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

Deliverable D1.3

Case study 2 Report: Secure Web Architectures

Abstract

Deliverable details

Deliverable version: *v0.1*
Date of delivery: *30.04.2015*
Editors: *Rigo Wenning*

Classification: *public*
Due on: *M30*
Total pages: *121*

List of Contributors:
Steven Van Acker, Lieven Desmet, Stephen Farrell, Martin Johns, Frank Piessens, Philippe De Ryck, Rigo Wenning

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

Document revision history

	Responsible	Date
Initial outline	KUL	December 12, 2014
Initial draft of 'Secure session management' section	KUL	December 12, 2014
Initial draft of 'JavaScript sandboxing' section	KUL	January 28, 2015
Draft including recent developments	W3C	March 26, 2015
Draft including XSS	SAP	April 22, 2015
First Revision and improvements	TCD	April 24, 2015
Final Revision	All	April 30, 2015

Executive Summary

For the Web, security was for a long time an issue that was devoted to the underlying network layer. The declarative approach of the Web of documents used HTML4 and CSS and did not offer a very high attacking surface. With the movement of the Web towards being the Open Web Platform for Applications, this has changed. More and more vital applications in the area of government, economy and social life use the web as their platform of choice. And those higher value applications and services made it much more interesting for attackers to spend time finding loopholes on the Open Web Platform. Additionally, the Community needed to respond to the attack against the Internet via Pervasive Monitoring and the challenges laid down in RFC7258[77].

The Community reacted to those challenges with a variety of improvements and tools that web application developers can use to increase security, confidentiality and privacy for work. This includes improvements in the protocol sector, the widespread use of encryption and a number of tools to address the vulnerabilities created by the new functionality introduced by the Open Web Platform. Some of those security tools are still in their infancy, but look promising. Some tools need support to take off. Some have the potential to have landscape changing effects by tying the trust chain to new actors.

After the systematic evaluation of the Web Security landscape by the Web-platform Security Guide[59], this document describes the toolbox now available at a rather high level and picks the three main topics where STREWS is able to provide high impact advances for Web security to the Community.

Secure Web Architectures thus suggests to further the compartimentalisation and modularisation of web applications by allowing developers to sandbox certain modules. While the Community has already thought about it very recently, STREWS comes with a well thought-out proposal that may allow the Community to advance more quickly.

Another area of active research with potential high impact is the question on how to secure sessions on the Web. Session management is a crucial component in every modern web application. It links subsequent requests together, enabling a rich and interactive user experience. But sessions can be hijacked with potentially high damages. STREWS makes suggestions on how to further improve sessions by suggesting the SecSess approach that effectively prevents the unauthorized transfer of an established session, by eradicating the bearer token properties of the session identifier.

Finally, Cross-site Scripting attacks remain to be one of the fastest growing and most severe threats on the Open Web Platform. Considerable work is already done in the IETF's WebSec Working Group and in the W3C's Web Application Security Working Group, but several open problems remain to be addressed in the future. STREWS suggests to include recent research advances in this field in the works already under way to improve mitigating tools like Content Security Policies.

Contents

Chapter 1

Introduction

The Web platform is omnipresent in modern life, and has known a short but intense growth. It has become the foundation of many recent innovations in the ICT sector and beyond. Web technologies are more and more used in mission-critical services, in critical infrastructures and in population-facing high-value services that are expected to be secure and resilient. The Web is a key enabler in all societal challenges of the Digital Agenda of the European Commission. Doctors use a browser to control their eHealth systems, people put their skills online to find a job. And when we talk about the intelligent fridge, we mean it talks to the shop using web technologies. It is thus no surprise that Privacy, Security and Online Trust are part of the Digital Agenda, or even core to it. Because if people do not trust the systems and the Web, people will not use those. Systems will fail to grow and scale up. Cases of data breaches and identity theft already now undermine the trust in our systems. Unfortunately neither the proposed NIS-Directive nor the various Cybersecurity initiatives give the Web the place it deserves in this debate. It is thus crucial that Web Security architecture is put back on the agenda of the European IT security debate. This document tries to give hints and orientations where further work can improve the overall level of security on the web platform. It takes into account the latest developments in the communities as seen and detected by the two central standardisation organisations IETF and W3C.

Web technologies are constantly evolving. And with the evolving platform, the security challenges change too. In October 2014, HTML5 became a W3C Recommendation. It is already widely implemented by browsers and will change the landscape of the Web. The evolution was consequent. HTML4.1 was mainly declarative and its layout determined by Cascading Style Sheets (CSS). While the same-origin (SOP) was sufficient as a security concept for the declarative Web, the platform became more and more interactive. With the addition of javascript and XMLHTTPRequest, the so called «Web 2.0» was created. HTML5 will transform the Web further into an application platform. At the same time, the services side encountered a strong move towards virtualisation. Both form together the innovation hotbed of today. Securing the services side is important. But it has to coincide with the right security capabilities in the browser. The Web community started already to work on the challenges that result form this new transformation. This report will summarize those developments based on the assumption that a reader can always turn back to the Web platform Security Guide (D1.1)[59] to look for a description of the basic security landscape and its challenges. This covers not only the initiatives that address browser security directly, but also the threats on the network layer. A lot can be drawn from the very successful STRINT workshop[80] STREWS had organised in 2014.

It seems useful to allow a mental map to be built by layering the responses of a Web security architecture along the lines of the threats from the network level up to the scripting level. This is why the first chapter will describe the work in the IETF concerning that layer. Further work on higher layers in W3C will be addressed. From this basis, the suggestions to improve the overall security with new work seem evident.

The Open Web Platform (OWP) is not a fixed architecture in the sense of strict set of components that work in a specific way together. It is rather a giant toolbox of interoperable components that can be combined in ever new and innovative ways. As a consequence, a document on Web Security Architecture can not address all the ways the components can be combined, but has to focus on the existing components and their security. The very concrete play of the components in a given scenario has to be subject to a more detailed review as was done with WebRTC in Deliverable D1.2. One could list the security considerations of the components of the OWP in an arbitrary way, it would make no difference. The listed order is thus one that is oriented towards readability, but should not prevent readers from jumping right to the component of their interest.

The remainder of this report is structured as follows: First, in Chapter 2 we give a comprehensive overview on recent security-centric development in the ongoing evolution of the Web towards a feature rich application platform. Then, we successively explore three key topics that concern the security of Web architectures: Web sessions, application compartimentalisation, and code integrity: To provide robust handling of the security characteristics of an Web application, the server requires fine-grained control of the client/server authentication, currently provided by web sessions. We explore the current state-of-the-art and research in Chapter 3 and show potential improvements for the future. In Chapter 4, we present current approaches towards JavaScript sandboxing and compartimentalisation, which will allow applications to realise more flexible, yet secure, architectures. Finally, constructive JavaScript-driven security can only protect reliably, as long the JavaScript code itself is trustworthy. The Web's main threat against code integrity are Cross-site Scripting (XSS) vulnerabilities. Thus, in Chapter 5 we provide a comprehensive overview on research in this area in the last decade and isolate open problems, which require further attention. The report concludes in Chapter 6 with a short summary and a list of identified hot spots for future research and standardisation activities.

Chapter 2

Recent Developments

When Tim Berners-Lee invented the Web, he started in simplicity. The first Hypertext transfer protocol (HTTP) was a very simple stateless network protocol. It was designed to transport a static file from one party to another in the most simple way. Cookies[146] added statefulness to the protocol already in 1997[1]. HTTP and the applications built on top of it have evolved quite dramatically. Rich AJAX-based web applications introduced interactiveness. More and more high value applications and services found their way to the Web. This requires an ever increasing level of security. The more attractive the asset to attack, the more effort will be put into bypassing the security measures in place. The challenges for the network layer go beyond a mere transport security. This section describes selected security tools layer by layer. Some of them did not have the impact the community had hoped for. Other tools are still in their infancy but show already a great potential. The goal is to review some of the more prominent failures to draw conclusions and to look at promising developments that require more attention and research to avoid failure.

2.1 IETF Web protocol improvements

The IETF has a well functioning process with respect to security. For every specification, the authors have to complement their technical descriptions with security considerations and before being finalised all "IETF Stream" specifications are assigned for review from the security Directorate review team. This helps to understand the risks involved while implementing and using Internet technology.

Until very recently (April 2015) the IETF had a dedicated Working Group on Web Security[1]. As that group has now completed all of it's deliverables, it has been closed - a mark of success in the IETF organisation. And of course the Web Security Working Group was not intended to cover all security developments for the Web. The reader ought not be confused into thinking the IETF is not working on Web security just because the websec working group has closed!

As previously noted in STREWS deliverables, the STRINT Workshop[80] was part of a process that has lead to a renewal in the energy of all IETF groups aiming to improve the security of the Internet. STRINT also contributed to the start of a range of new activities. And in November 2014, the Internet Architecture Board (IAB) also issued a statement on confidentiality on the Internet[30] stating:

> In 1996, the IAB and IESG recognized that the growth of the Internet depended on users having confidence that the network would protect their private information. RFC 1984[114] documented this need. Since that time, there has been evidence that the capabilities and activities of attackers are greater and more pervasive than

[1] now obsoleted by [17]

> previously known. The IAB now believes it is important for protocol designers, developers, and operators to make encryption the norm for Internet traffic. Encryption should be authenticated where possible, but even protocols providing confidentiality without authentication are useful in the face of pervasive surveillance as described in RFC 7258[77].

Aside from this high level trend, we briefly describe some of the more core IETF defined building blocks that influence secure web architectures. The sections below are presented in a "bottom-up" manner from the lower relevant layers.

2.1.1 Privacy Enhanced IP Addressing

At a low level, many Web privacy issues depend on the extent to which IP addresses are identifying.

IPv6 has the option to have Stateless Address Autoconfiguration (SLAAC). SLAAC generates IPv6 addresses from a prefix and an identifier of the device. They are augmented by some fixed bytes. This means a device is re-identifiable across networks, countries, time and space. This allows tracking of the device and also reveals information about the manufacturer and other things. The STRINT Workshop has shown how harmful this can be. But also tracking for advertisement and the re-targeting can be done using IPv6 addresses generated in this manner.

As one mitigation, the IETF created Privacy Extensions in IPv6. [196, 15]. These specifications remove the fixed relation of IPv6 address and the device identifier.

IPv6 privacy Extensions are supported in some operating systems but may be not useful for services requireing full and real end-to-end connectivity. One of the advantages of IPv6 is that there is no scarcity of IP addresses any more. As use and implementation of IPv6 is still behind the expectations and needs, those lower level protections are a nice evidence that privacy by design helps even on the protocol level.

With IPv6 however, the global prefix issued by the ISP or network still makes up the high order bits of the address. And as that needs to be aggretable for routing purposes, it is also likely to be identifying.

Even within the IPv4 world, deployment of so-called carrier grade NATs, means that more and more devices and people can be "behind" any given source address. However, logging by ISPs (at the behest of law enforcement or otherwise) is likely to ensure that such addresses remain identifying.

In addition there is now work starting in IEEE on randomised MAC addresses. This is directly useful for IPv6 addressing, but also to make re-identification harder. With a fixed MAC address, a colluding layer 2 (or access to layer 2 logs) can mean that an adversary can correlate which seemingly independent IP addresses were actually issued to or used by the same device or person.

It is also worth noting that Web proxies do not remove the need for privacy friendly forms of addressing since such proxies may include information about the source IP addresses they see (via the XFF or Forwarded HTTP header field [214]), which has seen significant deployment.

So the practical benefits achieved via privacy friendly addressing mechanisms are only of limited impact at time of writing of this document.

2.1.2 DNSSEC

The Domain Name System (DNS) is an essential part of the Web. It provides the names that are used in the Unique Resource Identifiers (URI) and define the origin in the same origin concept[2]. The DNS allows to lookup a name for a certain machine and get back the corresponding IP address. An important part of the system are so called caching name servers. Caching name servers (DNS caches) store DNS query results for a period of time determined in the configuration

[2] see D1.1 Chapter 3.13[59]

(time-to-live) of each domain-name record. An attacker can now try to poison the caching name server by inception of false data. As a result of so called cache poisoning attacks, e-mails can be redirected and copied before they are delivered to their final destination, voice over IP calls can be tapped by third parties, and - given the circular dependency of the registration process on the DNS - SSL certificates may not be as protective as one would hope.[144]

In recent history, also governments are starting to use the DNS system to manipulate the user experience. DNS queries to the name server of the user's Internet Service Provider are matched against a list of URIs with allegedly harmful content. If an entry of the list matches, the name server, instead of giving the right IP address, gives the IP address of a governmental server that then shows some landing page with «STOP» signs or other symbols for a dead end request.

DNSSEC was designed to deal with cache poisoning and a set of other DNS vulnerabilities such as man in the middle attacks and data modification in authoritative servers. Its major objective is to provide the ability to validate the authenticity and integrity of DNS messages in such a way that tampering with the DNS information anywhere in the DNS system can be detected.[144]

Despite the importance and usefulness, DNSSEC still lacks ubiquitous deployment. This is not only due to the difficulties in implementation, but also due to the fact that there are a range of middle boxes redirecting people that would be put out of business with DNSSEC. The governmental blockings are only one examples for certain techniques that will not work any more once DNSSEC is deployed. More importantly, most Hotel networks today depend also on the manipulation of the DNS request to redirect people to the payment portal or to some click-through page that contains disclaimers and terms of service.

Solutions to this problem are largely unknown. The current business models block the evolution of the network by the deployment of higher security solutions like DNSSEC mainly because neither research nor industry has found reasonable solutions to transition from world to another. DNSSEC also still contains major challenges concerning its technical deployment where further work is needed.

2.1.3 DNS Privacy

While the information in the DNS is public, the act of accessing that can be sensitive. For example some DNS names themselves may indicate interest in a medical condiition or political leanings, and any accesses to such domains could be sensitive from some parts of the Internet. For this reason the IETF are now developing DNS privacy mechanisms for use between so called "stub" and "recursive" resolvers, which is usually the link from the end user's computer to some DNS server. This work is being done in the IETF's DPRIVE working group. While it is early days for this, the existence of the working group does show that the privacy aspects of the Internet infrastructure are now being much more seriously considered.

DPRIVE may also impact on the Web in less obvious ways - for example without DPRIVE, a page of HTML loaded over TLS might still cause a sufficiently identifying set of DNS queries to be emitted by a browser, so that an observer can determine the specific page being rendered, even if TLS and HTTPS functioned perfectly. It is probably only the Web that would cause such large sets of DNS query to be emitted to be identifying.

2.1.4 Transport Layer Security (TLS)

The OWASP Top 10 of 2010 still mentioned transport layer security. This has been merged into the more general term of «sensitive data exposure». Transport layer security (TLS) was one of the early protections established on the Web. TLS, the specification behind «HTTPS» was first standardised in 1999[62]. TLS has been improved over time and is currently specified in RFC5246[63]. The IETF is also currently working on TLS 1.3[225] which is a fairly major update.

TLS is very widely used, both in the Web and far beyond, but for the Web most especially in higher value services such as online banking. It is one of the absolute cornerstones of Web security. As the name indicates, TLS is not end-to-end security, but a mere encryption and secure connection between two endpoints of the transporting pipes. This leads to limitations once it comes to higher level security and authentication requirements that would need a fully designed end-to-end security system. It is also the most prominent application using X.509v3 public key certificates.

Vulnerabilities

Because TLS is of such importance, it is under high scrutiny. Vulnerabilities are discovered on a regular basis. Most of the vulnerabilities are based on implementation errors, but a few are flaws in the protocol. Here are some of the most notable ones recently uncovered. Most of them are fixed in up to date software.

- The *heartbleed bug*[79] has shown vulnerabilities that have been fixed in the meantime.
- The *triple handshake attack*[27] uses a shortcut in the authentication mechanism to mount a man-in-the-middle attack
- The *FREAK attack*[26] made a lot of noise. The «Factoring RSA Export Keys» attack targets a class of deliberately weak export cipher suites. Support for these weak algorithms had remained in implementations such as OpenSSL, even though they are typically disabled by default. The researchers discovered that several implementations incorrectly allow the message sequence of export ciphersuites to be used even if a non-export ciphersuite was negotiated.

One of the key findings during the bugfixing of «*heartbleed*» and «*FREAK*» was that large parts of the Internet community were relying on OpenSSL without investing anything into it. OpenSSL had only one active contributor and maintainer at this time. For the EU, this raises the question of ensuring the resilience and sustainability of the core building blocks and foundations of its Digital Agenda. If the underlying tools used by *everybody* are not maintained well, the entire house may collapse. This shows the challenge for Web security: How to make sure that the security tools we are using are reasonably maintained and sustained. What especially the FREAK attack showed is that the tendency in Cybersecurity to look for better monitoring of people by using weaker encryption can backfire enormously. The STRINT report[78] showed the challenge we have to identify modules of the Web technology stack that were weakened on purpose and that are now vulnerable and create security risks.

TLS & Certification Authorities (CA)

Another challenge for TLS is that it relies on the trust that the CAs are reliable. But some CAs have issued certificates for sites where they hadn't authenticated the requester properly, or have issued subbordinate CA certificates to organisations that misissued certificates. And one of the main problematic aspects of the Web PKI is that any CA is allowed to issue certificats for any DNS name. [239] [286].

As discussed in the STRINT Workshop, governmental intelligence services may also try to undermine the transport encryption to benefit from the ability to mount man-in-the-middle-attacks. One incident was reported 2013 when security engineers in Google found an intermediate certificate that could be traced back to the french «*Agence nationale de la sÃĩcuritÃĩ des systÃĺmes d'information (ANSSI)*». ANSSI reported that the intermediate CA certificate was used in a commercial device, on a private network, to inspect encrypted traffic with the knowledge of the users on that network[148]. But it could also have been used for more. In March 2015, Google became aware that CNNIC, which had a root certificate in all major browsers,

issued an intermediate certificate for several Google domains[149]. Google subsequently banned the CNNIC certificate from its list of accepted root CAs. Mozilla followed soon thereafter[193].

Certificate misissuance incidents such as these are likely to continue but are also increasingly likely to be detected thanks to the deployment of technologies such as Certificate Transparency. [151] [3] Certificate transparency is of course not the only approach to detecting such mis-issuance, nor to mitigating the damage that ensues, public-key-pinning and DANE (both described below) are also relevant here.

DANE

A key challenge for the TLS protocol remains the trust chain. With the advent SSL/TLS in important eServices like online-banking and online shops, the key chain was done with X.509v3 certificates via a system of Certification authorities (CAs). The system with the CAs was overall rather successful as people actually do use the Web for higher value online services via TLS/SSL. But the certification system has two central challenges. The success of Phishing attacks showed how poorly designed the trust relation between users and services was, usability wise. Digging deeper on known and unknown certificates reveals raw X.509v3 certificates that are only useful for X.509v3 experts and even those are sometimes lost. Thus, nobody ever questioned bogus CA certificates and if the alarm of the browser shows an unknown certificate, people just click it away[277] or stop browsing.

W3C tried to augment the usability by issuing a Recommendation called Web Security Context: User Interface Guidelines[229]. Already in 2010 it said: «Public key certificates (see [PKIX]) are widely used in TLS [SSLv3] [TLSv11] [TLSv12], but have been the basis for the generation of many inappropriate warnings and other dialogs for users.» Has it changed? Yes, a little bit, because the «*Identity and Trust Anchor Signaling* defined by the Web Security Context: User Interface Guidelines[229] found wide deployment in browsers but many sites still don't use the possibility to show their brand name in the navigation bar.

Reading the certificate used for the STRINT Workshop[80] the following technical assertions are made in the certificate:

- Some entity Gandi asserts that www.w3.org is the www.w3.org that has their certificate
- the connection is using TLS 1.2, encrypted with AES 256 CBC, using SHA1 for message authentication and ECDHE RSA for the key exchange But, how secure is TLS 1.2, encrypted with AES 256 CBC, using SHA1 for message authentication and ECDHE_RSA for the key exchange, compared to the state of art?

And the legal or social questions of interest are:

- Who is Gandi, where are they and what do they do?
- Are they asserting the identity of an entity and which identity are they asserting corresponds to what?
- Who is www.w3.org, what legal entity is behind www.w3.org and how can I reach them in case of problems

This means, the really important messages for the generation of trust by the user are not really transported or displayed to the user, even if that user examines all of the detail of a public key certificae (which essentially none do). In comparison, the domain name system (DNS) offers much more relevant information since it is the relevant DNS name (together with the URI scheme and port number) that forms the Web origin [18] on all all other Web security properties are based.

[3] RFC6962 is an experimental RFC that is now being standardised within the IETF "trans" working group - `http://tools.ietf.org/wg/trans`

The DNS-Based Authentication of Named Entities (DANE) provides a way to directly bind the pubilc keys required for TLS with this DNS information, by storing those keys in the DNS itself.

DANE however relies on DNSSEC to ensure the security of this binding and DNSSEC has some key potential advantages over the CA model. First, and most importantly, DNSSEC directly ties the keys to the Domain Name System and is thus much closer to the trust model of the Web. So the keys associated with a domain name can only be signed by a key associated with the parent of that domain name; for example, the keys for "example.com" can only be signed by the keys for "com", and the keys for "com" can only be signed by the DNS root. This prevents an untrustworthy signer from compromising anyone's keys except those in their own subdomains. This also integrates better into the Internet protocols than the CA system with its arbitrary identifying strings.

Secondly, signed keys for any domain can be made accessible online through a straightforward query using the standard DNS protocol, so there is no problem distributing the signed keys.

Given these significant advantages, one would in theory immediately jump on DANE as the new support for TLS. This hasn't (yet) happened. First, there are CAs that have mis-issued certificates, but most have not. Large players in the field work well with the CA system as the marginal cost for them is much lower if one takes into account that IBM needs one certificate while the bakery around the corner needs one certificate too. As the larger players have higher impact on the implementation decisions, the interest in DANE remained low. There is a lagging browser implementation and small players are only starting to realize the benefits of DANE for them.

But the main problem for DANE is the currently miniscule deployment of DNSSEC. Deployment of DNSSEC has to be large enough for the networking effects to kick in. One significnat inhibitor is the functionality provided by badly engineered DNS middle boxes in the network, often in a non-patched DSL modem. This is perhaps the main reason why browsers have not adopted DANE - at present it would be less reliable for them. (There are also concerns about the overheads of DNSSEC and how those might impact on rendering performance.)

And DANE also requires the DNS registries to improve their technical support for DNSSEC, their verification mechanisms and business processes to accommodate the higher value attached to a DNS entry that is now also used for trust purposes. The community has so far not fully understood and documented the issues around a full DANE deployment.

In the medium to longer term, if DNSSEC deployment gains traction, then one could confidently expect DANE deployment to follow very quickly and with very similar coverage as DNSSEC. However, without widespread DNSSEC deployment, DANE will not succeed in significantly displacing nor enhancing the X.509-based Web PKI.

Key Pinning

Above, we described some of the problems with the Web PKI system in TLS, namely that any CA that chains up to a root in a browser can issue a public key certificate for any name. If this rogue CA is trusted by the user agent, a web page can appear "trusted" despite the fact that it comes from some attacker. In addition to DANE, key pinning [73] is another remedy against this vulnerability.

Key pinning uses a HTTP header that allows a web site to instruct user agents to remember or «*pin*» the host's (or a CA's) cryptographic key material over a period of time[72]. Key pinning is a trust-on-first-use (TOFU) mechanism. At first visit, the user agent follows the normal TLS cryptographic identity validation. Thus the user agent will not be able to prevent a man-in-the-middle-attack at the first connection to the server. This first request thus suffers from all the vulnerabilities indicated in subsection 2.1.4. But on further requests, key pinning provides significant value by allowing host operators to limit the number of certification authorities than can vouch for the host's identity, and allows user agents to detect in-process man-in-the-middle attacks after the initial communication.

The risk is though, that the web site operator can lose control over their own identity certification provided by their private key. An attacker using a bogus certificate for a man-in-the-middle attack could inject on first request their own pinned private key in the PKP header into the HTTP stream, and pin the user agent to its own keys. To avoid post facto detection, the attacker would have to be in a position to intercept all future requests to the host from that user agent. So if the operator had pinned only the key of the host's end entity certificate, the operator would not be able to serve their web site or application in a way that user agents would trust for the duration of their pin's max-age.

Key pinning has several issues, notably to act like a «*super cookie*» that allows to track the user. As key pinning also has the inherent ability to accidentally "brick" one's web site (if one gets the pins values wrong), there are also operational concerns with the scheme.

HSTS

HTTP Strict Transport Security (HSTS) [110] is yet another web security mechanism aiming to address the failing in the Web PKI. In this case, the threat is so-called "SSL strippng" and the mitigation is for a web site to indicate, again via an HTTP header field, that it ought only be accessed for resources accessed using https-URI schemed requests. In other words HSTS provides a way for a site to declare that it ought only ever be accessed over TLS.

TLS Best Current Practices

As the original definitions for many of the protocols using TLS, including HTTP, are of the order of a decade old, many of their specifications do not reflect the expericence gained in recent years in dealing with TLS vulnerabilities, nor do they use the most up-to-date cryptographic options. The IETF's Using TLS with Applications (UTA)[4] is developing modern recommendations for use of TLS that address these issues. The main specification for that is an Internet-Draft [244] that at the time of publication is just about to become RFC 7525. It is expected that browsers and web servers will over time adopt these improved recommendations.

TLS 1.3

Finally, the IETF's TLS working group [5] are also in the process of developing a new and fairly significantly different version of TLS, TLS 1.3. This work is expected to be mostly complete within calendar 2015 and, based on recent history, we expect adoption to proceed much more quickly than the curve seen for TLS1.2. (At the time TLS1.2 was finalised, there was not much perception of urgency for deployment of new TLS versions. That position seems to have changed.)

2.1.5 X-Frame-Options

In Section 9.3 of the STREWS deliverable D1.1, the Web-platform security guide[59], contains a full description of the issue of clickjacking. A clickjacking attack, also known as a UI redressing, attack, in one form redresses or «*redecorates*» 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.

[4] `https://tools.ietf.org/wg/uta/charters`
[5] `https://tools.ietf.org/wg/tls/charters`

One of the possible mitigations is to use of the X-Frame-Header. [35]. RFC7034[230] provides informational documentation about the current use and definition of the X-Frame-Options HTTP header field. The X-Frame-Options HTTP header field indicates a policy that specifies whether the browser should render the transmitted resource within a `<frame>` or an `<iframe>`. Servers can declare this policy in the header of their HTTP responses to prevent clickjacking attacks, which ensures that their content is not embedded into other pages or frames. Braun et al.[35] claim that the X-Frame-Option can also be utilised to mount a first defence against Cross-Site Scripting (XSS) and Cross-Origin Leaks as it allows the site to control iframes and framing in general and reduce it to the desired sources. This is certainly a way for a site to have more control over the execution context.

2.1.6 HTTP Origin Based Authentication - HOBA

All security experts in this world agree: Passwords are bad. But how to replace them. One idea is HTTP Origin Based authentication (HOBA)[6] now specified in RFC7486.[76] Current username/password authentication methods such as HTTP Basic, HTTP Digest, and web forms have been in use for many years but are susceptible to theft of server-side password databases. Instead of passwords, HOBA uses digital signatures in a challenge-response scheme as its authentication mechanism. HOBA also adds useful features such as credential management and session logout. In HOBA, the client creates a new public-private key pair for each host [7] to which it authenticates. These keys are used in HOBA for HTTP clients to authenticate themselves to servers in the HTTP protocol or in a JavaScript authentication program.

HOBA session management is identical to username/password session management, with a server-side session management tool or script inserting a session cookie – according to RFC6265[17]– into the output to the browser. Use of Transport Layer Security (TLS) for the HTTP session is still necessary to prevent session cookie hijacking.

HOBA keys are «*bare keys*», so there is no need for the semantic overhead of X.509 public key certificates, particularly with respect to naming and trust anchors. The Client Public Key (CPK) structures in HOBA do not have any publicly visible identifier for the user who possesses the corresponding private key, nor the web origin with which the client is using the CPK.

HOBA has the potential to make Web authentication a bit more secure. But its main challenge will be in achieving widespread deployment, as today web browser developers do not see the need to support such a mechanism, preferring to continue with forms-based authentication linked in to federated (and ultimately password based) login systems. HOBA may also have a niche as a 2nd factor authenticaiton scheme in some cases.

2.1.7 HTTP/2

The transport layer of the Web uses HTTP, the hypertext transport protocol most prominently laid down initially in RFC2616[81] and now obsoleted by RFC7230[83]. There are two developments noteworthy here. First HTTP/2 tries to be more efficient for high volume websites. And secondly, the question of encryption by default was discussed in the HTTP Working Group. Finally, the Group settled with a solution that has two specifications: The Hypertext Transfer Protocol version 2[25] and Opportunistic Security for HTTP[206].

With so called «*http-streams*», HTTP/2 introduces a new paradigm. A stream can be used to have an exclusive connection to a node. That node is then funnelling all needed objects from all different sources of a web page via the node into the exclusive connection to the client. This is especially useful when on a low bandwidth connection or on a device with lower computing power. A client can rely on a node that has much higher bandwidth. This node assembles all the sources for a certain page and streams the complete information into an exclusive channel with the mobile browser.

[6] See `https://hoba.ie` for an example implementation
[7] "web origin" RFC6454[18]

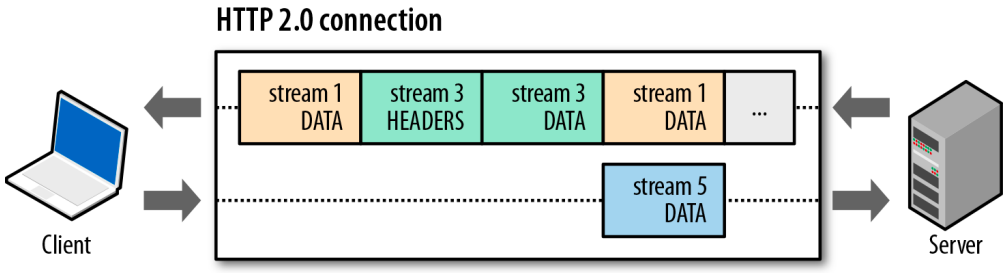


Figure 2.1: Schema of HTTP2 streaming, by Ilya Grigorik[91]

Each stream can be given an explicit dependency on another stream. A stream that depends on another stream is a dependent stream. When assigning a dependency on another stream, the stream is added as a new dependency of the parent stream.

```
  A                A
 / \       ==>    /|\
B   C            B D C
```

The combination of HTTP/2 exclusive streaming and encryption could be a mildly effective means against pervasive monitoring. HTTP/2 can either use TLS (see below) or the new opportunistic encryption specified in draft-ietf-httpbis-http2-encryption-01[206] The ISP only sees that the browser communicates with a certain node. Only the node knows which subelements are fetched and delivered. The ISP can't use deep packet inspection to determine the content of the stream coming to the user. On the other hand, if the node itself is tapped, the eavesdropper can acquire more information on the dependency between streams and has a single point for manipulations. It becomes legally interesting if the user is in one country and the node used for the exclusive stream is in another country. In this case, lawful interception becomes a challenge.

It is worth noting that the new emphasis on opportunistic encryption is a reaction to RFC7258[77] considering pervasive monitoring an attack. It was also one of the solutions retained by the STRINT Workshop, co-organized by the STREWS project[80]. But the STRINT Workshop also showed that we do not know yet enough about the vulnerabilities of opportunistic encryption and how to create the necessary half way encryption that allows legitimate lawful interception and prevents pervasive monitoring. A combination with the approach of DANE below is likely, but in its infancy.

2.2 Security improvements on the Open Web Platform

Security is also high up on the agenda in W3C. There is no formal requirement for security considerations in every specification. The Web has never been a place where the cathedral-approach had high recognition. Thus there is no master plan for security on the Web. Like the IETF[8], W3C has various Groups working on tools that may be used when implementing a service on the web. But traditionally, the activities around security in W3C have been smaller than those in the IETF.

The main Groups active in the area are the Web Cryptography Working Group (WebCrypto), the Web Application Security Working Group (WebAppSec), the Web Security Interest Group (SecurityIG) and the Technical Architecture Working Group (TAG).

[8] see section 2.1.7

2.2.1 TAG-Finding: Securing the Web

The STRINT Workshop[80] not only had repercussions in the IETF and in RFC7258[77], but the IAB Statement from November 2014[30] and the Montevideo Statement[9] of the I* (I-Star) Organisations[9] also had repercussions into the highest levels of W3C. In the Montevideo Statement the associated organisations re-asserted the benefits of the multistakehoder model. The Montevideo Statement is a high level commitment of all organisations responsible for coordination of the Internet technical infrastructure globally to address the security issues created by the fact that the current Internet infrastructure is to easy to wire tap by all kinds of malicious actors, including governments. The relevant statement is:

> *They (the I* Organisations) reinforced the importance of globally coherent Internet operations, and warned against Internet fragmentation at a national level. They expressed strong concern over the undermining of the trust and confidence of Internet users globally due to recent revelations of pervasive monitoring and surveillance.*

On 22 January 2015, the W3C Technical Architecture Group (TAG) issued a TAG Finding[208] entitled: «*Securing the Web.*» This TAG Finding can be seen as a part of W3C's commitment to the I* Statement. The TAG insists that the Web needs trustworthiness more than ever as the lack of trust is seen as an inhibitor of the further development of the Web:«*Important properties of authentication, integrity and increased confidentiality are currently best provided on the Web by Transport Layer Security (TLS)[63]. For the HTTP protocol, this means using "https://" URLs[83].*»

The TAG concluded that as much traffic as possible should transition to the TLS/HTTPS protocol. The TAG then identified issues and mitigation techniques when transitioning to and using encryption:

- Evaluating 'Powerful' Features Many new Web platform features offer increased capabilities, access to data and richer functionality. Some of these 'powerful' features also have significant security and privacy implications, and Working Groups should consider whether they ought only be used over an encrypted connection. The Web Applications Security Working Group (WebAppSec) has begun work on powerful-features[279] to document best practices in this area.

- Controlling Scheme-dependent Behaviors Existing specifications that change behavior based upon the URL scheme ('http://' vs. 'https://') should be examined to see if these differences can either be eliminated or controlled by authors, provided that there is no loss of security or surprising changes in behavior. For example, the referrer-policy specification[71] is offering more control over the Referrer HTTP header, as part of CSP2[280].

[9] I* is a roundtable of the following organisations:

- African Network Information Center (AFRINIC)
- American Registry for Internet Numbers (ARIN)
- Asia-Pacific Network Information Centre (APNIC)
- Internet Architecture Board (IAB)
- Internet Corporation for Assigned Names and Numbers (ICANN)
- Internet Engineering Task Force (IETF)
- Internet Society (ISOC)
- Latin America and Caribbean Internet Addresses Registry (LACNIC)
- Réseaux IP Européens Network Coordination Centre (RIPE NCC)
- World Wide Web Consortium (W3C)

- Managing Changing Links Updating sites from 'http://' to 'https://' necessitates changing links to resources, which is counter to the good practice of 'avoiding URI aliases'[10]. To mitigate such changes, Working Groups (in particular, those dealing with Linked Data) should consider how redirects (like 301 Moved Permanently) and Strict Transport Security[222] can be used to assert that a 'http://' origin has been replaced by an 'https://' one.

- Transitioning Mashups When transitioning from 'http://' to 'https://', applications that depend upon third-party resources ('mashups') that have not yet changed to 'https://' themselves can experience difficulties, because of the Mixed Content policy[278].

- Facilitating Shared Caching Adopting 'https://' has the side effect of disallowing shared HTTP caching[82]. Shared caching has a limited role on the Web today; many high traffic sites either discourage caching with metadata, or disallow it by already using 'https://'. However, shared caching is still considered desirable by some (e.g., in limited networks); in some cases, it might be so desirable that networks require users to accept TLS Man-in-the-Middle – which is a bad outcome for Web security overall. The TAG encourages the exploration of alternative mechanisms that preserve security more robustly, such as certain uses of Subresource Integrity[33].

- Browser Policy Extension APIs Similarly, adopting 'https://' makes the practice of imposing policy in intermediaries (e.g., in schools and workplaces, by parents, in prisons) more difficult. While TLS Man-in-the-Middle is one solution to this, it is a blunt one, sacrificing substantial security when they are used. Therefore, the TAG encourages development of facilities to enable imposition of policy – when it is necessary – in a more controlled way, e.g., as new APIs for Web browser extensions.

- Documentation of Best Practices Changing to 'https://' is often difficult, for a variety of reasons. The TAG encourages development of documentation to aid Web content creators, administrators and implementers in this process. In particular, since W3C has expertise in the implications upon content creators, the W3C documentation should focus on this audience.

- User Experience of Security Educating and interacting with users regarding security is notoriously difficult. Even so, the TAG encourages the implementer community to continuously challenge their assumptions in this space; for example, there is currently a discussion changing how 'http://' URLs are presented so that they are marked insecure, or even defaulting to 'https://' when a URL reference without a scheme is input. Where appropriate, the TAG also encourages these discussions to take place in W3C fora.

Many of the suggested improvements will be explored in more detail in the following sections. It has to be noted that the TAG is addressing some rather difficult areas of security that are in urgent need of research. This is certainly the old question of middle boxes that is also haunting the IETF and reappears here in the form of the issue of imposing policy in intermediaries. But there is also a new approach of the TAG towards the user interface. So far, browsers and most of the user facing technology components of the Web had the same model: While being as interoperable as possible into the system and the data exchanges, the user interface was a heavily guarded spaces for competition between the stakeholders and economic actors. But in case of security and privacy, the blind spot concerning user interfaces in web standardisation has shown its limits. On the other hand it is obvious that a standardisation in the area of user interfaces for security and privacy must be very prudent and leave as much space as possible for competition. *Securing the Web*[208] is not meant to be exhaustive by the TAG. There is WebAppSec, but there are also other new ways explored to improve the Security on the Open Web Platform.

[10] as expressed in the TAG Finding *Architecture of the World Wide Web, Volume One*[121]

2.2.2 Server Driven Policies

A confirmed trend in Web security technology is revolving around server driven browser enforcement. This is done via some policy document containing rules that is pushed towards the client as a part of a web application. The client is then responsible to enforce the policy correctly. On the protocol layer, X-Frame-Options, HSTS and Key pinning[11] were already described. On the Web layer, there is additional room for higher level policies.

Content Security Policy

JavaScript injection actively circumvents all protective isolation measures which are provided by the same-origin policy, and empowers the adversary to conduct a wide range of potential attacks, ranging from session hijacking, over stealing of sensitive data and passwords, up to the creation of self-propagating JavaScript worms.[12]. One attempt to mitigate those vulnerabilities are Content Security Policy (CSP). With a CSP 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 includes capabilities to:

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.

The Web Application Security Working Group started by specifying CSP 1.0[249]. But in 2015, the Group decided to abandon[250] work on CSP level 1 that was superseded by CSP level 2[281]. Level 2 makes breaking changes from Level 1, but also adds a number of rules to be used in a CSP. The goal of this specification is to reduce attack surface by specifying overall rules for what content may or may not do, thus preventing violation of security assumptions by attackers who are able to partially manipulate that content. CSP level 2 is actually in Candidate Recommendation, which means that W3C calls for implementation of the specification. The Working Group nevertheless things about an extension to CSP called CSP «*Next*».

There are also plans to work on a complement to CSP that will allow authors to instruct user agents to remember ('pin') and enforce a Content Security Policy for a set of hosts for a period of time. This is similar to key pinning [13] and extends this successful concept to CSPs.

This confirms the assumption that the potential of Content Security Policies is far from being fully exploited and that further work in this area is needed. It is thus interesting from a research perspective whether CSPs can have adverse effects like confirming monopolies of big players or excluding certain competitors. What does it mean to abuse a CSP?

User Interface Security Directives for Content Security Policy

CSPs can also be used to mitigate User Interface (UI) Redressing. An author has to restrict the behaviour of a web application, e.g. the origins where it can load its resources from or the ways it can execute scripts. The User Interface Security Directives for Content Security Policy[173] defines directives to restrict the presentation or the interactivity of a resource when its interaction with the user may be happening in an ambiguous or deceitful context due to the spatial and/or temporal contiguity with other content displayed by the user agent[14]. The User

[11] See sections 2.1.5, 2.1.4 and 2.1.4
[12] See later section 5.1.3
[13] see section 2.1.4
[14] for a full explanation of the clickjacking attack, also known as a UI redressing can be found in section 9.3 of D1.1, the Web-platform security guide: Security assessment of the Web ecosystem[59]

Interface Security Directives for Content Security Policy is thus an implementation cookbook for the CSPs in the context of web applications. This shows nicely that the current standardisation work is at a level of abstraction that addresses advanced developers and less the average SME using the Open Web Platform. Further action will be needed to translate not only the UI aspects of the new security toolbox into guidelines that are readable and usable for the breadth of the Web community.

Further work along the same lines is planned around Entry Point Regulation for Web Applications. This specification will define means to allow web applications to designate their entry points via a combination of headers and a policy manifest. User agents will be forced to restrict external navigations and other information flows into the application based on this policy to reduce the application's attack surface against Cross-Site Scripting and Cross Site Request Forgery. But Entry Point Regulation itself has some caveats as it allows the site to control deep links, bookmarks and similar features. Deep linking itself was already subject to several court decisions and is socially highly controversial. Thus Entry Point Regulation can be rather considered a dual use tool.

The user interface to security remains by far the most difficult issue to tackle. The W3C Workshop on Privacy and UserâĂŞCentric Controls[2] has shown that issues related to implementing and achieving adoption related to privacy and security may be similar to those for accessibility and internationalization. The latter are seen as a constant challenge and review-point for all specifications. The Workshop has raised the right questions, but wasn't able to come up with agreed answers despite the number of high quality contributions. It has been noted earlier already that the user interface is where browsers compete. So agreement in this area is very difficult as it faces to constant challenge not to overstep the line where space is reserved for competition between the actors.

Powerful features

The Web Application Working Group is now chartered to work on the issue of so called «*powerful features*». The charter itself calls it «*Upgrade Insecure Requests*». The chartering was controversial as it cuts across several Working Groups and Standardisation Organisations. The idea was then relabelled «*Privileged Contexts*»[282].

The specification provides guidelines for user agent implementors and specification authors for implementing features whose properties dictate that they be exposed to the web only within a trustworthy environment. This means mainly that security requirements are applied tightly and that encryption is used pervasively. It is interesting to see that the specification starts by defining use cases that the authors feel need a privileged context of execution. Features are considered powerful enough to restrict when it fits into one or more of the following categories:

1. The feature provides access to sensitive data (personally-identifying information, credentials, payment instruments, and so on)
2. The feature provides access to sensor data on a user's device (camera, microphone, and GPS being particularly noteworthy, but certainly including less obviously dangerous sensors like the accelerometer)
3. The feature provides access to or information about other devices a user has access to.
4. The feature exposes temporary or persistent identifiers, including identifiers which reset themselves after some period of time (e.g. session storage), identifiers the user can manually reset
5. The feature introduces some state for an origin which persists across browsing sessions.
6. The feature manipulates a user agent's native UI in some way which removes, obscures, or manipulates details relevant to a user's understanding of her context.

7. The feature introduces some functionality for which user permission will be required.

This rather broad list is considered non-exhaustive and reveals a certain opinion of the authors that privileges of the average web application should be evaluated in more depth. The specification still lacks maturity. With the broad definition of «*privileged context*», applying the necessary measures to said privileged context becomes more Important. The first thing to do is to apply transport encryption. Citing RFC7258[77] closes the loop to all other attempts to increase the amount of traffic encryption on the Web. But the guidelines go beyond a mere recommendation for more encryption. A past debate about the geolocation API carries fruit as the authors of the specification suggest to only allow the execution of the getCurrentPosition()[220] command in a privileged context, meaning a requirement for TLS transportation.

A further test is whether an origin is trustworthy. This first of all makes the local resources *Potentially Trustworthy*, namely localhost and local files. *Potentially Trustworthy* are also authenticated resources, which could cover TLS and also things like DANE[15]. When a specification allows user agents to extend this trust to other, vendor-specific URL schemes like 'app:' or chrome-extensions, it is opening multiple paths to abuse not by the attacker of the assumed threat model, but by the vendors themselves. This is also an opener for potentially big loopholes. But there is so far very little experience with setting a privileged context for a web application and further research in this area would be a good investment: What are the features that need to be reduced? What are the security features required for certain use cases? The Directive 2002/58EC[44] in its current version as amended by the Directive 2009/136EC[45] required the presence of an indicator in the UI of a device if the geolocation feature of that device is active. There are certainly more requirements where applications need a privileged context. Guidelines based on use cases will help tremendously to make the Web a more secure place. ENISA has already provided extensive guidelines in its «Threat Landscape and Good Practice Guide for Internet Infrastructure»[154]. It would be a good idea to translate the insights from those reports into a more technical specification that helps guide implementers of web applications to the right level of security. This is also what the ENISA report suggests in its recommendation 4.

2.2.3 Mashing up from various resources

Subresource Integrity

The Web has gained part of its exponential growth from the fact that people share many things. This accelerates greatly the development cycles. Because of the distributed nature of the Web and its architecture, services can be delivered and merged into the Web presence. The most prominent example is the business engine of the Web: Advertisement. Specialised companies provide scripts to a, e.g., a newspaper web site. This script then loads resources into the page that is constructed at runtime by the browser. But those so called mashups go far beyond the trivial example above. As the Web moves forward and becomes an application platform, not only text and images are merged on the fly to construct a page. Paradigms well known in computer science appear. Node.js is one of the most prominent JavaScript libraries and is widely used. A web application then uses functionality provided by the library in its own application code. On the Web, this does not mean to provide one's own copy of the library. Mostly, those libraries are dynamically integrated from the original source. This takes the form of «*<script src="http://example.org/include.js"></script>*». Once an attacker has access to include.js on example.org or can manipulate the script in transition, the attacker can inject malicious capabilities into the library. This hidden capabilities allow the attacker to subsequently gain full control over the web application and all its assets. As the script is used by many applications and web sites, this has the potential for a tidal wave of infections. A 2011 survey of 800 businesses revealed that 87% are using Google analytics for online measurement[42]. Google analytics is mainly a JavaScript library that is integrated into the web site one wants to measure.

[15] see section 2.1.4

If attackers would be able to inject malicious code into google-analytics.js they could gain access to 87% of the hosts on the Web. Is there a way to secure those shared libraries? For the moment, the only way to mitigate a possible attack somewhat is to use TLS when integrating those scripts into one's own web application. It is further recommended to also use HSTS and key pinning[16].

The Web Application Security Working Group started work on this issue. The idea is to not only authenticate the server as TLS does, but also the content. An author of a web application should be able to pin the content, ensuring that an exact representation of a resource, and only that representation, loads and executes. A specification on how to achieve this is currently under development. It is called «*Subresource Integrity*»[34]. The specification creates a validation scheme, extending several HTML elements with an integrity attribute that contains a cryptographic hash of the representation of the resource the author expects to load. The browser would then be able use the transported hash, remember it and alarm the user or the CSP reporting URI of a potential mismatch once the cryptographic function was executed. As for key pinning, this reduces the attacking surface to the first download of the content as later downloads can be compared to the pinned hash of the loaded application script. An obvious relation to the work of the Web Cryptography Working Group[17] unfortunately is only there on the level of the implementers. A further harmonisation of functionalities and their re-use seems desirable. Research will have to further explore the use of trust chains like trusted computing on the Open Web Platform. This also includes the use of hardware tokens to secure certain high value application environments.

Suborigin Namespaces

The Web Security Guide D1.1[59] explained the concept of same origin protections in section 3.1.3. The Web Origin Concept is laid down in RFC6454[18], reminding us that many web-related technologies have converged towards this common security model. In principle, user agents could treat every URI as a separate protection domain and require explicit consent for content retrieved from one URI to interact with another URI. Unfortunately, this design is cumbersome for developers because web applications often consist of a number of resources acting in concert. Instead, user agents group URIs together into protection domains called «*origins*». The origin is computed using an algorithm described in section 4 of RFC6454[18] that creates a triple of «*(uri-scheme, uri-host, uri-port)*».

In a mashup scenario where authors want to use scripts downloaded from the Web, those scripts would have a different origin than the rest of the web-site. To give authors flexibility, the suborigin namespaces work will allow authors to create applications that place themselves into a given namespace defined by the origin triple. The hope is to allow for easier development of modular applications and support privilege separation. This is certainly a helpful component with respect to the JavaScript sandboxing suggested in this document[18]. The work on suborigin namespaces is in its infancy and will benefit from feedback from the project contained in this report.

Confinement with Origin Web Labels

On a longer time scale, mashups need to be controlled in a more fine grained way. In a mashup, a web application may want to specify privacy and integrity policies on data. Currently this is imagined to be done by origin labels. This way browsing contexts can be confined according to the labels and policies set. This allows web applications to share data with less trusted code and impose restrictions on how the code can further share the data involved. This comes very close to the sandboxing suggestion given later in this document[19], but is mainly still a basic

[16] see section 2.1.4 and 2.1.4
[17] see below section 2.2.4
[18] See section 4
[19] see Section 4

idea without written content available. STREWS is funnelling the research results into to Web Application Working Group to help develop this further.

2.2.4 Javascript APIs

As the Web moves from the world's library to the world's application platform, access for programmers to features of devices or services is provided by Application Programming Interfaces (API). For APIs to work, they have to be known and interoperable. There are proprietary APIs to interface with some commercial software, but most of the APIs are described in specifications from standardisation organisations. The Open Web Platform is based on many of those APIs. Many of them are still under active development. To assess the Web Security Architecture toolbox, selected APIs are presented as they expose security features.

Credential Management API

In Section 2.1.6 it was already mentioned that weak passwords are one of the most used vulnerabilities on the Web. There are several approaches to create wallets. Today, all browsers have their own wallet format. A web application can currently not access those to help with the management of user credentials or identity providers. The API is also supposed to provide mechanisms for for use and lifecycle management of these common credential types. It remains to be seen how well this API will integrate into existing schemes and how much schemes will be able to cater to the new API.

Web Cryptography API

The Web Cryptography API defines a low-level interface to interacting with cryptographic key material that is managed or exposed by user agents. The API itself is agnostic of the underlying implementation of key storage, but provides a common set of interfaces that allow rich web applications to perform operations such as signature generation and verification, hashing and verification, encryption and decryption, without requiring access to the raw keying material.[246]

The Web Cryptography API was developed with 7 use cases in mind. Those were:

Multi-factor Authentication A web application may wish to extend or replace existing username/password based authentication schemes with authentication methods based on proving that the user has access to some secret keying material. Rather than using transport-layer authentication, such as TLS client certificates, the web application may wish to provide a rich user experience by providing authentication within the application itself.

Using the Web Cryptography API, such an application could locate suitable client keys, which may have been previously generated via the user agent or pre-provisioned out-of-band by the web application. It could then perform cryptographic operations such as decrypting an authentication challenge followed by signing an authentication response.

The authentication data could be further enhanced by binding the authentication to the TLS session that the client is authenticating over, by deriving a key based on properties of the underlying transport.

If a user did not already have a key associated with their account, the web application could direct the user agent to either generate a new key or to re-use an existing key of the user's choosing.

Protected Document Exchange When exchanging documents that may contain sensitive or personal information, a web application may wish to ensure that only certain users can view the documents, even after they have been securely received, such as over TLS. One way that a web application can do so is by encrypting the documents with a secret key, and then wrapping that key with the public keys associated with authorized users.

When a user agent navigates to such a web application, the application may send the encrypted form of the document. The user agent is then instructed to unwrap the encryption key, using the user's private key, and from there, decrypt and display the document.

Cloud Storage When storing data with remote service providers, users may wish to protect the confidentiality of their documents and data prior to uploading them. The Web Cryptography API allows an application to have a user select a private or secret key, to either derive encryption keys from the selected key or to directly encrypt documents using this key, and then to upload the transformed/encrypted data to the service provider using existing APIs.

This use case is similar to the Protected Document Exchange use case because Cloud Storage can be considered as a user exchanging protected data with himself in the future.

Document Signing A web application may wish to accept electronic signatures on documents, in lieu of requiring physical signatures. An authorized signature may use a key that was pre-provisioned out-of-band by the web application, or it may be using a key that the client generated specifically for the web application.

The web application must be able to locate any appropriate keys for signatures, then direct the user to perform a signing operation over some data, as proof that they accept the document.

Data Integrity Protection When caching data locally, an application may wish to ensure that this data cannot be modified in an offline attack. In such a case, the server may sign the data that it intends the client to cache, with a private key held by the server. The web application that subsequently uses this cached data may contain a public key that enables it to validate that the cache contents have not been modified by anyone else.

Secure Messaging In addition to a number of web applications already offering chat based services, the rise of WebSockets and RTCWEB allows a great degree of flexibility in inter-user-agent messaging. While TLS/DTLS may be used to protect messages to web applications, users may wish to directly secure messages using schemes such as off-the-record (OTR) messaging.

The Web Cryptography API enables OTR, by allowing key agreement to be performed so that the two parties can negotiate shared encryption keys and message authentication code (MAC) keys, to allow encryption and decryption of messages, and to prevent tampering of messages through the MACs.

Javascript Object Signing and Encryption (JOSE) A web application wishes to make use of the structures and format of messages defined by the IETF Javascript Object Signing and Encryption (JOSE) Working Group. The web application wishes to manipulate public keys encoded in the JSON key format (JWK), messages that have been integrity protected using digital signatures or MACs (JWS), or that have been encrypted (JWE).

Next steps The Workshop on Web Cryptography Next Steps[99] concluded that further work is needed to address authentication and the integration of hardware tokens. But the current Web Cryptography Working Group charter does not allow for the integration of hardware tokens. In the subsequent chartering discussion, the Working Group decided against developing an API to access security hardware elements. This is bad news for Europe, as access from the browser and the Open Web Platform to the security features of Identity cards will be more difficult and will require dedicated software. To achieve integration of dedicated software into a workflow on the Web is much harder than accessing the needed functionality from within a Web application or page. This will make eGovernment applications more difficult and burdensome. Research is needed to assess the arguments from the browser makers and find remedies so that a secure access to the Identity tokens for eGovernment becomes easy.

Permissions API

This API will allow web applications to be aware of the status of a given permission, to know whether it is granted, denied, or if the user will be asked whether the permission should be granted. It will will not address user agent implementations of permissions, including their scope, duration, granularity, or user interface and experience for indicating, configuring, or asking for permissions.

Permissions and authentication are closely related. In general work in this area is not easy. There have been many attempts to solve the permission and authentication issue for the Web. The struggle has gone on for more than 10 years. The Permissions API is low level and only allows a web application to access certain information about the permissions known to the browser. It is connected with the work of the Web Cryptography Working Group and its interface to the cryptographic trust chain in the browser as well as to hardware tokens.

While the work is currently basic, one can imagine that web applications may want to access all kinds of access control and authentication features in the future. This high potential is also a major burden for this work. Many people in business believe that owning the identity management will provide them considerable competitive advantages, hence the constant battle over those mechanisms. Currently they remain underdeveloped despite their high importance for many areas of life on the Web.

Chapter 3

Secure Session Management[1]

3.1 Session Management in the Modern Web

Recent developments have shown a broad activity in securing the information transport and authenticity of exchanges on the Web. New tools like DANE will help to determine whether a user is talking to the right service. Once the connection is established, it is important to remember what has been done and which options are available in a certain context. The shopping cart is the most prominent example of such a stateful service. This is done by session management. Session management is a crucial component in every modern web application. It links subsequent requests and temporary stateful information together, enabling a rich and interactive user experience. Unfortunately, as we will show in this section, the de facto standard cookie-based session management mechanism [2] is imperfect, which is why session management vulnerabilities rank second in the OWASP top 10 of web application vulnerabilities [285].

3.1.1 Session Management

In the early Web, when sites only consisted of static content, the HTTP protocol already offered an *Authorization* header [84], which could be used for authentication and authorization, even up until this day. The most common application of the *Authorization* header uses *Basic* authentication, where the user's credentials, more specifically a username and password, are base64-encoded, and included as the header value. This allows a server-side application to extract these credentials, verify them against a password file and make a decision on whether to allow the request or not. After a successful authentication, the browser will attach the user's credentials to every subsequent request to this origin. Note that since the browser remembers these credentials during the lifetime of its process, logging out of an account is only possible by closing the browser.

As web applications became more complex, developers wanted to integrate authentication with the application, streamlining the user experience within the same look and feel. To authenticate users, they embedded an HTML form, where the user had to enter a username and a password. By submitting the form, the username and password were sent to the server, where they could be validated. However, since the credentials are only sent in a single request after form submission, instead of in every subsequent request as with the *Authorization* header, web applications needed a way to keep track of the user's authentication state across requests, a challenge with the stateless HTTP protocol.

[1] Parts of this chapter have been disseminated in the PhD dissertation of Philippe De Ryck [56] (December 2014), the OWASP 2013 presentation, entitled *"Improving the Security of Session Management in Web Applications"*, and the SAC 2015 paper [58], entitled *"SecSess: Keeping your Session Tucked Away in your Browser"*.

[2] Cookies are a standard defined by RFC 2109[146], but there is no standard on how to do session management

A session management mechanism on top of HTTP is capable of associating multiple requests from the same user with a server-side session state, allowing the application to store useful session information, such as an authentication state or a shopping cart (illustrated in Figure 3.1). To keep track of this server-side session across multiple requests, the session is assigned an identifier, which the client needs to include in every request. Initially, this identifier was embedded in the URI as a parameter, for example like *http://example.com/index.html?SID=1234*. Unfortunately, this requires the web application to append the user's session identifier to every URI in the page, and repeat this action for every user. The introduction of cookies [19] addresses this problem. Cookies are server-provided key-value pairs, stored by the client in the cookie jar and attached to every request to the same domain. Essentially, cookies allow the server to store small pieces of data at the client side, which will be attached to future requests. Cookies can be used for storing simple settings, such as a language preference, or for building more complex systems, such as session management mechanisms. Today, cookie-based session management is the de facto standard, but many systems still allow parameter-based session management as a fallback mechanism, for example when cookies are not supported.

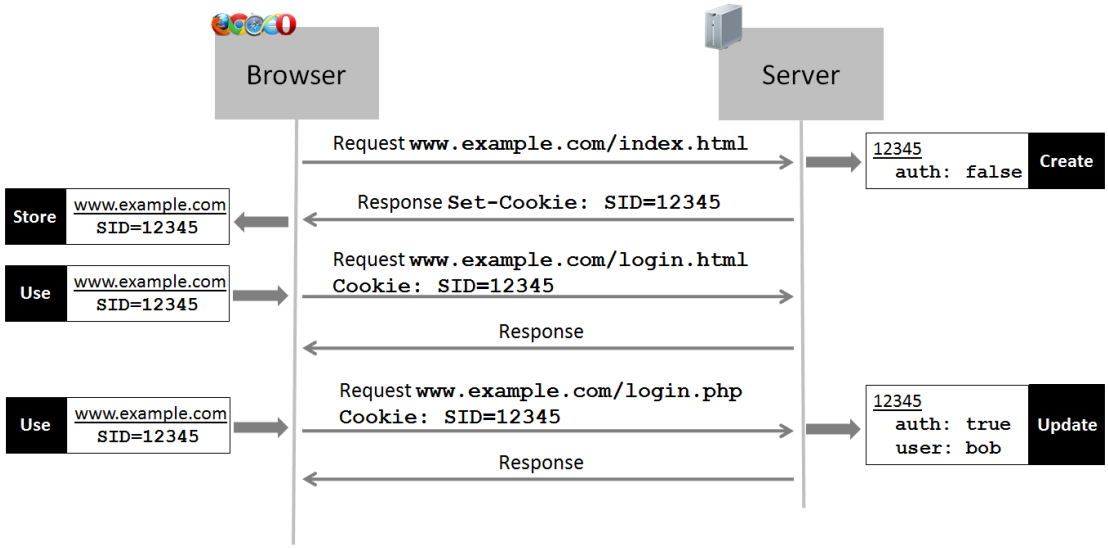


Figure 3.1: An illustration of cookie-based session management on top of HTTP. Presenting the session identifier (here 12345) to the server suffices to associate a request with the server-side session object.

In this de facto standard session management mechanism, the session identifier has a peculiar property, since it suffices to present the associated session identifier to the web application in order to associate a request with a specific session. This essentially makes the session identifier a *bearer token*, meaning that a request *bearing* the identifier is granted the privileges associated with the session. Since the server-side session state generally stores the user's authentication state, these privileges are comparable to those of an authenticated user. Note that this means that the session identifier and credentials are similar, as they both grant access to an authenticated session. Credentials, however, can be used to repeatedly gain access to the application, while the level of access granted by the session identifier is limited to the lifespan of the session.

3.1.2 Relevant Threat Models

The previous section introduced several building blocks of modern web applications, and already hinted at several of the attacks that are possible in the current Web. In this section, we present several threat models, often encountered in research on Web security, and in this case, relevant

for the discussion of client-side Web security. We explain the general capabilities for each threat model below, and provide additional details in the technical chapters, tailored towards a specific problem domain.

Forum Poster A *forum poster* [21] is the weakest threat model, representing a user of an existing web application, who does not register domains or host application content. A forum poster uses a web application, and potentially posts active content to the application, within the provided features. Additionally, a forum poster remains standards-compliant, and cannot create HTTP(S) requests other than those he can trigger from his browser.

Web Attacker The *web attacker* [8, 21, 22, 32] 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 specific attacker characteristic, such as a specific physical location, every user on the Web is able to become a web attacker. As a consequence, the capabilities of the web attacker are considered to be the baseline for threat models in the Web, with the forum poster as the exception to the rule.

Related-domain Attacker A *related-domain attacker* [32] is an extension of the web attacker, where the attacker is able to host content in a related domain of the target application. By hosting content in such a related domain, the attacker is able to abuse certain Web features, which are bound to the parent domain. A common case of a related-domain attacker is when the attacker is able to host content on a sibling or child domain of the target application, for example for the web sites of different departments within a company.

Related-path Attacker A *related-path attacker* is another extension of the web attacker, and represents an attacker that hosts an application on a different path than the target application, but within the same origin. This scenario occurs for example within the web hosting of Internet Service Providers (ISPs), which often offer each of their clients a web space under a specific path, all within the same origin. Academic papers aptly describe this attacker [119] and its conflicts with the Web's security model, albeit without giving it an explicit name.

Passive Network Attacker A *passive network attacker* [120] is considered to be an attacker who is able to passively eavesdrop on network traffic, but cannot manipulate or spoof traffic. 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 needed. Depending on the system under attack, the passive network attacker may require a specific physical location to eavesdrop on the network traffic. One common example of a passive network attack is an attacker eavesdropping on unprotected wireless communications, which are ubiquitous thanks to publicly accessible wifi networks and freely available hotspots.

In 2013, whistleblower Edward Snowden [263] revealed that intelligence services across the globe have powerful traffic monitoring capabilities. These *pervasive monitoring* capabilities are essentially passive network attacks, albeit on a very large scale compared to the traditional passive network attacker. In response to the Snowden revelations, the IETF has drawn up a best practice, stating that specifications should account for pervasive monitoring as an attack [75].

Active Network Attacker An *active network attacker* [8, 21, 120] is considered to be capable of launching 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, inject and block the contents of all unencrypted network traffic. An active network attacker is generally not considered to be capable of presenting valid certificates for HTTPS sites

that are not under his control, unless by means of obtaining fraudulent certificates, or by using attacks such as SSL stripping [174].

3.1.3 Common Threats against Web Sessions

The attackers described in the previous section can threaten currentely deployed session management mechanisms in various ways. Below, we introduce two concrete threats against session management mechanisms, which comprise the effects of several attacks commonly observed in the Web.

Violating Session Integrity

The first threat is the violation of the *integrity of a session*, as encountered in other papers [21, 8], where an attacker is able to manipulate the session state. For example, if an attacker can remove requests from a session, or insert requests into the session, the integrity of the session is compromised. Note that the attacker is not assumed to have full control over the session, which would allow a transfer to another machine or browser, which is a significantly more powerful attack, as will become clear in the next threat.

A first attack that violates the integrity of the session is cross-site request forgery (CSRF) [295], where an attacker tricks the user's browser to send requests to the target application, which interprets these requests as legitimate. A second attack is UI redressing [112], where the user is tricked into interacting with a seemingly innocuous page, while in fact he is interacting with a hidden page of the target application. In a third attack scenario, an attacker is able to take control of the client-side execution context, for example through a cross-site scripting (XSS) attack [234], allowing him to send arbitrary requests to the application's origin. Note that the main difference between a cross-site scripting attack and a cross-site request forgery (CSRF) attack is the level of control over the origin of the target application, since in a CSRF attack, the attacker mainly controls his own origin, and not that of the application under attack.

Unauthorized Transfer of a Session

The second threat to the security of web sessions is the *unauthorized transfer of a session*, essentially allowing an attacker to take control over the user's session. If an attacker succeeds in taking over a user's session, he can impersonate the user towards the target application, giving the attacker the same privileges as the user. While this threat is more powerful than violating the integrity of a session, the attacker is still bound within this single session, losing his access when the session is terminated, or when a re-authentication is required by the target application.

The most common way to perform an unauthorized session transfer is a session hijacking attack [204], by simply stealing the session identifier. Because this session identifier acts as a bearer token, the attacker can successfully impersonate the user towards the target application. A second attack session fixation [60], which has the same result as a session hijacking attack, but is technically more complicated to carry out. In a session fixation attack, the attacker first establishes a session with the target application, and subsequently transfers this session to the user's browser, causing the user's actions to be carried out within the attacker's session. When the user authenticates himself within this session, the attacker gains access to the user's authenticated session, giving him the same privileges as the user.

Improving session management in the Web is an active research topic, and many proposals effectively mitigate the unauthorized transfer of a session [6, 51, 97, 130], albeit without explicitly naming the security property. One paper [32] that investigates security issues when two applications are hosted on a sibling domain defines a violation of *session confidentiality* as the attacker learning the session identifier, and *session integrity* as the attacker being able to modify the session identifier, which respectively corresponds to a session hijacking and session fixation

attack. Even though these definitions are explicitly tailored towards session management mechanisms that use bearer tokens, our more generic definition of the threat inherently subsumes these bearer token-based definitions.

3.1.4 Current State-of-Practice Countermeasures

Since these attacks on session management mechanisms are well-known and well-documented, several countermeasures are available. Most relevant are the *HttpOnly* and *Secure* cookie flags, which respectively restrict script-based access to cookies, and prevent the transmission of cookies over insecure channels. While these countermeasures offer adequate protection if deployed correctly, they do not fundamentally prevent the unauthorized transfer of an established session, as the session identifier remains a bearer token. Additionally, these countermeasures are often not or incorrectly deployed, even within the Alexa top 100 sites [37], and new attacks that compromise secure deployments have been discovered [11].

Additionally, while the benefits of HTTPS deployments are evident, wide scale adoption on the Web is impeded by several intricacies. One often cited issue is the performance impact, an argument that has lost most of its relevance on modern hardware [150]. Second, HTTPS deployments are disproportionately more complex compared to HTTP deployments, putting a significant burden on system administrators. Examples of such complexities are creating keys, monitoring and renewing certificates, dealing with browser-approved certificate authorities, preventing mixed-content warnings and deploying shared hosting using TLS's Server Name Indication extension [68], if supported by the client. Additional to the complexity of deploying HTTPS, a wide-scale transition to HTTPS severely obstructs the operation of the so-called middleboxes, machines in between the endpoints that cache, inspect or modify traffic. These middleboxes are essential parts of the web infrastructure, for example by bringing the Web to developing nations through extensive caching and enabling efficient video transmission on mobile phone networks.

We acknowledge that wide-scale deployment of HTTPS remains imperative for securing the Web, but also recognize the long and tedious process. This explains why the recent revelations about pervasive monitoring on the Web have sparked multiple proposals looking to transparently upgrade the security properties of the HTTP channel when supported by the endpoints, a topic vividly discussed during the STRINT workshop organized by the STREWS project. One prominent proposal is to negotiate an encrypted HTTP channel without verifying the entities' authentication [207], which is even proposed as one of the available modes in the upcoming HTTP/2.0 specification. This eagerness to improve the security properties of the HTTP protocol, even by introducing them into the new version, shows that the HTTP protocol will be around for the near future. Therefore, it makes sense to not only upgrade the network-level protocol properties, but also to take the opportunity to improve the security properties of session management on top of the HTTP protocol.

3.2 Recent Research on Session Management Mechanisms

Securing session management is an active research topic, which has resulted in a few different approaches to address the problem. Here, we present the four most relevant academic papers. One thing these proposals have in common is that at some point, they rely on TLS to exchange credentials or establish a shared secret. Additionally, some solutions require extensive changes to legacy applications before they can be deployed, or integrate tightly with the authentication process. These restrictions make them incompatible with several common web scenarios, such as supporting legacy applications or third-party authentication services. Nonetheless, these solutions all have their merits and provide valuable insights into secure session management.

3.2.1 SessionLock

SessionLock [6] augments requests with an HMAC based on a shared session secret. The session secret is established over a TLS channel and stored in a secure cookie. For HTTP pages, it is stored in the fragment identifier, a part of the URL that is never sent over the network. SessionLock also supports setup over a non-TLS channel using Diffie-Hellman. At the client-side, SessionLock is implemented using a JavaScript library, making it vulnerable to injection attacks. Additionally, SessionLock requires all requests within the application to be made from JavaScript using AJAX, making it incompatible with most legacy applications.

3.2.2 BetterAuth

BetterAuth [130] is an authentication protocol for web applications, offering protection against several attacks, including network attacks, phishing and cross-site request forgery. BetterAuth considers a user's password to be a shared secret, and uses that shared secret to agree on a session secret over an insecure channel. The session secret is subsequently used to sign requests, offering authenticity. The strength of BetterAuth is that it protects against active network attackers. A disadvantage is that the password needs to be shared with the server, requiring an initial setup phase over TLS. Additionally, because BetterAuth depends on the password, it is incompatible with current third-party authentication services.

3.2.3 One-Time Cookies

One-Time Cookies [51] proposes to replace the static session identifier with disposable credentials based on a modified hash chain construction. Each credential can only be used once, but due to the hash chain construction, all valid credentials can be linked together, enabling session management. To share the initial credential, One-Time Cookies depend on the use of TLS during the authentication phase. Additionally, each credential that is used needs to be acknowledged by the server in the response, making sending parallel requests impossible.

3.2.4 HTTP Integrity Header

The HTTP Integrity Header [97] is an expired draft proposing to add integrity protection to HTTP, which includes a session management mechanism. The header depends on a key exchange, either over TLS or with a traditional Diffie-Hellman exchange, after which the integrity of the selected parts of a message is protected. The use of the HTTP Integrity header offers similar mutual authentication properties as the TLS protocol, albeit without explicitly protecting against active network attackers.

3.2.5 TLS Origin-Bound Certificates

Origin-Bound Certificates (OBC) [64] is an extension for TLS, that establishes a strong authentication channel between browser and server, without falling prey to active network attacks.

Within this secure channel, TLS-OBC supports the binding of cookies and third-party authentication tokens, which prevents the stealing of such bearer tokens. While TLS-OBC offers strong security guarantees, it obviously depends on TLS, making it inapplicable to web applications that do not deploy TLS.

3.3 Progress beyond the State-of-the-Art: Transparent Session Management with SecSess

3.3.1 Objectives for a New Session Management Mechanism

Based on the discussion of the current cookie-based session management mechanism and its associated threats, we identify four design objectives for a new session management mechanism. The first objective is a core design feature, focusing on preventing unauthorized session transfers. The three remaining objectives ensure the feasibility of deployment, covering induced overhead, compatibility with current applications and infrastructure, and a gradual migration path.

Preventing Unauthorized Session Transfer State-of-practice session management mechanisms are vulnerable to session transfer attacks by design, due to their reliance on the session identifier as a bearer token. Newly designed session management mechanisms should actively try to prevent session transfer attacks. Additionally, session management mechanisms should still support authorized transfers, such as desktop-to-mobile synchronization at the client side, and load balancing at the server side.

Minimal Overhead A crucial requirement for a new session management mechanism on the Web is a minimal overhead, well illustrated by browser vendors and web application developers focusing on minimizing page load times. Overhead in the Web is twofold, with on one hand performance overhead, such as additional computations, and on the other hand networking overhead, with increased message sizes and additional roundtrips. Especially the latter is considered problematic, since it delays the processing of the page and the loading of sub-resources, such as style sheets, images, etc.

Compatibility with Current Applications and Infrastructure A newly proposed session management mechanism should be compatible with the current Web and its peculiar deployment scenarios. Examples are the integration of third-party content in Web sites, and the redirection towards third-party service providers, such as a centralized authentication provider. On the infrastructure level, the Web deploys numerous middleboxes, such as web caches at various levels and content inspection systems at network perimeters.

Gradual Migration Finally, a new mechanism looking for adoption on the Web should support a gradual migration path, starting with early adopters on the client and server side, followed by a gradual increase of coverage in the Web. Key in this process is an application-agnostic opt-in session management mechanism, supporting implementation in current clients and server software or application development frameworks, thereby preventing the need for each individual application to incorporate the new mechanism. Additionally, backwards compatibility with parts of the Web that will not quickly adopt the new mechanism is also important, since the Web can not be updated in a single step.

3.3.2 The *SecSess* Approach

In this section, we propose SecSess, a lightweight session management mechanisms that effectively prevents the unauthorized transfer of an established session, by eradicating the bearer token properties of the session identifier. Below, we discuss the details of SecSess in three steps: (i) the actual session management mechanism, (ii) establishing the shared session secret and (iii) the resulting request flow, which is identical to the flow in cookie-based session management mechanisms. The essence of SecSess is establishing a shared secret used for session management between browser and server, which is stored in the browser, where it can not be obtained by

an attacker. SecSess effectively binds an established session to its initiating parties, thereby preventing the unauthorized transfer of an established session.

SecSess improves upon state-of-the-art research by remaining compatible with the flow used by cookie-based session management mechanisms. Additionally, SecSess is compatible with currently deployed middleboxes, a major requirement for building a roadmap towards adoption of an improved session management mechanism.

Session Management A core feature of a session management mechanism is actually keeping track of a session, which SecSess achieves by using a plain session identifier. The session identifier is provided by the server using a *Session* response header (response 1 in Figure 3.2). The browser attaches the session identifier to each request, using the newly introduced *Session* request header. Note that while the use of a session identifier strongly resembles traditional cookie-based session management, the session identifier is no longer considered to be a bearer token, and is useless without knowledge of the shared secret. Therefore, using a simple incremental counter as an identifier is sufficient.

Instead of using the session identifier as the bearer token, SecSess uses the shared secret to add an hash-based message authentication code (HMAC) to the request, thereby legitimizing the request within the session. Since this HMAC takes the request and the shared secret as input, only the browser and the server can compute the correct values. Incoming requests with an invalid HMAC are simply discarded by the server.

Note that the input for the HMAC should be chosen carefully. Technically, a network attacker can steal the valid HMAC from an eavesdropped request and attach it to a crafted request, having the crafted request reach the server first. In order to maintain a valid HMAC on the crafted request, the attacker can only modify the parts of the request that are not part of the input to the HMAC function. Including the URL of the request in this input prevents an attacker from directing the request to a different destination, but still allows him to modify sensitive information in the request headers and body (e.g. the destination account of a wire transfer). Therefore, the HMAC also covers the request headers containing sensitive data[3], and, if present, the request body. Covering the URL, request headers and request body in the HMAC does not prevent an attacker from taking the valid HMAC value and attaching it to a crafted request. However, it does ensure that the attacker can not change the sensitive data, hence limiting the contents of the crafted request to those of the original request, thereby reducing the problem to the common *double submission* problem [260].

Establishing the Shared Secret The shared session secret, needed to compute and verify HMACs on requests, is established using the Hughes variant [240] of the Diffie-Hellman key exchange algorithm, which allows to exchange the key even in the presence of eavesdropping attackers. In Figure 3.2, the server sends his public value (Y) after seeing the first request, in which the browser indicates support for the *Session* header. Using the server's public component Y, the browser can calculate the second public part (X), which the server needs to calculate the key. In the next request, the browser sends the public value X, allowing the server to calculate the full key and verify this and any subsequent requests, effectively establishing the session, as acknowledged in the second response.

Note that the advantage of the Hughes variant of Diffie-Hellman is that the browser can compute the key before the first request is sent. This is required to attach an HMAC to the first request, so the server can verify that the sender of the first and second request are in fact the same. Omission of the first HMAC allows an eavesdropper to respond to the first response, injecting his key material into the session, which is problematic when the first request already caused some server-side state to be stored in the session.

[3]Concretely, we include the following standard HTTP headers: *Authorization*, *Cookie*, *Content-Length*, *Content-MD5*, *Content-Type*, *Date*, *Expect*, *From*, *Host*, *If-Match*, *Max-Forwards*, *Origin*.

Preserving the Request Flow By design, SecSess is an application-agnostic session management mechanism, preserving the same flow of requests and responses as a currently deployed cookie-based session management mechanism. This property supports a gradual deployment, where client and server software can be upgraded to opt-in to SecSess next to cookie-based session management. If the client does not support SecSess, no *Session* header is sent, so the server simply defaults to cookie-based session management. Alternatively, if the client supports SecSess, but the server does not, the *Session* header will be ignored by the server, and the default cookie-based session management mechanism will be used.

Handling Modified Request Flows

Since the Web is a complex distributed system, where multiple simultaneous requests are fired by the browser, request flows often differ from the flows drawn on paper. One example of a modified flow are requests that arrive at the server in a different order than they were sent. A second example are middleboxes changing the request flow, such as a Web cache responding to a request, which will thus not be sent to the server. Since these scenarios are common in the Web, it is important that they are robustly handled by a newly introduced session management mechanism.

The design of SecSess explicitly takes modified request flows into account, and effectively achieves full compatibility with currently deployed web caches, both within the browser and on the intermediate network. First, by only adding integrity protection, the caching of content is effectively enabled. Second, SecSess is robust enough to deal with out-of-order requests and cached responses, which is fairly trivial once a session is established, but challenging during establishment. If the client's public component (request 2 in Figure 3.2) would get lost in transit, for example when an intermediate cache responds to a request, the server would not be able to complete the session establishment, effectively breaking the protocol. Concretely, SecSess addresses this by continuing to send the public component as long as the server has not confirmed the session establishment (response 2 in Figure 3.2), effectively preventing it from getting lost in a modified request flow. We discuss the concrete impact of this decision during the performance evaluation.

3.3.3 Compatibility with the Modern Web

To show the benefits of SecSess over traditional session management mechanisms, we evaluate the security properties of SecSess. Additionally, we provide empirical evidence of SecSess' compatibility with web caches, which are fundamental to the Web's infrastructure.

Security Evaluation

The security evaluation of SecSess with regard to the proposed in-scope threat model considers several concrete attack vectors. A first is the capability to run attacker-controlled scripts within the context of the target application. A session hijacking or session fixation attack using this attack vector will no longer succeed, since none of SecSess's data is available to the JavaScript environment. Additionally, the *Session* request and response headers contain only public information, of no use to an attacker.

Session transfer attacks can also be performed on the network level. Eavesdropping attacks on the session management mechanism are effectively mitigated by SecSess, since the shared secret used for calculation of the HMACs is never communicated over the wire, and the Hughes variant of the Diffie-Hellman key exchange can withstand passive attacks. Next to passive attacks, an attacker can also try to modify existing requests, or re-attach a valid HMAC to a crafted request. Such attempts will fail as well, because the HMAC is based on the contents of the request, effectively preventing any modifications to go unnoticed.

Compatibility with Web Caches

Web caches are widely deployed throughout the Web, enabling faster page loads and limiting the required bandwidth. Caches are often deployed in a transparent way, where they intercept HTTP traffic, and respond when they have the resource in cache. When a cache responds to a request, the request is never forwarded to the target server, resulting in a modified request flow. As stated in one of the proposed design objectives, newly proposed session management mechanisms should be able to cope with infrastructure components of the Web modifying request flows.

SecSess is robustly designed to be compatible with such modified request flows. We have confirmed this compatibility empirically by running experiments with two popular caches, Squid [247] and Apache Traffic Server [261], configured as a forwarding proxy. In our setup (Figure 3.3a), we add SecSess session management on top of the requests sent between the browser and the web servers of the Alexa top 1,000 sites. Since these servers do not know about SecSess, we have added a dedicated SecSess-proxy in between, which will handle the SecSess session management with the browser (full arrows), while forwarding the request to the actual web server (dashed arrows). Finally, we add the cache in between the browser and the SecSess-proxy. This setup allows us to test the establishment and maintaining of a session with traffic patterns from the Alexa top 1,000 sites. Additionally, when the cache responds to a request, the SecSess-proxy will never see the request. This effectively allows us to verify the robustness of SecSess when dealing with modified request flows. The results are shown in Figure 3.3b.

To maximize the potential of the cache, we visited each site in the Alexa top 1,000 twice. For the Squid run, 52,947 requests were sent to 5,167 distinct hosts. Of these requests, 5,008 were cached, of which 830 during session establishment, and 4,178 when an established session was already present. For the Apache Traffic Server run, we observed 44,173 requests to 4,660 hosts in total, of which 4,263 were served from the cache. 1,169 cached responses occurred during session establishment, and 3,094 with an established session. During these requests, SecSess robustly handled session management, without losing an established session, or failing to establish a session.

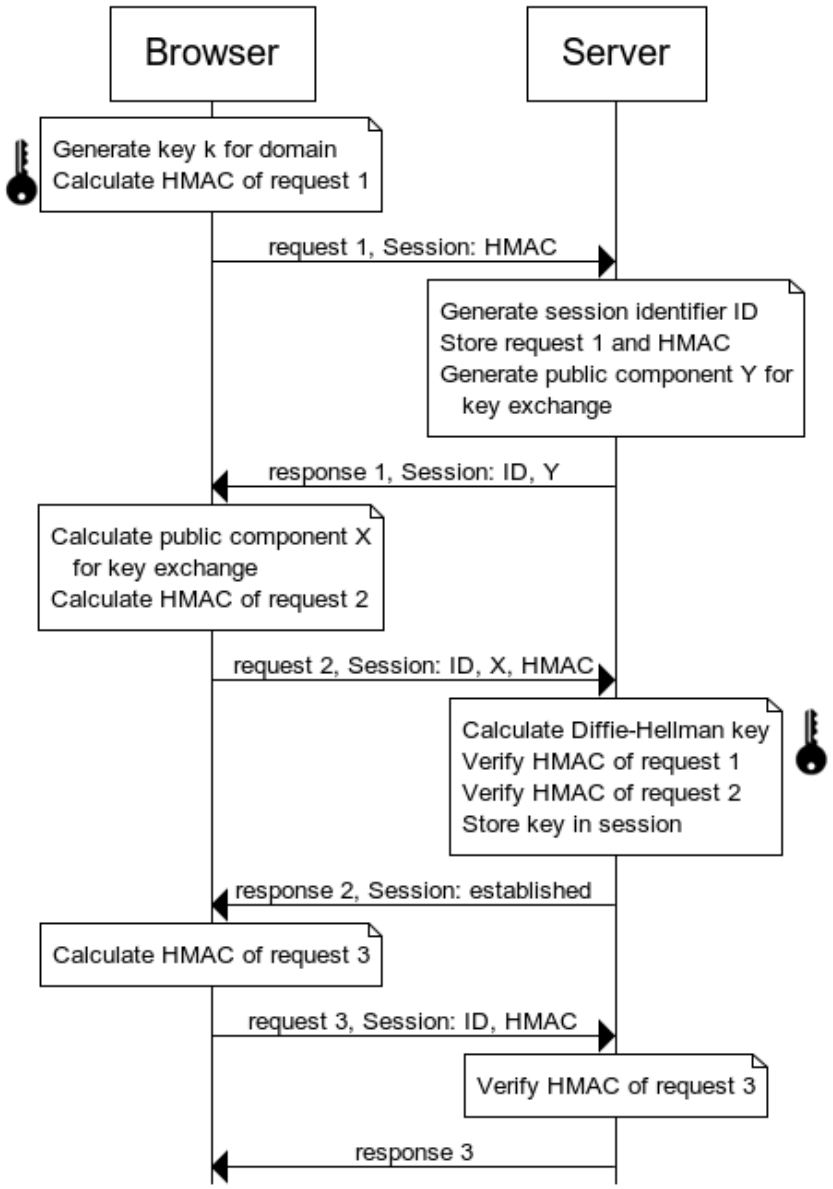
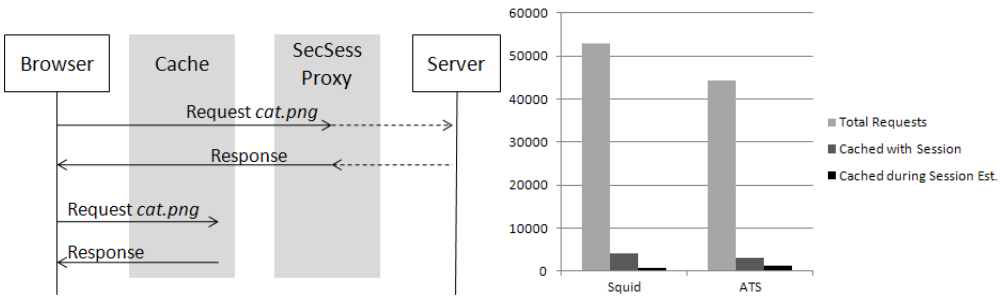


Figure 3.2: The flow of requests and responses used by SecSess, showing all the details.



(a) The setup for browsing the Alexa top 1,000. (b) The results from the cache.

Figure 3.3: The setup and results of the cache compatibility experiment.

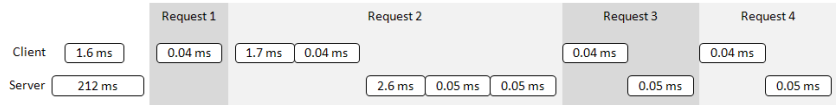


Figure 3.4: SecSess adds an average 4.3 milliseconds to the session establishment. The first step, where the shared secret is generated at the client side and the computation parameters are generated at the server side, takes a bit longer, but can be pre-computed offline or during idle times.

3.4 Roadmap towards Adoption

Pinning a certain state to a set of partners is a trendy concept. Here it is used to secure sessions. While most of the components for session management are standardised themselves, the session management itself is not formalized in a specification. Thus it is not sufficient to address a specific standard and make changes. It rather needs a variety of concerted actions that have to be combined. This includes proof of concept, implementation, deployment considerations and further standardisation.

3.4.1 Implementation

We have created a proof-of-concept implementation of SecSess, which not only allows us to perform a performance evaluation, but also provides valuable insights into the feasibility of deploying SecSess on the Web. At the client side, we have extended the Firefox browser with support for SecSess, heavily leveraging the support of OpenSSL's crypto library. At the server side, we have implemented a session management middleware module for the Express framework, which runs on top of Node.js, an event-driven bare metal web server. The middleware amounts to a mere 113 meaningful lines of code, and a binary module linking the OpenSSL library is 178 meaningful lines of code.

Figure 3.4 shows the performance overhead induced by SecSess on a session establishment timeline. To get correct measurements, we calculated 100 data points for each step, which contain the average computation time of 100 runs each, executed from within JavaScript code, both on the client-side (browser add-on) as the server side (Node.js)[4].

Most notable results are the very limited overhead at the client-side, especially after the session has been established (from request 3 onwards). At the server side, there is a significant pre-calculation overhead (212ms) for generating the required parameters. This overhead is induced by the Hughes variant of the Diffie-Hellman key exchange, which requires the inverse of the server's private component. Note that these parameters are session-independent, and can be pre-calculated offline in bulk, and read from a file on a per-need basis. After the parameters have been calculated, the additional overhead for actually establishing and maintaining a session is negligible.

In a Web context, network overhead can be caused by increased message sizes, but also by introducing additional requests or round trips in the flow of requests. By design, SecSess follows the same sequence of requests and responses used in currently existing applications, which deploy cookie-based session management mechanisms, so no additional requests or round trips are required. For brevity reasons, we can not go into much detail, but we have confirmed that, compared to the sizes of cookies used on the top 5,000 sites, SecSess leads to 25.58% reduction in the size of the session management headers in requests, a 9.19% increase in response headers after session establishment, and a 867.44% increase in response headers during the brief stage of session establishment, due to the transmission of the parameters to generate the secret.

[4]Experiments have been performed in a VirtualBox VM (Linux Mint 15), which was assigned 1 Intel i7-3770 core and 512 Mb of memory.

3.4.2 Deploying SecSess

Implementing SecSess's functionality at the client side is best done within the browser, who is responsible for managing sessions, sending requests and receiving responses. A browser-based implementation has full control over all outgoing requests, regardless whether they originate from within a page, JavaScript or the browser core. Currently, we implemented SecSess as a lightweight add-on, enabling easy development of a proof-of-concept for evaluation with the latest browser. The long-term deployment strategy integrates the current codebase into the browser core, enabling SecSess as an opt-in mechanism at the client side.

At the server side, SecSess can be supported through two complementary implementation strategies. The first option is to integrate SecSess directly into the web platform, while remaining transparent to the target applications, for example as a module within the web server (e.g. as an Apache module), or as part of the underlying application framework (e.g. our Express middleware implementation). The alternative strategy focuses on supporting legacy applications, where the underlying frameworks can not easily be upgraded. Implementing SecSess as a reverse proxy in front of the server provides secure session management towards the browser while shielding the server from these changes.

Since SecSess follows the same traffic flow as current cookie-based session management mechanisms, it remains fully compatible with advanced web technologies. One common example are load balancing techniques deployed at the server side, where multiple servers serve requests for the same target application. Since cookie-based mechanisms already require session tracking or state sharing between servers, SecSess can benefit from the same technology.

3.4.3 Next Steps

The road from inception to adoption of security measures for the Web is difficult, aptly illustrated by the slow and cumbersome adoption process of the HttpOnly cookie attribute [55]. For SecSess, research suggests to invest in a prototype-based strategy, using concrete implementations to show the benefits and feasibility of SecSess. This implementation-driven approach allows us to maximize the opportunities to gain traction within the community and to gain momentum for further standardisation.

We have optimized exposure by presenting SecSess at industry-driven events, such as OWASP's AppSec EU conference, and workshops backed by standardisation committees, such as STREWS' STRINT workshop. Participating in these events has resulted in a promising interaction with developers from the Firefox browser and the Apache web server, both interested in providing respectively client-side and server-side support for SecSess. STREWS will try to gain attention of the Web Application Security Working Group to evaluate interest to include SecSess as one of the items for the next charter. This has to be coordinated with the relevant IETF Working Groups, especially WebSec, as it also has protocol elements owned by the IETF.

In addition to these efforts, we are currently conducting a feasibility study of integrating SecSess directly into the browser core, an essential requirement for a wide-scale deployment on the Web. To extend our support towards multiple platforms, we are integrating SecSess into the Chromium browser, which is based on the Blink rendering engine.

Chapter 4

JavaScript sandboxing[1]

4.1 JavaScript inclusions in the Modern Web

JavaScript

In 1995, Netscape management told Brendan Eich to create a programming language to run in the web browser that "looked like Java." He created JavaScript in only 10 days [41]. In addition to browser plugins, JavaScript was another novel feature of Netscape Navigator 2.0 that supported Netscape's vision of the Web as a distributed operating system. In contrast with Java, which was considered a heavyweight object-oriented language and used to create Java applets, JavaScript would be Java's "silly little brother" [198], aimed towards non-professional programmers who would not need to learn and compile Java applets.

Listing 4.1 shows a simple example of JavaScript. When executed, the code will prompt for the user's name and birth-year. It will then calculate the user's age based on the current year and display it with a greeting using a pop-up. This JavaScript example makes use of the "prompt()" function, the "Date" object and the "alert()" function.

When an HTML document is about to be loaded, and before the rendering pipeline starts, the browser initializes an instance of the JavaScript engine and ties it uniquely to the webpage about to be loaded.

The webpage's developer can use JavaScript to interact with this rendering pipeline by including JavaScript in several ways. JavaScript can be executed while the pages is loading, using HTML <script> tags. These script tags can cause the browser to load external JavaScript and execute them inside the webpage's JavaScript execution environment. Script tags can also contain inline JavaScript, which will equally be loaded and executed. HTML provides a way to register JavaScript event handlers with HTML elements, which will be called when e.g. an

[1] Parts of this chapter have been disseminated in the PhD dissertation of Steven Van Acker [265] (January 2015).

Listing 4.1: Example JavaScript code prompting the user for name and birthyear, calculating age and displaying it in a pop-up.

```
1 var name = prompt("What is your name?");
2 var year = prompt("What year were you born?");
3
4 var today = new Date();
5 var age = today.getFullYear() - year;
6
7 alert("Hello "+name+", you are about "+age+" years young");
```

image has loaded, or the user hovers the mousepointer over a hyperlink. In addition, JavaScript can register these event handlers itself by querying and manipulating the DOM tree. Events are not only driven by the user, but can also be driven programmatically. For instance, JavaScript has the ability to use a built-in timer to execute a piece of JavaScript at a certain point in the future. Likewise, the *XMLHttpRequest* functionality available in the JavaScript engine allows a web developer to retrieve Internet resources in the background, and execute a specified piece of JavaScript code when they are loaded. Lastly, JavaScript has the ability to execute dynamically generated code through the *eval* function.

JavaScript APIs

JavaScript's capabilities inside a web page are limited to the APIs that are offered to it. Typical functionality available to JavaScript in a web page includes manipulating the DOM, navigating the browser and accessing resources on remote servers.

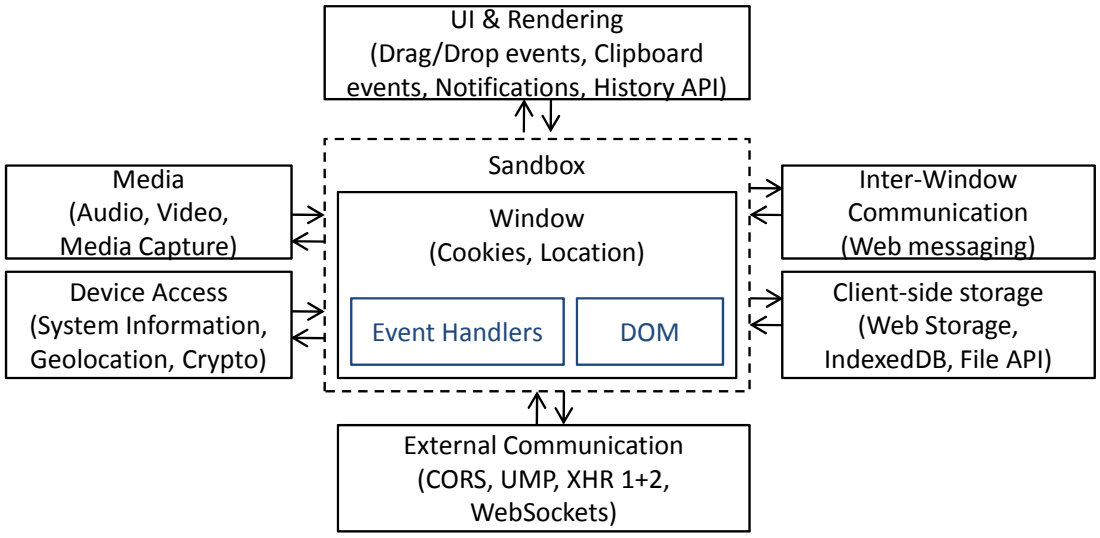


Figure 4.1: Synthesized model of the emerging HTML5 APIs, from [5].

In the new HTML 5 and ECMAScript 5 specifications, JavaScript gains access to more and powerful APIs. Figure 4.1 [57] shows a model of some of these new HTML 5 APIs, which are further explained below and in Section 4.2.2.

Inter-frame communication. facilitates communication between windows (e.g. between mashup components). This includes window navigation, as well as Web Messaging (postMessage).

Client-side storage. enables applications to temporarily or persistently store data. This can be achieved via Web Storage, IndexedDB or the File API.

External communication. features such as CORS, UMP, XMLHttpRequest level 1 and 2, WebSockets, raw sockets and Web RTC (real-time communication) allow an application to communicate with remote websites.

Device access. allows the web application to retrieve contextual data (e.g. geolocation) as well as system information such as battery level, CPU information, ambient sensors and high-resolution timers.

Media. enable a web application to play audio and video fragments, capture audio and video via a microphone or webcam and manage telephone calls through the Web Telephony API.

The UI and rendering. allow subscription to clipboard and drag-and-drop events, issuing desktop notifications, allow an application to go fullscreen, populating the history via the History API and create new widgets with Web Components API and Shadow DOM.

Web applications

A web application combines HTML code, JavaScript and other resources from several web servers, into a functional application that runs in the browser. Unlike typical desktop applications which need to be installed on a computer's hard disk, web applications are accessible through the web browser from anywhere and do not need to be installed.

A key component in today's web application, is JavaScript. JavaScript code in a web application executes in the browser and can communicate with a web server, which typically also executes code for the web application.

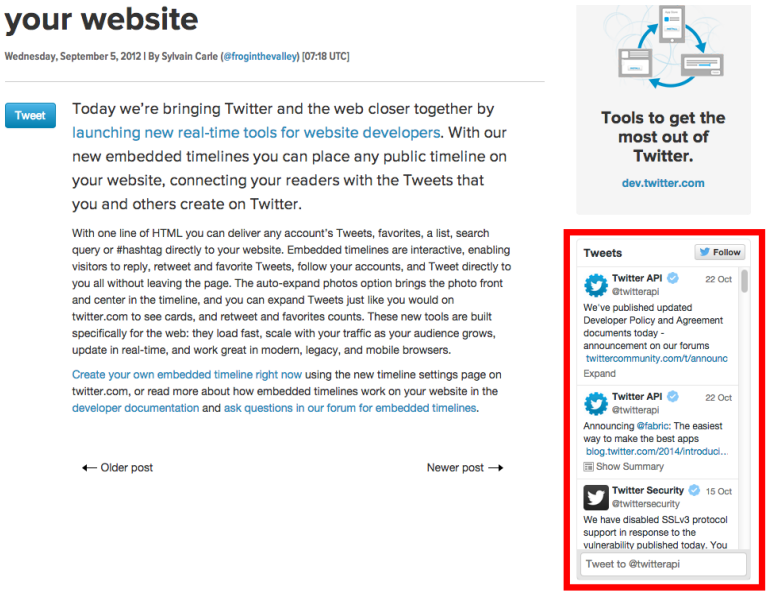


Figure 4.2: Example of an embedded live Twitter feed (indicated by the rectangle on the bottom right), from [264].

Consider a website wishing to display the latest tweets from a Twitter feed. Such a widget can be embedded into a web page, as shown in Figure 4.2. Without a client-side programming language such as JavaScript, the web server from which this web page is retrieved, could gather and insert the latest tweets at the moment the web page was requested, and insert them into the web page as HTML-formatted text. When rendered, the visitor would see the latest tweets, but they would not update themselves in the following minutes because the web page is static.

Another option is to use JavaScript on the client-side. When the web page is requested, the web server can insert JavaScript that regularly requests the latest tweets from the feed and updates the web page to display them. The result is an active web page that always displays the latest information.

This example consists of only one HTML page and requests information from one source. Today's web has many web applications combining a multitude of third-party resources. Examples are Facebook, YouTube, Google Maps and more.

The Same-Origin Policy

If web applications were allowed complete access to a browser, they would be able to interfere in the operation of other web applications running in the same browser. Given the powerful APIs briefly discussed in the previous section, a web application would be able to access another web application's DOM, local storage and data stored on remote servers.

To prevent this, web applications are executed in their own little universe inside the web browser, without knowledge of each other. The boundaries between these universes are drawn based on the *Same-Origin Policy* (SOP) [267].

When the root HTML document of web application is loaded from a certain URL, the *origin* of that web application is said to be a combination of the scheme, hostname and port-number of that URL. For instance, a web application loaded from `https://www.example.com` has scheme *https*, hostname *www.example.com* and, in this case implicit, port number *443*. The origin for this web application is thus *(https,www.example.com,443)* or `https://www.example.com:443`.

The same-origin policy is part of the foundation of web security and is implemented in every modern browser. The *Same-Origin Policy* (SOP) dictates that any code executing inside this origin only has access to resources from that *same* origin, unless explicitly allowed otherwise by e.g. a Cross-Origin Resource Sharing (CORS) [268] policy. In the previous example, the web application from `https://www.example.com:443` cannot retrieve the address book from a webmail application with different origin `https://mail.example.com:443` running in the same browser, unless the latter explicitly allows it.

Insecurely written web applications may allow attackers to breach the same-origin policy by executing their JavaScript code in that web application's origin. Once arbitrary JavaScript code can be injected into a web application, it can take over control and access all available resources in that web application's origin.

Consider a typical webmail application, such as Gmail, allowing an authenticated user to access his emails and contact list. The webmail application offers a user interface in the browser and can send requests to the webmail server to send and retrieve emails, and manipulate the contact list.

An attacker may manage to lure an authenticated user of this webmail application onto a specially crafted website. This website could try to contact the webmail server to send and retrieve emails and contact information, just as the web application would. However, the webmail application's origin is e.g. `https://webmail.com:443`, while the attacker's website is `https://attacker.com:443`. Because of the SOP, JavaScript running on the attacker's website has no access to resources of the webmail's origin.

Now consider what would happen if the webmail application is written insecurely, so that an attacker can execute JavaScript in its origin: `https://webmail.com:443`. Because the attacker's code runs inside the same origin as the webmail application, it has access to the same resources and can also read and retrieve emails and contact information. Because of the power of JavaScript, an attacker can do much more. Specially crafted JavaScript can compose spam email messages and send them out using the victim's email account, or it could erase the contact list. It could even download all emails in the mailbox and upload them to another server.

An attacker with the ability to execute JavaScript in a web application's origin can take full control of that web application. In the typical web application scenario, untrusted JavaScript can be executed in two ways: by including it legitimately from a third party, or by having it injected through a Cross-Site Scripting vulnerability in the web application or an installed browser plugin or extension.

Attacker model

When discussing Web security, it is important to keep in mind a typical web application with third-party JavaScript and the actors involved in it. Figure 4.3 shows such a typical web application where HTML and JavaScript from a trusted source are combined with JavaScript from

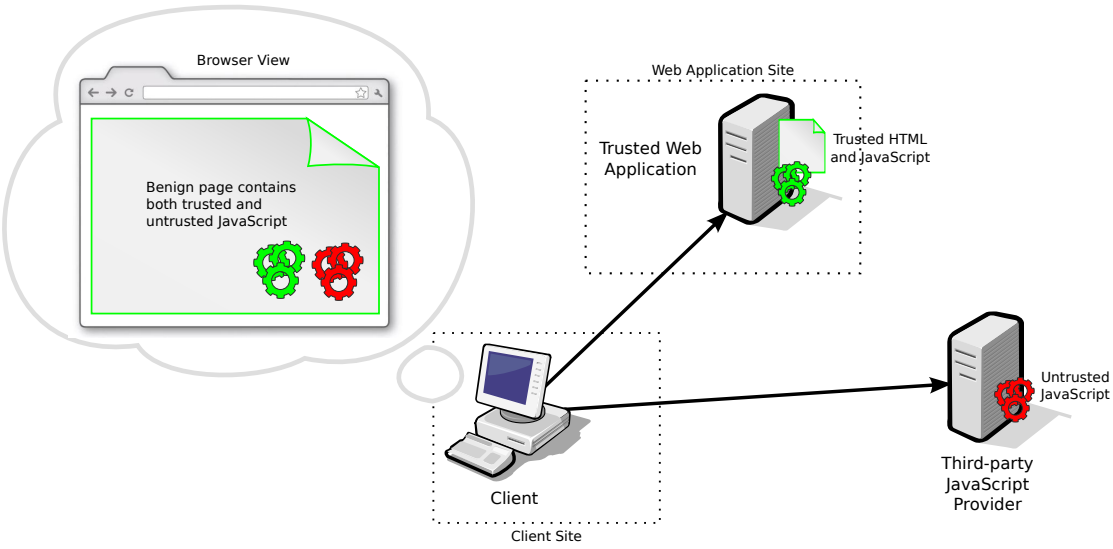


Figure 4.3: A typical web application with third-party JavaScript inclusion. The web application running in the browser combines HTML and JavaScript from a trusted source, with JavaScript from an untrusted source.

an untrusted source. Remember that all JavaScript, trusted or untrusted, running in a web application's origin has access to all available resources.

There are three actors involved in this scenario: The developer of the trusted web application and the server it is hosted on, the developer of the third-party JavaScript and the server it is hosted on, and the client's browser.

Both the client and the trusted web application have a clear motive to keep untrusted JavaScript from accessing the web application's resources. The client will wish to protect his own account and data. The trusted web application has its reputation to consider and will protect a user's account and data as well. Furthermore, the client does not need to steal information from himself and can use any of his browser's functionality without needing to use a remote web application. Likewise, the web application developer owns the origin in which the web application runs. Stealing data from his own users through JavaScript is not necessary.

It may be the case that the client has modified his browser and installed a browser plugin or extension. Such a plugin or extension may be designed to make the interaction with the web application easier or automated, potentially circumventing certain defensive measures put in place by the developer of the web application. In this scenario, the client is still motivated to protect his account and data, but may be exposing himself to additional threats through the installed browser plugins or extensions that form additional attack surface.

The third-party script provider however, does not necessarily share the same desire to protect a user's data. Even with the best of intentions, a third-party script provider may be compromised and serving malicious JavaScript without its knowledge. It may be the case that the script provider has an intrusion-detection system in place that will detect when it is serving malware, but this would be wishful thinking. In the worst case, the third-party script provider is acting maliciously on its own for whatever sinister reason. In any case, the client and trusted web application cannot trust a third-party script provider with their secrets.

The attacker model best associated with this actor is the *gadget attacker* [23]. A gadget attacker is a malicious actor who owns one or more machines on the Internet, but can neither passively not actively intercept network traffic between the client's browser and the trusted web application. Instead, the gadget attacker has the ability to have the trusted web application's developer integrate a gadget chosen by the attacker.

Third-party script inclusion

Web applications are built from several components that are often included from third-party content providers. JavaScript libraries like jQuery or the Google Maps API are often directly loaded into a web application's JavaScript environment from third-party script providers.

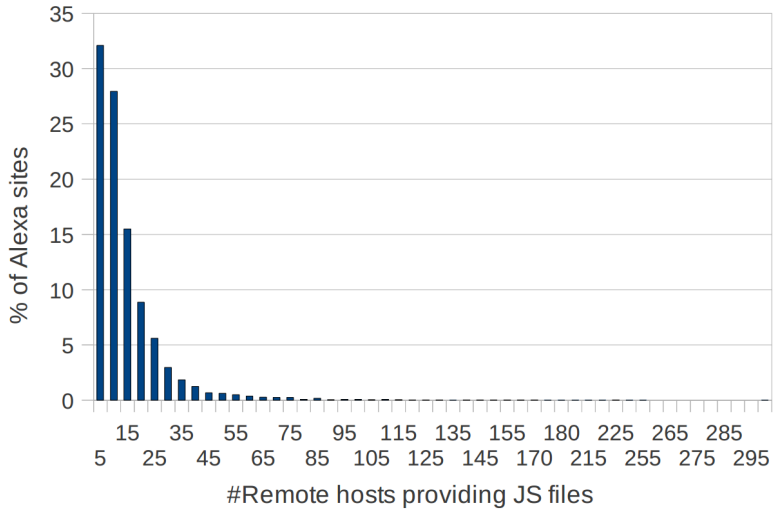


Figure 4.4: Relative frequency distribution of the percentage of top Alexa websites and the number of unique remote hosts from which they request JavaScript code, from [203].

In a large-scale study of the Web in 2012 [203], Nikiforakis et al. found that 88.45% of the top 10,000 web sites on the Web, include JavaScript from a third-party script provider. Figure 4.4 shows the distribution of the number of third-party script providers each web site includes. While about a third include JavaScript from at most 5 remote hosts, there are also web sites that include JavaScript from more than 295 different remote hosts.

Including JavaScript from remote hosts implicitly trusts these hosts not to serve malicious JavaScript. If these third-party script providers are untrustworthy, or if they have been compromised, a web application may end up executing untrusted JavaScript code.

As an example, consider jQuery, a popular multi-purpose JavaScript library used on 60% of the top million websites on the Web [36]. The host distributing jQuery was compromised in September 2014 [133], giving the attackers the ability to modify the library and possibly infect many websites that include the library directly from `http://jquery.com`. Fortunately, the attackers did not modify the jQuery library itself, but used the compromised server to spread malware instead. Although the JavaScript library itself was not tampered with, the jQuery compromise indicates the inherent security threat that third-party script inclusions can pose.

Listing 4.2: Example JavaScript calling eval() on user input, but only if its MD5 hash matches a given hash.

```
1  var cmd = prompt();
2
3  // MD5 algorithm computes a one-way hash
4  function md5(m) {
5      // ...
6      return m;
7  }
8
9  // verifies whether the given input is "alert('hello')"
10 if(md5(cmd) == "3b022ec21226e862450f2155ef836827") {
11     eval(cmd);
12 }
```

4.2 Recent Research on Sandboxing Mechanisms for JavaScript

The gadget attacker, as defined in Section 4.1, has the ability to integrate a malicious gadget into a trusted web application. This allows the attacker to execute any chosen JavaScript code in the JavaScript execution environment of this trusted web application's origin and access its sensitive resources.

Given this attacker model, we cannot stop the attacker from presenting a web application user's browser with chosen JavaScript. Instead, we can isolate the JavaScript and restrict its access to certain resources and functionality, by executing this code in a JavaScript sandbox.

From the typical web scenario architecture described in Section 4.1, keeping in mind our attacker model, there are only two possible locations that can be considered to deploy a JavaScript sandboxing mechanism: the trusted web application and the client's browser. The third-party script provider is considered untrustworthy.

The developer of the web application and the server hosting it, are trusted according to the attacker model. This server then offers a possible location to facilitate JavaScript sandboxing. Before serving the untrusted JavaScript from the third-party script provider to the client, the code can be reviewed and optionally rewritten to make sure it does not abuse the web application's available resources.

The client's browser provides a second location to sandbox JavaScript, because it is also considered trusted. With direct access to the JavaScript execution context, a JavaScript sandboxing system located at the client-side has better means to restrict access to resources and functionality.

4.2.1 JavaScript subsets and rewriting

JavaScript is a very flexible and expressive programming language which gives web-developers a powerful tool to build web-applications. However, this same powerful tool is also available to attackers wishing to execute malicious JavaScript code in a website visitor's browser.

Moreover, the powerful nature of JavaScript is problematic because it hinders code verification efforts which could prove safety properties for a given piece of JavaScript code.

Example: eval() Consider for instance the JavaScript fragment in Listing 4.2. When executed in a browser, this code will prompt a user to input a line of text. The one-way hashing algorithm MD5 is then used to compute a hash of this line of text. If the hash matches "3b022ec21226e862450f2155ef836827", the MD5 hash for "alert('hello')", then the line of text is passed to the eval() function and executed as JavaScript code.

Given that the MD5 hashing algorithm cannot easily be reversed, it is practically impossible for a code verification tool to automatically determine the effect of this code, prior to its

Listing 4.3: Example JavaScript using the "with" construct to place a new object at the front of the scope chain during the evaluation of the construct's body. This example is adapted from Miller et al. [185].

```
1  var o = {f:2, x:4};
2
3  console.log("before with: f == " + f);
4  console.log("before with: x == " + x);
5  console.log("before with: \"x\" in window == " + ("x" in window));
6
7  with(o) {
8      function f() { }
9      console.log("inside with: f == " + f);
10     var x = 3;
11     console.log("inside with: x == " + x);
12 }
13
14 console.log("after with: o.f == " + o.f);
15 console.log("after with: o.x == " + o.x);
16 console.log("after with: f == " + f);
17 console.log("after with: x == " + x);
18 console.log("after with: \"x\" in window == " + ("x" in window));
```

Listing 4.4: Output of example in Listing 4.3.

```
1  before with: f == function f() { }
2  before with: x == undefined
3  before with: "x" in window == true
4  inside with: f == 2
5  inside with: x == 3
6  after with: o.f == 2
7  after with: o.x == 3
8  after with: f == function f() { }
9  after with: x == undefined
10 after with: "x" in window == true
```

execution. The eval() function illustrates a feature of JavaScript which makes code verification difficult because of its dynamic nature. For this reason, eval() is considered evil [226] and should be used with the greatest care, or not be used at all.

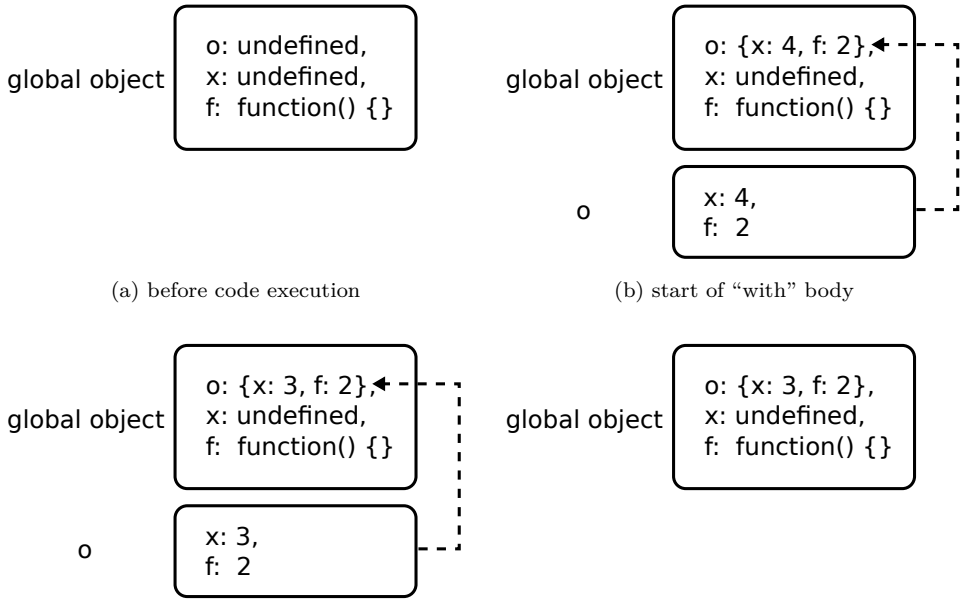
Example: Strange semantics and scoping rules As another example, the JavaScript fragment in Listing 4.3 illustrates some strange semantic rules in JavaScript, including the "with" construct. This particular example showcases some non-intuitive scoping rules associated with the scope chain. The scope chain consists of an ordered list of JavaScript objects which are consulted when unqualified names are looked up at runtime.

Before continuing, the reader is advised to read the code and try to predict what it will output. The actual output of the code in this example, is listed in Listing 4.4.

From the output, it appears that both "f" and "x" are already defined before they are even declared, but "x" has "undefined" as value. Using "with", the user-defined object "o" is pushed to the front of the scope chain. The new function "f()" is declared, but the subsequent "console.log()" call seemingly is not aware it. Instead, the value of "f" is retrieved from the first object in the scope chain ("o"), resulting in "2." Then, a variable "var x" is declared and assigned "3." The following "console.log()" call is aware of this declaration and outputs the correct value. Outside the with loop, the object "o" has changed to reflect the new value of "o.x", but did not record any change to "o.f".

The strange behavior in this example indicates that variable and function declarations have

different semantics in JavaScript. The discrepancy between variable and function declarations can be explained by a process called "variable hoisting." Variable hoisting examines the JavaScript code to be executed and performs all declarations before any code is actually run.



(a) before code execution (b) start of "with" body

(c) end of "with" body (d) after code execution

Figure 4.5: The scope chain during execution of the example in Listing 4.3. In this depiction, the scope chain grows down so that newly pushed objects are at the bottom.

A graphical representation of the scope chain during the execution of this example is shown in Figure 4.5 and can be used as a visual aid during the explanation.

Depicted in Figure 4.5a is the result of the variable hoisting before any code is run. The function "f()" and the variable "x" are declared on the global object. While the variable "x" has value "undefined", the function "f()" is declared and is assigned its value immediately.

Next, the object "o" is pushed to the front of the scope chain. The scope chain right after this push and right before the start of the "with" construct, is shown in Figure 4.5b. Any unqualified names are now looked up in the variable "o".

The third image shown in Figure 4.5c, depicts the state of the scope chain at the end of the "with" body. Here, the value of the property "x" of the object "o" has changed to "3" because of the assignment. Also note that the value of "f" has not changed because variable hoisting declares and initializes a function in a single step before the code is run, and so outside of the "with" body.

Finally, in Figure 4.5d, the scope chain is restored because the "with" body ended.

The strange scoping rules and semantics of "with" are difficult to reason about for uninitiated programmers. Widely-acknowledged as being a "JavaScript wart" [94], it is often recommended to not use the "with" construct because it may lead to confusing bugs and compatibility issues [192].

JavaScript subsets: verification and rewriting The goal of JavaScript code verification and rewriting is to inspect JavaScript code before it is executed in a browser, and ensure that it is not harmful.

In the light of the previous examples, it can be desirable to eliminate those constructs from the JavaScript language that hinder code verification efforts or cause confusion in general. At the

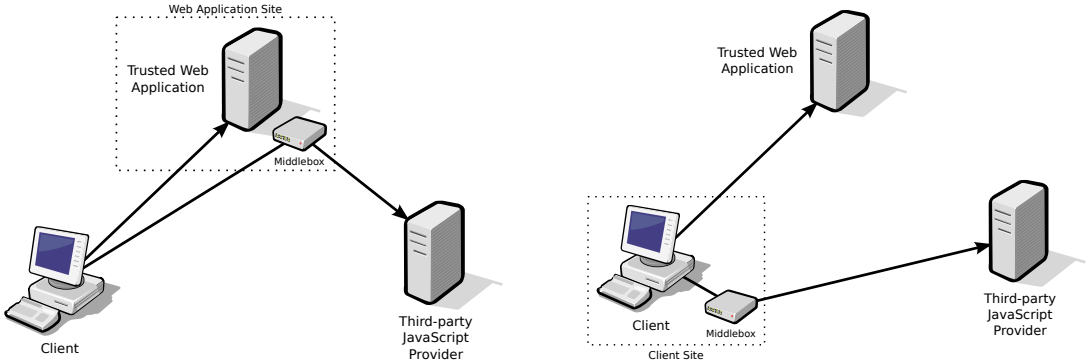
same time, it is also desirable to maintain as much of the language as possible so that JavaScript is still useful. Such a reduced version of JavaScript, with e.g. “eval()” and “with” construct missing, is called a JavaScript subset.

The usage of a JavaScript subset must be accompanied by a mechanism which verifies that a given piece of code adheres to the subset. A deviation from the subset’s specification can be handled in two ways: rejection and rewriting.

Rejection is the simpler of both options, treating a deviation from the subset as a hard error and refusing to execute the given piece of code.

Rewriting is a softer alternative, transforming the deviating piece of code into code which conforms to the subset. Such a rewriting phase can also introduce extra instrumentation in the code to ensure that the code behaves in a safe way at runtime.

Interception in a middlebox Both the JavaScript subset verification and rewriting steps necessitate the processing of raw third-party JavaScript code before it reaches the client’s browser. These steps are to be performed in a *middlebox*, a network device that sits on the network path between a client and a server. Such a middlebox may consist of a physical device unrelated to either client or server, but it may just as well be collocated with either client or server.



(a) Middlebox at web application site

(b) Middlebox at client site

Figure 4.6: Architectural overview of a setup where a middlebox is used for code verification and transformation, at the web application site and at the client site.

From the attacker model discussed in Section 4.1, we can eliminate the third-party script provider’s site as a possible location to verify and rewrite JavaScript. We are left with two possible locations for these tasks: the site of the trusted web application and the client’s site.

A middlebox at the site of the web application, as shown in Figure 4.6a, can equally be implemented as part of a separate network device such as a load-balancer, reverse proxy or firewall, or can be integrated to be part of the web-application.

A middlebox at the client’s site, as shown in Figure 4.6b, can either be a implemented as a proxy performing the required verification and translation steps, or as a browser plugin or extension, implementing the proxy’s behavior as part of the browser.

ECMAScript 5 strict mode ECMAScript 5 strict mode [190], or JavaScript strict, is a standardized subset of JavaScript with intentionally different semantics than normal JavaScript.

To use strict mode, a JavaScript developer must only place “use strict”; at the top of a script of function body, as shown in Listing 4.5. Strict mode will then be enforced for that entire script, or only in the scope of that function. JavaScript strict mode can be mixed with and function together with normal JavaScript.

Strict mode removes silent failures and turns them into hard errors that throw exceptions and halt JavaScript execution. For instance, accidentally creating a global variable by mistyping

Listing 4.5: JavaScript strict mode example.

```
1  "use strict";
2
3  var example = 123;
4  // the following fails because the name is misspelled
5  exmaple = 345;
6
7  // the following fails because of a duplicate key name
8  var obj = {p:1, p:2};
9
10 // the following fails because "with" is not allowed
11 with(obj) {
12     alert(p);
13 }
```

a variable name, will throw an error. Likewise, overwriting a non-writable global variable like "NaN" or defining an object with a duplicate key, causes strict mode to throw errors.

Strict mode simplifies variable names and allows better JavaScript engine optimization by removing the "with" construct. Through this construct, JavaScript engine optimizations may be confused about the actual memory location of a variable. In addition, strict mode changes the semantics of "eval()" so that it can no longer create variable in the surrounding scope.

Strict mode also introduces some fixes with regard to security. It is no longer possible to access the global object through the "this" keyword, preventing unforeseen runtime leaks. It is also no longer possible to abuse certain variables to walk the stack or access the "caller" from within a function.

Finally, strict mode forbids the use of some keywords that will be used in future ECMAScript versions, such as "private", "public", "protected", "interface", ...

Research in the area of JavaScript subsets and rewriting systems includes BrowserShield [224], CoreScript [291], ADsafe [49], Facebook JavaScript [262], Caja [185], Jacaranda [117], Microsoft Live Websandbox [183], Jigsaw [179], Gatekeeper [93], Blancura [85], Dojo Secure [145], ... The remainder of this section discusses a selection of work on JavaScript subsets and rewriting systems.

BrowserShield

Reis et al. have developed BrowserShield, a dynamic instrumentation system for JavaScript. BrowserShield parses and rewrites HTML and JavaScript in a middlebox, rewriting all function calls, property accesses, constructors and control structures to be relayed through specialized methods of the *bshield* object. A client-side JavaScript library then inserts this *bshield* object, which mediates access to DOM methods and properties according to a policy, into the JavaScript execution environment before any scripts run.

BrowserShield aims at preventing the exploitation of browser vulnerabilities, such as MS04-40 [181], a buffer overflow in the Microsoft Internet Explorer browser caused by overly long "src" and "name" attributes in certain HTML elements. To shield the browser from attacks against these vulnerabilities, BrowserShield rewrites both HTML and JavaScript, transforming them to filter out any detected attacks. BrowserShield does not use a JavaScript subset, because it needs to be able to rewrite any HTML and JavaScript found on the Internet to be effective.

Although sandboxing is not the main goal of BrowserShield, its rewriting mechanism provides all the necessary machinery to accomplish this goal by tuning the policy. For instance, BrowserShield could have a policy in place to mediate access to the sensitive eval() function. Listing 4.6 shows the output of BrowserShield's rewriting mechanism on a JavaScript example using the eval() function. After the rewriting step, any call to eval() in the original code is relayed through the "bshield" object, which can mediate access at runtime.

Listing 4.6: Example JavaScript code rewritten by BrowserShield.

```
1 // original JavaScript code
2 eval("...");
3
4 // rewritten by BrowserShield
5 bshield.invokeFunc(eval, "...");
```

A prototype of BrowserShield was implemented as a Microsoft ISA Server 2004 [180] plugin for evaluation. The plugin in this server-side middlebox is responsible for rewriting HTML and script elements, and injecting the BrowserShield client-side JavaScript library which implements the "bshield" object and redirects all JavaScript functionality through it. BrowserShield worked as expected during evaluation. The performance evaluation indicated a maximum slowdown of 136x on micro-benchmarks, and on average 2.7x slowdown on rendering a webpage.

ADsafe

The ADsafe subset, developed by Douglas Crockford, is a JavaScript subset designed to allow direct placement of advertisements on webpages in a safe way, while enforcing good coding practices. It removes a number of unsafe JavaScript features and does not allow uncontrolled access to unsafe browser components.

Examples of the removed unsafe JavaScript features are: the use of global variables, the use of "this", eval(), "with", using dangerous object properties like "caller" and "prototype". ADsafe also does not allow the use of the subscript operator, except when it can be verified that the subscript is numerical, e.g. a[i] is not allowed but a[+i] is allowed because "+i" will always produce a number. In addition, ADsafe removes all sources of non-determinism such as "Date" and "Math.random()".

To make use of ADsafe, widgets must be loaded and executed via the "ADSAFE.go()" method. These widgets must adhere to the ADsafe subset, although there is no verification built into ADsafe. Instead, it is recommended to verify subset adherence in any stage of the deployment pipeline with e.g. JSLint [134], a JavaScript code quality verification tool.

ADsafe does not allow JavaScript code to make use of the DOM directly. Instead, ADsafe makes a "dom" object available which provides and mediates access to the DOM.

No performance evaluation has been published about ADsafe by its author, who claim that ADsafe "will not make scripts bigger or slower or alter their behavior" [49]. This claim applies if advertisement scripts are written in the ADsafe subset directly, and not translated from full JavaScript.

Research on ADsafe has revealed several problems and vulnerabilities, which allow leaking the document object [258], launch a XSS attack [85], allow the guest to access properties on the host page's global object [219], prototype poisoning [168] and more.

Facebook JavaScript

Facebook JavaScript (FBJS) is a subset of JavaScript and part of the Facebook Markup Language (FBML) which was used to publish third-party Facebook applications on the Facebook servers. FBJS was designed to allow web application developers as much flexibility as possible while at the same time protecting site integrity and the privacy of Facebook's users.

The FBJS subset excludes some of JavaScript's dangerous constructs such as "eval", "with", "___parent___", "constructor" and "valueOf". A preprocessor rewrites FBJS code so that all top-level identifiers in the code are prefixed with an application-specific prefix, thus isolating the code in its own namespace.

Special care is also taken with e.g. the use of "this" and object indexing to retrieve properties, making sure that a Facebook application cannot break out of its namespace. The semantics of

Listing 4.7: Example JavaScript code making use of "this" semantics to return the global object and the code rewritten by FBJS to prevent FBJS code from breaking out of its namespace.

```
1 // original code
2 (function() { return this; })();
3
4 // code rewritten by FBJS
5 (function() { return ref(this); })();
```

"this" are dependent on the way and location that it is used. A code fragment such as the one listed in Listing 4.7 can return the global object, allowing FBJS code to break out of its namespace. To remedy this problem, the FBJS rewriter encloses all references to "this" with the function "ref()", e.g. "ref(this)". This "ref()" function verifies the way in which it is called at runtime, and prevent FBJS code from breaking out of its namespace. Similarly, the FBJS rewriter also encloses object indices such as "property" in "object["property"]" with "idx("property")" to also prevent that "this" is bound to the global object.

Research on FBJS has revealed some vulnerabilities [168, 167], which were addressed by the Facebook team.

Maffeis et al. [168] discovered that a specially crafted function can retrieve the current scope object through JavaScript's exception handling mechanism, allowing the "ref()" and "idx()" functions to be redefined. This redefinition in turn allows a FBJS code to break out of its namespace and take over the webpage.

After Facebook fixed the previous issues, Maffeis et al. [167] discovered another vulnerability which allows the global object to be returned on some browsers, by tricking the fixed "idx()" function to return an otherwise hidden property, through a time-of-check-time-of-use vulnerability [189].

Caja

Google's Caja, short for Capabilities Attenuate JavaScript Authority, is a JavaScript subset and rewriting system using a server-side middlebox. Caja represents an object-capability safe subset of JavaScript, meaning that any code conforming to this subset can only cause effects outside itself if it is given references to other objects. In Caja, objects have no powerful references to other objects by default and can only be granted new references from the outside. The capability of affecting the outside world is thus reflected by holding a reference to an object in that outside world.

The Caja subset removes some dangerous features from the JavaScript language, such as "with" and "eval()". Furthermore, Caja does not allow variables or properties with names ending in "___" (double-underscore), while at the same time marking variables and properties with names ending in "_" as private.

Listing 4.8: Example JavaScript code to be cajoled by Caja.

```
1 window.alert("hello world");
```

Caja's rewriting mechanism, known as the "cajoler," examines the guest code to determine any free variables and wraps the guest code into a function without free variables. Listing 4.8 shows some example code and its cajoled form is shown in Listing 4.9 (the "cajoledcode" variable). In addition, Caja adds inline checks to make sure that Caja's invariants are not broken and that no object references are leaked. The output of the cajoler is cajoled code, which is sent to a client's browser.

Listing 4.9: Conceptual cajoled code and tamed window.

```
1 var tamedwindow = tame(window);
2 var cajoledcode = function(param) {
3    param.alert("hello world");
4 };
5
6 cajoledcode(tamedwindow);
```

On the client-side, objects from the host webpage are "tamed" so that they only expose desired properties before being passed to the cajoled guest code. These tamed objects with carefully exposed properties are the only references that cajoled code obtains to the host page. In this way, all accesses to the DOM can be mediated by taming the global object before passing it to cajoled code. Listing 4.9 shows how the "window" object is tamed and passed to the cajoled form of Listing 4.8.

Discussion

The JavaScript language makes static code verification difficult, because of its dynamic nature (e.g. "eval()") and strange semantics (e.g. the "with" construct). JavaScript subsets eliminate some of JavaScript's language constructs so that code may be more easily verified. When required, JavaScript rewriting systems can transform the code so that policies can also be enforced at runtime.

This section discussed four JavaScript subsets and rewriting mechanisms: BrowserShield, ADsafe, Facebook JavaScript and Caja. Some of their features are summarized in Table 4.1.

It is noteworthy that all three JavaScript subsets remove "with" and "eval()" from the language, which is in line with the standardized JavaScript strict mode subset. The only available performance benchmarks are for BrowserShield, which rewrites code written in full JavaScript, and indicate a heavy performance penalty when rewriting JavaScript in a middlebox. Furthermore, the list of known weaknesses suggest that creating a secure JavaScript subset, although possible, is not an easy task.

JavaScript subsets and code rewriting have been used in real world web applications and have proved to be effective in restricting available functionality to selected pieces of JavaScript code. However, restricting the integration of third-party JavaScript code which conforms to a specific JavaScript subset, puts limitations on third-party JavaScript library developers which they are unlikely to follow without incentive. Even if these developers are willing to limit themselves to a JavaScript subset, they would need to create a version of their code for every subset that they need to conform too. For instance, the jQuery developers would need to create a specific version for use with FBJS, Caja, ADsafe etc. This is an unrealistic expectation.

The standardization of a JavaScript subset, such as e.g. strict mode, helps eliminate this disadvantage for third-party JavaScript providers. But even with a standardized JavaScript subset to aid with code verification, this verification step itself must still happen in a middlebox located at either the server-side or the client-side.

Opting for a middlebox on a server-side has the disadvantage that it changes the architecture of the Internet. From the browser's perspective, JavaScript code would need to be requested from the middlebox instead of directly downloading it from the third-party script provider. Although this poses no problem for generic JavaScript libraries such as jQuery, it does pose a problem for JavaScript code which is generated dynamically depending on the user's credentials, as is the case with e.g. JSONP. In the latter case the third-party script provider might require session information to prove a user's identity, which will not be provided by the browser when requesting said script from a server-side middlebox.

A client-side middlebox on the other hand, does not suffer from this particular problem because it has the option of letting the browser connect to it transparently, e.g. in case of a

Table 4.1: Comparison between prominent JavaScript sandboxing systems using subsets and rewriting systems.

System	Target application	Rewrites	Uses subset	Removed features	Performance	Known weaknesses
BrowserShield	Preventing browser exploitation	Y	N	n/a	max. 136x slowdown on micro-benchmarks, avg. 2.7x slowdown on user experience	
ADsafe	Advertising	N	Y	“eval()”, “with”, “this”, global vars, ...	no slowdown	[258] [85] [219] [168]
FBJS	Third-party widgets	Y	Y	“eval()”, “with”, ...	no data	[168] [167]
Caja	Third-party widgets	Y	Y	“eval()”, “with”, ...	no data	

Listing 4.10: Example JavaScript calling setTimeout() with unknown input.

```
1 var x = ...;
2 window.setTimeout(x, 1000);
```

web proxy. With a client-side middlebox, the web application developers lose control over the rewriting process. Users of the web application should setup the middlebox on the client-side in order to make use of this web application. But requiring users to install a middlebox next to their browser for a single web application, hurts usability and puts a burden on users which they might not like to carry.

From a usability viewpoint, it makes more sense to require only a single middlebox which can be reused for multiple web applications and to integrate this client-side middlebox into the browser somehow.

4.2.2 JavaScript sandboxing using browser modifications

The previous section showed that JavaScript contains several language constructs that cannot easily be verified to be harmless before executing JavaScript code. Instead of verifying the code beforehand, another approach is to control the execution of JavaScript at runtime and monitor the effect of the executing JavaScript to make sure no harm is done.

In a typical modular architecture of a browser, as explained in chapter 3 of the Web-platform Security Guide[59], the JavaScript environment is disconnected from other browser components. These other components, such as the DOM, the network layer, the rendering pipeline or HTML parser are not directly accessible to JavaScript code running in the JavaScript environment. Without these components, JavaScript is effectively side-effect free and is unable to affect the outside world.

The connection layer between the JavaScript engine and the different browser components, is an excellent location to mediate access to the powerful functionality that these components can provide. In order to enforce a policy at this location, the browser must be modified with a mechanism that can intercept, modify and block messages between the JavaScript engine and the different components.

Example: allowing only Function object parameters for setTimeout() Consider the example in Listing 4.10. In this example, the DOM API function "setTimeout()" is called with a parameter "x." The specification for the "setTimeout()" function in the Web application API standard [283] lists two versions: a version where "x" must be a Function object, and a version that allows it to be a String. Passing a string to the "setTimeout()" function is regarded as a bad coding practice and considered as evil as using "eval()" [135]. Because of the inherent difficulty in verifying JavaScript code before runtime, it can be desirable to enforce a policy at runtime which rejects calls to "setTimeout()" when a string is passed as an argument.

The "setTimeout()" function is provided by a browser component which implements timer functionality. To access this function, the JavaScript engine must send a message to this component to invoke the timer functionality, as shown in Figure 4.7. At this point, a browser modified with a suitable policy enforcement mechanism can intercept the message, and reject it if the given parameter is not a Function object.

Forms of browser modifications Browser modifications can take many forms, but they can generally be split into three groups: browser plugins, browser extensions and browser core modifications. Browser plugins and browser extensions can add extra functionality to the browser that can be used to enforce a JavaScript sandboxing technique. They are however limited in the modifications they can make in the browser environment.

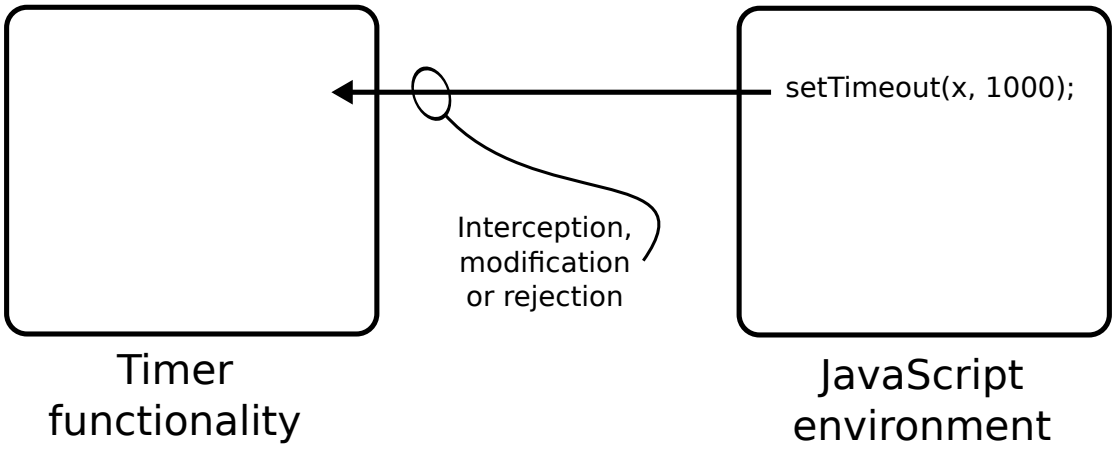


Figure 4.7: Executing the "setTimeout()" function will send a message from the JavaScript environment to the component implementing timer functionality, which can be intercepted, modified or rejected by a policy enforcement mechanism in a modified browser.

Listing 4.11: Example whitelist policy implemented in BEEP's *afterParseHook* function, from [123].

```
1  if (window.JSSecurity) {
2    JSSecurity.afterParseHook =
3      function(code, elt) {
4        if (whitelist[SHA1(code)]) return true;
5        else return false;
6      };
7    whitelist = new Object();
8    whitelist["478zB3KkS+UnP2xz8x62ugOxvd4="] = 1;
9    whitelist["AO0q/aTVjJ7EWQIsGVeKfdg4Gdo="] = 1;
10   ... etc. ...
11 }
```

For more advanced modifications to the browser, such as e.g. the JavaScript engine or the HTML parser, it is typically the case that neither plugins nor extensions are suitable. Therefor, modifying the browser core itself is required.

Research on JavaScript sandboxing through some form of browser modification, includes BEEP [123], ConScript [178], WebJail [5], Contego [165], AdSentry [65], JCShadow [212], Escudo [122], ...

Browser-Enforced Embedded Policies (BEEP)

Jim et al. introduce Browser-Enforced Embedded Policies, a browser modification that introduces a callback mechanism, called every time JavaScript is about to be executed. The callback mechanism provides a hook named *afterParseHook* inside the JavaScript environment, which can be overridden by the web developer.

Every time a piece of JavaScript is to be executed, the browser calls the *afterParseHook* callback to determine whether the piece of JavaScript is allowed to execute or not. To be effective, BEEP must be the first JavaScript code to load in the JavaScript environment, in order to set up the *afterParseHook* callback.

The authors experimented with two types of policies: whitelisting and DOM sandboxing.

In the whitelisting policy approach, illustrated in Listing 4.11, the *afterParseHook* callback function receives the script to be executed, and hashes is with the SHA-1 hashing algorithm.

Listing 4.12: Example HTML with the "noexecute" attribute to be used with BEEP's DOM sandboxing policy.

```
<div class="noexecute">
  <!-- possibly-malicious content starts here -->
  <script>
      alert("hello world");
  </script>
  <!-- possibly-malicious content ends here -->
</div>
```

Listing 4.13: A node-splitting attack against the example in Listing 4.12. Notice how the enclosing "div" element with "noexecute" attribute is closed by an attacker-injected closing "div" element.

```
<div class="noexecute">
  <!-- possibly-malicious content starts here -->
  </div><script>
      alert("hello world");
  </script><div>
  <!-- possibly-malicious content ends here -->
</div>
```

This hash is then compared with a list of hashes for allowed scripts. If the hash is found among this whitelist, the *afterParseHook* callback returns "true" and the script is executed.

In the DOM sandboxing policy approach, illustrated in Listing 4.12, HTML elements in the web page are clearly marked with a "noexecute" attribute if they can potentially contain untrusted content such as third-party advertising. When a script is about to be executed, the *afterParseHook* callback function receives both the script and the DOM element from which the execution request came. The *afterParseHook* callback function then walks the DOM tree, starting from the given DOM element and following the references to parent nodes. For each DOM node found in this walk, the callback function checks for the presence of a "noexecute" attribute. If such an attribute is found, the *afterParseHook* callback function returns false, rejecting script execution.

The authors report two problems with this last approach. First, in an attack to which the authors refer to as "node-splitting," an attacker may write HTML code into the webpage, allowing him to break out of the enclosing DOM element on which a "noexecute" attribute is placed. Shown in Listing 4.13, an attacker could easily break out of the DOM sandboxing policy by closing and opening the enclosing "div" tag which has the "noexecute" attribute set, hereby escaping its associated policy of rejecting untrusted scripts. Second, an attacker can introduce an HTML frame, which creates a child document. The *afterParseHook* callback function inside this child document would not be easily able to walk up the parent's DOM tree to check for "noexecute" attributes.

BEEP was implemented in the Konqueror and Safari browsers, and partially in Opera and Firefox. Performance evaluation indicates an average of 8.3% and 25.7% overhead on the load-time of typical webpages for a whitelist policy and DOM sandboxing policy respectively.

ConScript

Meyerovich et al. present ConScript, a client-side advice implementation for Microsoft Internet Explorer 8. ConScript allows a web developer to wrap a function with an advice function using *around advice*. The advice function is registered in the JavaScript engine as *deep advice* so that it cannot be altered by an attacker.

Listing 4.14: Example HttpOnly cookie policy defined on a script element using ConScript, adapted from ConScript [178].

```
<head>
  <script policy='
    let httpOnly: K -> K = function(_ : K) {
      curse(); throw "HTTP-only cookies"; };
    around(getField(document, "cookie"), httpOnly);
    around(setField(document, "cookie"), httpOnly);
    '>
  </script>
</head>
```

As with BEEP, ConScript's policy enforcement mechanism must be configured before any untrusted code gains access to the JavaScript execution environment. ConScript introduces a new attribute "policy" to the HTML <script> tag, in which a web developer can store a policy to be enforced in the current JavaScript environment. When the web page is loaded, ConScript parses this "policy" attribute and registers the contained policy.

Unlike shallow advice, which is within reach of attackers and must be secured in order to prevent tampering by an attacker, ConScript registers the advice function as "deep advice" inside the browser core, out of reach of any potential attacker.

Listing 4.14 shows a ConScript policy being defined in the head of a web page. The policy in this particular example enforces the usage of "HttpOnly" [182] cookies, a version of HTTP cookies which cannot be accessed by JavaScript. To achieve this goal, the policy defines a function "HttpOnly" which simply throws an exception, and registers this function as "around" advice on the getter and setter of the "cookie" property of the "document" object, from which regular cookies are accessible in JavaScript.

Using *around advice* as an advice function allows a policy writer full freedom to block or allow a call to an advised function, possibly basing the decision on arguments passed to the advised function at runtime.

ConScript uses a ML-like subset of JavaScript with labeled types and formal inference rules as its policy language, which can be statically verified for common security holes. To showcase the power of ConScript and its policy language, the authors define 17 example policies addressing a variety of observed bugs and anti-patterns, such as: disallowing inline scripts, restricting XMLHttpRequests to encrypted connections, disallowing cookies to be leaked through hyperlinks, limiting popups and more.

ConScript was implemented in Microsoft Internet Explorer 8 and its performance evaluated. On average, ConScript introduces a slowdown during micro-benchmarks of 3.42x and 1.24x after optimizations. The macro-benchmarks are reported to have negligible overhead.

WebJail

Van Acker et al. propose WebJail, a JavaScript sandboxing mechanism which uses deep advice functions like ConScript.

In WebJail, HTML iframe elements are used as the basis for a sandbox. A new "policy" attribute for an iframe element allows a web developer to specify the URL of a WebJail policy, separating concerns between web developers and policy makers.

The authors argue that an expressive policy language such as ConScript's can cause confusion with the integrators who need to write the policy, thus slowing the adoption rate of a sandboxing mechanism. In addition, they warn for a scenario dubbed "inverse sandbox," in which the policy language is so expressive that an attacker may use it to attack a target web application by sandboxing it with a well-crafted policy. For instance, if the policy language is the JavaScript language, an attacker may define a policy on an iframe to intercept any cookie-access and

Listing 4.15: Example WebJail policy allowing inter-frame communication, external communication to Google and YouTube, but disallowing access to the Device API, from [5].

```
{
  "framecomm": "yes",
  "extcomm": ["google.com", "youtube.com"],
  "device": "no"
}
```

transmit these cookies to an attacker-controlled host. A target web-application could then be loaded into this iframe and would, upon accessing its own cookies, trigger the policy mechanism which leaks the cookies to the attacker.

To avoid this scenario, WebJail abstracts away from an overly expressive policy language and defines its own secure composition policy language. Based on an analysis of sensitive JavaScript APIs in the HTML5 specifications, we divided the APIs into nine categories. The policy consists of a file written in JSON, describing access rights for each of these categories. Access to a category of sensitive JavaScript APIs in WebJail can be granted or rejected with "yes" or "no," or determined based on a whitelist of allowed parameters. Listing 4.15 shows an example WebJail policy which allows inter-frame communication (framecomm: yes), external communication to Google and YouTube (extcomm: ["google.com", "youtube.com"]), but disallowing access to the Device API (device: no).

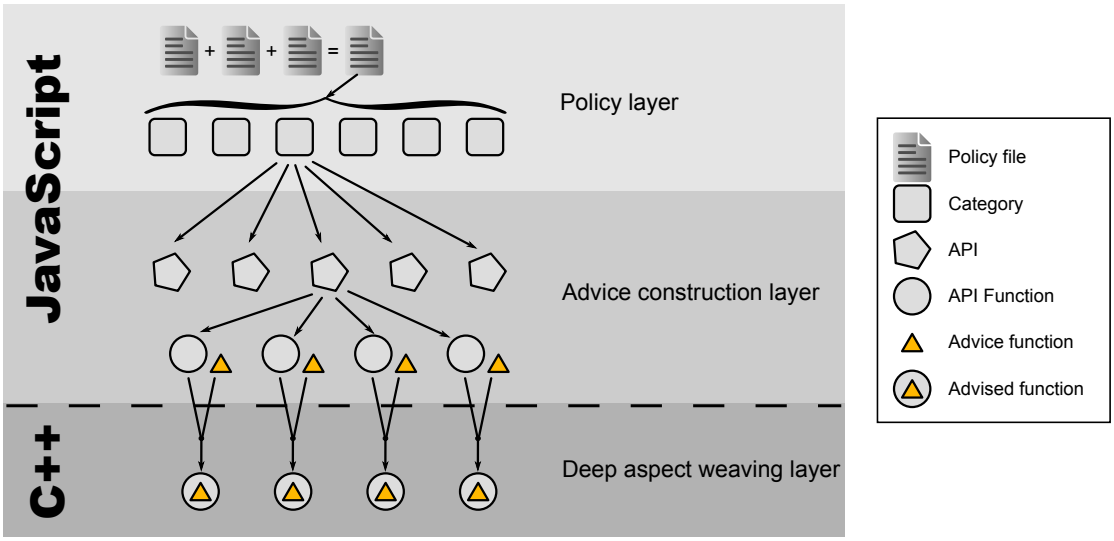


Figure 4.8: The WebJail architecture consists of three layers: the policy layer, the advice construction layer and the deep aspect weaving layer, from [5].

WebJail's architecture, depicted in Figure 4.8 consists of three layers to process an integrator's policy and turn it into deep advice. The policy layer reads an iframe's policy and combines with the policies of any enclosing iframes. Policy composition is an essential step to ensure that an attacker cannot easily escape the sandbox by creating a child document without a policy defined on it. The advice construction layer processes the composed policy and creates advice functions for all functions in the specified JavaScript APIs. Finally, the deep aspect weaving layer combines the advice functions with the API functions, turning them into deep advice and locking them safely inside the JavaScript engine.

WebJail was implemented in Mozilla Firefox 4.0b10pre for evaluation. The performance evaluation indicated an average of between 6.4% and 27% for micro-benchmarks and an average

Listing 4.16: Example usage of Contego and its capability bitstring, from [165].

```
<div cap="110001111"> ... </div>
<!--
  Capability bitstring:
    1 AJAX POST request allowed
    1 AJAX GET request allowed
    0 Cookie setting not allowed
    0 Cookie getting not allowed
    0 Cookie using not allowed
    1 HTTP GET request allowed
    1 HTTP POST request allowed
    1 Hyperlink click allowed
    1 Button submit click allowed
-->
```

of 6 ms loadtime overhead for macro-benchmarks.

Contego

Luo et al. design and implement Contego, a capability-based access control system for browsers.

In a capability-based access control model, the ability of a principal to perform an action is called a capability. Without the required capability, the principal cannot perform the associated action.

Contego's authors identified a list of capabilities in browsers, among which: performing Ajax requests, using cookies, making HTTP GET requests, clicking on hyperlinks, ... They list three types of actions that can be associated with those capabilities, based on where they originate: HTML-induced actions, JavaScript-induced actions and event-driven actions.

Contego allows a web developer to assign capabilities to elements in the DOM tree, by assigning a bit-string to the "cap" attribute. Each bit in the bit-string indicates whether a certain capability should be enabled ("1") or disabled ("0") for all DOM elements enclosed by the element on which the capabilities apply. The meaning of each bit in the bit-string is shown in Listing 4.16, which also shows an example policy disabling access to cookies.

The authors warn about a node-splitting attack when an attacker is allowed to insert content into a element. Just as with BEEP's DOM sandboxing policy, care should be taken to avoid that an attacker can insert a closing tag and escape the policy. In addition, Contego has measures in place to ensure that an attacker cannot override capability restrictions by e.g. setting a new "cap" attribute either in HTML or in JavaScript. Cases where principals with different capabilities interact are handled by restricting the actions to the conjunction of the capability sets.

To implement Contego in the Google Chrome browser, the authors extended the browser with two new components: the binding system and the enforcement system. The binding system assigns and tracks individual principal's capabilities within a webpage. The enforcement system then uses the information from the binding system to allow or deny actions at runtime.

The performance evaluation shows an average overhead of about 3% on macro-benchmarks.

AdSentry

Dong et al. propose AdSentry, a confinement solution for JavaScript-based advertisements, which executes the advertisements in a special-purpose JavaScript engine.

An architectural overview of AdSentry is shown in Figure 4.9. Next to the regular JavaScript engine, AdSentry implements an additional JavaScript engine, called the shadow JavaScript engine, as a browser plugin. The browser plugin is built on top of the Native Client (NaCl) [90]

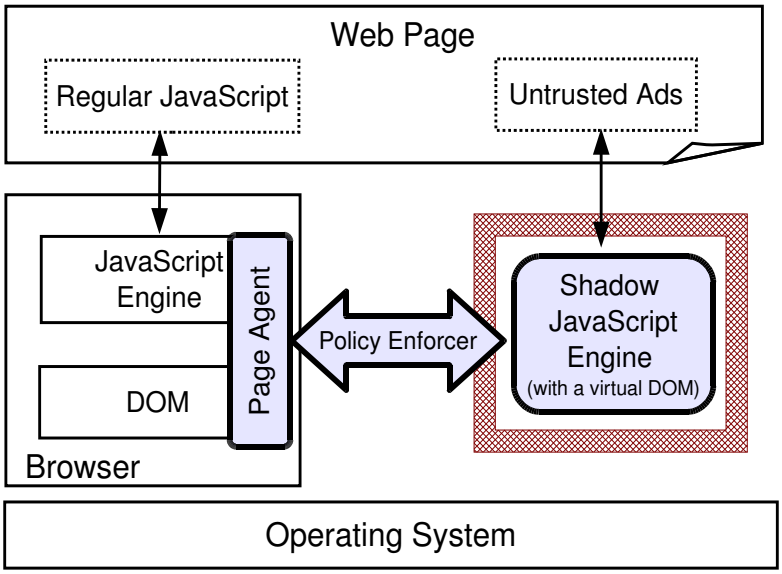


Figure 4.9: The AdSentry architecture: advertisements are executed in a shadow JavaScript engine which communicates with the Page Agent via the policy enforcer, from [65].

Listing 4.17: Format of the communication protocol used between AdSentry's shadow JavaScript engine and the Page Agent, from [65].

```
1 msg ::= command data
2 command ::= script | callFunc | getProp
3                | setProp | return
4 data ::= <text>
```

sandbox, which protects the browser and the rest of the operating system from drive-by-download attacks occurring inside the sandbox.

Advertisements can either be explicitly marked for use with AdSentry, or they can be automatically detected by AdBlock Plus. When an advertisement is detected in a webpage, AdSentry assigns it a unique identifier and communicates with the shadow JavaScript engine to request that the code be executed there. The shadow JavaScript engine then creates a new JavaScript context with its own global object and virtual DOM and executes the advertisement.

The virtual DOM inside the shadow JavaScript context has no access to the real webpage on which the advertisement is supposed to be rendered. Instead, the methods of the virtual DOM are stubs which trigger the shadow JavaScript engine to communicate with a Page Agent in the real JavaScript engine, requesting access on behalf of the advertisement. The communication between the Page Agent and the shadow JavaScript engine is facilitated with a data exchange protocol, shown in Listing 4.17. This communication channel is also where AdSentry's enforcement mechanism operates, granting or blocking access to the real webpage's DOM according to a user-specified policy. No information is given on how this policy can be specified.

AdSentry was implemented in Google Chrome, and uses a standalone version of SpiderMonkey, Mozilla's JavaScript engine, as the shadow JavaScript engine. The performance evaluation indicates an average overhead of 590x on micro-benchmarks when traversing the boundary between the shadow JavaScript engine and the Page Agent, and an around 3% to 5% overall loadtime overhead on macro-benchmarks.

Table 4.2: Comparison between prominent JavaScript sandboxing systems using a browser modification.

System	Target application	Isolation unit	Restricts	Policy expressiveness	Deployment	Browser	Performance	Known weaknesses
BEEP	restrict scripts	entire JS environment	execution of JS scripts	full JavaScript to indicate "accept" or "reject"	*afterParseHook* implementation by integrator	Konqueror, Safari, partially Opera, partially Firefox	8.3% to 25.7% macro	node-splitting
ConScript	sandboxing	entire JS environment	??? anything	high: own JS subset	"policy" attribute on script element	MSIE	1.24x to 3.42x micro, negligible macro	
WebJail	sandboxing	entire JS environment + subframes	access to sensitive APIs	yes/no/whitelist	"policy" attribute on iframe element	Firefox	6.4% to 27% micro, 6 ms macro	
Contego	restrict capabilities	element	capabilities	bitstring	"cap" attribute on div element	Chrome	3% macro	
AdSentry	advertisement	shadow JavaScript engine	access to the DOM	???	???	Chrome	590x micro, 3% to 5% macro	

Discussion

This section discussed five browser modifications that aim to isolate and restrict JavaScript code in the web browser: BEEP, ConScript, WebJail, Contego and AdSentry. Some of their features are summarized in Table 4.2.

JavaScript sandboxing through a browser modification allows the integration of third-party scripts written in the full JavaScript language. Web applications can be built with a much richer set of JavaScript libraries, since those JavaScript libraries are not confined to a subset of JavaScript.

In addition, a browser modification can control the execution of JavaScript inside the browser, allowing the construction of efficient custom-built machinery to enforce a sandboxing policy, ensuring low overhead.

However, modified browsers pose a problem with regard to dissemination of the software and compatibility with browsers and browser versions. End-users must take extra steps in order to enjoy the protection of this type of JavaScript sandboxing systems.

Because end-users do not all use the same browser, it becomes impossible to assure that all end-users can keep using their own favorite browser. In the most fortunate case, the developers of this browser core modification may find a way to port their sandboxing system to all browsers. Even if this is the case, a browser core modification is a fork in a browser's code base and must be maintained to keep up with changes in the main code base, which can be a significant time investment.

Likewise, a browser plugin or extension implementing a certain JavaScript sandboxing system, must also be created for all browser vendors and versions, to enable a wide range of users to make use of it. Such a plugin or extension must equally be maintained for future releases of browsers, which can also require a significant time investment.

All in all, modifying a browser through a fork of browser code, a browser plugin or a browser extension in order to implement a JavaScript sandboxing system, is acceptable for a prototype, but proves difficult in a production environment.

An alternative approach is to convince major browser vendors to implement the browser modification as part of their main code base, or even better, pass it through the standardization process so that all browser vendors will implement it. This approach will ensure that the sandboxing technique ends up in a user's favorite browser automatically and that the code base is maintained by the browser vendors themselves.

Unfortunately, getting a proposal accepted by the standardization committees is not a straightforward task, partly because no solution is widely accepted as being "The Solution."

In recent years, the standardization process has yielded new and powerful functionality that could be used to build a JavaScript sandboxing system. Through this approach, a JavaScript sandboxing system would not need any browser modification at all and work out of the box on all browsers that support the latest Web standards.

4.2.3 JavaScript sandboxing without browser modifications

The previous section showed that a sandboxing mechanism implemented as a browser modification, can be used to restrict JavaScript functionality available to untrusted code at runtime. A browser modification is useful for proof-of-concept evaluation of a sandboxing mechanism, but proves problematic in a production environment. Not only must a browser modification be maintained with new releases of the browser on which it is based, but end-users must also be convinced to install the modified browser, plugin or extension.

Given the powerful nature of JavaScript, it is possible to isolate and restrict untrusted JavaScript code at runtime, without the need for a browser modification. This approach is challenging because the enforcement mechanism will execute in the same execution environment as the untrusted code it is trying to restrict. Special care must be taken to ensure that the

untrusted code cannot interfere with the enforcement mechanism, and this without any added functionality to protect itself from the untrusted code.

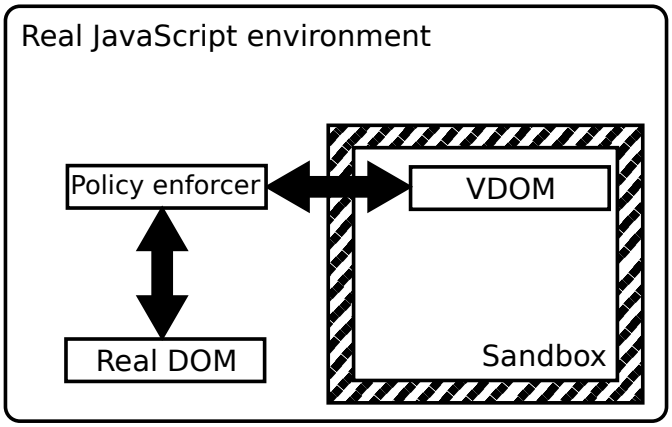


Figure 4.10: Relationship between the real JavaScript environment and a sandbox. The sandbox can only interact with a Virtual DOM, which forwards it via the policy enforcer to the real DOM.

Isolation unit and communication channel Following the same rationale as in the previous section, a good approach is to create an isolated unit (or sandbox) which is completely cut off from any sensitive functionality, reducing it to a side-effect free execution environment. Figure 4.10 sketches the relationship between a sandbox and the real JavaScript environment.

Any untrusted code executed in the sandbox, will not be able to affect the outside world, except through a virtual DOM introduced into this sandbox. To access the outside world, the isolated code must make use of the virtual DOM, which will forward the access request over a communication channel to an enforcement mechanism. If the access is allowed, the enforcement mechanism again forwards the access request to the real JavaScript environment.

New and powerful ECMAScript 5 functionality The rise of Web 2.0 resulted in the standardization of ECMAScript 5, which brought new and powerful functionality to mainstream browsers. This new functionality can help with the isolation and restriction of untrusted JavaScript code.

An example of such functionality is the WebWorker API, or WebWorkers [269]. WebWorkers allow web developers to spawn background workers to run in parallel with a web page. These workers are intended to perform long-running computational tasks in the background, while keeping web pages responsive to user interaction.

WebWorkers have a very restricted API available to them, which only allows them to do very basic tasks such as set timers, perform XMLHttpRequests or communicate through "postMessage()". In particular, WebWorkers have no access to the DOM. Communication between WebWorkers and a web page is achieved through the postMessage API.

Having new ECMAScript 5 functionality in place in browsers today, opens new options for JavaScript sandboxing mechanisms which previously required browser modifications or code verification/transformation in a separate middlebox.

For instance, because WebWorkers restrict JavaScript code from accessing the DOM and other sensitive JavaScript functionality, they can be used as the isolation unit for a JavaScript sandboxing mechanism. TreeHouse, discussed farther in this section, uses WebWorkers as its isolation unit.

Research on JavaScript sandboxing without browser modification includes Self-protecting JavaScript [216, 169], AdJail [160], Object Views [177], JSand [7], TreeHouse [115], SafeScript [161],

Listing 4.18: Simplified version of Self-protecting JavaScript's creation of a wrapper around the "alert()" function, allowing it to be called maximum twice.

```
1  var wrapper = (function (original) {
2      // counter keeps state across
3      // multiple function calls
4      var counter = 0;
5
6      // create and return the wrapper
7      return function(m) {
8          if(counter < 2) {
9              original(m);
10             counter++;
11         }
12     }
13 })(window.alert);
14
15 window.alert = wrapper;
```

IceShield [104], SafeJS [39], Two-tier sandbox [215], Virtual Browser [38], ... A selection of this work is discussed in the following sections.

Self-Protecting JavaScript

Phung et al. propose a solution where DOM API functions are replaced by wrappers which can optionally call the original functions, to which the wrapper has unique access. The wrappers can be used to enforce a policy and, with the ability to store state inside the wrapper function's scope, allow the enforcement of very expressive policies. Access to sensitive DOM properties can also be limited by defining a getter and setter method on them which implements a restricting policy.

An example of how a DOM function is replaced with a wrapper, is shown in Listing 4.18. In this example, a wrapper for the function "alert()" is created with a built-in policy to only allow the function to be called twice. A reference to the original native implementation of "alert()" is kept inside the wrapper's scope chain, making it only accessible by the wrapper itself. Finally, the original "alert()" function is replaced by the wrapper.

It is vital that the wrappers are created and put in place of the original DOM functions before any other JavaScript runs inside the JavaScript environment, to achieve full mediation. If any untrusted JavaScript code is run before the wrappers are in place, an attacker may keep copies of the original DOM functions around, thus bypassing any policies that are placed on them later.

The authors warn that references to DOM functions can also be retrieved through the "contentWindow" property of newly created child documents. To prevent this, access to the "contentWindow" property is denied.

A bug in the "delete" operator of older Firefox browsers also allows overwritten DOM functions to be restored to references to their original native implementations, by simply deleting the wrappers.

A performance evaluation of Self-protecting JavaScript revealed a average of 6.33x slowdown on micro-benchmarks, and a 5.37% average overhead for macro-benchmarks.

Magazinius et al. [169] analyzed Self-protecting JavaScript and uncovered several weaknesses and vulnerabilities that allow the sandboxing mechanism to be bypassed by an attacker.

They note that the original implementation does not remove all references to DOM functions from the JavaScript environment, leaving them open to abuse from attackers. The "alert()" function for instance, has several aliases (such as "window.___proto___.alert"), which must all be replaced with a wrapper for Self-protecting JavaScript to be effective.

Equally, simply denying access to the "contentWindow" property is not sufficient to prevent references to DOM functions from being retrieved from child documents. These references can also be access from child documents through the "frames" property of the "window" object, or from the parent document through the "parent" property of the "window" object.

They also point out that Self-protecting JavaScript is vulnerable to several types of prototype poisoning attacks, allowing an attacker to get access to the original, unwrapped DOM functions as well as the internal state of a policy wrapper.

Lastly, they remind that an attacker could abuse the caller chain during a wrapper's execution, by gaining access to the non-standard "caller" property available in functions, allowing an attacker to gain access to the unwrapped DOM functions.

Finally, Magazinius et al. offer solutions to remedy these vulnerabilities by making sure any functions and objects used inside a wrapper are disconnected from the prototype chain to prevent prototype poisoning, and coercing parameters of functions inside wrappers to their expected types in order to further reduce the attack surface.

AdJail

Ter Louw et al. propose AdJail, an advertising framework which enforces JavaScript sandboxing on advertisements.

AdJail allows a web developer to restrict what parts of the web page an advertisement has access to, by marking HTML elements in that web page with the "policy" attribute. This "policy" attribute contains the AdJail policy that is in effect for a certain HTML element and its sub-elements.

The AdJail policy language allows the specification of what HTML elements can be read or written to, and whether that access extends to its sub-elements. The web developer can also define a policy to enable or disable images, Flash or iframes, restrict the size of an advertisement to a certain height and width and allow clicked hyperlinks to open web pages in a new window.

By default, an advertisement is positioned in the "default ad zone," an HTML element that aids the web developer in positioning the advertisement in the web page. The default policy is set to "deny all."

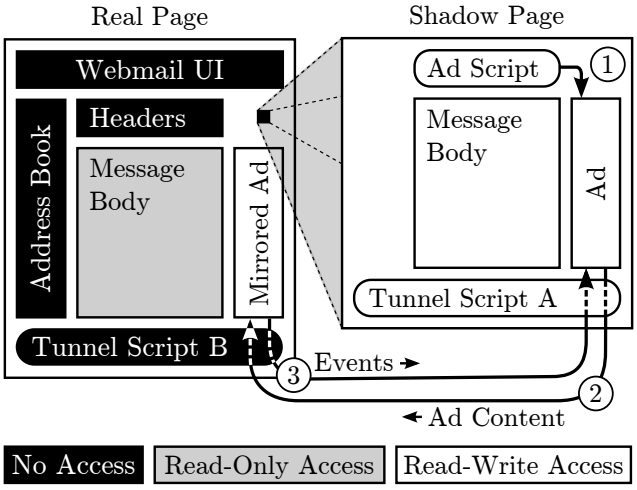


Figure 4.11: Overview of AdJail, showing the real page, the shadow page and the tunnel scripts through which they communicate and on which the policy is enforced, from [160].

An overview of AdJail is shown in Figure 4.11. The advertisement is executed in a "shadow page," which is a hidden iframe with a different origin, so that it is isolated from the real web

Listing 4.19: Pseudo-code showing how an Object View around an object "obj" can be used to intercept reading and writing a property, and intercepting a function call, from [177].

```
var wrapper = ...;
var obj = { prop: 123, func: function() {
    alert("hello world");
    }
};

defineSetter(wrapper, "prop",
    function(x) {
        obj.prop = x;
    });

defineGetter(wrapper, "prop",
    function() {
        return obj.prop;
    });

wrapper.func = function() {
    obj.func(arguments);
};

alert(wrapper.prop); // displays 123
wrapper.prop = 456;  // sets obj.prop to 456
wrapper.func()       // displays "hello world"
```

page. Those parts of the real web page's DOM that are marked as readable by the advertisement, are replicated inside the shadow page before the advertisement executes.

Changes made by the advertisement inside the shadow page, are detected by hooking into the DOM of the shadow page, and communicated to the real page through a tunnel script. The changes are replicated on the real page if allowed by the policy. Likewise, events generated by the user on the real page, are communicated to the shadow page so that the advertisement can react to them.

Because AdJail is aimed at sandboxing advertisements, special care must be taken to ensure that the advertisement provider's revenue stream is not tampered with. In particular, AdJail takes special precautions to ensure that content is only downloaded once, to avoid duplicate registration of "ad impressions" on the advertisement network. Furthermore, AdJail leverages techniques used by BLUEPRINT [162] to ensure that an advertisement does not inject scripts into the real webpage.

Performance benchmarks indicate that AdJail has an average overhead of 29.7% on ad rendering, increasing the rendering time from an average of 374 ms to 532 ms. Further analysis showed that AdJail has an average overhead of 25% on the entire page loadtime, increasing it from 489 ms to 652 ms.

Object Views

Meyerovich et al. introduce Object Views, a fine grained access control mechanism over shared JavaScript objects.

An "Object View" is a wrapper around an object that only exposes a subset of the wrapped object's properties to the outside world. The wrapper consists of a proxy between the wrapped object and the outside world, and a policy that determines what properties should be made available through the proxy.

Sketched in Listing 4.19, an Object View contains a getter and setter method for each property on the wrapped object, and a proxy function for each function object. Writing a value to a property on an Object View, triggers the setter function which may eventually write the value to

Listing 4.20: A declarative policy rule specifying that a DOM element of class “example” and its subtree are read-only. If a method “shake” exists, it may be read and invoked as a method.

```
1 {
2     "selector": "(//*[@class=âĂŸexampleâĂŹ]) | (//*[@class=âĂŸexampleâĂŹ]//*)",
3     "enabled": true,
4     "defaultFieldActions": {read: permit},
5     "fields": {shake: {methCall: permit}}
6 }
```

the wrapped object’s respective property. The getter function works in a similar way for reading properties. Using a property of an Object View as a function and calling it, triggers the proxy function. Object Views are applied recursively to a proxy function’s return value.

Creating two Object Views that wrap the same object, poses a problem with regard to reference equality. Although comparing the underlying objects of both object views would result in an equality, this would not be the case for the two wrapping Object Views. This inconsistent view can be prevented by only wrapping an object with an Object View once, and returning that same Object View every time a new Object View for the underlying object is requested.

Object views offer a basis for fine-grained access control through an aspect system. Each getter, setter and proxy function on an Object View can be combined with an “around” advice function, allowing the enforcement of an expressive policy.

Because of its size and complexity, manually wrapping the entire DOM with object views would be a difficult and error-prone process. Instead, the authors advocate a declarative policy system which is translated into advice for the Object Views.

The declarative policy is specified by a set of rules consisting of an XPath [270] selector to specify a set of DOM nodes and an Enabled flag to indicate that the selected nodes may be accessed. Optionally, each rule can be extended with default and specific rules for each field of a DOM element. An example rule, shown in Listing 4.20, specifies that all DOM elements of class “example” and its subtree can be accessed (Enabled = true) and is by default read-only (defaultFieldActions). A specific rule for a field called “shake” allows that field to be read and invoked as a method.

The authors discuss using Object Views in two scenarios: a scenario where JavaScript is rewritten[2] to make use of Object Views for same-origin usage, and a scenario where Object Views are used in cross-origin communication between frames.

In the latter scenario, each frame provides an Object View around its enclosed document to only expose the view required by the other. Communication between the frames is handled by marshaling requests for the other side to a string and transmitting it with “postMessage()”. Because each Object View has its own built-in policy, the communication channel does not need to enforce a separate policy.

The performance of Object Views was evaluated on a scenario where several objects are wrapped in a view, but where the communication between Object Views is not marshaled and transmitted with “postMessage()”. For this scenario, the average overhead is between 15% and 236% on micro-benchmarks.

JSand

Agten et al. propose JSand, a JavaScript sandboxing mechanism based on Secure ECMAScript (SES).

Secure ECMAScript (SES) is a subset of ECMAScript 5 strict which forms a true object-capability language, guaranteeing that references to objects can only be obtained if they were

[2]This work could also be listed under Section 4.2.1, but since the published paper mostly focuses on the cross-origin communication which does not require browser modifications, it is listed in this section instead.

explicitly passed to an object-capability environment.

Without a reference to the DOM, JavaScript code running in a SES environment cannot affect the outside world. JSand wraps the global object using the Proxy API [69] and passes a reference to this proxied global object to the SES environment. Any access to the global object from inside the SES environment, will traverse the proxy wrapper on which a policy can be enforced.

Without additional care, JavaScript inside the SES environment with access to this proxied global object, can invoke methods that return unwrapped JavaScript objects. Such an oversight can cause a reference to the real JavaScript to leak into the SES environment, making JSand ineffective. To avoid this, JSand wraps return values recursively, according to the Membrane Pattern [186]. In addition, JSand preserves pointer equality between wrappers around the same objects, by storing created wrappers in a cache and returning an existing wrapper if one already exists.

Using the Membrane pattern, any access to the outside world from inside the SES environment, can be intercepted and subjected to a policy enforcement mechanism. The authors do not specify a specific policy implementation, but point out that JSand's architecture allows for expressive fine-grained and stateful policies.

There are two important incompatibilities between the SES subset and ECMAScript 5 code, which makes legacy JavaScript incompatible with JSand.

The first is the mirroring of global variables with properties on the global object and vice versa. When a global variable is created under ECMAScript 5, a property with the same name is created on the global object. Similarly, a property created on the global object results in the creation of a global variable of the same name. This ECMAScript 5 behavior is not present in SES and can cause legacy scripts who depend on that behavior, to break.

Second, because SES is a subset of ECMAScript 5 strict, it does not support the "with" construct, does not bind "this" to the global object in a function call and does not create new variables during "eval()" invocations. Legacy scripts making use of this behavior will also break in SES.

To be backwards compatible with legacy JavaScript that does not conform to SES, JSand applies a client-side JavaScript rewriting step where needed before sandboxing the guest JavaScript code. The UglifyJS [184] JavaScript parser is used to parse JavaScript into an Abstract Syntax Tree (AST). This tree is then inspected and modified for legacy ECMAScript 5 constructs that will break in SES. In particular, JSand rewrites guest code so that the mirroring of global variables and properties of the global object in ECMAScript 5, is replicated explicitly. JSand also finds all occurrences to the "this" keyword and replaces it with an expression that replaces it with "window" if its value is undefined, thus also replicating ECMAScript 5 behavior.

JSand's performance evaluation indicates an average 9x slowdown for function-calls than traverse the membrane wrapper, resulting in an average of 31.2% overhead in user experience when interacting with a realistic web application. The load-time of a web application is increased on average by 365% for legacy web applications using ECMAScript 5 code which requires the rewriting step. The authors expect that this rewriting step will not be needed in the future, so that the average load-time overhead will drop to 203%.

TreeHouse

Ingram et al. propose TreeHouse, a JavaScript sandboxing mechanism built on WebWorkers. As explained previously, WebWorkers are parallel JavaScript execution environments without a usual DOM, which can only communicate through postMessage.

An overview of TreeHouse's architecture is shown in Figure 4.12. TreeHouse loads guest JavaScript code into a WebWorker to isolate it from the rest of a web page. WebWorkers do not have a regular DOM, so TreeHouse installs a broker with a virtual DOM inside the WebWorker that emulates the DOM of a real webpage. When this virtual DOM is accessed, the broker first consults the policy to determine whether access is allowed. If access is allowed, the broker

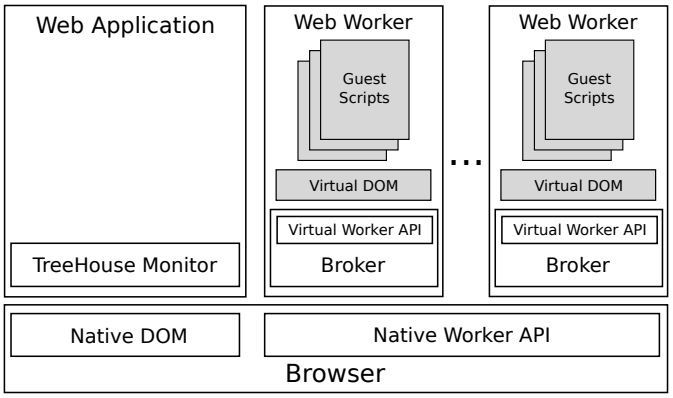


Figure 4.12: TreeHouse architectural overview. Sandboxes consist of WebWorkers with a virtual DOM. Access to this virtual DOM is mediated by broker according to a policy. If access is allowed, the request is forwarded to the real page's monitor, from [115].

Listing 4.21: TreeHouse integration in a web page. Guest code is loaded into <script> tags with type "text/x-treehouse-javascript" so that they are automatically sandboxed. The policy is also specified in a <script> element marked with a "data-treehouse-sandbox-policy" attribute, from [115].

```
1  <script src="tetris.js"
2    type="text/x-treehouse-javascript"
3    data-treehouse-sandbox-name="worker1"
4    data-treehouse-sandbox-children="#tetris">
5  </script>
6  <script src="tetris-policy.js"
7    type="text/x-treehouse-javascript"
8    data-treehouse-sandbox-name="worker1"
9    data-treehouse-sandbox-policy>
10 </script>
```

then forwards the access request to the real page's "TreeHouse Monitor" using "postMessage()," which handles the access to the real page's DOM.

TreeHouse offers two deployment options to web developers wishing to use its sandboxing mechanism. One option is to create a sandbox with a policy and load JavaScript in it manually using the TreeHouse API. Another option, is more user-friendly and allows a web developer to specify guest code to be sandboxed, in actual <script> elements. These <script> elements should have their "type" attribute set to "text/x-treehouse-javascript" to prevent them from being executed by the JavaScript engine in the host page. The special script type is also automatically detected by the TreeHouse Monitor, which will create sandboxes and load the script inside them.

An example use of TreeHouse is shown in Listing 4.21. Here, the first <script> element shows how a sandbox is created called "worker1," with access to the DOM element with id "#tetris" and its subtree. The "tetris.js" script is then loaded inside the sandbox and executed. The second <script> tag references the sandbox "worker1" and indicates through the "data-treehouse-sandbox-policy" attribute that the script "tetris-policy.js" should be interpreted as a policy instead of guest JavaScript code.

A TreeHouse policy consists of a mapping between DOM elements and rules. There are three types of rules: a rule can be expressed by a boolean, a function returning a boolean, or a regular expression. If the rule has a boolean value of True, access to the associated DOM element is allowed. If the rule is a function, that function is invoked at policy enforcement time

Listing 4.22: SafeScript used on a webpage. After loading the rewriter and API implementation, a namespace is created and the guest code is loaded. Afterwards, the guest code is transformed so that the property resolution mechanism is locked to the created namespace, and the transformed code is executed, from [161].

```
1  <!-- the transformation tool -->
2  <script src='rewriter.js'></script>
3  <!-- an APIâĂŹs implementation -->
4  <script src='interface0.js'></script>
5
6  <script>
7  var namespace0 = $_sm[0]();
8  var script0_code = load_script('http://3rd.com/main.js');
9  exec_script(transform(script0_code, namespace0));
10 </script>
```

by the broker, and access is allowed if the return value is True. Finally, if the rule is a regular expression, it refers to a property. If the regular expression matches a property's name, then the guest code is allowed to set a value to that property.

Because WebWorkers are concurrent by design, they present a problem when multiple TreeHouse sandboxes try to access to same DOM element in a real page. Such simultaneous access would cause a race condition and result in undefined behavior. To prevent such a race condition, TreeHouse allows a DOM element to only be accessed by one sandbox.

Another concurrency problem arises when the guest code makes use of a synchronous method such as "window.alert()". The guest code will expect the JavaScript execution to block, waiting for the end-user to click away the pop-up window. In reality, TreeHouse's communication channel between the host page and the WebWorkers is asynchronous because "postMessage()" is asynchronous. When calling "window.alert()" in the guest code, the broker would send an asynchronous message to the host page, and let code execution in the sandbox resume immediately. This conflicts with the guest code's expected behavior. The authors chose not to handle this case and raise a runtime exception when guest code calls synchronous methods.

The performance benchmarks for TreeHouse show an average slowdown of 15x to 176x for macro-benchmarks, and an average of 7x to 8000x slowdown on micro-benchmarks for method invocations on the DOM.

SafeScript

Ter Louw et al. propose SafeScript, a client-side JavaScript transformation technique to isolate JavaScript code in namespaces.

SafeScript makes use of Narcissus [191], a JavaScript meta-interpreter, to rewrite JavaScript code on the client-side and instrument the code so that it can interpose on the property resolution mechanism. Narcissus is a full JavaScript interpreter and can correctly handle all of JavaScript's strange semantics, its scoping, prototype chains and thus also the property resolution mechanism.

Through this rewriting step, SafeScript can separate JavaScript code in namespaces by manipulating the property resolution mechanism for each sandboxed script so that it ultimately resolves to its own isolated global object. Because property resolution is under SafeScript's control, it can effectively mediate access to the real DOM when sandboxed JavaScript guest code requests access to it.

Listing 4.22 shows how SafeScript can be used to sandbox a given JavaScript. In this example, the "rewriter.js" script contains SafeScript's transformation code and "interface0.js" contains an API implementation for a "namespace 0." After creating the namespace with "$_sm[0]()", the guest code is loaded from a third-party host, transformed so that the property resolution mechanism is locked to namespace 0, and then executed.

SafeScript ensures that any dynamically generated JavaScript code is also transformed and isolated in a namespace. In order to do so, SafeScript traps methods such as “eval()”, “setTimeout()”, which can inject JavaScript code into the execution environment directly. To capture JavaScript code that is indirectly injected, SafeScript monitors methods such as “document.write()” and properties like “innerHTML.” HTML written through these injection points must first be parsed and have its JavaScript code extracted before it can be transformed by SafeScript.

Despite its many optimizations, SafeScript’s performance benchmarks indicate an average slowdown of 6.43x on basic operations such a variable incrementation, because SafeScript rewrites every variable statement. The macro-benchmark reveals an average slowdown of 64x.

Discussion

This section discussed six JavaScript sandboxing mechanisms that do not require any browser modifications: Self-protecting JavaScript, AdJail, Object Views, JSand, TreeHouse and SafeScript. Some of their features are summarized in Table 4.3.

Besides Self-protecting JavaScript, which protects all access-routes to the DOM API through enumeration, all solutions isolate untrusted JavaScript in an isolation unit. The isolated JavaScript cannot access the DOM directly, but must communicate with the real web page and request access, which is then mediated by a policy enforcement mechanism.

JavaScript sandboxing systems that do not require browser modifications leverage existing standardized powerful functionality that is available in browsers today. The advantage of this approach is that standardized functionality is, or in the near future will be, available in all browsers and thus the sandbox works out of the box for all Internet users.

Much of the new browser functionality incorporated in the previously discussed JavaScript sandboxing systems, was not designed for sandboxing and may not perform well enough for a seamless user experience.

In the future that may change, because browser vendors optimize their code for speed to compete with other browser vendors. When new browser functionality becomes more popular, it will undoubtedly also be optimized for speed, automatically increasing the performance of the JavaScript sandboxing systems making use of it.

Web standards keep evolving, so that we can expect more advanced browser functionality in the future. This new functionality can then be used to design and implement yet more powerful JavaScript sandboxing systems. Ideally, this new functionality will also bring APIs dedicated to JavaScript sandboxing, providing purpose-built mechanisms to isolate code in a sandbox and communicate with that sandbox.

When such specialized JavaScript APIs are adopted and implemented, future JavaScript sandboxing mechanisms will no longer need to rely on repurposed functionality, making them simpler and faster.

Table 4.3: Comparison between prominent JavaScript sandboxing systems not requiring browser modifications.

System	Target application	Isolation unit	Communication	Policy expressiveness	Deployment	Performance	Known weaknesses
Self-protecting JavaScript	sandboxing	JavaScript environment	n/a	high	library	6.33x micro, 5.37% macro	[169]
AdJail	advertisements	shadow page	postMessage	read/write elements + enable/disable images/other	"policy" attribute	25% macro	
Object Views	sandboxing	iframe	postMessage	get/set/call	declarative policy with XPath	15% to 236% micro	
JSand	sandboxing	SES environment	Membrane/Proxy API	high	VDOM implementation	9x micro, 203% to 365% macro on loadtime, 31.2% macro on user experience	
TreeHouse	sandboxing	WebWorker	postMessage	high	script elements with custom type	7x to 8000x micro, 15x to 176x macro	
SafeScript	sandboxing	namespace	VDOM implementation	high?	VDOM implementation	6.43x micro, 64x macro	

4.3 Browser components for efficient Secure Sandboxing of JavaScript

The previous section showed that there is a large body of research into JavaScript sandboxing solutions, but that there is not a clear winner in terms of the perfect solution.

Browser modifications are powerful and can sandbox JavaScript efficiently, because of their prime access to the JavaScript execution environment. Unfortunately, the software modifications are difficult to distribute and maintain in the long run unless they are adopted by mainstream browser vendors.

JavaScript sandboxing mechanisms without browser modifications leverage existing browser functionality to isolate and restrict JavaScript. This approach can be slower but requires no redistribution and maintenance of browser code. In addition, it automatically works on all modern browsers.

From the review of relevant research, we can distinguish a number of components that are always present in a JavaScript sandboxing solution. By analyzing the advantages and disadvantages of these solutions, we can describe the ideal form of these components.

Isolation unit

Untrusted JavaScript should be isolated from the regular JavaScript environment so that a policy enforcement mechanism can identify on which running JavaScript code it should act.

The isolation units used in today's JavaScript sandboxing mechanisms are not adequate: they are either not lightweight, or not free of side-effects. For instance, AdJail uses an iframe as its isolation mechanism, but does not prevent JavaScript running inside from access the DOM. JSand uses a SES sandbox as its isolation mechanism, which can mediate access to the DOM, but at a performance penalty. TreeHouse uses WebWorkers, giving executing JavaScript access to a stripped-down version of the DOM, but still allows access to some DOM functionality. Other JavaScript sandboxing mechanisms have similar disadvantages.

Ideally, an isolation unit should be light-weight and provide a clean, side-effect free environment in which untrusted JavaScript can be run and from which it can not escape.

Virtual DOM and complete mediation mechanism

Once untrusted JavaScript code is isolated in an isolation unit, it may require access to browser functionality available in the JavaScript environment outside of its isolation unit.

This untrusted code has certain expectations about the environment in which it executes. For instance, it may assume to have access to XMLHttpRequest functionality through certain standardized properties and functions in the DOM.

The isolation unit must thus provide this environment, or Virtual DOM, so that any untrusted code running inside the isolation unit will not break due to expected but unavailable basic functionality.

The Virtual DOM must be able to provide the requested functionality (e.g. XMLHttpRequest) while at the same time mediating access to it through a mediation mechanism.

Virtual DOM implementations of today's JavaScript sandboxing mechanisms are not optimal. AdJail and TreeHouse for instance, set up a communications channel between the isolation unit and the hosting page, marshalling requests and responses between them through the postMessage() functionality. This marshalling of data inflicts a large performance hit. JSand and Object Views (in one of its forms) have no communications channel between the isolation unit and hosting page. Instead, the mediation mechanism is implemented using the membrane pattern through the Proxy API or equivalent. This approach is fast but requires wrapping the entire DOM, which can involve a lot of extra code.

Care must be taken to never leak any unmediated references to the real DOM into the isolation unit. Such an oversight would allow the untrusted code to break out of the isolation unit.

Ideally, browsers would provide functionality so that the Virtual DOM can automatically be wrapped using the membrane pattern in such a way that no unwrapped references to the DOM are passed to the untrusted JavaScript code.

Policy language

The mediation mechanism should be configurable through a policy, enumerating which DOM functionality that is available inside the isolation unit and describing any restrictions imposed on the usage of that functionality. For instance, a policy might dictate that access to document.location is permitted, but only to read properties and not alter them.

Such a policy language should be fine-grained and expressive enough without becoming a security problem of its own. WebJail describes an "inverse sandbox" scenario where a policy language becomes so expressive that an attacker may abuse it to steal information from other origins.

4.4 Roadmap towards Adoption

Isolation unit

Mozilla Firefox extensions have access to functionality that allows them to execute JavaScript code in a sandbox with a selected global object and origin. This functionality, through the `Components.utils.evalInSandbox()` method, would be ideal if it was also available in the JavaScript execution environment of a web page. Unfortunately, this functionality is only available for browser extensions. On its own, `evalInSandbox()` would also not allow access mediation to a virtual DOM. Wrapping the selected global object using the Proxy API and membrane pattern could achieve this, but it is unclear whether `evalInSandbox()` works in conjunction with a Proxy object.

Web Workers allow JavaScript code running in a web page to spawn a background thread in which JavaScript can be executed concurrently. This mechanism allows a web page to offload computationally expensive JavaScript into a separate thread, preventing the main thread from being blocked or slowed down. JavaScript code running inside a Web Worker has access to only a subset of the functions and classes available in a web page's DOM. In particular, Web Workers have no access to the DOM itself. In addition, Web Workers communicate with the main web page through postMessage, which hinders both performance and synchronous operations. An adaptation of Web Workers (e.g. a SandboxWorker) could be ideal for JavaScript sandboxing if it were possible to replace the global object inside the Web Worker with a custom object. Again, a Proxy object would allow a JavaScript sandboxing mechanism to mediate access to the DOM, if this Proxy object could replace the global object in a SandboxWorker. Because Web Workers execute in parallel with the main thread, a SandboxWorker will need to address synchronisation issues when accessing the real DOM.

Support for Virtual DOM creation

The virtual DOM needs to reflect the methods and properties available in the real DOM as accurately as possible, because running JavaScript code has certain expectations with regard to the appearance of a "standard DOM". Using the Proxy API to wrap the real DOM, presents executing JavaScript code with an apparently real DOM while at the same time allowing full mediation.

To prevent that references to the real DOM leak from a Proxy object, the membrane pattern should be used. The implementation of this membrane pattern needs code to mimic the structure of the DOM, the DOM classes and their properties and methods. The construction of a virtual DOM thus involves a full description of the real DOM, which is not available through any JavaScript API. Such a description has to be imported from elsewhere, as JavaScript code, JSON or even IDL listings.

Since the browser itself has the information about the structure of its own DOM, it should not be necessary to import it from elsewhere. Sharing this information from the browser in a structured way through a JavaScript API, could ease the construction of a virtual DOM.

Chapter 5

Cross-Site Scripting (XSS)

5.1 Cross-Site Scripting Vulnerabities in the Modern Web

Back when the Web was invented, it was intended to serve as a distribution mechanism for static, scientific documents written in HTML. With the advent of server-side scripting languages such as PHP and the introduction of client-side scripting technologies such as JavaScript, Flash or Silverlight, the Web became a fully-fledged runtime environment for sophisticated applications. In fact, operating systems such as Chrome OS or Firefox OS merely provide an execution environment for a browser such that a user can interact with web applications. As opposed to static documents, such applications frequently generate content on-the-fly based on inputs provided by the user, the user's browser, the underlying database, or other (potentially untrusted) sources.

In some cases, this input is processed by the web application in such a way that a maliciously crafted input may divert the web application's intended control flow [153, 253, 292]. Besides classical injection problems such as SQL injection [96] or Command injection [253], a considerable portion of the work focusses on so called *Cross-Site Scripting* (XSS) attacks. In the basic version of such attacks, malicious HTML tags are embedded by an attacker into Web requests issued by a client. This type of vulnerability was encountered in the late nineties by a group of Microsoft security engineers, who also coined the term [232]. The work of this group led to an advisory published by CERT in February 2000 [40], which is, to the best of our knowledge, the first known written documentation of XSS. The description given in this advisory "might nowadays be classified as the reflected/non-persistent form of the attack" [232].

5.1.1 The Ongoing Evolution of Cross-site Scripting

Within the last decade, a huge amount of research has been conducted to investigate the nature of such injection attacks. A multitude of different flavors of this problem emerged and our understanding of the vulnerability has fundamentally changed. This evolution can be observed best when reviewing existing research papers. The early approaches only consider reflected or persistent XSS (e.g., [29, 142, 197]). In 2005, XSS gained special attention when the infamous Samy worm [137] hit the social network MySpace and Klein published his paper on "DOM Based Cross Site Scripting or XSS of the Third Kind" [143]. Since then, a lot of different flavors of the problem emerged and, hence, the understanding of what XSS really is has changed significantly. Recent works include many other dimensions such as DOM-based XSS [143, 153], Flash-based XSS [4], mutation-based XSS [107] or CSS-based XSS [106].

5.1.2 XSS Today

Until today, XSS is a serious and widespread security issue. Only recently, an XSS vulnerability caused a major security breach of the Ubuntu Forum: two million user credentials were stolen,

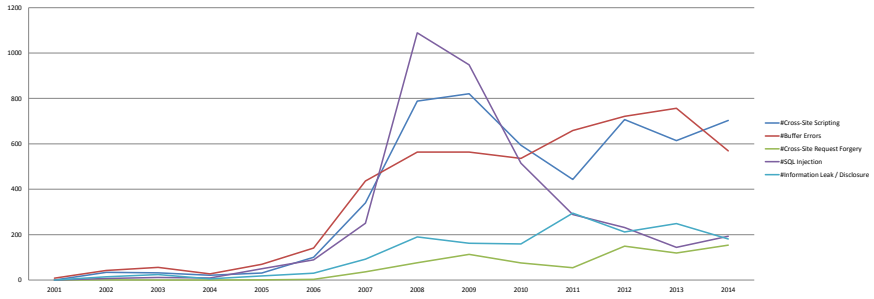


Figure 5.1: Count of entries in the CVE database assigned to the XSS category [188]. Data for 2014 was compiled on October 15th 2014.

when an administrative account was hijacked via a persistent XSS attack [284].

The Open Web Application Security Project (OWASP) regularly lists XSS as one of the top three security vulnerability problems on the Web [210]. In its yearly analysis report [242], the company Whitehat Security lists XSS vulnerabilities as the most wide-spread vulnerability: according to this study, 43% of all discovered serious vulnerabilities can be accounted to XSS. An examination of MITRE's Common Vulnerabilities & Exposures (CVE) database [188] also shows, that XSS has constantly been the second most reported vulnerability class since 2008 and even takes the top spot for 2014 (see Fig. 5.1). In this statistic it is especially notable that no clear trend is visible, which suggests that the overall problem of XSS is in decline: Both SQL Injection and XSS expose a peak in the timeframe between 2007 and 2009, coinciding with the field of Web applications moving into the focus of the security community. But, while the frequency of SQL Injection reports has visibly declined over the following years, no such trend can be observed for the number of reports of XSS.

5.1.3 Overview

In this chapter, we consolidate the existing knowledge on Cross-Site Scripting, especially since the notion of this type of vulnerability obviously changed a lot over time. Our objective is to systematically review the existing literature related to XSS and present a comprehensive overview of this research field. This also enables us to identify open problems and potential research topics that still need to be addressed. We achieve these goals by first elaborating on the different dimensions of the problem space. More specifically, we discuss the different causes behind such attacks and examine the different options of approaching the problem. We then provide a classification scheme for past, current, and future research in this area. On the one hand, this classification is based on the different angles from which the problem can be addressed. On the other hand, our classification is divided into the different places where the approach is deployed/implemented. By applying this classification scheme to the current research landscape consisting of more than 80 papers that we reviewed, we identify the underlying trends and concepts applied to conduct, detect, mitigate, and prevent XSS vulnerabilities and attacks. Furthermore, we identify several problems that are not well-understood so far and where additional research is needed.

To summarize, in this chapter we provide the following information:

- We introduce a systematic classification scheme for research on XSS by analyzing the different dimensions and facets of the problem space.
- We provide a comprehensive overview of the current research landscape by applying our classification scheme to this research area. Thereby, we analyze the different approaches and study the underlying concepts.

- We discuss the advantages and drawbacks of main research streams and point out several open research problems for future work in this area.

5.2 Dimensions of Cross-Site Scripting

According to OWASP, "Cross-Site Scripting attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites" [210]. While being correct for a large portion of XSS attacks, this definition does not fully address the complete problem space. Thus, we propose a precise definition of XSS by analyzing the different flavors and dimensions of the problem space.

In this deliverable, we consider Cross-Site Scripting to be *any* injection vulnerability that allows the injection of arbitrary client-side code or markup into a web application. This includes JavaScript injection, HTML injection, CSS injection and the injection of other past or upcoming client-side scripting or markup languages (e.g., VBScript, SVG, etc.). As such, we define XSS attacks and vulnerabilities as follows:

Definition 1 (Cross-site Scripting Attack) *A Cross-Site Scripting Attack is an attack in which adversary-controlled data is injected into a web application in an unauthorized fashion. Thereby, the injected content is crafted in such a way that a users's browser interprets it as client-side code or markup without the user's consent.*

Definition 2 (Cross-site Scripting Vulnerability) *A Cross-Site Scripting Vulnerability is a programming error within a web application or the corresponding execution environment which unintentionally allows an attacker to conduct a Cross-Site Scripting Attack.*

An XSS attack is caused by *adversary-controlled data* since the adversary can either directly inject data into a given web application or she can trick the victim into performing the attack steps himself (so called *self-XSS attack*). It is *unauthorized* since the attacker injects data into the application without a user's consent. The attack manifests when the user's browser *interprets* the injected data as client-side code or markup, again without the user's consent. An XSS vulnerability is a software fault that can be activated via an external trigger as part of an XSS attack. Note that the vulnerability can be either located in the web application itself or within the execution environment (e.g., *universal XSS* attacks where an adversary exploits a browser bug to inject data). Again, we emphasize that XSS vulnerabilities are *unintentional* to exclude legitimate use cases where code is injected (e.g., as part of browser plugins such as *Greasemonkey* or developer sites such as `jsfiddle.net`).

To illustrate our definition further, the following examples provide an overview of different facets of XSS attacks:

- **HTML Injection:** As described in the initial advisory, XSS is an attack that allows the injection of malicious HTML tags [40]. Hence, XSS is not limited to the injection of scripts, but to the injection of anything that can be expressed with HTML. This of course includes JavaScript enclosed in script tags or event handlers, but additionally also includes any kind of plugin content embedded via object or embed tags (e.g., Flash, PDFs, or Java) and all the other tags that can be used for attacks such as defacement [86, 211] or phishing [92, 147].
- **JavaScript Injection:** One way to inject JavaScript into a web application is to use HTML script tags or event handlers as described above. However, sometimes it is also possible to directly inject JavaScript without the need for HTML, for example when JavaScript code is directly generated by the application from user input. This generation can take place anywhere within the application, for example at the client-side when strings are converted to code via the `eval()` function or at the server-side when scripts are dynamically generated.

- **CSS Injection:** Besides the well-known and understood ways of injecting malicious code via HTML or JavaScript, it is also possible to inject malicious Cascading Style Sheets (CSS) into an application. As demonstrated by Heyes et al. [108] and Heiderich et al. [106], it is possible to inject stylesheets that are capable of leaking sensitive information without the need for JavaScript.

The underlying Cross-site Scripting vulnerability for an injection of such (client-side) content can be caused by different entities. Depending on the specific entity, the exploitation, detection, mitigation, or prevention techniques must be adapted to account for the specific environment. To supplement our definition, we thus classify XSS vulnerabilities according to the causes as explained in the following.

5.2.1 Caused by Server-side Code

The most well-known type of XSS is the server-side variant. Thereby, the injection problem is caused by the server-side code used to generate dynamic HTML or script content. Generally, there are two types of server-side XSS problems:

Non-persistent/Reflected

In a reflected XSS vulnerability, the server "reflects" user input contained in a request into the response handed back to the browser (see Listing 5.1 for an example). Hence, a malicious payload executes in the very same browser that also generated the request. In order to exploit such a vulnerability, an attacker, therefore, needs to lure a victim into generating the malicious request in his browser. This can either be done by luring the victim into opening a malicious URL through phishing or social engineering, or by luring the victim onto an attacker controlled web page that is capable of generating the malicious request (e.g., by embedding an iframe or submitting a hidden form).

Listing 5.1: Exemplary reflected XSS vulnerability

```
1 [...]
2 <h1>Hello <?php echo $_GET["name"] ?></h1>
3 [...]
```

Persistent

In a persistent XSS vulnerability, the application also outputs potentially malicious user input in an uncontrolled fashion. Opposed to reflected XSS, the user input is not directly reflected back to the user, but stored on the server (e.g., in a database or file). Depending on the application's behavior, this data is then outputted to other users of the application. Hence, the attacker model in this kind of an attack is a little bit different: instead of luring a user into clicking a malicious link, this time the adversary simply injects the payload directly into the application. Afterwards, she only needs to wait until the payload is delivered to another user of the same application. Hence, it is easier for the adversary to conduct the attack as social engineering or phishing is not necessary. Due to the persistent nature of the attack, it can be used to construct XSS worms [46, 47, 54, 137, 139, 255], whereas the server-side backend of the web application serves as a distribution channel for the worm.

5.2.2 Caused by Client-side Code

Until 2005, Cross-Site Scripting was believed to be a server-side vulnerability. Back then, Klein published a paper on a variant that he called DOM-based XSS or "XSS of the third kind" (while server-side reflected and persistent XSS were considered to be the other two variants) [143]. As opposed to server-side XSS, DOM-based XSS is caused by client-side code, i.e., by legitimate

JavaScript executed in the browser (see Listing 5.3 for an example). Although there are still newly published research papers that consider XSS to be a server-side problem only (see for example [213]), the research community has adapted its definition of XSS in the meantime. Lekies et al. demonstrated that client-side XSS is a prevalent problem nowadays [153].

Listing 5.2: Exemplary persistent, client-side XSS vulnerability

```
1 var html = "<a href='"
2   + location.href
3   + "'>Home</a>";
4 localStorage.setItem("cached", html);
5
6 [...]
7
8 var html = localStorage.getItem("cached");
9 document.write(html);
```

Listing 5.3: Exemplary non-persistent, client-side XSS

```
1 document.write("<script
2   src='http://example.org/test.php?r="
3   + document.referrer + "'></script >");
```

As opposed to Klein's paper, we consider two variants of client-side XSS: non-persistent and persistent:

Non-persistent/Reflected

Client-side non-persistent XSS vulnerabilities are similar to their server-side counterparts, as can be seen when comparing Listing 5.1 with Listing 5.3. Nevertheless, there are some very important differences:

- *Attack Detection:* While server-side XSS attacks are caused by malicious input to the server (i.e., everything that is contained in the HTTP request), client-side XSS is caused by malicious inputs to the client-side (i.e., any attacker controllable data in a web application's DOM). Hence, traditional intrusion detection systems that are deployed at the server-side will not necessarily be able to detect client-side XSS attacks.
- *Vulnerability Detection:* Another difference is the possibility to detect vulnerabilities. While for the server-side there are many sophisticated tools, there are almost no tools to detect client-side XSS vulnerabilities. Furthermore, many web applications make use of remote script includes [201]. Hence, the amount of source code that is executed within such a client-side application is significantly different from the amount of code that is hosted by the originating server. Scanning and patching vulnerabilities is thus much harder.
- *Scope:* As server-side XSS is caused by dynamic server-side scripts, server-side vulnerabilities are not present in static HTML pages. Client-side XSS vulnerabilities, however, are caused by dynamic client-side scripts. Hence, static HTML pages are potentially vulnerable to client-side XSS as soon as JavaScript code is embedded. Furthermore, client-side XSS is not necessarily caused by HTML and JavaScript, but can also be caused by plugin-based content such as Flash [4].
- *Client-side Environment:* While the server-side is under full control of the web site provider, the client-side is completely controlled by the user. Hence, no assumptions can be made on the client-side infrastructure. Many client-side XSS vulnerabilities are caused by browser quirks (e.g., by the CSS expression statement in Internet Explorer). Furthermore, the browser is currently a moving target. With the introduction of HTML5 related technologies a lot of new ways to conduct XSS attacks are being revealed nowadays. Hence, it is difficult to deploy and maintain filters and countermeasures.

Persistent

A variant of XSS that has not yet receive a low of attention is client-side persistent XSS (see Listing 5.2 for an example). Until recently, browsers did not offer a lot of client-side storage functionalities except cookies. However, with the introduction of HTML5 and related technologies, several of such client-side storage technologies were introduced. Examples are Web Storage, the File API, indexed Databases, and the Offline AppCache. Several papers demonstrated the real-world existence of such vulnerabilities [101, 153, 152].

5.2.3 Caused by the Infrastructure

While server-side and client-side XSS vulnerabilities are caused by faulty application-level code, infrastructure-based XSS vulnerabilities are caused by bugs or malicious code in the underlying infrastructure. The infrastructure is comprised of the server, the client, and the network.

Server

A common server infrastructure XSS problem can occur with server-wide error pages or similar vulnerable default behavior [221]. Besides erroneous configuration, server induced XSS can also be caused by the server directly. For example, in 2006 most web applications served by the Apache HTTP server were susceptible to XSS due to the server's wrong handling of the `Expect`-header [294].

Client

Client-side infrastructure XSS vulnerabilities are mainly caused by the browser or plug-ins and extensions executed by the browser. One popular example for this category is Universal XSS (UXSS). UXSS is a bug class that causes an injection vulnerability in arbitrary web applications. Thereby, the application itself does not need to expose a vulnerable code fragment. The attacker simply misuses a browser bug to inject the malicious payload. Instances of this particular problem occurred for example in Opera [138], Adobe Reader [245], or in Internet Explorer 8 [155].

Network

Network-based XSS attacks occur whenever entities on the network are able to manipulate HTTP responses. For example, Gilad et al. demonstrated how IP spoofing can be used to conduct a so-called *Off-Path Attack* that is capable of injecting malicious scripts into an XSS-free web application [89]. Another example for a network-based XSS was conducted by an American hotel chain: visitors that used the hotel's Wi-Fi noticed that strange advertisements were shown on web pages that normally did not include banner advertisements. The hotel intercepted the HTTP responses and injected the advertisement code into the web page [28].

5.3 The Facets of XSS

In the previous section, we defined the terms *XSS attack* and *XSS vulnerability* and reviewed their different dimensions and causes. Based on these insights, we now examine the different viewpoints from which XSS can be approached and study corresponding approaches to tackle the problem.

5.3.1 XSS as an Information Flow Problem

A necessary precondition of an XSS vulnerability is that an attacker can inject data (e.g., HTML tags, JavaScript code, or CSS markup) into a given web application and this data is later on executed. Hence, we can treat XSS as an information flow problem: in a first step, we need

to identify sources that an attacker can potentially control. Afterwards, we analyze how data is processed within the application and if attacker-controlled data eventually reaches a security sensitive sink. This kind of information flow analysis is a well-understood field and many different techniques exist to implement information flow analysis (e.g., [61, 233, 241]).

However, unlike monolithic systems, web applications are a composite of various detached components, typically consisting of at least the web browser, the application server, and a server-side persistence system or database. And again, each of these components consists of several subsystems. For example, the web browser is a set of parsers for complex languages such as JavaScript, HTML, XML, CSS, SVG, ActionScript, etc. In practice, many information flows span more than one of these components and subcomponents, including information flows spanning both server-side and client-side code [107] or data that is temporarily stored in persistence, either on the server [141] or client-side [153]. As a consequence, covering such complex flows is a major challenge for protective systems.

5.3.2 XSS as a Sanitization Problem

From a different standpoint, XSS can be seen as a sanitization problem: if we manage to check both the input and output of a web application for potential XSS attacks, we can filter malicious content and prevent it from causing harm. Sanitization is a general concept used in many security contexts and can be applied to XSS vulnerabilities as well. Typical challenges for such an approach are for example the placement of sanitizers, the problem of deciding if a particular piece of data is actually malicious or benign, and the various encodings used in modern web applications.

One of the main difficulties in sanitizers placement and correctness results from the many different contexts in which injections can occur. Depending on the surroundings of the injection point, sanitizers have to filter different commands and control characters. For example, if user input is written to the document via the PHP function `echo()` between an opening and a closing `div` tag, the input has to be filtered for HTML control characters. If user input is used within a call to JavaScript's `eval` function, the input has to be filtered for JavaScript control characters. And although it is not easy for a security-unaware developer to determine this context, it gets even more difficult when multiple contexts are combined. Then, sanitizers have to be applied in exactly the right order. Listing 5.4 provides an example in which a JavaScript context is nested within a URL context, which again is nested in an HTML attribute context.

Listing 5.4: Example for nested contexts

```
1 <?php
2   $url  = "'javascript:alert("
3   $url .= $_GET['name']."')";
4   echo "<a href=".$url.">";
5 ?>
```

5.3.3 XSS as a Mitigation Problem

On a conceptional level, XSS is related to well-explored code injection vulnerability classes, such as memory corruption vulnerabilities in binary applications [256]: an attacker can inject input to a given application that leads to an unauthorized change in the control flow. For memory corruption attacks, several (more or less successful) approaches for exploit mitigation have been proposed, such as StackGuard [48], *Instruction Set Randomization* [140] / *Adress Space Layout Randomization* (ASLR), *Control-Flow Integrity* (CFI) [3], or *Write Integrity Testing* (WIT) [10]. Interestingly, the underlying concepts of some of these approaches were also applied in the context of XSS protection, for example in systems such as *Document Structure Integrity* [195] or Noncespaces [95]. We will more closely review such mitigation options later on.

The complexity of the XSS problem suggests that in the near and midterm future, XSS will still be a major concern: despite a decade of research on this topic, novel attacks that bypass all existing mitigation and prevention approaches are published frequently. Hence, exploring applicable mitigation approaches from other areas and studying if/how they can be applied to the XSS problem is a worthwhile effort. The challenging part is to identify which methods can be transfered: XSS is a hard problem due to aspects like the involvement of several parties (e.g., a web application and several browsers that all have their own quirks), a lot of encoding/transformation that happens on the different levels, or the fact that JavaScript is hard to analyze. Thus, we need to carefully analyze which mitigation strategies can be applied.

5.4 Research Classification

In the last years, a large body of work on different aspects of the Cross-Site Scripting problem has been published. The individual papers focus on specific dimensions of the problem and it is hard to keep track of the overall picture. Especially the identification of "blind spots" (i.e., areas where still open problems exist and research work is needed) has become a challenge. In the following, we thus introduce a general classification of the different research areas that we will use in the rest of this chapter to provide a comprehensive overview of the current research landscape. Furthermore, this systematic overview also enables us to identify future opportunities for research.

Our classification is divided into two orthogonal axes that highlight the distinct facets of the problem. On the first axis, we organize the existing literature according to the different technical purposes to approach the problem. The individual classes of this axis are (*i*) attack, (*ii*) detection, (*iii*) mitigation, and (*iv*) prevention. Note that it is sometimes hard to assign a method to a single class, especially since the classes are slightly interconnected. However, we attempt to provide a classification according to the primary goal of a given method to systematically review the existing works. Orthogonal to this classification is the point where the presented approaches are deployed. We also subdivide this axis into several classes. A method can be deployed either at the (*i*) server-side, (*ii*) client-side, or it is a (*iii*) hybrid approach that requires a (close) interaction between both sides. If we only consider the defense approaches combined with the deployment aspects, this leaves us with a 3×3 matrix on which we can organize the existing literature. In the following, we explain the individual aspects of our classification in greater detail and provide a justification for our breakdown of the field.

5.4.1 Purpose of Technical Measure

In a first step, we categorize the different purposes of the technical measures to tackle the XSS problem in four classes according to the angle from which the problem is addressed, namely: *Exploitation*, *Detection*, *Mitigation*, and *Prevention*.

A given paper is classified according to its primary focus. The following list gives a short overview of the four classes while we provide detailed characterizations in Sections 5.4.1 to 5.4.1:

Exploitation: The primary focus of the paper is on exploring specific facets connected to the conduction of XSS attacks.

Detection: The paper's technical contribution targets to detect XSS vulnerabilities or XSS attacks, without additional steps toward prevention or mitigation.

Mitigation: The paper's technical measure does not address the underlying XSS vulnerability (see Def. 2). Instead, the effects of an XSS attack (see Def. 1) are thwarted.

Prevention: The paper proposes a measure to directly prevent the causing XSS vulnerability.

While there are papers that appear to fit into more than one category (e.g., detection and prevention), the definition of categories was designed to be is sharp enough to clearly classify current research. Our categorization is not based on the potential of a technique to fulfill a certain task, but on the actual features described and evaluated in the papers.

For instance, some of the detection papers' techniques could, in theory, be extended towards prevention or mitigation. However, in the paper itself the authors purposefully focused on detection only, likely due to technical circumstances (e.g., performance overhead, false positives, compile- time vs. runtime detection, etc.). Thus, from the paper alone, the merits of a given detection approach to be used for other purposes cannot be concluded.

In a similar fashion, some approaches that focus on prevention or mitigation could potentially be able to include reporting functionality for detection purposes. But, if this option has not been evaluated by the paper's authors, we cannot assess the techniques actual detection capabilities and the potential quality of such usage of the approach. Thus, in our scheme the approach's primary functionality determines its position in the classification. In the remainder of this section, we explore the four classes in more detail:

Exploitation

The goal of many papers is to explore a method to exploit a particular XSS vulnerability, to bypass an existing defense strategy, or to execute code in the context of an XSS vulnerability. Such works contribute novel attack techniques and demonstrate shortcomings of existing defenses. As such, they are valuable since they enable us to study the attack surface and offer new insights into potential weaknesses of existing prevention or mitigation approaches.

Detection

The second class of approaches to tackle the problem includes different ways to detect XSS vulnerabilities or XSS attacks. This includes approaches that, in their presented form, are only detecting attacks or vulnerabilities, but not to mitigate or prevent them (e.g., due to shortcomings in areas such as false positive rates, performance, or completeness). We can subdivide this class in two aspects. On the one hand, we can proactively detect XSS vulnerabilities by analyzing a given application for potential vulnerabilities. Such an analysis can be either carried out statically (i.e., by analyzing the code without actually executing it) or dynamically (i.e., running an application on a particular set of inputs and observing the behavior during runtime). Both approaches have their individual strengths and weaknesses that enable them to detect particular kinds of vulnerabilities. On the other hand, we also need reactive ways to detect successful XSS attacks. This is mainly due to the complexity of the problem: despite many years of effort, no comprehensive and sound way to prevent all kinds of XSS attacks has been presented so far. As a result, we need means to detect successful attacks such that we can react on them and mitigate the effects.

Mitigation

This class represents approaches in which vulnerabilities still exist, but cannot be exploited due to a modified runtime environment. For example, CSP is such a technique: although CSP prevents vulnerabilities from being exploited by whitelisting legitimate code, the vulnerability itself still exists. Another example would be SessionSafe [126] or SessionShield [205] that do not stop JavaScript injection, but protect the cookie from being accessed via JavaScript.

There are different ways to mitigate a given XSS attack and we identified three unique approaches to do so. First, the injection is successful, but the mitigation approach prevents the browser from executing the script (e.g., via a randomization of the XML namespace). Second, the injection of arbitrary code or markup can be successful, but the approach mitigates the actual effect of this injection (e.g., by identifying and blocking illegitimate scripts). Third, the injection is successful and is also executed, but the unauthorized code is contained in a specific

environment such that it cannot cause harm (e.g., via sandboxing or runtime restriction of potentially malicious scripts). The different approaches have their own unique characteristics as we will discuss later on.

Prevention

Inspired by the *security by design* principle, a class of existing work focuses on ways to proactively prevent XSS vulnerabilities in the first place. Such papers introduce methods that either approach potential root causes of the problem or utilize approaches to prevent the underlying problem in a generic way. These methods tackle the underlying problems and thus enable us to anticipate potential attacks. For example, a clear separation between code and data prohibits a large class of XSS vulnerabilities since an attacker cannot smuggle data into the application that is later on interpreted as code, thus addressing a root cause. Other examples are the use of whitelisting or dynamic taint analysis to study the information flow within a given application with the goal of preventing unauthorized control or data flow.

5.4.2 Point of Deployment

As a second step, we discuss how the existing work can be categorized according to the place where the approach is deployed/implemented. Historically, XSS vulnerabilities were regarded as server-side problems and only later on researchers realized that also the client-side can be influenced. Accordingly, our broad classification on this axis is divided into server- and client-side approaches depending on where the proposed method is deployed and there are also hybrid approaches that require an interaction between both sides.

Server

As mentioned above, XSS can be seen as a server-side problem and thus a certain fraction of the existing literature proposed mechanisms to address the problem on the server side. First, such an approach can aim at the (web) application layer and implement mechanisms to attack, detect, mitigate, or prevent XSS vulnerabilities at this level. Second, the method can be applied one level deeper, namely at the runtime layer. This can be either the framework on top of which a web application is developed or the underlying infrastructure such as the webserver or the interpreter that executes the application (e.g., a PHP interpreter). Third and a bit complementary to the previous two aspects, a server-side approach can utilize a preprocessing step in which the input (or later on output) of an application is sanitized.

Note that this classification can be extended by the method of deployment. On the one hand, a given application can be transformed or instrumented in an automated way such that a solution can be deployed without (much) human interaction (e.g., by instrumenting the interpreter or rewriting the source code of a given application). On the other hand, manual work or code changes could be necessary, for example if the source code of an application needs to be adopted to include the solution.

Client

The situation at the client side is very similar compared to the server side. Again, the problem can first be tackled at the application layer, for example by instrumenting the code that is executed on the client side while a user interacts with the application. Second, a solution can be deployed one level deeper, in this case at the browser layer. This includes all works in which a browser is instrumented to mitigate or prevent attacks. Third and again a bit complementary, a preprocessing can be carried out to sanitize the input received by the browser and/or application. In general, these levels correspond to their counterpart on the server side.

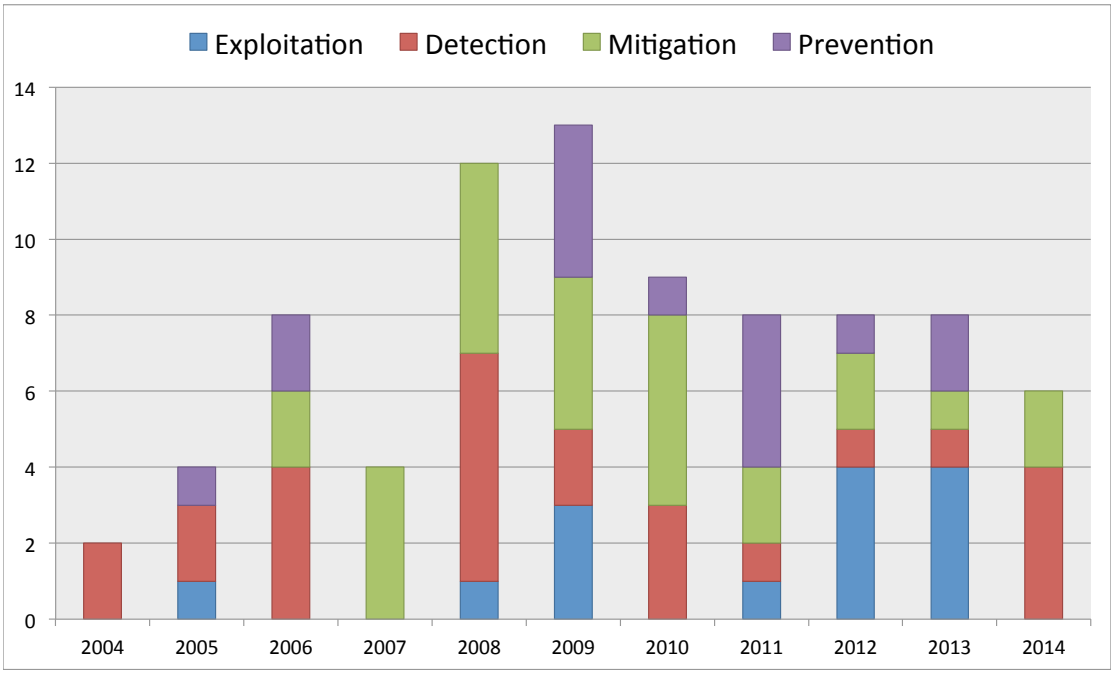


Figure 5.2: The XSS research landscape (papers per year)

Hybrid

The separation between server- and client-side approaches for deployment is not always sharp in the sense that there are also solutions that rely on both ends of the communication. As such, there are a few tools that do not fall into our two broader classes and hence there is a class for hybrid solutions. Note that such approaches are typically costly since they require changes on both ends.

5.5 Current Research Landscape

In this section, we present a comprehensive overview on conducted research that is directly related to XSS. Following the structure of the proposed research classification, we group the surveyed works into the classes *Attack*, *Detection*, *Prevention*, and *Mitigation*. In addition, related survey papers and research on XSS worms is presented in *Others*.

The data basis of this categorization consists of 80 technical papers, accompanied by relevant survey papers and several notable non-academic publications. These papers were collected through a systematic survey of the proceedings of the main academic security conferences and selected security workshops in the timespan between 2004 and 2014. Figure 5.2 shows the categorized technical publications by year and category.

5.5.1 Exploitation

First, we provide a brief overview of publications with main or substantial focus on exploiting XSS vulnerabilities. As a substantial portion of such offensive research was conducted in the non-academic security community, we also discuss selected approaches that have not been published separately at academic venues.

New Attack Methods

A comprehensive amount of work flew into the identification of novel XSS attack techniques.

One important paper in this category was authored by Klein [143]. In this paper, he extends the notion of cross-site scripting from a server-side-only vulnerability to a server- and client-side problem. He was the first who systematically demonstrated that XSS cannot only be caused by server-side code, but also by client-side code, i.e., the legitimate JavaScript running in the user's browser. Due to its dynamic nature, client-side XSS imposes very different requirements with regards to Detection, Prevention and Mitigation capabilities. Hence, it is important to distinguish these different kinds.

Another interesting attack technique was presented by Bojinov et al. [31]. The technique called Cross-Channel Scripting (XCS) injects malicious script code into an application via non-HTTP channels. XCS are applicable to all Web-based applications that output user-controllable data received from such a non-HTTP channel in an uncontrolled fashion. If an attacker is able to gain control over such a channel he can inject data to it and subsequently wait for another user to visit the vulnerable web interface.

Very similar to the XCS attack, Zhang et al. present a technique that allows the injection of malicious scripts via social networking APIs [296]. Often developers do not suspect APIs of popular social networks to return malicious data, however those APIs only forward user generated data to the caller of such a functionality. Zhang et al. utilize this fact to inject XSS payloads into Web-based applications calling these APIs. To do so they create content containing JavaScript or HTML code within the social network. Even if the social network itself might not be vulnerable to injection attacks, the malicious content might by returned to a vulnerable web application by a corresponding API method.

Filter Evasion

Besides identifying novel attacks, another track of research looked into the circumvention of existing countermeasures and, thus, into reenabling known attacks. Hereby, a special focus was put onto the circumvention of input validation and filtering techniques.

Heiderich et al. discuss the dangers of novel functionalities for XSS filters with the example of the SVG format [105]. Although SVG was primarily constructed to describe scaleable vector graphics, the XML-based format also introduces novel, sometimes unexpected ways to execute JavaScript code. Hence, blacklisting-based filters that are not aware of these new functionalities fail to correctly sanitize/filter SVG-based formats. Another work by Heiderich et al. presents a technique to circumvent widely deployed XSS filters by misusing browser-based parser quirks and mutations [107]. The idea behind the attack is that many XSS filters do not take browser-specific parsing optimizations into account. By misusing the corresponding mutations within user-provided data, a seemingly legitimate value can be transformed into a malicious XSS payload. As most of the sanitizer and filter functions are executed before the browser-based mutation, malicious content is undetectable for the filter functions.

Furthermore, Barth et al. [20] and Magazinius et al. [171] utilize lax parsing behavior to create documents that are accepted by both HTML and PDF parsers: Using the results of a systematic analysis of the content sniffing mechanisms of web browsers, Barth et al. were able to create PDF documents that also contain a sufficient amount of HTML mark-up to trick the content-sniffing algorithms of the examined browsers to render the PDF as HTML, hence, executing contained JavaScript in uploaded PDFs. Taking the opposite direction, Magazinius et al. showed that fairly harmless content injection in HTML documents could be used to create HTML documents that are also parsed by a PDF viewer, including the potentially contained JavaScript. As the respective input validation procedures are specialized for only one of the two document formats, the authors demonstrated how to use the observed behavior for filter circumvention. An earlier related attack, combining the syntactic characteristics of the GIF and the JAR formats, has been presented by McFeters et al. [176].

Information Exfiltration

As we will discuss in Sections 5.5.3 and 5.8.3, a subset of browser-based XSS mitigation techniques, including CSP [271] and sandboxed iFrames [109], are on the verge of becoming sufficiently wide spread to have substantial impact. Therefore, recently first techniques have been proposed that achieve the exfiltration of sensitive information using XSS attacks that evade existing mitigations [106, 293, 43]. We will explore this emerging trend in Section 5.8.3.

Generating Attack Payloads

One particular stream of work addresses the generation of XSS attack payloads for vulnerability validation. As a validation mechanism alone does not yield major benefits, these techniques are often used in conjunction with additional detection capabilities. Hence, we will discuss the generation of attack payloads in the next section.

5.5.2 Detection

Besides investigating novel attack techniques, a considerable amount of research was conducted to identify techniques capable of detecting either XSS vulnerabilities or XSS attacks.

Detection of Vulnerabilities

Approaches to detect vulnerabilities have the advantage that implementation faults can be detected in a proactive way. Different mechanisms and techniques proposed in the program analysis literature can be applied to web applications. The actual implementation of such an analysis can utilize different techniques such as model checking, data-flow analysis, or symbolic execution, typically depending on the desired precision and completeness. Three different properties are relevant for the analysis phase. First, the analysis itself can be performed on different levels, either only within a given function (*intraprocedural analysis*) or the interaction of functions can be analyzed as well (*interprocedural analysis*). Second, the execution context can be examined in detail or neglected, which then typically reduces the precision of the analysis. A third aspect of static analysis deals with the way how the flow of data or code is analyzed.

An important analysis technique that is used in many contexts is so called *taint analysis*. As discussed in Section 5.3.1, XSS can predominantly be regarded as an information flow problem, in which unsanitized data paths from untrusted sources to security sensitive sinks have to be stopped. To achieve this, a well established approach is *(dynamic) data tainting*. Untrusted data is outfitted with *taint* information, which is only cleared, if the data passes a dedicated sanitization function. If data which still carries taint information reaches a sink, the application can react appropriately, for instance through auto-sanitization or completely stopping the HTTP response. Dynamic taint tracking was initially introduced by Perl in 1989 [273] and since then has been adopted for numerous programming languagesand frameworks [241]. Most related publications use SQL injection as their motivating scenario and validation target. For brevity reasons, we only discuss taint tracking approaches that explicitly aim at preventing XSS in the following.

Static Analysis With static analysis, the code of a given application is analyzed without actually executing it. Different methods are used to reason about the properties of the code. One of the first tools in this area was *WebSSARI* [113], a code analysis tool for PHP applications developed by Huang et al. based on a CQual-like type system [87, 88].

Pixy is a static taint analysis tool presented by Jovanovic et al. for automated identification of XSS vulnerabilities in PHP web applications [131, 132]. The tool performs an interprocedural, context-sensitive, and flow-sensitive data flow analysis to detect vulnerabilities in a given application. Combined with a precise alias analysis, the tool is capable of detecting a variety of vulnerabilities. In a similar paper, Livshits and Lam demonstrated how a context-sensitive

pointer alias analysis together with string analysis can be realized for Java-based web applications [159].

Xie and Aiken introduced a static analysis algorithm based on so-called block and function summaries to detect vulnerabilities in PHP applications [288]. The approach performs the analysis in both an intraprocedural and an interprocedural way. Dahse and Holz refined this approach and demonstrated that a fine-grained analysis of built-in PHP features and their interaction significantly improves detection accuracy [53, 52].

Wasserman and Su implemented a static code analysis approach to detect XSS vulnerabilities caused by weak or absent input validation [274]. The authors combined work on taint-based information flow analysis with string analysis previously introduced by Minamide [187].

A tool for static analysis of JavaScript code was developed by Saxena et al [236]. The tool named Kudzu employs symbolic execution to find client-side injection flaws in Web applications. Jin et al employ static analysis[125] to find XSS vulnerabilities within HTML5-based Mobile Apps. Interestingly, they found new ways to conduct XSS attacks within such app by abusing the special capabilities of mobile phones. We will discuss this issue in detail in Section 5.8.2.

Dynamic Taint Tracking A complementary approach is dynamic analysis, in which a given application is executed with a particular set of inputs and the runtime behavior of the application is observed. Many of these approaches employ dynamic taint tracking and consist of two different components. A detection component to identify potentially vulnerable data flows and a validation component to eliminate false positives. As discussed in Section 5.5.1, most of these validation components aim at generating a valid exploit to demonstrate the vulnerability.

The first taint tracking paper that aimed at automatically generating Cross-Site Scripting attacks was authored by Martin et al. [175]. The presented mechanism expects a program adhering to the Java Servlet specification and a taint-based vulnerability specification as an input, and generates a valid Cross-Site Scripting or SQL injection attack payload with the help of dynamic taint tracking and model checking. While this approach requires a security analyst to manually write a vulnerability specification, Kieyzun et al. focus on the fully automatic generation of taint-based vulnerability payloads [141]. Furthermore, they extend the dynamic tainting to the database. Hence, as opposed to the first approach, this approach is able to also detect server-side persistent XSS vulnerabilities.

Lekies et al. implemented a browser-based byte-level taint-tracking engine to detect reflected and persistent client-side XSS vulnerabilities [153]. By leveraging the taint-information, their approach is capable of generating a precise XSS payload matching the injection context. A similar approach was taken by FLAX [237], which also employs taint-tracking in the browser. Instead of using an exploit generation technique, FLAX utilizes a sink-aware fuzzing technique, which executes variations of a predefined list of context-specific attack vectors.

Dynamic Testing Very similar to the dynamic tainting approach is dynamic testing. In dynamic testing, an application is executed and fed with several benign or malicious inputs. By observing the program's output, the testing tool determines certain potentially malicious situations and aims to validate the existence of a vulnerability.

SecuBat [136] is a general purpose Web vulnerability scanner that also has XSS detection capabilities. To achieve its goals, SecuBat employs three different components: a crawling component, an attack component, and an analysis component. Whenever the crawling component discovers a suspicious feature, it passes the page under investigation to the attack component, which then scans this page for web forms. If a form is found, appropriate payloads are inserted into the fields of the form and submitted to the server. The response of the server is then interpreted by the analysis component. Snuck [67] is another dynamic testing tool. In a first step, Snuck uses a legitimate test input to dynamically determine a possible injection and the corresponding context via XPATH expressions. Based on the determined context, the tool chooses a set of predefined attack payloads and injects them into the application.

While SecuBat and Snuck focus on server-side security problems, FlashOver [4] focuses on a client-side problem. More specifically, FlashOver detects client-side reflected XSS in Adobe Flash applets. It does so by decompiling the source code and statically detecting suspicious situations. Then it constructs an attack payload and executes the exploit via dynamic testing.

Sanitization As discussed earlier, XSS can also be seen as a sanitization problem. However, choosing and placing the correct sanitizers manually is a very hard problem. Hence, different publications looked into the detection of missing, incorrect, or misplaced sanitizers.

Saner [16] and BEK [111] are two approaches that evaluate the correctness of a given sanitization function. Saner combines static and dynamic analysis techniques to identify incorrect and thus bypassable sanitization functions. BEK is a language for writing and analyzing sanitizers in a sophisticated fashion. Thereby, BEK can be used in two different fashions. It can be used to either analyze existing filters by translating them into BEK, or to initially construct filters that can be translated into other languages after a security evaluation.

Another approach for detecting sanitization problems is ScriptGard [238], which can be used to detect *context-mismatched* or *inconsistent multiple* sanitization. In order to do so, ScriptGard uses positive tainting and server-side instrumentation. In that way, it can either serve as a detection tool or as a runtime prevention technique.

Detection of Attacks

Aa vulnerabilities can never be completely avoided in complex applications, another stream of work was conducted in order to detect and mitigate real-world attacks.

The first academic paper that proposed such an XSS detection mechanism was written by Ismail et al. [116] in 2004. Their system acts as a proxy on the network. The system utilizes two different modes to detect XSS. Both work by matching input parameters to the response of the server. If a match is found, an attack is detected and stored in a collection database.

Xssds [129] is a system for server-side detection of successful XSS attacks. Xssds consists of two separate components: For one, incoming HTTP request and the JavaScript content of outgoing responses are compared to detect non-persistent XSS. To account for server-side processing steps on the incoming data, the system utilizes subsequence matching instead of simple string matching. Furthermore, to detect persistent XSS, Xssds employs a learning based approach that builds a dataset of the application's legitimate scripts. In the production phase, scripts that cannot be found in this set are flagged as a potential XSS attacks. A second training-based, server-side technique was presented by Madou et al. [166]. The system observes the server-side code points that write directly into the HTTP response. For these code points, the characteristics of the produced HTML and JavaScript code are learned, represented by keyword occurrence, and counted. If in productive use of the web application the output characteristics of one of these code points changes in a suspicious fashion, a potential XSS attack is alerted.

Hallaraker and Vigna [98] report on a client-side detection approach that relies on continuos monitoring of JavaScript execution. The system utilizes a set of signatures that represent potential adversary behavior (e.g., cookie stealing). If the browser executes JavaScript that matches one of the signatures, a possible attack is noted.

Opposed to the methods presented before, the approach by Athanasopoulos et al. [13] does not take action during runtime. Instead, they propose to detect XSS offline by automatically analyzing the log files of a given web server.

5.5.3 Mitigation

Beyond detecting XSS problems, another line of research emerged that focusses on ways to mitigate an attack. As opposed to prevention techniques that completely prevent an injection, mitigation techniques aim at prohibiting a successful attack: although an attacker is still able

to inject code into the application, it is not possible for her to conduct any useful, malicious actions. Hereby, we distinguish between three types of mitigation techniques as discussed in the following.

Prohibiting Execution of Injections

The first class of mitigation techniques focusses on prohibiting the meaningful interpretation of the injected content. Thus, although an adversary might be able to inject content into the application, the content will not be regarded/evaluated by the browser. Most of the research papers in this category focus on *Instruction Set Randomization* (ISR) [140]. Thereby, attack-related commands or tags are randomized in such a way that the attacker is not able to guess the correct version that needs to be injected for a successful attack.

One of the first papers that utilized ISR to thwart XSS was NonceSpaces [95]. NonceSpaces uses randomized XML namespaces to communicate with a modified browser. At each request, the server generates a random namespace, prepends it to all legitimate HTML tags and attributes, and communicates the random namespace to the browser in the form of a policy. When the browser receives the response, it only interprets the tags prepended with correct namespaces and regards everything else as untrusted content. As the namespace is generated randomly after the request is received at the server side, the attacker cannot guess the namespace and, hence, is not able to create a meaningful payload.

xJS [14] also utilizes ISR to protect against XSS. Instead of using XML namespaces, xJS makes use of the XOR operation and a secret XOR key. As opposed to NonceSpaces, the XOR operations are only applied to JavaScript code and not to HTML tags. By communicating the key and the XORed code to a xJS-aware browser, the code can be executed normally. An attacker is again not able to guess the secret XOR key and, hence, cannot produce valid script code.

Finally, modern browsers support *sandboxed iFrames* [109], which enable the developer to explicitly isolate untrusted portions from the rest of the document or to forbid JavaScript execution completely in suspicious regions of the document.

Identifying and Blocking Illegitimate Scripts

A second approach for mitigating XSS attacks is to tell malicious and legitimate scripts apart on the parser level.

More specifically, injected code is handed over to the corresponding parser that uses different approaches to determine whether a script is malicious or not. One of the first proposals in this category was made in 2007 with BEEP [124]. With this technique, Web applications are able to specify a list/policy of allowed scripts. When executing scripts, a BEEP-aware browser checks whether a script under investigation is on the whitelist. If so, the script is executed and if not it will be blocked. Similar to BEEP, SOMA [209] also applies a whitelisting approach. However, SOMA does not only focus on script includes, but takes any kind of embeddable content into account. Finally, BEEP's approach was picked up by Stamm et al. who proposed the Content Security Policy (CSP) [248]. The CSP aims at defending against all kinds of XSS vulnerabilities and already achieved a vast browser adoption. The idea behind CSP is to only execute whitelisted scripts. In order to achieve this, the policy forbids several JavaScript/HTML language features such as inline scripts and string-to-code conversions. Any script that is executed by the browser needs to be specified within an external script include. Such includes are only allowed to be fetched from whitelisted servers. As the attacker is typically not able to control the content of such a whitelisted server, she is not able to invoke any script execution. A disadvantage of CSP is the fact that Web sites need to adapt to the proposed scheme, by removing inline scripts, inline stylesheets, and inline event handlers. Doupé et al, therefore, proposed a system capable of automatically rewriting a Web application's HTML code at runtime [66].

Another stream of research focusses on detecting and blocking malicious injection attacks within a victim's browser. The Firefox NoScript extension [172] was a pioneer in this area by introducing the first client-side XSS filter. By investigating an outgoing request for malicious scripts via regular expressions, the extension is able to detect and defuse attacks, before a request leaves the victim's browser.

Similar to NoScript's filter, Microsoft also introduced client-side protection capabilities in Internet Explorer 8 [231]. The filter intercepts and observes HTTP requests and responses. If the filter detects script content within the request via regular expression-based heuristics, it constructs a signature for the suspicious value and scans the corresponding response for matches to this signature. If a match is found, the XSS filter marks this as a server-side reflected XSS attack and neuters the payload by replacing certain characters.

By analyzing NoScript's and Microsoft's XSS filter, Bates et al. identified some issues with the regular expression-based filtering [24]. They propose a slightly different mechanism that does not work on the network layer, but after the initial parsing step of the browser. Thereby, the filter validates wether a parsed code token (e.g., a script tag) is contained in the request. If so, a server-side reflected XSS attack is flagged and hence execution of JavaScript is prohibited. However, as the approach applies a strict string matching to identify tokens within the request, the filter cannot cope with custom processing of a string. So if a Web application, on the server-side, somehow transforms an input string into something slightly different, the filter is not able to detect an attack. Therefore, Pelizzi et al. proposed a slightly adapted version, that does not filter on the basis of strict string matching, but by using partial, similarity-based matching [213]. Based on the work of Bates et al. and Pelizzi et al., Stock et al. propose a new filter design, based on dyanmic data tainting [252]. Whenever a tainted data value, gets transformed into dangerous code or markup the browser automatically blocks the execution. Hence, attacker-controllable input cannot be turned into code and hence, attacks are infeasible in their modified browser.

To dynamically identify legitimate scripts, XSS-Guard [29] generates every HTTP request twice on the server-side: once using the actual data of the request and once using dummy values. The second instance is called the "shadow page", a concept used in many other contexts as well [256]. The tool checks that for both pages the same server-side code is executed. Furthermore, the application framework ensures, that for the shadow page operations, all conditionals result in the same decision as for the actual page. In consequence, the shadow page is completely free of all dynamic values, which potentially could have been provided by the attacker. Before sending the response to the browser, both pages are parsed and all script content is detected. In cases where the actual page contains a script that cannot be found in the shadow page, it is recognized to be injected and, hence, removed.

Wurzinger et al. propose Swap, a proxy-based system that inserts the legitimate scripts only after is has been ensured that the HTML does not contain injected scripts [287]. During server-side processing, all legitimate scripts are added to the HTML in the form of non-standard HTML tags (e.g., `<script1>`) that carry an unambiguous script ID. When the HTML assembly has terminated, the result is parsed. As the legitimate scripts still only exist as non-supported tags, all encountered scripts are flagged as illegitimate and, hence, malicious. If no injected scripts can be found, the non-standard tags are retrieved and replaced with the corresponding scripts, identified through the script IDs.

Runtime Detection and Containment

The third and last set of mitigation techniques focus on detecting and prohibiting malicious actions. Instead of blocking an injected script, these approaches focus on restricting the script's capabilities at runtime, thus, ideally thwarting the adversary's ability to achieve the goal of his attack.

SessionSafe [126] is a server-driven approach that confines a successful XSS exploit to the HTML document in which the attack payload was injected. Through a URL randomization technique, the attack script is unable to further interact with the vulnerable application.

Furthermore, SessionSafe enforces that each document of the application is rendered using a unique subdomain, thus, leveraging the Same-origin Policy to prevent that the attack propagated from one document to the next. Furthermore, a series of approaches were discussed, in which JavaScript wrappers are utilized to mediate access to security sensitive JavaScript APIs, thus, in preventing attacker injected script code to conduct unwanted activities. The approach was initially proposed by Phung et al. [217] and hardened by Magazinius et al. [170]. Promising further development of this technique, using ECMA Script 5's object freezing to protect the wrappers' integrity were recently presented by Heiderich [102].

Livshits and Erlingsson proposed to extend the Same-origin Policy using a novel HTML-attribute [158]. This attribute can create isolated parts within a DOM tree that have only access to selected parts of the DOM tree. If an XSS injection point exists in such a constraint area, the injected script can only access and alter a very restricted set of the attacked webpage's elements (e.g., only values that are located within the reach of the script can be leaked to the attacker).

Finally, several approaches exist that are directly targeted at end-users. Utilizing such techniques, a user can achieve attack mitigation even in cases of a vulnerable Web application that does not provide protection or mitigation capabilities itself. As these approaches have no prior insight in the specifics of the respective application's logic, their general mitigation approach is to prevent the leakage of security sensitive information. For one, Nikiforakis et al. [205] and Tang et al. [259] propose to mitigate session hijacking attempts by automatically identifying session cookies and preventing further access to them by JavaScript. In a related fashion, Vogt et al. used dynamic taint tracking in the Web browser to stop leakage attempts of security sensitive information, such as session identifiers, cookies, or passwords. Kirda et al. proposed Noxes [142], a client-side proxy that prevents XSS-induced leakage of sensitive information through preventing the creation of dynamic HTTP requests to cross-domain hosts. In addition, Stock and Johns [251] propose browser-based defenses against XSS attacks which abuse the browser's built-in password managers to steal the user's stored password.

Mitigation and containment of XSS Worms

The term *XSS Worm* refers to an XSS attack that self-propagates on the vulnerable application [137], as already mentioned in Section 5.2.1. XSS Worms are independent of specific attack vectors and mainly rely on certain characteristics of the vulnerable application. Hence, potential detecting and preventing strategies for XSS Worms do not differ from detecting/preventing other XSS problems. However, their specific exploitation and propagation behavior has led to several research approaches to mitigate and contain XSS Worm outbreaks [290, 254, 157, 50]. Due to the narrow focus of these papers, we omit a deeper discussion for brevity reasons.

5.5.4 Prevention

We now discuss all approaches that either aim to remove XSS vulnerabilities altogether or fully thwart injection attempts, before they reach the browser. Such methods typically either attempt to address particular root causes of XSS vulnerabilities or use generic approaches to prevent them. Compared to mitigation approaches, they have the advantage of comprehensively tackling the problem.

Dynamic Taint Propagation

As discussed previously, taint analysis is an important program analysis technique and we now review how this method is used in the context of preventing XSS vulnerabilities. Both the tool by Nguyen-Tuong et al. [200] and CSSE [218] implement taint tracking on byte-level, which is more fine grained compared to Perl's approach of always tainting the full string [273]. This is in both cases achieved through a modification of PHP's string type implementation, thus

requiring a patched interpreter. The method by Nguyen-Tuong et al. [200] does not allow any tainted information to be present in the HTTP response. Hence, all tainted data, regardless of its destination in the document, is required to pass a sanitization function. If tainted data is found in the HTML, the response is stopped. In contrast, CSSE [218] parses the HTML before sending the HTTP response and checks for tainted information in code contexts. Only if such code is found, the response is stopped, while tainted data values are still permitted. In addition, Xu et al. [289] also present byte-level taint tracking. While XSS is used as one of the example protection policies in this paper, the presented policy (stopping tainted `script` tags) is rather simplistic and leads to many false positives in practice.

Sekar presents a system that hooks into an existing Web server/application infrastructure to apply a technique called *taint inference* [243]. If input to a Web application or an approximation of the input is also present in the output of the Web application, the output is marked as tainted. Based on the taint information and a user supplied policy the system is then able to detect or block an attack.

Mui and Frankl [194] propose to implement dynamic taint tracking indirectly through utilizing a specific unicode character encoding. More precisely, for each existing character in a given charset, a complementary character is introduced that signifies the tainted state of this character. As long as data that is encoded in this fashion is processed by systems that recognize the specific characteristics of this encoding (i.e., complementary characters have to be regarded as equivalent for most purposes), the taint status of the data is maintained automatically. If such data reaches a parser, such as the browser's HTML parser, tainted code fragments can be detected and disarmed precisely.

Another taint-based approach that aims at bridging the client-server gap is Dsi [195]. Dsi aims to prevent attackers from altering the document structure. They do so by enforcing Document Structure Integrity through isolating untrusted user input from the rest of a document via dynamic tainting and confinement on the parser-level.

Separation of Code and Data

In essence, XSS is a form of *code* injection by the means of attacker controlled *data*. Thus, a promising approach is to enable a Web application to cleanly separate data and code while generating the application's HTML content [223]. This way, at the point of generating the final mark-up and script code, the Web application is able to reliably apply matching sanitization to the *data* parts of the response. This general approach has been realized for a set of Web frameworks.

Robertson and Vigna present a novel Web framework that is built on top of the semantics of Haskel's Monads [227]. Instead of resorting to strings for server-side HTML assembly, the tree structure of an HTML document is kept in the form of internal data nodes. In the resulting framework, markup and script code can be added to such documents using type-safe APIs. Hence, a direct string-to-code conversion, one of the problems underlying XSS attacks, is impossible. Johns et al. combine a type-safe API and an abstraction layer to achieve similar results for the J2EE application server [128]. The API is used to assemble HTML documents in a pre-parsed state on the server-side, kept in a dedicated data type which strictly separates data and code fragments. The abstraction layer receives the HTTP response's body through this type and robustly serializes the resulting HTML. Due to the precise context information given by the pre-parsed document, the abstraction layer can chose the matching sanitization functions for the document's code fragments. A similar path is followed by Samuel et al., who introduced a *context-sensitive auto-sanitization* (CSAS) engine [235]. Instead of targeting Web frameworks that allow full server-side freedom in assembling the final mark-up, they augment more restrictive Web templating languages.

Blueprint [163] is a system that, similar to Dsi, aims to bridge the gap between client and server. It prevents XSS by removing the initial parsing and construction step of a Web page's document structure. Instead of sending a single string, Blueprint sends an additional

"Blueprint" of the legitimate document structure to the client. As the client is aware of this structure, attacker-controlled content is not able to alter the document structure and hence is not able to insert node that trigger the execution of scripts.

Sanitizer Usage and Placement

To address the problem of correct sanitizer usage, Saxena et al. present ScriptGard, a framework for testing sanitizer placement errors and runtime auto-sanitation [238]. The authors first present a taint-based approach to identify two previously unknown sanitation problems namely context-mismatched sanitation and inconsistent multiple sanitation. After a training period, the system is then able to automatically apply the correct sanitation routines at runtime.

Livshits and Chong propose a shift in the way how sanitizers are placed in Web applications: instead of letting the developer place the sanitizers manually during development (a process that is known to be potentially incomplete and error prone) they propose to place the sanitizers completely automatically after the source code has been finalized [156]. To do so, their system utilizes static analysis to identify all potentially problematic data flows in the application. Using automatic reasoning, the system chooses the context-dependent correct sanitizer and positions it at the algorithmic best place on the respective data path. For cases in which at compile time the correct sanitizer cannot be determined, the system injects additional runtime checks.

Protection against Specific Attacks

Besides general purpose prevention approaches, several techniques have been proposed that target only a specific subset of all potential XSS vulnerabilities. For example, to counter potential XSS vulnerabilities through the native JavaScript APIs `innerHTML` and `localStorage`, Heiderich et al. [107] and Lekies et al. [152] propose corresponding safe API alternatives. In both cases, the secure API variants can be utilized by Web application developers using a JavaScript library that wraps and mediates the native API.

As discussed in Sec. 5.5.1, Barth et al. identified a potential attack vector using file uploads of documents which are handled ambiguously by Web browsers [20]. To counter such attacks, they generalized the content sniffing algorithms of Web browsers and propose a secure alternative that is no longer susceptible to said attacks. Furthermore, they described a complementary server-side filtering approach that closely mimics the observed content sniffing behavior of vulnerable browsers and, thus, reliably detects potentially harmful documents. In a similar fashion, Heiderich et al. [105] propose a server-side sanitization approach for SVGs that counters their discovered attacks (see Sec. 5.5.1) and Magazinus et al. generalize such attacks using so-called *polyglots* [171].

5.6 Deployability Considerations

As motivated in Section 5.4.2, based on an approach's point of deployment, we propose a coarse grained classification in the three general classes *Server* (the approach takes action on the server-side of the application, i.e., as part of the server-side infrastructure or within the application's source code), *Client* (the approach is implemented as part of the client-side infrastructure, i.e., the browser or in the form of a client-side proxy) or *Hybrid* (approaches that have a foothold both on the server and the client side). As it can be seen in Table 5.1, the set of surveyed approaches is fairly well spread over the different classes, with a slight bias towards server-side techniques and notably missing detection approaches in the "hybrid" category.

Furthermore, we pay special attention to the deployment requirements of a specific subset of all approaches – namely detection, mitigation, and prevention techniques that have been designed to be adopted by application providers for continuous, productive use. In such scenarios, the to be expected cost of adoption is a major factor. While changing an application's framework is

Table 5.1: Classification of approaches according to point of deployment

	Detection	**Mitigation**	**Prevention**
Server	[13] [16] [111] [116] [129] [131] [132] [136] [141] [113] [175] [187] [274] [288] [125] [266] [52] [53] [166]	[50] [290] [126] [157] [29] [287]	[105] [128] [235] [238] [156] [218] [227] [243] [163] [200] [289]
Client	[98] [153] [237][4]	[24] [142] [205] [254] [197] [259] [172] [213] [231] [251] [252]	[20] [107] [152]
Hybrid		[14] [95] [124] [170] [209] [217] [248] [158] [66]	[195] [194]

probably feasible in most cases, converting the whole user-base of a commercial application to use an experimental browser add-on might not.

To get an impression of how the server-driven approaches perform under this criteria, we refined our categories as follows (starting with the lowest adoption effort and ending with the highest):

- *Changing server-side infrastructure (SSI)*: The approach requires changes to the server-side infrastructure, e.g., the utilized Web framework, the application server, or an outgoing HTTP response rewriting component.
- *Automatic code changes (ACC)*: The approach requires the alteration of the application's source code. This change can be conducted automatically by a code-rewriting technique.
- *Manual code changes (MCC)*: The approach requires manual changes in the application's source code or source code that was written from scratch using novel technology introduced by the approach.
- *Changing client-side infrastructure (CSI)*: The approach requires explicit support by the client-side infrastructure, e.g., the Web browser.

A given approach can appear in more than one of these classes, for instance if server-side code changes and a modified browser engine are required by the approach. Please refer to Table 5.2 for an overview on the surveyed papers (please note, that compared to Table 5.1 the total number of detection approaches is significantly lower, as we here only consider server-driven *attack* detection techniques). Based on this representation of the collected data, it becomes apparent, that robust protection capabilities often correlate with costly deployment requirements. We will revisit this observation in Section 5.7.1.

5.7 Analysis and Discussion

Based on our classification and review of the existing literature on XSS, we now discuss the general lessons learned.

Table 5.2: Fine grained deployment effort classification of server-driven approaches

Class	Approach	SSI	ACC	MCC	CSI
Detection	[116] [129] [166]	X			
Mitigation	[243] [287] [29]	X			
	[170] [217]			X	
	[126]	X		X	
	[24] [142] [205] [259] [197] [172] [213] [231] [252] [251]				X
	[14] [95]	X			X
	[124] [209] [248] [158]			X	X
	[66]		X		X
Prevention	[200] [156] [289] [218]	X			
	[238]		X		
	[105] [107] [128] [235] [152]			X	
	[227] [163]	X		X	
	[20]				X
	[195] [194]	X			X

SSI: Server-side infrastructure, *AAC*: Automatic code changes, *MCC*: Manual code changes, *CSI*: Client-side infrastructure

5.7.1 Revisiting the Facets of XSS

In Section 5.3, we discussed different viewpoints of how XSS vulnerabilities can be approached. We now revisit this classification and examine how the surveyed approaches address the raised problems.

XSS as an Information Flow Problem

As discussed previously, XSS can be seen as an information flow problem since the path of untrusted data from an attacker controlled source to a security sensitive sink is a mandatory precondition for all XSS vulnerabilities. Hence, it is not surprising that a considerable amount of approaches attempt to address XSS through identifying insecure data flows: For one, static analysis is utilized to detect vulnerable data flows on compile time [131, 132, 288, 274, 113] and prevent vulnerabilities via automatically place sanitizers [156] on such flows. Furthermore, several approaches use dynamic tracking of information flows for vulnerability detection [141, 153, 237, 175] and prevention [200, 218, 289, 194, 195].

A survey of the presented techniques results in three general observations:

- *Coverage:* With the exception of a few dynamic taint trackers [141, 195, 194], the presented approaches limit the detection of vulnerable flows only to single components, i.e., either the server-side or the client-side code. However, in practice complex flows can span the application's full scope, including server-code, persistent storage, client-side JavaScript, and the DOM tree. Hence, sufficient coverage is a challenge for both static and dynamic techniques.

- *Performance:* While performance is not necessarily a critical characteristic for static detection techniques, it is crucial for techniques that are designed to instrument the productive

use of a Web application at runtime. Depending on the situation, the overhead introduced during the taint tracking phase might be too high and prevent a wide adoption. The surveyed techniques report on overhead ranging from 0.09% to 8% in best case scenarios and 1.74% to 106% in the worst case. However, due to a missing common benchmarking approach (see Sec. 5.8.5) and highly diverse measurement methodologies a comparative or qualitative evaluation is difficult.

- *Sanitization:* Finding an insecure flow solves only one half of the problem for prevention purposes. Equally important is choosing the correct sanitization for the given code context (see Sec. 5.3.2). We will elaborate this aspect further in the following section.

XSS as a Sanitization Problem

Deducting from the surveyed literature, we can isolate two distinct problem classes in respect to sanitization:

- *Sanitizer selection and placement:* Even in case that a potentially insecure data flow has been identified, it is not always straightforward to choose the matching sanitizing approach, as secure sanitizing is highly dependent on the specific syntactic context of the injection point. Weinberger et al. [276] list a total of six challenges in XSS sanitization, including "sanitizing nested contexts", "browser transductions", and "dynamic code evaluation". Based on their observations, the authors conducted a systematic study on auto-sanitization in 14 commercially-used Web frameworks and noticed that these framework often fail to apply the correct sanitization. Furthermore, Saxena et al. [238] conducted a systematical analysis of a large legacy application and found that out of 25,209 observed information flows, 6.1% were improperly sanitized. Livshits and Chong acknowledge the challenges and address the problem by injecting runtime checks for sanitizer usage, in cases in which the exact syntactic code context cannot be determined on compile time [156].

- *Sanitizer correctness:* Many approaches, especially the taint-tracking based techniques, assume that sanitizers are always correct. Hence, if tainted data passes a sanitizer, taint is removed or code constructs are flagged as non-vulnerable. However, in practice sanitizers are often faulty, especially when they are not written by security experts. Evidence for this was given by systematic studies, using the test approaches for sanitizer correctness Saner [16] and Bek [111], which uncovered several insecurities in commercial sanitizers.

XSS as a Mitigation Problem

As discussed in Section 5.5.3, all mitigation techniques have in common that the application's underlying injection vulnerability remains untouched. Hence, the attacker maintains her ability to inject arbitrary data into the page's HTML content.

Within the class of mitigation approaches exists a hierarchy in respect to the robustness of the offered protection capabilities: Most powerful are approaches that prohibit the client-side interpretation of injected code completely (see Sec 5.5.3), such as NonceSpaces or xJS. Regardless of the nature of the attacker's injected code, the injection leads to no noticeable effect on the client-side.

Next in this order reside techniques that aim to differentiate between legitimate and injected scripts on runtime (see Sec. 5.5.3). These approaches either function via approximation, as it is the case with browser-based XSS filers [172, 231, 24], which are prone to false negatives, or rely on additional, server-provided policy information [124, 209, 248], which might be incomplete or too weak. The potentially worst protection capabilities are provided by techniques that allow the injected script to be executed, but aim to limit its potential actions (see Sec.5.5.3). An inherent characteristic of such approaches is that they are geared against a predefined attacker behavior, which they try to stop (e.g., sessions hijacking [205, 259, 126] or stealing of sensitive

information [142, 197]). If confronted with an attacker, that leaves the preconceived attack path, such measures are potentially weak and can be bypassed.

However, when evaluated under the aspect of deployability, the order almost completely reverses (see Table 5.2): Many of the containment approaches can be adopted comparatively easily, either through a simple `script` include [217, 170, 102] or a client-side proxy for individual end-user protection [142, 205]. Opposed to this, NonceSpaces or xJS require non-trivial changes both on the server- and client-side.

5.7.2 Reoccurring Strategies

Based on the data collected in Section 5.5, we are able to identify several strategies to deal with XSS that have been utilized by the surveyed approaches.

Reducing Complexity

The landscape of Web technologies and components is highly diverse and complex: a Web application is a combination of several loosely coupled systems, including (but not restricted to) the Web server, browser, and database, utilizing a mix of various programming, mark-up, and domain-specific languages. Thus, it is nontrivial to find a common set of assumptions that apply to all Web applications on which protective approaches could rely on. This makes a general solution to address XSS vulnerabilities a complex problem. Consequently, an often utilized strategy is to reduce the degrees of freedom in composing and implementing Web applications, thus, creating a common, reliable foundation that can be leveraged by defensive techniques. For one, a set of approaches to reduce the developer's freedom in composing the client-side of the application: CSP [248] restricts the usage of inline scripts and unsafe string-to-code conversions. CSAS [235] demands the usage of a Web templating engine to compose the UI instead of free-from HTML. Going even a step further, approaches that implement strict code/data separation impose strong restrictions on the server-side code as well, requiring the application to be written with specific combinations of type-safe languages and Web frameworks [227, 128].

Overcoming the Server/Browser Gap

A general characteristic for the majority of XSS vulnerabilities is, that the vulnerability resides on the server-side, while the attack manifests itself on the client-side in the browser. This is a problem for defensive techniques, as the server has only a very limited direct insight into how the browser interprets the HTTP response. Several approaches try to overcome this problem through server-side emulation of browser behavior [129, 29, 287, 200, 163]. However, the resulting necessity to fully parse the HTML on the server-side results in noticeable run-time overhead and latency, that might hinder the adoption of such tools. Furthermore, the diversity of browsers implies that discrepancy in the interpretation of content can potentially lead to bypasses of the defense methods. An alternative strategy can be found in approaches that take action both on the server and the client. In most such cases, the server provides additional information that allows the client to separate legitimate from injected content [163], thus, permitting robust client-side enforcement. Such additional information could be given by enforcement policies, as it is the case with CSP [248] or BEEP [124], or using specific encodings which cannot be injected by the attacker [95, 14, 195]. The main drawback of such solutions is their cost of adoption, as discussed in Section 5.6.

Instruction Set Randomization

A third reoccurring technique is Instruction Set Randomization (ISR) [140]. ISR is used in several variations for attack mitigation [95, 195, 14] and prevention [287]. Furthermore, on a conceptional level, techniques that require secret nonces to be present in HTML tags (e.g., `jail`-tag [70] or CSP's script nonce [272]) utilize the same underlying principle.

5.8 Open Research Problems

Our analysis also enables us to identify open problems and topics for future research on XSS as discussed in the following.

5.8.1 Client-side XSS

As it was demonstrated in 2013 by Lekies et al [153], client-side XSS (see Sec. 5.2.2) is a widespread problem in modern Web sites. In this study, the authors found DOM-based XSS vulnerabilities in roughly 10% of the Alexa Top 5000. However, only a small number of the approaches presented in Section 5.5 explicitly consider this specific subclass of vulnerabilities. These papers either implement variations of dynamic taint-tracking in the browser [237, 153, 195, 194] or address specific subproblems, such as second-order code injection [152] or client-side XSS caused by plugins [4]. It remains to be evaluated, to which degree the other approaches can be adapted to the specifics of client-side XSS.

5.8.2 XSS Outside of the Browser

Ever since HTML was adopted to create UIs of general purpose applications, XSS is no longer confined to the Web browser. Such attacks were for instance demonstrated for Skype [164, 12] and widgets for the Mac OS X Dashboard [228]. Lately, Web frontend technologies even found their way into mobile applications via WebViews that are embedded in the app's UI. A survey in 2012 found that up to 75% of all interviewed mobile app developers plan to use Web UI technologies in the future [257]. Apps using this technology are susceptible to XSS as well, leading to the execution of arbitrary code [199]. Up to now, only little systematic research has been conducted on XSS outside of the browser. However, given the growing importance of this development approach and the potentially enhanced privileges of injected scripts, the topic warrants further attention and should be studied in detail.

5.8.3 CSP and Script-less Attacks

The future for CSP appears to be bright. With the exception of Internet Explorer, all major browsers already support the basic CSP directives. The next iteration of the standard, CSP 1.1, will bring several enhancements for developers, such as script nonces or a JavaScript API [272] that have the potential to address current drawbacks of CSP 1.0 [275]. A strong CSP provides very robust mitigation guarantees that only break under certain circumstances, such as vulnerable server-side JavaScript generation [127] or insecure JSONP consumption [293]. Hence, CSP has the opportunity to become a major factor in the battle against XSS exploits, in case of substantial adoption of the standard in the future. However, CSP only mitigates JavaScript injection, while mark-up injection still remains possible. For this reason, offensive research emerged, that focus on XSS exploits that do not rely on JavaScript execution. For instance, Heiderich et al. [106] showed how to leverage HTML/CSS injection in combination with SVG fonts to leak sensitive data from an application. Additional information exfiltration attacks that function *without* scripting have been documented by Chen et al. [43] and Zalewski [293]. The capabilities and likelihood of such attacks are still subject to future research. Furthermore, defenses against script-less attacks are needed.

5.8.4 Self-XSS

The term *Self-XSS* describes a class of social engineering attacks in which a user is tricked to copy & paste `javascript:`-URLs into the browser's address bar, causing the browser to execute the contained script in the context of the top-level document. As JavaScript's syntax allows

several obfuscation tricks (e.g., encoding the full script without alpha-numeric characters [103]), the nature of the to-be-pasted URL is easily hidden and social engineering attacks are viable.

As a response to the growing number of reported Self-XSS attacks, Facebook introduced protective measures, but remains vague about the countermeasure's technical details [74]. Furthermore, Firefox disabled the `javascript:`-URL scheme for URLs pasted in the address bar [100]. While this closes the address bar attack vector, other methods remain open, including using the Firefox Web console [202] or tricking the user to create rogue bookmarklets. None of the other major browsers have followed Firefox's example yet.

On a technical level, Self-XSS offers the same offensive capabilities as all JavaScript injection techniques. Hence, it has to be rated on an equivalent severity level. However, up to this point only little research has been done in respect to applicable defensive measures and thus research on containing and mitigating such injections is necessary.

5.8.5 No Common Evaluation Methodology

A common methodology to measure and compare competing approaches is direly missing. This applies to performance, quality, and compatibility measurements all alike. For instance, the performance of memory corruption countermeasures is frequently measured using the SPEC benchmarking suite or through recompilation of well-known open source programs, such as Firefox. A similar shared measurement procedure does not exist for Web security research. Likewise, no suitable dataset is available to evaluate the quality of proposed techniques. For this reason, several papers resort to data provided by the site `xssed.com`, which offers a public archive of reported vulnerabilities. However, `xssed.com`'s data is of unvalidated quality and consist of only a specific subset of XSS, namely non-persistent server-side XSS that is easy to archive. For compatibility evaluations, typically the top sites of the Alexa ranking[1] are crawled and examined. As Jackson pointed out [118], crawling the Alexa sites exposes the evaluated mechanism only to a subset of the available Web technologies and code patterns, namely the public, information-centric Web. However, Web *applications* that implement rich functionality cannot be covered easily by automated crawling processes. Another problem of the Alexa-based evaluation approach is the reproducibility of the results. Even crawling one page twice in a time frame of a few seconds from the same computer could reveal different results due to dynamically generated parts of a Web page such as dynamic advertisement code. Furthermore, with network jitter and changing content, the Alexa Top sites are a moving target. Repeating or reviewing such an evaluation is therefore not possible, a serious deficit for scientific work. Thus, the creation of shared approaches and datasets for measuring essential characteristics of Web security research would be of significant benefit to the whole discipline.

[1] Alexa Top 500 Global Sites: `http://www.alexa.com/topsites`

Chapter 6

Conclusion

Web Security Architecture is a trendy topic. Substantial investment from the Community and research cause very rapid development. After years of insufficient attention, the potential for high scaling damages have helped put Web Security where it belongs: a central point of review for emerging new web technologies. A toolbox helps Web application developers to mitigate threats. But there is still a long way to go. Developments on the protocol layer need to be confirmed and have to be proven to be implementable. The transformation of the Web into a platform for applications makes additional research necessary.

To mitigate and prevent known vulnerabilities, the Web Security Architecture has to be improved with:

- *An improved session management for web applications and sites alike.* The current practice of using the session identifier as the bearer token has been shown to be error- and vulnerability prone. Instead, more sophisticated mechanism are needed that provide stronger bindings between the authenticated parties and tie better into the stateful workflows of Web applications.

  To exemplify such mechanisms, we have discussed SecSess, which proposes a potential improvement: , SecSess uses the shared secret to add an hash-based message authentication code (HMAC) to the request, thereby legitimizing the request within the session. Since this HMAC takes the request and the shared secret as input, only the browser and the server can compute the correct values. Incoming requests with an invalid HMAC are simply discarded by the server.

- *A reliable JavaScript sandboxing mechanism.* Untrusted code has to be isolated, but current isolation techniques have too many disadvantages. Work on confinement needs more research and standardisation to support a virtual DOM creation to allow full mediation.

- *A better mitigation of XSS vulnerabilities.* This report consolidates the existing knowledge on Cross-Site Scripting, especially since the notion of this type of vulnerability changed a lot over time. As there are different causes behind such attacks the report examines the different options of approaching the problem. This leads to a classification scheme for past, current, and future research in this area that also serves to direct current attempts to mitigate the issue. Based on this classification, it becomes obvious that the XSS problem is far from being "solved", as multiple areas exist for which no comprehensive protection/mitigation mechanism exist.

This list of identified, problematic areas is far from complete. It merely reflects closely the contents of this report. The fast evolving area of Web application security will require significant attention both from research and standardization in the future, so that the Web remains the fast incubator of innovation that is has been in the past, without jeopardizing the Web's users' security in the process.

References

Bibliography

[1] Ietf web security working group.

[2] *W3C Workshop on Privacy and User–Centric Controls*, 2014.

[3] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[4] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen, and Frank Piessens. FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. In *ASIA Computer and Communications Security (ASIACCS)*, May 2012. partner: KUL; project: WebSand, NESSoS.

[5] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. WebJail: least-privilege integration of third-party components in web mashups. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 307–316. ACM, 2011.

[6] Ben Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceedings of the 17th international conference on World Wide Web*, pages 517–524. ACM, 2008.

[7] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank 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.

[8] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304. IEEE, 2010.

[9] Adiel A. Akplogan, John Curran, Paul Wilson, Russ Housley, Fadi Chehadé, Jari Arkko, Lynn St. Amour, Raúl Echeberría, Axel Pawlik, and Jeff Jaffe. Montevideo statement on the future of internet cooperation, oct 2013.

[10] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. *IEEE Symposium on Security and Privacy*, 2008.

[11] Nadhem J AlFardan and Kenneth G Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[12] Adrian Asher. Explaining the cross site scripting bug in skype for windows. [online], `http://blogs.skype.com/2011/07/15/explaining-the-cross-site-scri/`, July 2011.

[13] Elias Athanasopoulos, Antonis Krithinakis, and Evangelos P. Markatos. Hunting cross-site scripting attacks in the network. In *Workshop on WEB 2.0 Security and Privacy (W2SP)*, 2010.

[14] Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P. Markatos, and Thomas Karagiannis. xjs: practical xss prevention for web application development. In *USENIX conference on Web application development*, WebApps'10, Berkeley, CA, USA, 2010. USENIX Association.

[15] T. Aura. Cryptographically Generated Addresses (CGA). RFC 3972 (Proposed Standard), March 2005. Updated by RFCs 4581, 4982.

[16] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, SP '08, pages 387–401, Washington, DC, USA, 2008. IEEE Computer Society.

[17] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.

[18] A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), December 2011.

[19] Adam Barth. HTTP state management mechanism. *IETF Proposed Standard*, 2011.

[20] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *IEEE Symposium on Security and Privacy*, SP '09, pages 360–371, Washington, DC, USA, 2009. IEEE Computer Society.

[21] Adam Barth, Collin Jackson, and John 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.

[22] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[23] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. *Communications of the ACM*, 52(6):83–91, 2009.

[24] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *International Conference on the World Wide Web (WWW)*, WWW '10, pages 91–100, New York, NY, USA, 2010. ACM.

[25] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol version 2. Internet-Draft draft-ietf-httpbis-http2-17, Internet Engineering Task Force, February 2015. Work in progress.

[26] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. Freak: Factoring rsa export keys.

[27] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over tls.

[28] John Biggs. Hotels inject banner ads into the wi-fi they charge you for. [online], `http://techcrunch.com/2012/04/06/now-you-know-hotels-inject-banner-ads-into-the-wi-fi-they-charge-you-for/`, April 2012.

[29] Prithvi Bisht and V. N. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, DIMVA '08, pages 23–43, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] Internet Architecture Board. Iab statement on internet confidentiality.

[31] Hristo Bojinov, Elie Bursztein, and Dan Boneh. Xcs: cross channel scripting and its impact on web applications. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '09, pages 420–431, New York, NY, USA, 2009. ACM.

[32] Andrew Bortz, Adam Barth, and Alexei Czeskis. Origin cookies: Session integrity for web applications. *Web 2.0 Security and Privacy (W2SP)*, 2011.

[33] Frederik Braun, Devdatta Akhawe, Joel Weinberger, and Mike West. Subresource Integrity. W3C Working Draft, W3C, March 2014. Work in progress. `http://www.w3.org/TR/2014/WD-SRI-20140318/`.

[34] Frederik Braun, Devdatta Akhawe, Joel Weinberger, and Mike West. Subresource Integrity. W3C Working Draft, W3C, April 2015. Work in progress. `http://www.w3.org/TR/2014/WD-SRI-20140318/`.

[35] Frederik Braun and Mario Heiderich. X-frame-options: All about clickjacking? Sep 2013.

[36] BuiltWith. jQuery Usage Statistics. `http://trends.builtwith.com/javascript/jQuery`.

[37] Stefano Calzavara, Gabriele Tolomei, Michele Bugliesi, and Salvatore Orlando. Quite a mess in my cookie jar! *To appear at WWW 2014*, 2014.

[38] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, Yan Chen, and Xitao Wen. Virtual browser: a virtualized browser to sandbox third-party JavaScripts with enhanced security. In Heung Youl Youm and Yoojae Won, editors, *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, pages 8–9. ACM, 2012.

[39] Damien Cassou, Stéphane Ducasse, and Nicolas Petton. Safejs: Hermetic sandboxing for javascript. *CoRR*, abs/1309.3914, 2013.

[40] CERT. Advisory ca-2000-02 malicious html tags embedded in client web requests, February 2000.

[41] Charles Severance. JavaScript: Designing a Language in 10 Days. `http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.html`.

[42] Graham Charlton. Companies spending more on web analytics: survey.

[43] Eric Y. Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *IEEE Symposium on Security and Privacy*, 2012.

[44] European 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.

[45] European Commission et al. *Directive 2009/136/EC of the European Parliament and of the Council of 25 November 2009 amending Directive 2002/22/EC on universal service and users' rights relating to electronic communications networks and services, Directive 2002/58/EC concerning the processing of personal data and the protection of privacy in*

the electronic communications sector and Regulation (EC) No 2006/2004 on cooperation between national authorities responsible for the enforcement of consumer protection laws, volume 337, pages 11–36. December 2009.

[46] Lucian Constantin. Xss worm hits orkut, September 2010.

[47] Lucian Constantin. Facebook hit by xss worm, March 2011.

[48] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[49] Douglas Crockford. ADsafe – making JavaScript safe for advertising. `http://adsafe.org/`.

[50] Arshan Dabirsiaghi. Building and stopping next generation xss worms. In *OWASP Application Security Conference (OASC)*, 2008.

[51] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Transactions on Internet Technology (TOIT)*, 12(1):1, 2012.

[52] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[53] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *USENIX Security Symposium*, 2014.

[54] Dancho Danchev. Twitter hit by multiple variants of xss worm, April 2009.

[55] P. De Ryck, L. Desmet, F. Piessens, and M. Johns. *Primer on Client-Side Web Security*. SpringerBriefs in Computer Science. Springer, 2014.

[56] Philippe De Ryck. *Client-Side Web Security: Mitigating Threats against Web Sessions*. PhD thesis, December 2014.

[57] Philippe De Ryck, Lieven Desmet, Pieter Philippaerts, and Frank Piessens. A security analysis of next generation web standards. Technical report, G. Hogben and M. Dekker (Eds.), European Network and Information Security Agency (ENISA), July 2011.

[58] Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. SecSess: Keeping your session tucked away in your browser. In *Proceedings of the 30th ACM Symposium on Applied Computing (SAC)*, 2015.

[59] Philippe De Ryck, Wouter Joosen, Frank Piessens, Martin Johns, Elwyn Davies, Bert Bos, Thomas Roessler, Lieven Desmet, Sebastian Lekies, Jan Tobias Mühlberg, and Stephen Farrell. Web-platform security guide: Security assessment of the web ecosystem. Technical report.

[60] Philippe De Ryck, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Serene: self-reliant client-side protection against session fixation. In *Distributed Applications and Interoperable Systems*, pages 59–72. Springer, 2012.

[61] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5), 1976.

[62] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176, 7465.

[63] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176, 7465.

[64] Michael Dietz, Alexei Czeskis, Dirk Balfanz, and Dan S Wallach. Origin-bound certificates: A fresh approach to strong client authentication for the web. In *USENIX Security Symposium*, pages 16–16, 2012.

[65] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In Robert H'obbes' Zakon, John P. McDermott, and Michael E. Locasto, editors, *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 297–306. ACM, 2011.

[66] Adam Doupé, Weidong Cui, Mariusz H Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. dedacota: Toward preventing server-side xss via automatic code and data separation. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[67] Fabrizio dÃŢAmore and Mauro Gentile. Automatic and context-aware cross-site scripting filter evasion. *Department of Computer, Control, and Management Engineering Antonio Ruberti Technical Reports*, 1(4), 2012.

[68] Donald Eastlake 3rd. Transport layer security (TLS) extensions: Extension definitions. *RFC 6066*, 2011.

[69] ECMAScript. Harmony Direct Proxies. `http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies`.

[70] Brendan Eich. Javascript: Mobility & ubiquity. presentation at the Dagstuhl Seminar 09141, February 2007.

[71] Jochen Eisinger and Mike West. Referrer Policy. Technical report.

[72] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. Internet-Draft draft-ietf-websec-key-pinning-21, Internet Engineering Task Force, October 2014. Work in progress.

[73] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), April 2015.

[74] Facebook. Keeping you safe from scams and spam. [online], `https://www.facebook.com/notes/facebook-security/keeping-you-safe-from-scams-and-spam/10150174826745766`, May 2011.

[75] Stephen Farrel and Hannes Tschofenig. Pervasive Monitoring is an Attack. *RFC Best Current Practice (RFC 7258)*, 2014.

[76] S. Farrell, P. Hoffman, and M. Thomas. HTTP Origin-Bound Authentication (HOBA). RFC 7486 (Experimental), March 2015.

[77] S. Farrell and H. Tschofenig. Pervasive Monitoring Is an Attack. RFC 7258 (Best Current Practice), May 2014.

[78] S. Farrell, R. Wenning, B. Bos, M. Blanchet, and H. Tschofenig. STRINT workshop report. Internet-Draft draft-iab-strint-report-00, Internet Engineering Task Force, April 2014. Work in progress.

[79] Stephen Farrell. Heartbleed - what can we learn? Talk, IESG Retreat, Cancun, Mexico, May 2014.

[80] Stephen Farrell, Rigo Wenning, and Hannes Tschofenig. W3c/iab workshop on strengthening the internet against pervasive monitoring (strint).

[81] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585.

[82] R. Fielding, M. Nottingham, and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Caching. RFC 7234 (Proposed Standard), June 2014.

[83] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230 (Proposed Standard), June 2014.

[84] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. *RFC 2616*, 1999.

[85] Matthew Finifter, Joel Weinberger, and Adam Barth. Preventing Capability Leaks in Secure JavaScript Subsets. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010.

[86] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress, 2011.

[87] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. *SIGPLAN Not.*, 34(5), May 1999.

[88] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. *SIGPLAN Not.*, 37(5), May 2002.

[89] Yossi Gilad and Amir Herzberg. Off-path attacking the web. In *USENIX Workshop on Offensive Technologies (WOOT)*, pages 41–52, 2012.

[90] Google Chrome Developers. Native Client. `https://developer.chrome.com/native-client`.

[91] Ilya Grigorik. Http/2 all the things!

[92] Jeremiah Grossman. Blackhat talk; phishing with superbait, October 2005.

[93] Salvatore Guarnieri and V. Benjamin Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In Fabian Monrose, editor, *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 151–168. USENIX Association, 2009.

[94] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In Theo D'Hondt, editor, *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150. Springer, 2010.

[95] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[96] WG Halfond, Jeremy Viegas, and Alessandro 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.

[97] P. Hallam-Baker. Http integrity header. *IETF Internet Draft*, 2012.

[98] Oystein Hallaraker and Giovanni Vigna. Detecting malicious javascript code in mozilla. In *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 85–94. IEEE, 2005.

[99] Harry Halpin. Web cryptography next steps - w3c workshop on authentication, hardware tokens and beyond.

[100] Nathan Hammond. Social Engineering Issue with "javascript:" URLs. Mozilla Bugzilla bug bug #527530, 2009.

[101] Steve Hanna, Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. The Emperors New APIs: On the (In)Secure Usage of New Client Side Primitives. In *Workshop on WEB 2.0 Security and Privacy (W2SP)*, 2010.

[102] M. Heiderich. *Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM*. PhD thesis, University Bochum, 2012.

[103] M. Heiderich, E.A.V. Nava, G. Heyes, and D. Lindsay. *Web Application Obfuscation: '-/WAFs..Evasion..Filters//alert(/Obfuscation/)-'*. Elsevier insights. Elsevier Science, 2011.

[104] Mario Heiderich, Tilman Frosch, and Thorsten Holz. Iceshield: Detection and mitigation of malicious websites with a frozen DOM. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*, volume 6961 of *Lecture Notes in Computer Science*, pages 281–300. Springer, 2011.

[105] Mario Heiderich, Tilman Frosch, Meiko Jensen, and Thorsten Holz. Crouching tiger - hidden payload: security risks of scalable vectors graphics. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '11, pages 239–250, New York, NY, USA, 2011. ACM.

[106] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg 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.

[107] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z Yang. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[108] Gareth Heyes, Eduardo Vela Nava, and David Lindsay. CSS: The Sexy Assassin. Talk at the Microsoft Blue Hat conference, `http://technet.microsoft.com/en-us/security/cc748656`, October 2008.

[109] Ian Hickson. The iframe element, November 2013.

[110] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), November 2012.

[111] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *USENIX Security Symposium*, SEC'11, Berkeley, CA, USA, 2011. USENIX Association.

[112] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schechter, and Collin Jackson. Clickjacking: attacks and defenses. In *USENIX Security Symposium*, pages 22–22, 2012.

[113] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *International Conference on the World Wide Web (WWW)*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[114] IAB and IESG. IAB and IESG Statement on Cryptographic Technology and the Internet. RFC 1984 (Informational), August 1996.

[115] Lon Ingram and Michael Walfish. Treehouse: Javascript sandboxes to help web developers help themselves. In *Proceedings of the USENIX annual technical conference*, 2012.

[116] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on*, volume 1, pages 145–151. IEEE, 2004.

[117] Jacaranda. Jacaranda. `http://jacaranda.org`.

[118] Collin Jackson. Crossing the Chasm: Pitching Security Research to Mainstream Browser Vendors. Invited talk at the USENIX Sec Conference, August 2011.

[119] Collin Jackson and Adam Barth. Beware of finer-grained origins. In *Web 2.0 Security and Privacy (W2SP)*, 2008.

[120] Collin Jackson and Adam 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.

[121] Ian Jacobs and Norman Walsh. Architecture of the World Wide Web, Volume One. Technical report, dec 2004.

[122] Karthick Jayaraman, Wenliang Du, Balamurugan Rajagopalan, and Steve J. Chapin. ESCUDO: A fine-grained protection model for web browsers. In *2010 International Conference on Distributed Computing Systems, ICDCS 2010, Genova, Italy, June 21-25, 2010*, pages 231–240. IEEE Computer Society, 2010.

[123] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[124] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *International Conference on the World Wide Web (WWW)*, WWW '07, pages 601–610, New York, NY, USA, 2007. ACM.

[125] Xing Jin, Xunchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. 2014.

[126] Martin Johns. Sessionsafe: implementing xss immune session handling. In *European Symposium on Research in Computer Security (ESORICS)*, ESORICS'06, pages 444–460, Berlin, Heidelberg, 2006. Springer-Verlag.

[127] Martin Johns. PreparedJS: Secure Script-Templates for JavaScript. In *10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '13)*, LNCS. Springer, July 2013.

[128] Martin Johns, Christian Beyerlein, Rosemaria Giesecke, and Joachim Posegga. Secure code generation for web applications. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, ESSoS'10, pages 96–113, Berlin, Heidelberg, 2010. Springer-Verlag.

[129] Martin Johns, Björn Engelmann, and Joachim Posegga. Xssds: Server-side detection of cross-site scripting attacks. In *Annual Computer Security Applications Conference (ACSAC)*, ACSAC '08, pages 335–344, Washington, DC, USA, 2008. IEEE Computer Society.

[130] Martin Johns, Sebastian Lekies, Bastian Braun, and Benjamin Flesch. Betterauth: Web authentication revisited. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 169–178. ACM, 2012.

[131] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, SP '06, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[132] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Workshop on Programming languages and analysis for security*, PLAS '06, pages 27–36, New York, NY, USA, 2006. ACM.

[133] jQuery. Update on jQuery.com Compromises. `http://blog.jquery.com/2014/09/24/update-on-jquery-com-compromises/`.

[134] JSLint,The JavaScript Code Quality Tool. `http://www.jslint.com/`.

[135] JSLint Error Explanations. Implied eval is evil. Pass a function instead of a string. `http://jslinterrors.com/implied-eval-is-evil-pass-a-function-instead-of-a-string`.

[136] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *International Conference on the World Wide Web (WWW)*, WWW '06, pages 247–256, New York, NY, USA, 2006. ACM.

[137] Samy Kamkar. I'll never get caught. i'm popular!, April 2005.

[138] Mathias Karlsson. Universal xss in opera. [online], `http://blog.detectify.com/post/32947196572/universal-xss-in-opera`, May 2012.

[139] Dawn Kawamoto. Worm lurks behind myspace profiles, July 2006.

[140] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM Conference on Computer and Communications Security (CCS)*, pages 272–280. ACM, 2003.

[141] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *International Conference on Software Engineering*, ICSE '09, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society.

[142] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *ACM Symposium On Applied Computing (SAC)*, SAC '06, pages 330–337, New York, NY, USA, 2006. ACM.

[143] Amit Klein. Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium, Articles*, 4, 2005.

[144] Olaf Kolkman. Dnssec howto, a tutorial in disguise. Technical report, jul 2009.

[145] Kris Zyp. Secure Mashups with dojox.secure. http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-dojoxsecure/.

[146] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2109 (Historic), February 1997. Obsoleted by RFC 2965.

[147] Mohit Kumar. Exploiting google persistent xss vulnerability for phishing, November 2011.

[148] Adam Langley. Further improving digital certificate security.

[149] Adam Langley. Maintaining digital certificate security.

[150] Adam Langley. Overclocking ssl. *Online at https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html*, 2010.

[151] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962 (Experimental), June 2013.

[152] Sebastian Lekies and Martin Johns. Lightweight integrity protection for web storage-driven content caching. *Web 2.0 Security and Privacy (W2SP)*, 2012.

[153] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later-large-scale detection of dom-based xss. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[154] Cédric Lévy-Bencheton, Louis Marinos, Rossella Mattioli, Thomas King, Christoph Dietzel, Stumpf Jan, et al. Threat landscape and good practice guide for internet infrastructure. January 2015.

[155] David Lindsay and Eduardo Vela. Universal xss via ie8s xss filters. [online], http://blackhat.com/html/bh-eu-10/bh-eu-10-briefings.html#Lindsay, April 2010.

[156] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 385–398. ACM, 2013.

[157] Benjamin Livshits and Weidong Cui. Spectator: detection and containment of javascript worms. In *USENIX Annual Technical Conference*, ATC'08, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.

[158] Benjamin Livshits and Ãšlfar Erlingsson. Using web application construction frameworks to protect against code injection attacks. 2007 workshop on programming languages and analysis for security. In *In Proc. Programming Languages and Analysis for Security*, pages 95–104, 2007.

[159] V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, 2005.

[160] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 371–388. USENIX Association, 2010.

[161] Mike Ter Louw, Phu H. Phung, Rohini Krishnamurti, and Venkat N. Venkatakrishnan. Safescript: Javascript transformation for policy enforcement. In Hanne Riis Nielson and Dieter Gollmann, editors, *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*, volume 8208 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2013.

[162] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers, 2009.

[163] Mike Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symposium on Security and Privacy*, SP '09, pages 331–346, Washington, DC, USA, 2009. IEEE Computer Society.

[164] Miroslav Lucinskij. Skype videomood XSS. Posting to the full disclosure mailinglist, `http://seclists.org/fulldisclosure/2008/Jan/0328.html`, January 2008.

[165] Tongbo Luo and Wenliang Du. Contego: Capability-based access control for web browsers - (short paper). In Jonathan M. McCune, Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse, and Yolanta Beres, editors, *Trust and Trustworthy Computing - 4th International Conference, TRUST 2011, Pittsburgh, PA, USA, June 22-24, 2011. Proceedings*, volume 6740 of *Lecture Notes in Computer Science*, pages 231–238. Springer, 2011.

[166] Matias Madou, Edward Lee, Jacob West, and Brian Chess. Watch what you write: Preventing cross-site scripting by observing program output. In *OWASP AppSec Europe*, 2008.

[167] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating javascript with filters, rewriting, and wrappers. In Michael Backes and Peng Ning, editors, *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, volume 5789 of *Lecture Notes in Computer Science*, pages 505–522. Springer, 2009.

[168] Sergio Maffeis and Ankur Taly. Language-Based Isolation of Untrusted JavaScript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 77–91. IEEE Computer Society, 2009.

[169] Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In Tuomas Aura, Kimmo Järvinen, and Kaisa Nyberg, editors, *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers*, volume 7127 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 2010.

[170] Jonas Magazinius, Phu H. Phung, and David Sands. Safe wrappers and sane policies for self protecting JavaScript. In Tuomas Aura, editor, *The 15th Nordic Conference in Secure IT Systems*, LNCS. Springer Verlag, October 2010. (Selected papers from AppSec 2010).

[171] Jonas Magazinius, Billy K. Rios, and Andrei Sabelfeld. Polyglots: crossing origins by crossing formats. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '13, pages 753–764, New York, NY, USA, 2013. ACM.

[172] Georgio Maone. Noscript.

[173] Giorgio Maone, David Lin-Shung Huang, Tobias Gondrom, and Brad Hill. User Interface Security Directives for Content Security Policy. W3C Working Draft, W3C, March 2014. Work in progress. `http://www.w3.org/TR/2014/WD-UISecurity-20140318/`.

[174] Moxie Marlinspike. Sslstrip. *Online at `http://www.thoughtcrime.org/software/sslstrip/`*, 2009.

[175] Michael Martin and Monica S. Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *USENIX Security Symposium*, SEC'08, pages 31–43, Berkeley, CA, USA, 2008. USENIX Association.

[176] Nate McFeters, Rob Carter, and John Heasman. Extreme Client-Side Exploitation. Talk at the Black Hat US conference, 2008.

[177] Leo A. Meyerovich, Adrienne Porter Felt, and Mark S. Miller. Object views: fine-grained sharing in browsers, 2010.

[178] Leo A. Meyerovich and V. Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 481–496. IEEE Computer Society, 2010.

[179] James Mickens and Matthew Finifter. Jigsaw: Efficient, low-effort mashup isolation. In E. Michael Maximilien, editor, *3rd USENIX Conference on Web Application Development, WebApps'12, Boston, MA, USA, June 13, 2012*, pages 13–25. USENIX Association, 2012.

[180] Microsoft. Microsoft Internet Security and Acceleration (ISA) Server 2004. `http://technet.microsoft.com/en-us/library/cc302436.aspx`.

[181] Microsoft. Microsoft Security Bulletin MS04-040 - Critical. `https://technet.microsoft.com/en-us/library/security/ms04-040.aspx`.

[182] Microsoft. Mitigating Cross-site Scripting With HTTP-only Cookies. `http://msdn.microsoft.com/en-us/library/ms533046(VS.85).aspx`.

[183] Microsoft Live Labs. Live Labs Websandbox. `http://websandbox.org`.

[184] Mihai Bazon. UglifyJS. `https://github.com/mishoo/UglifyJS/`.

[185] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.

[186] Mark Samuel Miller. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2006. AAI3245526.

[187] Yasuhiko Minamide. Static Approximation of Dynamically Generated Web Pages. In *International Conference on the World Wide Web (WWW)*, 2005.

[188] MITRE. Common vulnerabilities and exposures - the standard for information security vulnerability names.

[189] MITRE. CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition. `http://cwe.mitre.org/data/definitions/367.html`.

[190] Mozilla. JavaScript Strict Mode Reference. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode`.

[191] Mozilla. The Narcissus meta-circular JavaScript interpreter. `https://github.com/mozilla/narcissus`.

[192] Mozilla. The "with" statement. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with`.

[193] Mozilla and Kathleen Wilson. The mcs incident and its consequences for cnnic. *Mozilla Security Blog*.

[194] Raymond Mui and Phyllis Frankl. Preventing web application injections with complementary character coding. In *European Symposium on Research in Computer Security (ESORICS)*, ESORICS'11, pages 80–99, Berlin, Heidelberg, 2011. Springer-Verlag.

[195] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[196] T. Narten, R. Draves, and S. Krishnan. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 4941 (Draft Standard), September 2007.

[197] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Symposium on Network and Distributed System Security (NDSS)*, 2007.

[198] Netscape 2.0 reviewed. `http://www.antipope.org/charlie/old/journo/netscape.html`.

[199] Matthias Neugschwandtner, Martina Lindorfer, and Christian Platzer. A View to a Kill: WebView Exploitation. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2013.

[200] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. *Automatically hardening web applications using precise tainting*. Springer, 2005.

[201] 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 *ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, October 2012.

[202] Nick Nikiforakis. Firefox and self-xss. [online], `http://blog.securitee.org/?p=114`, January 2012.

[203] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote JavaScript inclusions. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 736–747. ACM, 2012.

[204] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. Sessionshield: lightweight protection against session hijacking. In *Engineering Secure Software and Systems*, pages 87–100. Springer, 2011.

[205] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, LNCS. Springer, February 2011.

[206] M. Nottingham and M. Thomson. Opportunistic Security for HTTP. Internet-Draft draft-ietf-httpbis-http2-encryption-01, Internet Engineering Task Force, December 2014. Work in progress.

[207] Mark Nottingham. Opportunistic encryption for HTTP URIs. 2013.

[208] Mark Nottingham. Securing the web. Technical report, jan 2015.

[209] Terri Oda, Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. Soma: mutual approval for included content in web pages. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '08, pages 89–98, New York, NY, USA, 2008. ACM.

[210] OWASP. Cross-site scripting (xss), September 2013.

[211] Stefano Di Paola. 23rd ccc conference: Subverting ajax, December 2006.

[212] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards fine-grained access control in javascript contexts. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, pages 720–729. IEEE Computer Society, 2011.

[213] Riccardo Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *ASIA Computer and Communications Security (ASIACCS)*, May 2012.

[214] A. Petersson and M. Nilsson. Forwarded HTTP Extension. RFC 7239 (Proposed Standard), June 2014.

[215] Phu H. Phung and Lieven Desmet. A two-tier sandbox architecture for untrusted JavaScript. In *JSTools '12 Proceedings of the Workshop on JavaScript Tools, Beijing,13 June, 2012*, pages 1–10, 2012.

[216] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[217] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting javascript. In *ACM Conference on Computer and Communications Security (CCS)*, CSS, pages 47–60, New York, NY, USA, 2009. ACM.

[218] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, RAID'05, pages 124–145, Berlin, Heidelberg, 2006. Springer-Verlag.

[219] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: type-based verification of javascript sandboxing. In *USENIX Security Symposium*, SEC'11, Berkeley, CA, USA, 2011. USENIX Association.

[220] Andrei Popescu. Geolocation API Specification. *W3C Proposed Recommendation*, 2012.

[221] Rapid7. Xss due to unescaped hostnames. [online], `http://www.rapid7.com/db/vulnerabilities/http-apache-host-xss`, October 2002.

[222] R. Raszuk, J. Heitz, A. Lo, L. Zhang, and X. Xu. Simple Virtual Aggregation (S-VA). RFC 6769 (Informational), October 2012.

[223] Donald Ray and Jay Ligatti. Defining Code-injection Attacks. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2012.

[224] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *OSDI '06: Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[225] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-tls13-05, Internet Engineering Task Force, March 2015. Work in progress.

[226] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2011.

[227] William Robertson and Giovanni Vigna. Static enforcement of web application integrity through strong typing. In *USENIX Security Symposium*, SEC'09, pages 283–298, Berkeley, CA, USA, 2009. USENIX Association.

[228] Thomas Roessler. When Widgets Go Bad. Lightning talk at the 24C3 conference, `http://log.does-not-exist.org/archives/2007/12/28/2160_when_widgets_go_bad.html`, December 2007.

[229] Thomas Roessler and Anil Saldhana. Web Security Context: User Interface Guidelines. W3C recommendation, W3C, August 2010. http://www.w3.org/TR/2010/REC-wsc-ui-20100812/.

[230] D. Ross and T. Gondrom. HTTP Header Field X-Frame-Options. RFC 7034 (Informational), October 2013.

[231] David Ross. Ie8 security part iv: The xss filter, July 2008.

[232] David Ross. Happy 10th birthday cross-site scripting!, December 2009.

[233] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1), 2006.

[234] Mike Samuel, Prateek Saxena, and Dawn 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.

[235] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '11, pages 587–600, New York, NY, USA, 2011. ACM.

[236] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[237] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Symposium on Network and Distributed System Security (NDSS)*. The Internet Society, 2010.

[238] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '11, pages 601–614, New York, NY, USA, 2011.

[239] Bruce Schneier. Comodo group issues bogus ssl certificates.

[240] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & sons, 2007.

[241] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[242] WhiteHat Security. Website security statistics report, May 2013.

[243] R. Sekar. An efficient black-box technique for defeating web application attacks ? In *Symposium on Network and Distributed System Security (NDSS)*, 2009.

[244] Y. Sheffer, R. Holz, and P. Saint-Andre. Recommendations for Secure Use of TLS and DTLS. Internet-Draft draft-ietf-uta-tls-bcp-11, Internet Engineering Task Force, February 2015. Work in progress.

[245] Ofer Shezaf. The universal xss pdf vulnerability. [online], https://www.owasp.org/images/4/4b/OWASP_IL_The_Universal_XSS_PDF_Vulnerability.pdf, June 2007.

[246] Ryan Sleevi and Mark Watson. Web Cryptography API. W3C Working Draft, W3C, December 2014. W3C Candidate Recommendation.

[247] Squid Project Maintainers. squid: Optimising Web Delivery. *Online at http://www.squid-cache.org/*, 2014.

[248] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *International Conference on the World Wide Web (WWW)*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

[249] Brandon Sterne and Adam Barth. Content Security Policy 1.0. W3C Candidate Recommendation, W3C, November 2012. http://www.w3.org/TR/2012/CR-CSP-20121115/.

[250] Brandon Sterne and Adam Barth. Content Security Policy 1.0. W3C Candidate Recommendation, W3C, November 2012. http://www.w3.org/TR/2012/CR-CSP-20121115/.

[251] Ben Stock and Martin Johns. Protecting users against xss-based password manager abuse. In *ACM Conference on Computer and Communications Security (CCS)*, pages 183–194. ACM, 2014.

[252] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise client-side protection against dom-based cross-site scripting. In *USENIX Security Symposium*, 2014.

[253] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. *SIGPLAN Not.*, 41(1):372–382, January 2006.

[254] Fangqi Sun, Liang Xu, and Zhendong Su. Client-side detection of xss worms by monitoring payload propagation. In *European Symposium on Research in Computer Security (ESORICS)*, ESORICS'09, pages 539–554, Berlin, Heidelberg, 2009. Springer-Verlag.

[255] Symantec. Js.yamanner@m, June 2006.

[256] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*, 2013.

[257] Darryl Taft. 75% of Developers Using HTML5. [online], http://www.eweek.com/c/a/Application-Development/75-of-Developers-Using-HTML5-Survey-508096/, 2012.

[258] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated analysis of security-critical javascript apis. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 363–378. IEEE Computer Society, 2011.

[259] Shuo Tang, Nathan Dautenhahn, and Samuel T. King. Fortifying web-based applications automatically. In *ACM Conference on Computer and Communications Security (CCS)*, CCS '11, 2011.

[260] TechNoesis. 4 ways to prevent duplicate form submission. *Online at http://technoesis.net/prevent-double-form-submission/*, 2013.

[261] The Apache Software Foundation. Apache Traffic Server. *Online at http://trafficserver.apache.org/*, 2014.

[262] The FaceBook Team. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

[263] The Guardian. Edward Snowden. *Online at http://www.theguardian.com/world/edward-snowden*, 2013.

[264] Twitter. How to embed Twitter timelines on your website. https://blog.twitter.com/2012/embedded-timelines-howto.

[265] Steven Van Acker. *Isolating and Restricting Client-Side JavaScript*. PhD thesis, January 2015.

[266] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Frank Piessens, and Wouter Joosen. Monkey-in-the-browser: Malware and vulnerabilities in augmented browsing script markets. In *USENIX Security Symposium*, pages 525–530. ACM, 2014.

[267] W3C. Same Origin Policy - Web Security. http://www.w3.org/Security/wiki/Same_Origin_Policy.

[268] W3C. W3C Standards and drafts - Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.

[269] W3C. Web Workers. http://dev.w3.org/html5/workers/.

[270] W3C. XML Path Language (XPath) 2.0. http://www.w3.org/TR/xpath20/.

[271] W3C. Content Security Policy 1.0. W3C Candidate Recommendation, http://www.w3.org/TR/2011/WD-CSP-20111129/, November 2012.

[272] W3C. Content Security Policy 1.1. W3C Editor's Draft 13, https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html, November 2013.

[273] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

[274] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering*, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[275] Joel Weinberger, Adam Barth, and Dawn Song. Towards client-side html security policies. In *USENIX Workshop on Hot Topics in Security*, HotSec'11, Berkeley, CA, USA, 2011. USENIX Association.

[276] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A systematic analysis of xss sanitization in web application frameworks. In *European Symposium on Research in Computer Security (ESORICS)*, ESORICS'11, pages 150–171, Berlin, Heidelberg, 2011. Springer-Verlag.

[277] Rigo Wenning. Trust & usability on the web, a social/legal perspective.

[278] Mike West. Mixed Content. *W3C Working Draft*, sep 2014.

[279] Mike West. Requirements for Powerful Features. Technical report, dec 2014.

[280] Mike West, Adam Barth, and Dan Veditz. Content Security Policy Level 2. W3C Working Draft, W3C, July 2014. Work in progress. http://www.w3.org/TR/2014/WD-CSP2-20140703/.

[281] Mike West, Adam Barth, and Dan Veditz. Content Security Policy Level 2. W3C Working Draft, W3C, February 2015. Work in progress. http://www.w3.org/TR/2014/WD-CSP2-20140703/.

[282] Mike West and Yan Zhu. Privileged Contexts. Technical report, apr 2015.

[283] WHATWG. HTML Living Standard - Timers. `https://html.spec.whatwg.org/multipage/webappapis.html#timers`.

[284] Zack Whittaker. Ubuntu forums hacked; 1.82m logins, email addresses stolen, July 2013.

[285] Jeff Williams and Dave Wichers. Owasp top 10. *OWASP Foundation*, 2013.

[286] Mike Wood. Fraudulent certificates issued by comodo, is it time to rethink who we trust?

[287] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. Swap: Mitigating xss attacks using a reverse proxy. In *ICSE Workshop on Software Engineering for Secure Systems*, IWSESS '09, pages 33–39, Washington, DC, USA, 2009. IEEE Computer Society.

[288] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium*, volume 15, pages 179–192, 2006.

[289] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, pages 121–136, 2006.

[290] Phillip Porras Yinzhi Cao, Vinod Yegneswaran and Yan Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[291] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 237–249. ACM, 2007.

[292] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *International Conference on the World Wide Web (WWW)*, WWW '09, pages 961–970, New York, NY, USA, 2009. ACM.

[293] Michal Zalewski. Postcards from the post-XSS world. [online], `http://lcamtuf.coredump.cx/postxss/`, December 2011.

[294] Thiago Zaninotti and Amit Klein. Apache HTTPd "Expect" Header Handling Client-Side Cross Site Scripting Vulnerability (CVE-2006-3918). [online]. `http://www.frsirt.com/english/advisories/2006/2963`, (05/05/07), July 2006.

[295] William Zeller and Edward W. Felten. Cross-site request forgeries: Exploitation and prevention. Technical report, Princeton University, 2008.

[296] Yuqing Zhang, Xiali Wang, Qihan Luo, and Qixu Liu. Cross-site scripting attacks in social network apis. In *Workshop on WEB 2.0 Security and Privacy (W2SP)*, 2013.