD method acts in the same way as the GET method, but does not return the content of a resource, only the response metadata, such as headers. This method is used to check the existence or modification time of a resource.

OPTIONS	The OPTIONS method allows the client to ask information about the communication capabilities of the server. This is not used often in traditional web applications, but new technologies [253] use this to check whether a certain request is safe to execute or not.
PUT & DELETE	The PUT and DELETE methods are intended for direct manipulation of the resource identified by the URI. A PUT request asks the server to store the payload under the URI's location, and a DELETE requests asks to delete the identified resource. In practice, web applications rarely use either of these methods.
CONNECT	The CONNECT method allows the use of a proxy for a specific request.
TRACE	With the TRACE method, a server simply echoes back all request data, intended for debugging purposes.

As well as the HTTP methods defined in the standard, a web developer can also define custom HTTP methods with their corresponding behavior. An example of such an extension to HTTP is the WebDAV protocol [259], which introduces methods such as MKCOL, PROPFIND, LOCK, UNLOCK, etc.

A.4.2 HTTP Headers

Both HTTP requests and responses support variable fields called *headers*, in the format *Name: Value*. The presence and value of these headers determine the meaning of requests and responses, such as specifying the content type, or enabling functionality such as redirecting the browser to a different location, requesting additional authentication from the browser, etc. Browsers, servers and web applications are not limited to the standardized header fields, but can also define custom headers to enable new or application-specific functionality.

In this section, we cover several relevant request and response headers as defined in the HTTP specification, and refer to the available documentation for the details on all available headers [99].

Request Headers

The HTTP specification defines several request headers, supported by major browsers in the Web. In addition to these HTTP-specific headers, other web technologies, such as *Cross-Origin Resource Sharing* [253] or *Do Not Track* [101] often define their own request headers, for which support is added when the browsers adopt these technologies. Web applications sending requests from JavaScript, through the XMLHttpRequest object, also have the possibility to add custom headers to the outgoing request. Below, we discuss four request headers defined in the HTTP specification, relevant for the remainder of this document.

User-Agent The User-Agent request header specifies which type of user agent is making the request, typically accompanied by additional information, such as a specific version, the underlying operating system, etc. An example value of this header can be seen in Listing A.4 and Listing A.5.

Referer The Referer header tells the server where the request originated from. For example, if the user clicks on a link in a document, the URI of this document will be the value of the Referer header. This enables web servers to keep track of referring web sites, but also poses a potential privacy violation with respect to the user. In specific sensitive cases, the browser does not add a Referer header (e.g., when coming from a secured web site).

Cookie A cookie is a server-provided key-value pair, stored by the client. Using the Cookie request header, the client is able to send these values to the server, allowing the server to add stateful information to the request. Cookies sent in the Cookie header must have been set by the server using the Set-Cookie header. Cookies have somewhat complex behavior, which is explained in detail in Section A.5.

Authorization The Authorization header allows the client to provide authentication information, such as a username and a password. The Authorization header is an application-independent mechanism, supported by all browsers, but rarely used. The credentials used by the browser can be entered through a popup dialog, or as part of the URI. The value of the header depends on the scheme that is used. Two well-known schemes are the Basic scheme, where the username and password are base64 encoded, and the Digest scheme, which applies a hash function to the credentials before sending them. An example header using the Basic scheme: *Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==*.

From a server's perspective, request headers should be treated as potentially untrusted, since no guarantees about their origin can be given. A header can be added by the browser or the web application, but the entire request can also be forged by an adversary, as will be explained in Part II of this document.

Response Headers

With a similar structure to request headers, several response headers have been standardized by the HTTP specification or in the specifications of other web technologies [253, 216]. Web applications running on the server also have the possibility to override headers, or even add custom headers. Naturally, browsers do not understand custom headers out of the box, but client-side application code can access custom response headers through the XMLHttpRequest object. Below, we briefly cover four relevant response headers, as defined by the HTTP specification [99].

Content-Type The Content-Type header allows the server to specify the MIME type of the content in the response, as well as the character encoding (charset) used. A Content-Type header indicating an HTML response with UTF-8 character encoding has the following value: *Content-Type: text/html; charset=utf-8*. The Content-Type header is optional, and in its absence, the browser will independently attempt to determine the content type of the response: be aware, however, that this is ultimately a guess and may result in the content being rendered incorrectly or being dumped into a file rather than displayed as expected.

Location The Location header can be used to redirect the browser to a different location to retrieve the requested content, by supplying the new URI. Example uses of redirects through the Location header are moved documents, URI rewriting techniques or URI shortening services. A Location header has to be combined with an appropriate response code, as will be explained in the next section.

Set-Cookie The Set-Cookie header can be used to store key-value pairs on the client. Cookies stored on the client are automatically attached with the Cookie header to any future requests sent to the same domain, hence can be used to share state among different requests. Cookies have somewhat complex behavior, which is explained in detail in Section A.5.

WWW-Authenticate Whenever the browser requests a resource that requires additional authentication, the WWW-Authenticate header is sent in the response, accompanied by the 401 response code. The header contains a challenge that indicates the authentication scheme (e.g., Basic or Digest), and potentially relevant parameters. The WWW-Authenticate response header is closely associated with the Authorization request header.

A.4.3 HTTP Response Codes

Every HTTP response has a status code, indicating the result of the server's attempt to understand and satisfy the request. The HTTP specification defines numerous response codes, and custom response codes are valid as well [99]. The HTTP specification defines 5 classes of response codes, indicated by the first digit of the three-digit response code. The trailing two digits have no special meaning, and simply define a status code.

Below, we briefly discuss the four common classes, and the most relevant status codes within each class. One class (the 1xx codes) is rarely used, and therefore not covered here.

2xx - Success A 2xx status code indicates that the request was handled successfully. Most common is 200, indicating a handled request with the results embedded in the response. Alternatively, a 204 code indicates that the request is handled, but no content is embedded in the request.

3xx - Redirection A 3xx status code means that the resource needs to be retrieved from a different location, defined by the Location response header. The exact status code defines the reason to look elsewhere, such as *301 Moved Permanently* or *303 See Other*. When following the redirect, modern browsers change the method of a POST request to GET, and remove the payload of the request.

4xx - Client Error A 4xx status code indicates problematic behavior by the client, such as requesting a non-existent resource (404), trying to access a resource without authorization (401), or not being allowed to access a resource (403).

5xx - Server Error The 5xx class is reserved to indicate processing errors at the server-side. Most common is the *500 Internal Server Error* status code, indicating a general problem at the server-side during the processing of the request. One example that often leads to these kind of errors are unhandled exceptions in the server software, causing the default handler to return an error message to the client.

A.4.4 HTTPS: HTTP with Security

HTTPS is the combination of HTTP and SSL/TLS, where the latter adds entity authentication, confidentiality and integrity to the underlying HTTP connection, effectively securing the communication between both parties. The use of HTTPS generally thwarts attacks on the network level, such as eavesdropping or man-in-the-middle attacks, preventing the theft or manipulation of sensitive information, such as authentication credentials or payment information.

This section explains how HTTPS works in detail, starting with SSL/TLS and its infrastructure, including certificates and certificate authorities. In the second part of this section, we discuss the impact of HTTPS on the Web, the changes in introduces to the plaintext HTTP protocol, as well as the advantages and disadvantages.

Transport Layer Security

Technically, HTTPS consists of the traditional HTTP protocol, deployed on top of a network connection secured by TLS or its predecessor SSL [77] (further referred to as TLS). Setting up a TLS connection requires some initialization steps, performed during a networking protocol "handshake" (mutual exchange of messages). The handshake ensures that both parties agree on a cipher suite to use, and exchange the required parameters, such as digital security certificates.

The security features that TLS can offer depend on the configuration parameters used in the handshake, but conventionally, a TLS connection set up with the default cipher suite and a valid server certificate can offer unforgeable entity authentication of the server, together with

confidentiality and integrity of the data sent over the connection. Entity authentication of the client can be achieved by having the client present a valid certificate as well, for example using an electronic identity card [67].

The security guarantees offered by TLS heavily depend on the use of correct certificates at the server side, and optionally at the client-side as well. These in turn rely on the cryptographic mechanism known as *public key encryption*. Using public key encryption involves creating associated public and private key pairs. A certificate is handed out by a Certificate Authority (CA), and contains the public key of a party and a signature of the CA, asserting that the public key has been vetted as belonging to the party that holds the corresponding private key. For example, on the Web, a certificate can be requested with any CA for `secure.example.com` by its administrator using the public key of his privately generated key pair. The CA will typically verify that the requesting party actually controls the domain, for example by sending a confirmation mail to the domain's `info` or `webmaster` address. Once the verification passes, the CA signs the provided public key with his own CA key, and provides the administrator with a signed certificate, vetting the authenticity of the provided public key and the verified domain name.

Whenever a browser connects to a secure server, the server will present its certificate, signed by a CA. The browser is able to verify the signature, in order to ensure that the public key presented by the server is actually approved by a CA, and not some key generated by an attacker. Verification of the signature is achieved using the CA's public key, which in turn needs to be vetted by a CA. This chain continues, until a root CA is reached. Since no-one can guarantee the validity of a root CA's key, they are built-in into the browser, effectively ending the chain there. As can be seen the whole hierarchy of trust is dependent on having reliable and long-standing root CAs.

Whenever a browser encounters an invalid or unverifiable certificate, the user is typically warned about some potential insecure situation. Failure to verify a certificate can have many reasons, some harmless, some serious. For example, if a certificate is expired, browsers will complain. Similarly, if a certificate is signed by a CA that is not linked through a verifiable chain to a root CA, a browser cannot verify the validity. The situation often occurs with so-called self-signed certificates, which are signed by the same person that created the keys. While such certificates can be used to set up a TLS connection, they can be easily forged by an attacker, enabling man-in-the-middle attacks breaching the security guarantees offered by TLS.

Impact on the Web

From the point of view of the HTTP protocol, when using HTTPS messages are now sent over a TLS-secured TCP connection instead of a plain TCP connection. This has some consequences for the HTTP protocol and web content. First, the scheme of secure URIs is HTTPS, causing secure and non-secure content on the same domain to be from different origins. A second consequence is the omission of the `Referer` header whenever a request from an HTTPS resource to an HTTP resource is made. Additionally, cookies can be tied to secure connections only, making them invisible for non-secure resources.

Obviously, HTTPS has many advantages over non-secure connections. The security guarantees offered by TLS help in protecting sensitive information, prevent the theft of authentication credentials and help ensuring the identity of the involved parties.

Unfortunately, TLS also has some disadvantages. A first disadvantage surrounds the CA system, with the trust placed in a long list of root CAs, and the hassle introduced by generating key pairs, requesting certificates, paying CAs, etc. Especially with the recent compromises of CAs [262] and the resulting consequences, the system has received some criticism. Other disadvantages are the circumvention of web caches, which cannot deal with encrypted content, and the problem with mixed content, where warnings are generated whenever HTTPS resources include content from non-HTTPS origins, inviting attackers through the front door.

A.5 Sharing State between Client and Server

Cookies are small chunks of data, provided by a web application, stored by a browser. On every request to the web application, the browser will attach the stored cookies, allowing the web application to store temporary state in a cookie. Cookies are typically used for session management (See Section 4.2), or to store temporary session information, such as user preferences or a shopping cart. Cookies have gained a questionable reputation due to their use as a tracking mechanism (See Intermezzo 8).

In the default cookie behavior [22], the web application sends a new cookie to the client using the *Set-Cookie* response header (e.g., containing a new session identifier when the client starts a new session with the server), after which the browser stores the cookie and the web application's domain in the so-called *cookie jar*. When a new request to a server is triggered, the browser searches the cookie jar for all cookies associated with the target domain, and attaches them to the request using the `Cookie` header. Cookies behave similarly for URIs based on numerical IP addresses, where each IP address acts as a unique domain name.

The server can specify several optional attributes when setting a cookie. A first attribute is the lifetime, indicating how long the cookie should be kept by the browser. If the lifetime is omitted, the cookie only lives as long as the browser session, and is deleted when the browser is closed. A second attribute is the domain attribute, which allows a web application to specify the domain a cookie should be associated with. If present, an application can limit the cookie scope to a specific subdomain of the application, or broaden the scope to all subdomains of the application, with the registered domain as the maximum level. If omitted, the cookie is associated with the domain of the resource that set the cookie. A third attribute allows the application to specify a path for the cookie, telling the browser to only attach the cookie when a request targets a resource within that specific path. Scoping a cookie to a specific path eases cookie management when hosting multiple applications under the same domain, but does not prevent these applications from accessing each others cookies through JavaScript.

A second set of relatively recently introduced cookie attributes, consisting of *HttpOnly* and *Secure*, address two important security issues. The former, HttpOnly, restricts a cookie for use in HTTP headers only, preventing any JavaScript-based access. This basically prevents a malicious script with the application's security context from stealing or manipulating sensitive cookies. The latter, Secure, restricts the use of a cookie to HTTPS connections only. This effectively prevents an attacker from stealing such cookies by eavesdropping on unencrypted traffic, or even from tricking the browser into revealing the cookies on a forged HTTP request.

Cookies, their attributes and the way different browsers handle them are complex topics, with lots of ambiguities. We cannot cover all peculiarities in this document, and refer to specialized literature for further reading [265].

Bibliography

[1] Privoxy. *Online at* http://www.privoxy.org, 2013.

[2] RefControl. *Online at* https://addons.mozilla.org/en-us/firefox/addon/refcontrol/, 2013.

[3] B. Aboba, D. Simon, and P. Eronen. Extensible authentication protocol (eap) key management framework. *RFC Proposed Standard (RFC 5247)*, 2008.

[4] B. Adida. Beamauth: two-factor web authentication with a bookmark. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 48–57. ACM, 2007.

[5] B. Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceedings of the 17th international conference on World Wide Web*, pages 517–524. ACM, 2008.

[6] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 1–10, 2012.

[7] D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304. IEEE, 2010.

[8] W. Alcorn. Browser Exploitation Framework (BeEF). *Online at* http://beefproject.com, 2013.

[9] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. Schuldt. On the security of rc4 in tls and wpa. In *USENIX Security Symposium*, 2013.

[10] N. J. AlFardan and K. G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[11] Apache Software Foundation. Apache tomcat - migration guide. *Online at* http://tomcat.apache.org/migration-7.html, 2013.

[12] P. Arzamendi. Exploiting vulnerabilities in multifunction printers. *Online at* http://www.slideshare.net/403Labs/exploiting-vulnerabilities-in-multifunction-printers, 2011.

[13] J. Ashkenas. backbone.js. *Online at* http://backbonejs.org/, 2013.

[14] Associated Press. New nuclear sub is said to have special eavesdropping ability. *Online at* http://www.nytimes.com/2005/02/20/politics/20submarine.html?_r=0, 2005.

[15] J. Aubourg, J. Song, and H. R. M. Steen. XMLHttpRequest. *W3C Working Draft*, 2012.

[16] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *NDSS*, 2011.

[17] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. Vex: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, volume 10, pages 339–354, 2010.

[18] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 247–262. IEEE, 2012.

[19] R. Barnes. Use cases and requirements for JSON object signing and encryption (JOSE). *IETF Internet Draft*, 2013.

[20] D. Baron. Preventing attacks on a user's history through css :visited selectors. *Online at* `http://dbaron.org/mozilla/visited-privacy`, 2010.

[21] A. Barr. Google may ditch 'cookies' as online ad tracker. *Online at* `http://www.usatoday.com/story/tech/2013/09/17/google-cookies-advertising/2823183/`, 2013.

[22] A. Barth. Http state management mechanism. *IETF Proposed Standard*, 2011.

[23] A. Barth. The Web Origin Concept. *RFC 6454*, 2011.

[24] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010.

[25] A. Barth and C. Jackson. Protecting browsers from frame hijacking attacks. *Online at* `http://seclab.stanford.edu/websec/frames/navigation/`, 2008.

[26] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

[27] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[28] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100, 2010.

[29] M. Belshe and R. Peon. Spdy protocol. *IETF Internet Draft*, 2012.

[30] M. Belshe, M. Thomson, A. Melnikov, and R. Peon. Hypertext transfer protocol version 2.0. *IETF Internet Draft*, 2013.

[31] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan. WebRTC 1.0: Real-Time Communication Between Browsers. *W3C Working Draft*, 2012.

[32] S. Berinato. Data breach notification laws, state by state. *Online at* `http://www.csoonline.com/article/221322/cso-disclosure-series-data-breach-notification-laws-state-by-state`, 2008.

[33] R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, S. Pfeiffer, and I. Hickson. HTML 5.1 Specification. *W3C Working Draft*, 2013.

[34] R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, S. Pfeiffer, and I. Hickson. HTML 5.1 Specification - The sandbox Attribute. *W3C Working Draft*, 2013.

[35] R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, S. Pfeiffer, and I. Hickson. Html5 - a vocabulary and associated apis for html and xhtml. *W3C Editor's Draft*, 2013.

[36] T. Berners-Lee, R. T. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. *RFC Internet Standard (RFC 3986)*, 2005.

[37] A. Bhargav-Spantzel, A. C. Squicciarini, S. Modi, M. Young, E. Bertino, and S. J. Elliott. Privacy preserving multi-factor authentication with biometrics. *Journal of Computer Security*, 15(5):529–560, 2007.

[38] J. Blagdon. W3c and whatwg finalize split on html5 spec, forking 'unlikely'. *Online at* `http://www.theverge.com/2012/7/22/3175248/html5-fork-w3c-whatwg`, 2012.

[39] S. Block and A. Popescu. DeviceOrientation Event Specification. *W3C Working Draft*, 2011.

[40] H. Bojinov, E. Bursztein, and D. Boneh. Xcs: cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431. ACM, 2009.

[41] J. Bonneau and S. Preibusch. The password thicket: Technical and market failures in human authentication on the web. In *WEIS*, 2010.

[42] A. Bortz, A. Barth, and A. Czeskis. Origin cookies: Session integrity for web applications. *Web 2.0 Security and Privacy (W2SP)*, 2011.

[43] J. Brookman, H. West, S. Harvey, and E. Newland. Tracking Compliance and Scope. *W3C Working Draft*, 2013.

[44] D. C. Burnett and N. Anant. Media Capture and Streams. *W3C Working Draft*, 2012.

[45] J. Burns. Cross site reference forgery. Technical report, Information Security Partners, 2005.

[46] E. Butler. Firesheep. *Online at* `http://codebutler.com/firesheep`, 2010.

[47] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium*, 2011.

[48] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX Conference on Security*, 2012.

[49] D. Carlisle, P. Ion, and R. Miner. Mathematical Markup Language (MathML) Version 3.0 . *W3C Recommendation*, 2010.

[50] T. Çelik, E. J. Etemad, D. Glazman, I. Hickson, P. Linss, and J. Williams. Selectors Level 3. *W3C Recommendation*, 2011.

[51] I. E. D. Center. Making HTML safer: details for toStaticHTML (Windows Store apps using JavaScript and HTML). *Online at* `http://msdn.microsoft.com/en-us/library/ie/hh465388.aspx`, 2012.

[52] CERT. Microsoft Internet Explorer buffer overflow in PNG image rendering component. *Vulnerability Note VU#189754*, 2005.

[53] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 227–238. ACM, 2011.

[54] K. Z. Chen, G. Gu, J. Zhuge, J. Nazario, and X. Han. Webpatrol: Automated collection and replay of web-based malware scenarios. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 186–195. ACM, 2011.

[55] P. Chen, N. Nikiforakis, L. Desmet, and C. Huygens. A dangerous mix: Large-scale analysis of mixed-content websites. In *Proceedings of the 16th Information Security Conference (ISC 2013)*, 2013.

[56] M. Coates. Putting Users in Control of Plugins. *Online* `https://blog.mozilla.org/security/2013/01/29/putting-users-in-control-of-plugins/`, 2013.

[57] E. Commission et al. Directive 2002/58/ec of the european parliament and of the council of 12 july 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector. *Official Journal L*, 201(31):07, 2002.

[58] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. *RFC Proposed Standard (RFC 3280)*, 2008.

[59] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM, 2010.

[60] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, pages 33–48, 2011.

[61] A. Czeskis, M. Dietz, T. Kohno, D. Wallach, and D. Balfanz. Strengthening user authentication through opportunistic cryptographic identity assertions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 404–414. ACM, 2012.

[62] A. Czeskis, A. Moshchuk, T. Kohno, and H. J. Wang. Lightweight server support for browser-based csrf protection. In *Proceedings of the 22nd international conference on World Wide Web*, pages 273–284. International World Wide Web Conferences Steering Committee, 2013.

[63] D. Dahl and R. Sleevi. Web Cryptography API. *W3C Working Draft*, 2013.

[64] E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, and J. Watt. Scalable Vector Graphics (SVG) 1.1 (Second Edition). *W3C Recommendation*, 2011.

[65] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *USENIX Security Symposium*, pages 267–282, 2009.

[66] B. Damele and M. Stampar. Sqlmap. *Online at* `http://sqlmap.org/`, 2012.

[67] D. De Cock, C. Wolf, and B. Preneel. The belgian electronic identity card (overview). In *Sicherheit*, volume 77, pages 298–301, 2006.

[68] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen. Security of web mashups: a survey. In *Information Security Technology for Applications, 15th Nordic Conference in Secure IT Systems (NordSec 2010), LNCS*, pages 223–238, 2011.

[69] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *Engineering Secure Software and Systems*, pages 18–34. Springer, 2010.

[70] P. De Ryck, L. Desmet, W. Joosen, and F. Piessens. Automatic and precise client-side protection against csrf attacks. In *Computer Security–ESORICS 2011*, pages 100–116. Springer, 2011.

[71] P. De Ryck, L. Desmet, P. Philippaerts, and F. Piessens. A security analysis of next generation web standards. Technical report, European Network and Information Security Agency (ENISA), July 2011.

[72] P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. A security analysis of emerging web standards-html5 and friends, from specification to implementation. In *Proceedings of the International Conference on Security and Cryptography (SECRYPT)*, pages 257–262, 2012.

[73] P. De Ryck, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Serene: self-reliant client-side protection against session fixation. In *Distributed Applications and Interoperable Systems*, pages 59–72. Springer, 2012.

[74] S. E. Deering. Internet protocol, version 6 (ipv6) specification. *RFC Draft Standard (RFC 2460)*, 1998.

[75] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590. ACM, 2006.

[76] M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 382–391, 2009.

[77] T. Dierks. The transport layer security (TLS) protocol version 1.2. *RFC 5246*, 2008.

[78] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *USENIX Security Symposium*, pages 16–16, 2012.

[79] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[80] Django Software Foundation. Django. *Online at* `https://www.djangoproject.com/`, 2013.

[81] T. Dougan and K. Curran. Man in the browser attacks. *International Journal of Ambient Computing and Intelligence (IJACI)*, pages 29–39, 2012.

[82] T. Duebendorfer and S. Frei. Why silent updates boost security. *TIK, ETH Zurich, Tech. Rep*, 302, 2009.

[83] G. Dumpleton. mod_wsgi. *Online at* `http://code.google.com/p/modwsgi/`, 2013.

[84] T. Duong and J. Rizzo. BEAST - Here Come The XOR Ninjas. 2011.

[85] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, volume 6205 of *Lecture Notes in Computer Science*, pages 1–18. 2010.

[86] Electronic Frontier Foundation. Https everywhere. *Online at* `https://www.eff.org/https-everywhere`, 2013.

[87] EllisLab. CodeIgniter. *Online at* , 2013.

[88] EMC. RSA SecurID - Two-Factor Authentication Security Token. *Online at* `http://www.emc.com/security/rsa-securid.htm`, 2013.

[89] Ettercap Project. Ettercap home page. *Online at* `http://ettercap.github.io/ettercap/`, 2013.

[90] European Union Agency for Network and Information Security (ENISA). ENISA Threat Landscape, Mid-year 2013. *Online at* `https://www.enisa.europa.eu/activities/risk-management/evolving-threat-environment/enisa-threat-landscape-mid-year-2013/`, 2013.

[91] C. Evans, C. Palmer, and R. Sleevi. Public key pinning extension for HTTP. *IETF Internet Draft*, 2013.

[92] Evidon Inc. Ghostery. *Online at* `http://www.ghostery.com/`, 2013.

[93] Facebook. Facebook login. *Online at* `http://developers.facebook.com/docs/facebook-login/`, 2013.

[94] N. Falliere and E. Chien. Zeus: King of the bots. *Symantec Security Response (Online at* `http://courses.isi.jhu.edu/malware/papers/ZEUS.pdf` *)*, 2009.

[95] S. Farrell, P. Hoffman, and M. Thomas. HTTP Origin-Bound Authentication (HOBA). *IETF Internet Draft*, 2013.

[96] Federal Trade Commission. Ftc settlement puts an end to "history sniffing" by online advertising network charged with deceptively gathering data on consumers. *Online at* `http://ftc.gov/opa/2012/12/epic.shtm`, 2012.

[97] V. Fergal Glynn. Static Code Analysis. *Online at* `http://www.veracode.com/security/static-code-analysis`, 2013.

[98] D. Ferguson, J. Manico, and K. Wall. Forgot password cheat sheet. *Online at* `https://www.owasp.org/index.php/Forgot_Password_Cheat_Sheet`, 2013.

[99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *RFC 2616*, 1999.

[100] R. T. Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, 2000.

[101] R. T. Fielding and D. Singer. Tracking Preference Expression (DNT). *W3C Working Draft*, 2013.

[102] D. Florêncio, C. Herley, and B. Coskun. Do strong web passwords accomplish anything. *Proc. Usenix Hot Topics in Security*, 2007.

[103] C. S. Foundation. CakePHP: the rapid development php framework. *Online at* `http://cakephp.org/`, 2013.

[104] E. F. Foundation. Panopticlick. *Online at* `https://panopticlick.eff.org/`, 2013.

[105] B. Frank, I. Poese, Y. Lin, G. Smaragdakis, A. Feldmann, B. Maggs, J. Rake, S. Uhlig, and R. Weber. Pushing cdn-isp collaboration to the limit. *ACM SIGCOMM CCR*, 43(3), 2013.

[106] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication. *RFC Draft Standard (RFC 2617)*, 1999.

[107] S. Friedl and A. Popov. Transport layer security (tls) application layer protocol negotiation extension. *IETF Internet Draft*, 2013.

[108] B. S. Fung and P. P. Lee. A privacy-preserving defense mechanism against request forgery attacks. In *Trust, Security and Privacy in Computing and Communications*, pages 45–52. IEEE, 2011.

[109] F. Gadaleta, Y. Younan, and W. Joosen. Bubble: A javascript engine level countermeasure against heap-spraying attacks. In *Engineering Secure Software and Systems*, pages 1–17. Springer, 2010.

[110] F. Galiegue, K. Zyp, and G. Court. JSON schema: core definitions and terminology. *IETF Internet Draft*.

[111] Y. Gluck, N. Harris, and A. Prado. BREACH: Reviving the CRIME Attack. *Online at* `http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf`, 2013.

[112] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. Http extensions for distributed authoring–webdav, 1999.

[113] D. Goldman. The Internet's most dangerous sites. *Online at* `http://money.cnn.com/gallery/technology/security/2013/05/01/shodan-most-dangerous-internet-searches/index.html`, 2013.

[114] Google. AngularJS. *Online at* `http://angularjs.org/`, 2013.

[115] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Toward secure embedded web interfaces. In *USENIX Security Symposium*, 2011.

[116] A. C. Grant. Search for trust: An analysis and comparison of ca system alternatives and enhancements. 2012.

[117] J. Grossman. Intranet hacking attacks found in the wild. *Online at* `http://jeremiahgrossman.blogspot.be/2008/01/intranet-hacking-attacks-found-in-wild.html`, 2008.

[118] J. Grossman, B. Livshits, R. G. Bace, G. Neville-Neil, and C. Cole. Browser security case study: Appearances can be deceiving. *ACM Queue*, 10(11):30, 2012.

[119] S. Guarnieri and V. B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium*, pages 151–168, 2009.

[120] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 115–130. IEEE, 2011.

[121] S. Guha, B. Cheng, and P. Francis. Privad: Practical Privacy in Online Advertising. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[122] HAK5. Wifi pineapple. *Online at* `https://wifipineapple.com/`, 2013.

[123] W. Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, USA*, pages 13–15, 2006.

[124] P. A. Hallgren, D. T. Mauritzson, and A. Sabelfeld. Glasstube: a lightweight approach to web application integrity. In *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 71–82. ACM, 2013.

[125] E. Hammer-Lahav, D. Recordon, and D. Hardt. The oauth 2.0 authorization protocol. *Network Working Group Internet-Draft*, 2011.

[126] R. Hansen. Intranet hacking. *Online at* `http://www.sectheory.com/intranet-hacking.htm`, 2007.

[127] D. H. Hansson. Ruby on Rails. *Online at* `http://rubyonrails.org/`, 2013.

[128] N. Heath. Malicious Chrome and Firefox extensions found hijacking Facebook profiles. *Online at* `http://www.zdnet.com/malicious-chrome-and-firefox-extensions-found-hijacking-facebook-profiles-7000015277/`, 2013.

[129] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 760–771, 2012.

[130] D. Hepper. Gmail CSRF vulnerability explained. *Online at* `http://daniel.hepper.net/blog/2008/11/gmail-csrf-vulnerability-explained/`, 2008.

[131] I. Hickson. HTML5 Web Messaging. *W3C Candidate Recommendation*, 2012.

[132] I. Hickson. Server-Sent Events. *W3C Candidate Recommendation*, 2012.

[133] I. Hickson. The WebSocket API. *W3C Candidate Recommendation*, 2012.

[134] I. Hickson. Web Workers. *W3C Candidate Recommendation*, 2012.

[135] I. Hickson. [whatwg] administrivia: Update on the relationship between the whatwg html living standard and the w3c html5 specification. *Online at* `http://lists.w3.org/Archives/Public/public-whatwg-archive/2012Jul/0119.html`, 2012.

[136] I. Hickson. Web Storage. *W3C Proposed Recommendation*, 2013.

[137] J. Hodges, C. Jackson, and A. Barth. HTTP strict transport security (HSTS). *RFC Proposed Standard (RFC 6797)*, 2012.

[138] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: attacks and defenses. In *USENIX Security Symposium*, pages 22–22, 2012.

[139] J. Ichnowski and J. Manico. Owasp's java xml templates. *Online at* `http://code.google.com/p/owasp-jxt/`, 2013.

[140] J. Ichnowski, J. Manico, and J. Long. Owasp java encoder project. *Online at* `https://www.owasp.org/index.php/OWASP_Java_Encoder_Project`, 2013.

[141] L. Ingram and M. Walfish. Treehouse: Javascript sandboxes to help web developers help themselves. In *Proceedings of the USENIX annual technical conference*, 2012.

[142] ISO. *Information Processing, Text and Office Systems, Standard Generalized Markup Language (SGML) = Traitement de l'information, systèmes bureautiques, langage standard généralisé de balisage (SGML). First edition, 1986-10-15*. International Organization for Standardization, Geneva, Switzerland, 1986. International Standard ISO 8879-1986. Federal information processing standard; FIPS PUB 152.

[143] ITSecTeam Security Research. Havij advanced SQL injection. *Online at* `http://www.itsecteam.com/products/havij-v116-advanced-sql-injection/`, 2012.

[144] C. Jackson and A. Barth. ForceHTTPS: protecting high-security web sites from network attacks. In *Proceedings of the 17th international conference on World Wide Web*, pages 525–534. ACM, 2008.

[145] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from dns rebinding attacks. *ACM Transactions on the Web (TWEB)*, 3(1):2, 2009.

[146] M. Jakobsson. *The Death of the Internet*. Wiley, 2012.

[147] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 270–283, 2010.

[148] M. Johns. Sessionsafe: Implementing xss immune session handling. In *Computer Security–ESORICS 2006*, pages 444–460. Springer, 2006.

[149] M. Johns, B. Braun, M. Schrank, and J. Posegga. Reliable protection against session fixation attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1531–1537. ACM, 2011.

[150] M. Johns, S. Lekies, B. Braun, and B. Flesch. Betterauth: Web authentication revisited. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 169–178. ACM, 2012.

[151] M. Johns, S. Lekies, and B. Stock. Eradicating dns rebinding with the extended same-origin policy. In *USENIX Security Symposium*, 2013.

[152] M. Johns and J. Winter. Requestrodeo: Client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference*, 2006.

[153] Y. Katz. Handlebars.js: Minimal templating on steroids. *Online at* *http://handlebarsjs.com/*, 2013.

[154] G. Keizer. Google builds stronger Flash sandbox in Chrome. *Online at* *http://www.computerworld.com/s/article/9230094/Google_builds_stronger_Flash_sandbox_in_Chrome*, 2012.

[155] S. M. Kelly. LastPass passwords exposed for some internet explorer users. *Online at* *http://mashable.com/2013/08/19/lastpass-password-bug/*, 2013.

[156] A. King. Club nintendo japan hacked, user details could be compromised. *Online at* *http://wiiudaily.com/2013/07/club-nintendo-japan-hacked/*, 2013.

[157] G. Klyne. Uniform Resource Identifier (URI) Schemes. *Online at* *http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml*, 2013.

[158] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 443–457. IEEE, 2012.

[159] A. Kostiainen. Vibration API. *W3C Last Call Working Draft*, 2013.

[160] E. Kovacs. CSRF Vulnerability in eBay Allows Hackers to Hijack User Accounts. *Online at* *http://news.softpedia.com/news/CSRF-Vulnerability-in-eBay-Allows-Hackers-to-Hijack-User-Accounts-Video-383316.shtml*, 2013.

[161] E. Kovacs. Vodafone germany hacked, details of 2 million users stolen. *Online at* *http://news.softpedia.com/news/Vodafone-Germany-Hacked-Details-of-2-Million-Users-Stolen-382458.shtml*, 2013.

[162] G. Kumparak. Drupal.org hacked, user details exposed and reset. *Online at* `http://techcrunch.com/2013/05/29/drupal-org-hacked-user-details-exposed-and-reset/`, 2013.

[163] M. Lamouri. The Network Information API. *W3C Working Draft*, 2012.

[164] A. Langley. Overclocking ssl. *Online at* `https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html`, 2010.

[165] A. Langley. ChaCha20 and Poly1305 based Cipher Suites for TLS. *IETF Internet Draft*, 2013.

[166] P. Laskov and N. Šrndić. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 373–382. ACM, 2011.

[167] LastPass.com. LastPass. *Online at* `https://lastpass.com`, 2013.

[168] B. Laurie, A. Langley, and E. Kasper. Certificate transparency. *RFC Experimental (RFC 6962)*, 2013.

[169] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 3 Core Specification. *W3C Recommendation*, 2004.

[170] T. Leithead, J. Rossi, D. Schepers, B. Höhrmann, P. Le Hégaret, and T. Pixley. Document Object Model (DOM) Level 3 Events Specification. *W3C Working Draft*, 2012.

[171] S. Lekies and M. Johns. Lightweight integrity protection for web storage-driven content caching. *Web 2.0 Security and Privacy (W2SP)*, 2012.

[172] S. Lekies, W. Tighzert, and M. Johns. Towards stateless, client-side driven cross-site request forgery protection for web applications. In *Sicherheit*, pages 111–121, 2012.

[173] P. Leon, B. Ur, R. Shay, Y. Wang, R. Balebako, and L. Cranor. Why johnny can't opt out: a usability evaluation of tools to limit online behavioral advertising. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 589–598, 2012.

[174] B. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying Web Browser Extension-sâĂŹ Compliance with Private-Browsing Mode. In *European Symposium on Research in Computer Security*, pages 57–74. Springer, 2013.

[175] J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *Information Security Technology for Applications*, pages 239–255. Springer, 2012.

[176] G. Maone. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience! *Online at* `http://noscript.net/`, 2013.

[177] G. Maone. NoScript Application Boundaries Enforcer (ABE). *Online at* `http://noscript.net/abe/`, 2013.

[178] G. Maone, D. L.-S. Huang, T. Gondrom, and B. Hill. User Interface Safety Directives for Content Security Policy. *W3C Working Draft*, 2012.

[179] M. Marlinspike. New tricks for defeating ssl in practice. *BlackHat DC, February*, 2009.

[180] M. Marlinspike. Sslstrip. *Online at* `http://www.thoughtcrime.org/software/sslstrip/`, 2009.

[181] B. Martin, M. Brown, A. Paller, and S. Christey. Cwe/sans top 25 most dangerous programming errors. *Online at* `http://cwe.mitre.org/top25/pdf/2009_cwe_sans_top_25.pdf`, 2009.

[182] M. Masnick. FLYING PIG: The NSA Is Running Man In The Middle Attacks Imitating Google's Servers. *Online at* `http://www.techdirt.com/articles/20130910/10470024468/flying-pig-nsa-is-running-man-middle-attacks-imitating-googles-servers.shtml`, 2013.

[183] J. Mayer and J. Mitchell. Third-party web tracking: Policy and technology. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 413–427, 2012.

[184] A. M. McDonald. Cookie clearinghouse faq. *Online at* `http://cch.law.stanford.edu/faq/`, 2013.

[185] N. Mehta, J. Sicking, E. Graff, A. Popescu, J. Orlow, and J. Bell. Indexed Database API. *W3C Last Call Working Draft*, 2013.

[186] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 481–496, 2010.

[187] M. S. Miller. Secure EcmaScript 5. *Online at* `http://code.google.com/p/es-lab/wiki/SecureEcmaScript`, 2011.

[188] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript. *Online at* `http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf`, 2008.

[189] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage. Re: Captchas-understanding captcha-solving services in an economic context. In *USENIX Security Symposium*, volume 10, 2010.

[190] S. J. Murdoch. Hardened stateless session cookies. In *Security Protocols XVI*, pages 93–101. Springer, 2011.

[191] F. Muttis and A. Sacco. HTML5 heap sprays. *Online at* `http://exploiting.files.wordpress.com/2012/10/html5-heap-spray.pdf`, 2012.

[192] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 736–747, 2012.

[193] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Security and Privacy*, 2013.

[194] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. Sessionshield: lightweight protection against session hijacking. In *Engineering Secure Software and Systems*, pages 87–100. Springer, 2011.

[195] N. Nikiforakis, S. Van Acker, F. Piessens, and W. Joosen. Exploring the ecosystem of referrer-anonymizing services. In *Privacy Enhancing Technologies*, pages 259–278. Springer, 2012.

[196] N. Nikiforakis, Y. Younan, and W. Joosen. Hproxy: Client-side detection of ssl stripping attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 200–218. Springer, 2010.

[197] Oracle. Overview of java security models. *Online at* `http://docs.oracle.com/cd/E12839_01/core.1111/e10043/introjps.htm`, 2003.

[198] W. Palant and T. Faida. Adblock plus. *Online at* `https://adblockplus.org/en/chrome`, 2013.

[199] R. Pelizzi and R. Sekar. A server-and browser-transparent csrf defense for web 2.0 applications. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 257–266. ACM, 2011.

[200] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proc. of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, 2009.

[201] A. Popescu. Geolocation API Specification. *W3C Proposed Recommendation*, 2012.

[202] J. Prins. Diginotar certificate authority breach – 'operation black tulip'. *Fox-IT*, 2011.

[203] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.0 Specification. *W3C Recommendation*, 1997.

[204] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 Specification. *W3C Recommendation*, 1999.

[205] A. Ranganathan and J. Sicking. File API. *W3C Working Draft*, 2012.

[206] Rapid7. Metasploit. *Online at* `http://www.metasploit.com/`, 2013.

[207] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[208] D. Raywood. Linux forum ubuntu hacked and user details collected. *Online at* `http://www.scmagazineuk.com/linux-forum-ubuntu-hacked-and-user-details-collected/article/303896/`, 2013.

[209] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Transport layer security (tls) renegotiation indication extension. *RFC Proposed Standard (RFC 5746)*, 2010.

[210] M. Richardson and M. Bilenko. Predictive client-side profiles for personalized advertising. In *Proceedings of the 17th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD-2011)*, 2011.

[211] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39. ACM, 2010.

[212] I. Ristic. Internet ssl survey 2010. *BlackHat DC, July*, 2010.

[213] J. Rizzo and T. Duong. The CRIME Attack. *Online at* `https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit?pli=1#slide=id.g1d134dff_1_222`, 2012.

[214] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the 14th Usenix Security Symposium*, volume 1998, 2005.

[215] D. Ross. IE 8 XSS Filter Architecture / Implementation. *Online at* *http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx*, 2008.

[216] D. Ross and T. Gondrom. HTTP Header Field X-Frame-Options. *IETF Internet Draft*, 2013.

[217] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *Web 2.0 Security and Privacy (W2SP)*, 2, 2010.

[218] G. Rydstedt, B. Gourdin, E. Bursztein, and D. Boneh. Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies*, pages 1–8. USENIX Association, 2010.

[219] A. P. Sabzevar and A. Stavrou. Universal multi-factor authentication using graphical passwords. In *Signal Image Technology and Internet Based Systems, 2008. SITIS'08. IEEE International Conference on*, pages 625–632. IEEE, 2008.

[220] J. Samuel and B. Zhang. Requestpolicy: Increasing web browsing privacy through control of cross-site requests. In *Privacy Enhancing Technologies*, pages 128–142. Springer, 2009.

[221] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 587–600, 2011.

[222] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[223] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 601–614, 2011.

[224] S. Schoen and E. Galperin. Iranian man-in-the-middle attack against google demonstrates dangerous weakness of certificate authorities. *Online at* *https://www.eff.org/deeplinks/2011/08/iranian-man-middle-attack-against-google*, 2011.

[225] S. Schultze. Web browser security user interfaces: Hard to get right and increasingly inconsistent. *Online at* *https://freedom-to-tinker.com/blog/sjs/web-browser-security-user-interfaces-hard-get-right-and-increasingly-inconsistent/*, 2011.

[226] M. Schwartz. Hackers target java 6 with security exploits. *Online at* *http://www.informationweek.com/security/vulnerabilities/hackers-target-java-6-with-security-expl/240160443*, 2013.

[227] H. E. Security. HP Fortify Static Code Analyzer (SCA). *http://www.hpenterprisesecurity.com/products/hp-fortify-software-security-center/hp-fortify-static-code-analyzer*, 2013.

[228] R. Shay, P. G. Kelley, S. Komanduri, M. L. Mazurek, B. Ur, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. Correct horse battery staple: exploring the usability of system-assigned passphrases. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, 2012.

[229] Y. Sheffer. Recommendations for Secure Use of TLS and DTLS. *IETF Internet Draft*, 2013.

[230] O. Shezaf, J. Grossman, and R. Auger. Web hacking incidents database. *Online at http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Database*, 2013.

[231] SHODAN. SHODAN - Computer Search Engine. *Online at www.shodanhq.com*, 2013.

[232] R. Siles. Session management cheat sheet - renew the session id after any privilege level change. *Online at https://www.owasp.org/index.php/Session_Management_Cheat_Sheet#Renew_the_Session_ID_After_Any_Privilege_Level_Change*, 2013.

[233] R. Siles. Session management cheat sheet - session id properties. *Online at https://www.owasp.org/index.php/Session_Management_Cheat_Sheet#Session_ID_Properties*, 2013.

[234] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 463–478, 2010.

[235] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *Network and Distributed System Security Symposium*, 2013.

[236] D. Song. dsniff. *Online at http://www.monkey.org/~dugsong/dsniff/*, 2000.

[237] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, pages 921–930, 2010.

[238] B. Sterne and A. Barth. Content Security Policy 1.0. *W3C Candidate Recommendation*, 2012.

[239] J. Steven. Password storage cheat sheet. *Online at https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet*, 2013.

[240] P. Stone. Pixel perfect timing attacks with html5. *Online at http://www.contextis.com/files/Browser_Timing_Attacks.pdf*, 2013.

[241] B. Stone-Gross, R. Abman, R. A. Kemmerer, C. Kruegel, D. G. Steigerwald, and G. Vigna. The underground economy of fake antivirus software. In *Economics of Information Security and Privacy III*, pages 55–78. Springer, 2013.

[242] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. In *USENIX Security Symposium*, 2011.

[243] M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security Symposium*, pages 371–388, 2010.

[244] The Guardian. Edward Snowden. *Online at http://www.theguardian.com/world/edward-snowden*, 2013.

[245] The jQuery Foundation. jQuery. *Online at http://jquery.com/*, 2013.

[246] M. Toussain and C. Shields. Subterfuge. *Online at http://kinozoa.com/blog/subterfuge-documentation/*, 2013.

[247] D. Turner. Ambient Light Events. *W3C Working Draft*, 2012.

[248] E. Uhrhane. File API: Directories and System. *W3C Working Draft*, 2012.

[249] E. Uhrhane. File API: Writer. *W3C Working Draft*, 2012.

[250] B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. L. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. How does your password measure up? the effect of strength meters on password creation. In *Proceedings of the 21st USENIX conference on Security symposium*, 2012.

[251] US-CERT. Oracle Java Contains Multiple Vulnerabilities. *Alert (TA13-064A)*, 2013.

[252] S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 307–316, 2011.

[253] A. van Kesteren. Cross-Origin Resource Sharing. *W3C Candidate Recommendation*, 2013.

[254] A. van Kesteren and J. Gregg. Web Notifications. *W3C Working Draft*, 2012.

[255] A. van Kesteren, A. Gregor, L. Hunt, and Ms2ger. DOM4. *W3C Working Draft*, 2012.

[256] L. Von Ahn, M. Blum, N. J. Hopper, and J. Langford. Captcha: Using hard ai problems for security. In *Advances in CryptologyâĂŤEUROCRYPT 2003*, pages 294–311. Springer, 2003.

[257] J. Weinberger, A. Barth, and D. Song. Towards client-side html security policies. In *Workshop on Hot Topics on Security (HotSec)*, 2011.

[258] M. West. Securing the client side. *Online at* `https://mikewest.org/2013/02/securing-the-client-side-devoxx-2012`, 2012.

[259] E. J. Whitehead Jr and M. Wiggins. WebDAV: IEFT standard for collaborative authoring on the Web. *Internet Computing, IEEE*, 2(5):34–40, 1998.

[260] Wi-Fi Alliance. Wi-fi protected access: Strong, standards-based, interoperable security for todayâĂŹs wi-fi networks. *Online at* `http://www.ans-vb.com/Docs/Whitepaper_Wi-Fi_Security4-29-03.pdf`, 2003.

[261] D. Wichers. Owasp top 10. *Online at* `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`, 2013.

[262] Wikipedia Contributors. Certificate authority (version 573015368). *Online at* `http://en.wikipedia.org/w/index.php?title=Certificate_authority&oldid=573015368#CA_compromise`, 2013.

[263] E. Z. Yang. HTML Purifier. *Online at* `http://htmlpurifier.org/`, 2013.

[264] M. Zalewski. Postcards from the post-xss world. *Online at* `http://lcamtuf.coredump.cx/postxss/`, 2011.

[265] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

[266] W. Zeller and E. W. Felten. Cross-site request forgeries: Exploitation and prevention. Technical report, Princeton University, 2008.