Server-driven Outbound Web-application Sandboxing
FP7-ICT-2009-5, Project No. 256964

https://www.websand.eu

Deliverable D4.3
Secure Composition Policies and Server-driven Enforcement

Abstract

This deliverable reports on the end-to-end enforcement of secure composition policies in WebSand. The report investigates the problem of insecure script inclusions, and describes and discusses the developed techniques to achieve server-driven client-side enforcement.

Deliverable details

Deliverable version: *v1.0* Classification: *public*
Date of delivery: *30.06.2013* Due on: *M33*
Editors: *Lieven Desmet and Frank Piessens* Total pages: *158*

List of Contributors:
Pieter Agten, Yoran Brondsema, Ping Chen, Lieven Desmet, Martin Johns, Wouter Joosen, Nils Kunze, Jonas Magazinius, Martín Ochoa, Jan Tobias Mühlberger, Nick Nikiforakis, Pieter Philippaerts, Frank Piessens, Phu H. Phung, Andrei Sabelfeld, David Sands, Steven Van Acker

Project details

Start date: *October 01, 2010* Duration: *36 months*
Project Coordinator: *Martin Johns*
Partners: SAP, Siemens, CHALMERS, KUL, UNI PASSAU

Document revision history

	Responsible	Date
Integration of section 2	Lieven Desmet	June 4, 2013
Integration of section 3	Lieven Desmet	June 6, 2013
Integration of section 4	Lieven Desmet	June 10, 2013
Integration of section 6	Lieven Desmet	June 18, 2013
Integration of section 5	Martin Johns	June 21, 2013
Initial revision	Lieven Desmet and Frank Piessens	June 23, 2013
Review	Sebastian Lekies	June 28, 2013
Final version	Lieven Desmet and Frank Piessens	June 28, 2013

Executive Summary

Modern web applications strongly rely on JavaScript to deliver highly responsive user interfaces, and to enrich the website with additional functionality. To do so, a web developer can integrate in-house developed JavaScript libraries, or can simply rely on relevant third-party JavaScript libraries.

In this deliverable, we will explore the risks involved in integrating third-party JavaScript, and discuss the server-driven enforcement mechanisms developed in work package 4 of the WebSand project to securely integrate untrusted third-party JavaScript code as part of a website.

We start with a brief summary of the web security model and the impact of this security model on integrating third-party JavaScript in a website. In addition, we assess the use of script inclusion on the top 10,000 most popular websites to illustrate the widespreadness of this type of code assembly on the web, and the urge to securely compose JavaScript coming from potentially untrusted third-party providers.

Based on the security-sensitive operations available in the JavaScript context, we propose the least-privilege secure composition policies. Moreover, this deliverable also investigates how the richness of JavaScript APIs not only affects the security but also the privacy of end-users in web fingerprinting frameworks.

We propose and discuss a variety of enforcement techniques, as they have been developed within WebSand. These enforcement techniques range from a security-enhanced browser (WebJail) to JavaScript security architectures that runs on top of mainstream browsers (Two-tier sandbox, JSand and PreparedJS). The implementation strategies mainly vary in their mode of deployment, but also make different trade-offs in terms of legacy support, precision, efficiency and maintainability.

We selected JSand as most promising technique for the server-driven enforcement. To this extent, we have further matured the JSand prototype implementation to support a representative selection of DOM operations, and applied it to the most frequently used third-party scripts.

The server-driven enforcement is successfully realized and works as follows. While processing a web request at the server-side, the client-side code to set up the security architecture and the secure composition policy are pushed towards the browser as part of the web page. Before loading the third-party JavaScript code in the browser environment, the JavaScript security architecture is set up, and a script-specific sandbox environment is created based on the provided secure composition policy. Once fully configured, the third-party JavaScript code is loaded in the sandbox environment and securely executed.

Contents

1 Introduction

Modern web applications intensively use JavaScript to create highly responsive user interfaces, and to enrich the website with additional functionality (ranging from integration with social media sites, to context-sensitive advertisements and tools for website analytics). To do so, a web developer can integrate in-house developed JavaScript libraries, or can simply rely on relevant third-party JavaScript libraries.

In this deliverable, we will explore the risks involved in integrating third-party JavaScript, and discuss the server-driven enforcement mechanisms developed in work package 4 of the WebSand project to securely integrate untrusted third-party JavaScript code as part of your website. This section provides the overview of the research conducted in work package 4, whereas Sections 2 to 6 provide more detailed descriptions in the form of published research papers.

In Subsection 1.1, the web security model is briefly discussed, as well as the impact of this security model on integrating third-party JavaScript in a website. In addition, an assessment of script inclusions on the top 10,000 most popular websites underpins the widespreadness of this type of code assembly on the web, and the urge to securely compose JavaScript coming from potentially untrusted third-party providers.

Subsection 1.2 briefly summarizes the set of security-sensitive operations, as they are available in the JavaScript context. Next, the security-sensitive operations are bundled in logical categories, in order to enable the specification of least-privilege secure composition policies. Finally, this deliverable also investigates how the richness of JavaScript APIs not only affects the security but also the privacy of end-users in web fingerprinting frameworks.

In Subsection 1.3, a variety of enforcement techniques is proposed and discussed, as they have been developed within WebSand. These enforcement techniques range from a security-enhanced browser to a JavaScript security architecture that runs on top of mainstream browsers.

Finally, Subsection 1.4 discusses how the server-driven enforcement of least-privilege composition is achieved within WebSand.

1.1 Web security model

The security model in which third-party JavaScript code executes is based on the de facto security policy of the Web: the Same Origin Policy (SOP). The SOP states that scripts from one origin should not be able to access content from other origins. This prevents scripts from stealing data, cookies or login credentials from other sites.

1.1.1 Integration of scripts

The two most-widespread techniques to integrate third party JavaScript into a web application are through *script inclusion* or via *iframe integration*. Loading components from different origins in iframes causes them to be separated by the SOP. Using script inclusion causes the script to be loaded in the protection domain of the including page, which is a straightforward way to achieve interaction between components. Communication with the origin of the page containing the script can be achieved using the XMLHttpRequest functionality of the DOM.

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

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

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

1.1.2 JavaScript inclusions: assessing the state of practice

The majority of third-party JavaScript APIs are integrated via script inclusion, giving the external script provider full control over the client-side part of their website. This can be the case if the script provider has malicious intentions, but can as well happen if the script provider gets compromised over time.

To better understand this relationship between script provider and website owner, we have examined if and how the 10,000 most important sites integrate third-party JavaScript code, as reported in more detail in Section 2.

We examined 3,300,000 pages of the top 10,000 websites (according to Alexa), and extracted 8,439,799 remote script inclusions. 88.45% of the 10,000 web sites included at least one remote JavaScript library, and there are even sites in the top Alexa list that trust up to 295 remote hosts.

To assess the quality/security of web sites as well as script providers, we defined the Quality of Maintenance (QoM) metric and performed an empirical evaluation of various domains. As expected, we discovered that low-maintenance domains often include JavaScript libraries from low-maintenance providers and high-maintenance domains, instead, tend to prefer high-maintenance providers, showing that they are indeed concerned about the providers they include.

However, we also found that, for sites with high-maintenance scores, one out of four of their inclusions comes from providers with a low maintenance score, which are potential "weak spots" in their security perimeter.

1.2 Secure composition policies

As a first step to better understand the impact of running arbitrary JavaScript code, the upcoming HTML5 specification and accompanying APIs have been studied, and a list of 86 security-sensitive operations available in the JavaScript execution context has been enumerated. This includes, among others, access to sensitive information in the DOM, the history and cookies, sensitive device information, inter-frame communication, cross-domain communication, and various features in media, UI, and rendering. This list of security-sensitive operations is a key ingredient in defining privileges granted to third-party JavaScript code as part of the secure composition policies.

As reported in deliverable D4.1, these security-sensitive operations have been bundled into nine disjoint categories, based on their functionality (as illustrated in Table 1). The least-privilege composition policy for a third-party script expresses for each of the categories whether or not the specific privilege is allowed or not, and if allowed, the composition policy can further restrain by specifying a whitelist. For instance, for external communication, the policy can allow remote communication such as XHR, but restrict it to a set of trusted domains specified in the *DestinationDomainSet*.

Note that with the advent of HTML5, a whole range of new JavaScript APIs has been defined, drastically increasing the functionality that can be accessed by JavaScript code. As a consequence, if an attacker can get a handle on JavaScript code included in your website, the attacker is able to control all this functionality.

Categories and APIs (# op.)	Whitelist
DOM Access	ElemReadSet, ElemWriteSet
DOM Core (17)	
Cookies	KeyReadSet, KeyWriteSet
cookies (2)	
External Communication	DestinationDomainSet
XHR, CORS, UMP (4)	
WebSockets (5)	
Server-sent events (2)	
Inter-frame Communication	DestinationDomainSet
Web Messaging (3)	
Client-side Storage	KeyReadSet, KeyWriteSet
Web Storage (5)	
IndexedDB (16)	
File API (4)	
File API: Dir. and Syst. (11)	
File API: Writer (3)	
UI and Rendering	
History API (4)	
Drag/Drop events (3)	
Media	
Media Capture API (3)	
Geolocation	
Geolocation API (2)	
Device Access	SensorReadSet
System Information API (2)	
Total number of security-sensitive operations: 86	

Table 1: Overview of the sensitive JavaScript operations from the HTML 5 APIs, divided in categories.

In our least-privilege composition policy, we mainly focused on the security of the web application. In addition, we have also investigated what the risks are with respect to the privacy of the end user (Section 6).

In particular, we have studied the problem of web-based fingerprinting to get a good understanding of how real-life web fingerprinting providers work, and to assess to what extent this can be controlled or limited by enforcing composition policies on third-party code. As part of this ongoing research, we can already conclude that although we might be able to block some of the functionality used by web fingerprinting libraries, we will probably never be able to hide all the complexity and sophistication browsers possess (and what typically is abused by web fingerprinting libraries) with a server-driven solution.

1.3 Enforcing secure composition policies

Our implementation strategy has been to pursue multiple tracks to enforce least-privilege integration of JavaScript code (as proposed in Deliverable 4.1), ranging from an enhanced browser environment to a JavaScript security architecture deployed in legacy browsers. In this subsection, we report on the various implementations for secure composition of 3rd party JavaScript components. They mainly vary in their mode of deployment, but also make different trade-offs in terms of legacy support, precision, efficiency and maintainability.

1.3.1 WebJail

The first prototype, WebJail, is a client-side security architecture that enforces the least-privilege composition policy on iframe integrated code via a modified browser. In this prototype, the secure composition policy is provided by the server-side, but the full enforcement is realized in an enhanced client-side environment.

Via deep aspect technology in the browser (as proposed by the authors of ConScript), every access to security-sensitive operations is mediated in the browser core, and can be configured via policy-driven advices. The composition policy is specified by the integrator and pushed as an attribute of the iframe tag to the client. On receipt of the composition policy, the enhanced browser triggers the loading of the security architecture and enforces the provided composition policy.

More details on WebJail can be found in Deliverable 4.1 and Deliverable 4.2.

1.3.2 Two-tier sandbox

The second prototype is a client-side security architecture, developed in JavaScript, that enforces modular and fine-grained security policies for untrusted JavaScript code. The two-tier sandbox architecture builds on top of the Secure ECMAScript (SES) library.

The two-tier sandbox architecture combines a coarse-grained outer sandbox with a fine-grained inner sandbox. The architecture allows to enforce application-specific and stateful fine-grained policies without browser modification or pre-processing of the code (e.g. in line with the Self-Protecting JavaScript mechanism), while the baseline API in the outer sandbox ensures a failsafe fallback in case of badly written policies.

More details on the two-tier sandbox architecture can be found in Section 3.

1.3.3 JSand

The third prototype is a client-side security architecture, developed in JavaScript, that enforces the least-privilege composition policy without (major) browser modifications. This implementation strategy is inspired by earlier results on the two-tier sandbox architecture, but matures the sandboxing technology as well as enables the enforcement of the least-privilege composition policy.

The JavaScript security architecture builds on top of the Secure ECMA-Script (SES) library, and provides a sandbox environment for third party scripts. The SES sandbox isolates the JavaScript execution, and limits its capabilities according to the object-capability model. To enforce the secure composition policies, a baseline API has been developed to provide the necessary capabilities towards the sandboxed script, while mediating access to security-sensitive APIs via wrappers and the Proxy API. In this prototype, both the secure composition policy as well as the security architecture are provided by the server-side, and execute in legacy browser environments.

More details on JSand can be found in Section 4.

1.3.4 PreparedJS

The fourth prototype, PreparedJS, builds upon the Content Security Policy (CSP) to securely compose third-party scripts. PreparedJS facilitates the use of CSP in web applications by offering a templating format for JavaScript (in line with SQL's prepared statement model), and strongly binds scripts and policies by applying a light-weight script checksumming scheme.

In combination with the base-line protection provided by CSP, PreparedJS is able to prevent the full spectrum of potential XSS attacks. PreparedJS can be realized as a native browser component while providing backwards compatibility with legacy browsers that cannot handle PreparedJS's script format.

More details on PreparedJS can be found in Section 5.

1.4 Server-driven policy enforcement

In this subsection, we briefly discuss the selection of the WebSand WP4 policy enforcement implementation, describe the server-driven enforcement architecture and illustrate in more detail how this is realized with JSand.

1.4.1 Selection of the policy enforcement implementation

The four secure composition prototypes explore different implementation strategies to enforce the secure composition of third-party code. All implementation strategies have reported successful isolation of third-party JavaScript. They mainly vary in their mode of deployment, but also make different trade-offs in terms of legacy support, precision, efficiency and maintainability.

WebJail focuses on the secure integration of third-party iframes and realizes this enforcement by mediating access to security-sensitive operations in the browser core. The other three prototypes focus on the secure integration of scripts, and test the feasibility of realizing this enforcement without (major) client-side modifications.

The two-tier sandbox architecture allows to apply application-specific, stateful fine-grained policies. By combining the fine-grained enforcement mechanism with a more coarse-grained outer sandbox, the technique ensure a baseline protection in case the policy writer mistakenly introduces vulnerabilities while expressing the fine-grained policies.

The JSand approach is more focused towards the enforcement of the least-privilege composition policy, which tends to be a good balance between the fine-grained policies of the two-tier sandbox approach and the very coarse-grained policies of the Same Origin Policy and the Content Security Policy.

PreparedJS enriches the Content Security Policy model, both in terms of usability for the web developer, and security towards white-listing the intended third-party scripts. This latter technique provides an additional layer of protection on top of the mechanisms specified above, to protect websites against trusted scripts and script providers that get compromised over time.

Based on the intermediate results of the various prototypes, we have selected the JSand prototype as basis in work package 4 for the server-driven enforcement of secure composition policies. We have further matured the prototype implementation to support a representative selection of DOM operations, and applied it to the most frequently used third-party scripts.

1.4.2 Server-drive enforcement architecture

The server-driven enforcement works as follows. First a set of secure composition policies is expressed by the website owner or the security officer in charge. The least-privilege composition policy, as discussed in Deliverable D4.1, expresses for each of the nine categories whether or not scripts should have access to these security-relevant operations. Next, a sandbox environment is configured as part of the web application, by selecting the appropriate secure composition policy and the code that needs to be executed as part of the sandbox. Both of these steps are part of the development or deployment of the web application.

During execution, the client-side code to set up the security architecture and the secure composition policy are pushed towards the browser as part of the web page. Before loading the third-party JavaScript code, the JavaScript security architecture is set up, and a script-specific sandbox environment is created based on the provided secure composition policy. Once fully configured, the third-party JavaScript code is loaded in the sandbox environment and executed.

1.4.3 Server-driven enforcement with JSand

Listing 1 illustrates how Google Maps can be integrated into a webpage with JSand. In line 4, the JSand framework gets initialized on the webpage with the JSP tag `jsand:initialize`, and a new sandbox environment is loaded by means of the `jsand:sandbox` tag.

The value of the policy attribute of the latter one (ie. *googlemapsNoGeolocation*) refers to one of the secure composition policies that have been made preconfigured for this application context by the security officer (expressed in listing 2). Such a policy consists of the least-privilege composition policy, and optionally one or more JavaScript APIs that are preloaded as part of the sandbox environment. For instance, in case of the exemplary *googlemapsNoGeolocation* policy of listing 1, the Google Maps API is preloaded as part of the sandboxed context.

In addition to preloaded APIs, the developer can add additional JavaScript code blocks and references to JavaScript files, similar to a non-JSand environment. In this example, a code block is added with the `jsand:code` tag to set some page-specific variables (lines 6 to 10), and an external JavaScript file is loaded on line 11 with the `jsand:script` tag.

The server-side implementation of JSand in JEE automatically transforms this input to the output shown in listing 3.

The `JSandLib.js` JavaScript library on line 3 of listing 3 encapsulates the core logic of JSand. This reusable library enables the creation of JSand sandboxes: the loading of the SES environment, and the configuration of the DOM wrappers and proxies based on the application-specific composition policy. This functionality is applied in lines 4-26 of listing 3 by calling the `sb.Sandbox` constructor.

1.5 Overview of this deliverable

In the remainder of this deliverable, the following research papers report and discuss in more detail the WebSand results achieved in this work package:

```
<%@taglib uri="/WEB-INF/tlds/jsand" prefix="jsand"%>
<html>
  <head>
    <jsand:initialize/>
    <jsand:sandbox policy="googlemapsNoGeolocation">
      <jsand:code>
        canvasID = "map_canvas";
        failcity = "New York";
        failpos = new google.maps.LatLng(40.69, -73.95);
      </jsand:code>
      <jsand:script src="googlemaps-geolocation.js"/>
    </jsand:sandbox>
  </head>
  <body>
    <div style="width: 300px; height: 300px;"
        id="map_canvas"></div>
  </body>
</html>
```

Listing 1: Integrating GoogleMaps with JSand

```
{ "domaccess-read":"yes",
  "domaccess-write":"yes",
  "cookies-read":"yes",
  "cookies-write":"yes",
  "extcomm":"yes",
  "framecomm":"yes",
  "storage-read":"yes",
  "storage-write":"yes",
  "ui":"yes",
  "media":"yes",
  "device":"yes" }
```

Listing 2: The exemplary *googlemapsNoGeolocation* secure composition policy

```
<html>
  <head>
    <script src="/JSand-JEE/JSandLib.js"></script>
    <script>
     window.addEventListener("load",function() {
       var sandbox = new sb.Sandbox(
         { "domaccess-read":"yes",
           "domaccess-write":"yes",
           "cookies-read":"yes",
           "cookies-write":"yes",
           "extcomm":"yes",
           "framecomm":"yes",
           "storage-read":"yes",
           "storage-write":"yes",
           "ui":"yes",
           "media":"yes",
           "device":"yes"});
       sandbox.load("maps3-12-10-fixed.js", false,
         function(){
           sandbox.eval("canvasID = \"map_canvas\"; failcity =
              \"New York\"; failpos = new
              google.maps.LatLng(40.69, -73.95); ",
           function(){
             sandbox.load("googlemaps-geolocation.js", false,
                function(){});
           });
         });
       });
</script>
  </head>
  <body>
    <div style="width: 300px; height: 300px;"
        id="map_canvas"></div>
  </body>
</html>
```

Listing 3: Server output of integrating GoogleMaps with JSand

- Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, Giovanni Vigna, **You are what you include: Large-scale evaluation of remote JavaScript inclusions**, Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012), pages 736-747, Raleigh, NC, USA, 16-18 October 2012 (Section 2)

- Phu H. Phung, Lieven Desmet, **A two-tier sandbox architecture for untrusted JavaScript**, Proceedings of the Workshop on JavaScript Tools (JSTools '12), pages 1-10, Beijing, China, 13 June 2012 (Section 3)

- Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, Frank Piessens, **JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications**, Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC 2012), pages 1-10, Orlando, Florida, USA, 3-7 December 2012 (Section 4)

- Martin Johns, **PreparedJS: Secure Script-Templates for JavaScript**, in 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '13), July 2013 (Section 5)

- Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, Giovanni Vigna, **Cookieless monster: Exploring the ecosystem of web-based device fingerprinting**, IEEE Security and Privacy, San Francisco, 19-22 May 2013 (Section 6)

Section 7 summarizes the contributions of this deliverable.

2 You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions[1][2]

2.1 Introduction

The web has evolved from static web pages to web applications that dynamically render interactive content tailored to their users. The vast majority of these web applications, such as Facebook and Reddit, also rely on client-side languages to deliver this interactivity. JavaScript has emerged as the de facto standard client-side language, and it is supported by every modern browser.

Modern web applications use JavaScript to extend functionality and enrich user experience. These improvements include tracking statistics (e.g., Google Analytics), interface enhancements (e.g., jQuery), and social integration (e.g., Facebook Connect). Developers can include these external libraries in their web applications in two ways: either (1) by downloading a copy of the library from a third-party vendor and uploading it to their own web server, or (2) by instructing the users' browsers to fetch the code directly from a server operated by a third party (usually the vendor). The safest choice is the former, because the developer has complete control over the code that is served to the users' browsers and can inspect it to verify its proper functionality. However, this choice comes with a higher maintenance cost, as the library must be updated manually. Another downside is that by not including remote code from popular Content Distribution Networks, the developer forces the users' browsers to download scripts from his own servers even if they are identical with scripts that are already available in the browsers' cache. Moreover, this method is ineffective when the library loads additional, remotely-hosted, code at run time (e.g., like Google Analytics does). A developer might avoid these drawbacks by choosing the second option, but this comes at the cost of trusting the provider of the code. In particular, the provider has complete control over the content that is served to the user of the web application. For example, a malicious or compromised provider might deface the site or steal the user's credentials through

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

[2]For KU Leuven, this research was done with the financial support from the Prevention against Crime Programme of the European Union, the IBBT, the Research Fund KU Leuven, and the EU-funded FP7 projects NESSoS and WebSand. For UCSB, this work was supported by the Office of Naval Research (ONR) under Grant N000140911042, the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537.

DOM manipulation or by accessing the application's cookies. This makes the provider of the library an interesting target for cyber-criminals: after compromising the provider, attackers can exploit the trust that the web application is granting to the provider's code to obtain some control over the web application, which might be harder to attack directly. For example, on the 8th of December 2011 the domain distributing qTip2, a popular jQuery plugin, was compromised [2] through a WordPress vulnerability. The qTip2 library was modified, and the malicious version was distributed for 33 days.

It is generally known that developers should include JavaScript only from trustworthy vendors, though it is frightening to imagine the damage attackers could do when compromising a JavaScript vendor such as Google or Facebook. However, there has been no large-scale, in-depth study of how well the most popular web applications implement this policy. In this paper, we study this problem for the 10,000 most popular web sites and web applications (according to Alexa), outlining the trust relationships between these domains and their JavaScript code providers. We assess the maintenance-quality of each provider, i.e., how easy it would be for a determined attacker to compromise the trusted remote host due to its poor security-related maintenance, and we identify weak links that might be targeted to compromise these top domains. We also identify new types of vulnerabilities. The most notable is called "Typosquatting Cross-site Scripting" (TXSS), which occurs when a developer mistypes the address of a library inclusion, allowing an attacker to register the mistyped domain and easily compromise the script-including site. We found several popular domains that are vulnerable to this attack. To demonstrate the impact of this attack, we registered some domain names on which popular sites incorrectly bestowed trust, and recorded the number of users that were exposed to this attack.

The main contributions of this paper are the following:

- We present a detailed analysis of the trust relationships of the top 10,000 Internet domains and their remote JavaScript code providers
- We evaluate the security perimeter of top Internet domains that include code from third-party providers.
- We identify four new attack vectors to which several high traffic web sites are currently vulnerable.
- We study how the top domains have changed their inclusions over the last decade.

The rest of this paper is structured as follows. Section 2.2 presents the setup and results of our large-scale crawling experiment for the discovery of remote JavaScript inclusions. Section 2.3 presents the evolution of JavaScript inclusions of popular web sites and our metric for assessing the quality of maintenance of a given JavaScript provider. In Section 2.4 we introduce four new types of vulnerabilities discovered during our crawl. Section 2.5 reviews some techniques that web applications can utilize to protect themselves against malicious third-party JavaScript libraries. Section 2.6 explores the related work and Section 2.7 concludes.

2.2 Data Collection

In this section, we describe the setup and results of our large-scale crawling experiment of the Alexa top 10,000 web sites.

2.2.1 Discovering remote JavaScript inclusions

We performed a large web crawl in order to gather a large data set of web sites and the remote scripts that they include. Starting with Alexa's list of the top 10,000 Internet web sites [5], we requested and analyzed up to 500 pages from each site. Each set of pages was obtained by querying the Bing search engine for popular pages within each domain. For instance, the search for "site:google.com" will return pages hosted on Google's main domain as well as subdomains. In total, our crawler visited over 3,300,000 pages of top web sites in search for remote JavaScript inclusions. The set of visited pages was smaller than five million since a portion of sites had less than 500 different crawlable pages.

From our preliminary experiments, we realized that simply requesting each page with a simple command-line tool that performs an HTTP request was not sufficient, since in-line JavaScript code can be used to create new, possibly remote, script inclusions. For example, in the following piece of code, the inline JavaScript will create, upon execution, a new remote script inclusion for the popular Google-Analytics JavaScript file:

```
var ishttps = "https:" == document.location.protocol;
var gaJsHost = (ishttps)?
     "https://ssl." : "http://www.");
var rscript = "";
rscript += "\%3Cscript src='" + gaJsHost;
rscript += "google-analytics.com/ga.js' type=";
rscript += "'text/javascript'\%3E\%3C/script\%3E";

document.write(unescape(rscript));
```

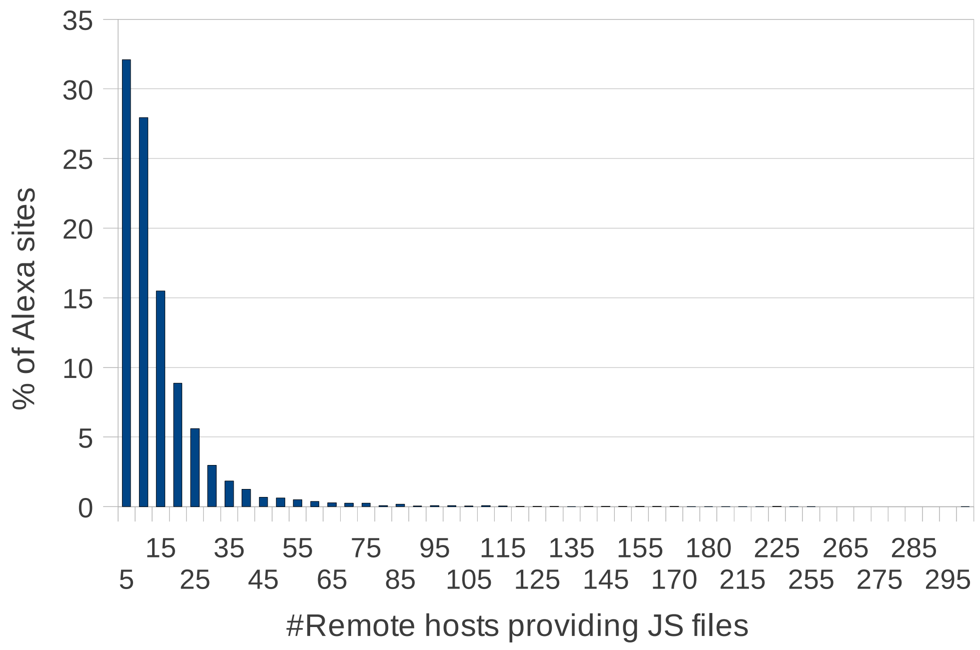

Figure 1: Relative frequency distribution of the percentage of top Alexa sites and the number of unique remote hosts from which they request JavaScript code

To account for dynamically generated scripts, we crawled each page utilizing HtmlUnit, a headless browser [3], which in our experiments pretended to be Mozilla Firefox 3.6. This approach allowed us to fully execute the inline JavaScript code of each page, and thus accurately process all remote script inclusion requests, exactly as they would be processed by a normal Web browser. At the same time, if any of the visited pages, included more remote scripts based on specific non-Firefox user-agents, these inclusions would be missed by our crawler. While in our experiments we did not account for such behaviour, such a crawler could be implemented either by fetching and executing each page with multiple user-agents and JavaScript environments, or using a system like Rozzle [14] which explores multiple execution paths within a single execution in order to uncover environment-specific malware.

2.2.2 Crawling Results

Number of remote inclusions The results of our large-scale crawling of the top 10,000 Internet web sites are the following: From 3,300,000 pages, we extracted 8,439,799 inclusions. These inclusions map to 301,968 unique

[3]HtmlUnit-http://htmlunit.sourceforge.net

Offered service	**JavaScript file**	**% Top Alexa**
Web analytics	www.google-analytics.com/ga.js	68.37%
Dynamic Ads	pagead2.googlesyndication.com/pagead/show_ads.js	23.87%
Web analytics	www.google-analytics.com/urchin.js	17.32%
Social Networking	connect.facebook.net/en_us/all.js	16.82%
Social Networking	platform.twitter.com/widgets.js	13.87%
Social Networking & Web analytics	s7.addthis.com/js/250/addthis_widget.js	12.68%
Web analytics & Tracking	edge.quantserve.com/quant.js	11.98%
Market Research	b.scorecardresearch.com/beacon.js	10.45%
Google Helper Functions	www.google.com/jsapi	10.14%
Web analytics	ssl.google-analytics.com/ga.js	10.12%

Table 2: The ten most popular remotely-included files by the Alexa top 10,000 Internet web-sites

URLs of remote JavaScript files. This number does not include requests for external JavaScript files located on the same domain as the page requesting them. 88.45% of the Alexa top 10,000 web sites included at least one remote JavaScript library. The inclusions were requesting JavaScript from a total of 20,225 uniquely-addressed remote hosts (fully qualified domain names and IP addresses), with an average of 417 inclusions per remote host. Figure 1 shows the number of unique remote hosts that the top Internet sites trust for remote script inclusions. While the majority of sites trusts only a small number of remote hosts, the long-tailed graph shows that there are sites in the top Alexa list that trust up to 295 remote hosts. Since a single compromised remote host is sufficient for the injection of malicious JavaScript code, the fact that some popular sites trust hundreds of different remote servers for JavaScript is worrisome.

Remote IP address Inclusions From the total of 8,439,799 inclusions, we discovered that 23,063 (0.27%) were requests for a JavaScript script, where the URL did not contain a domain name but directly a remote IP address. These requests were addressing a total of 324 unique IP addresses. The number of requesting domains was 299 (2.99% percent of the Alexa top 10,000) revealing that the practice of addressing a remote host by its IP address is not widespread among popular Internet sites.

By geolocating the set of unique IP addresses, we discovered that they were located in 35 different countries. The country with most of these IP addresses is China (35.18%). In addition, by geolocating each domain that included JavaScript from a remote IP address, we recorded only 65 unique cases of cross-country inclusions, where the JavaScript provider and the web application were situated on different countries. This shows that if a web application requests a script directly from a remote host through its IP address, the remote host will most likely be in the same country as itself.

In general, IP-address-based script inclusion can be problematic if the IP addresses of the remote hosts are not statically allocated, forcing the script-including pages to keep track of the remote servers and constantly update their links instead of relying on the DNS protocol.

Popular JavaScript libraries Table 2 presents the ten most included remote JavaScript files along with the services offered by each script and the percentage of the top 10,000 Alexa sites that utilize them. There are several observations that can be made based on this data. First, by grouping JavaScript inclusions by the party that benefits from them, one can observe that 60% of the top JavaScript inclusions do not directly benefit the user. These are JavaScript libraries that offer Web analytics, Market Research, User tracking and Dynamic Ads, none of which has any observable effect in a page's useful content. Inclusions that obviously benefit the user are the ones incorporating social-networking functionality.

At the same time, it is evident that a single company, Google, is responsible for half of the top remotely-included JavaScript files of the Internet. While a complete compromise of this company is improbable, history has shown that it is not impossible [32].

2.3 Characterization of JavaScript Providers and Includers

In this section, we show how the problem of remote JavaScript library inclusion is widespread and underplayed, even by the most popular web applications. First, we observe how the remote inclusions of top Internet sites change over time, seeking to understand whether these sites become more or less exposed to a potential attack that leverages this problem. Then, we study how well library providers are maintaining their hosts, inquiring whether the developers of popular web applications prefer to include JavaScript libraries from well-maintained providers, which should have a lower chance of being compromised, or whether they are not concerned about this issue.

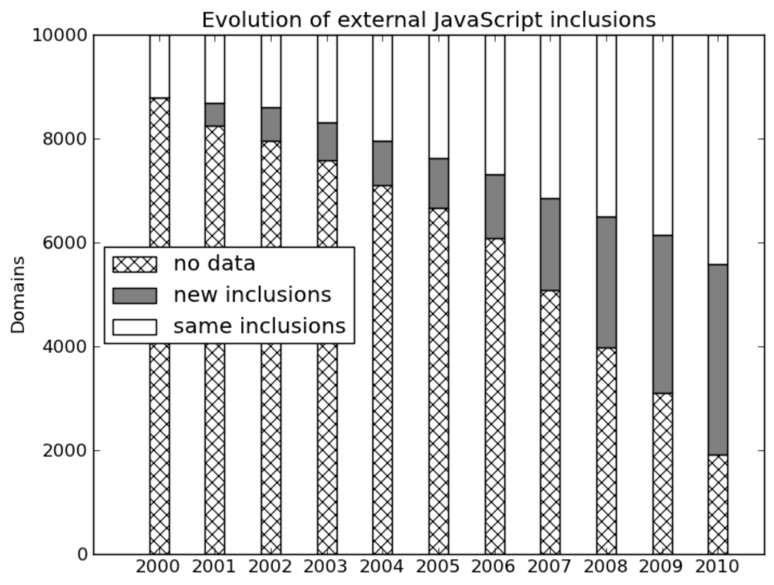

Figure 2: Evolution of remote JavaScript inclusions for domains ranked in the top 10,000 from Alexa.

2.3.1 Evolution of remote JavaScript Inclusions

In the previous section, we examined how popular web sites depend on remote JavaScript resources to enrich their functionality. In this section, we examine the remote JavaScript inclusions from the same web sites in another dimension: time. We have crawled *archive.org* [4] to study how JavaScript inclusions have evolved through time in terms of new remote dependencies and if these increase or decrease over time.

To better understand how JavaScript is included and how the inclusions change over time, we examine each page from different snapshots that span across several years. For the same pages that we crawled in Section 2.2, we have queried *archive.org* to obtain their versions for past years (if available). For each domain, we choose one representative page that has the most remote inclusions and the highest availability since 2000. For every chosen page we downloaded one snapshot per year from 2000 to 2010. Every snapshot was compared with the previous one in order to compute the inclusion changes.

In Figure 2, one can see the evolution of remote JavaScript inclusions for domains ranked in the top 10,000 from Alexa. For every year, we show how the inclusions from the previous available snapshot changed with the addition of new inclusions or if they remained the same. A *new inclusion* means that the examined domain introduced at least one new remote script inclusion since the last year. If the page's inclusions were the same as the previous

Year	No data	Same inclusions	New inclusions	% New inclusions
2001	8,256	1,317	427	24.48%
2002	7,952	1,397	651	31.79%
2003	7,576	1,687	737	30.40%
2004	7,100	2,037	863	29.76%
2005	6,672	2,367	961	28.88%
2006	6,073	2,679	1,248	31.78%
2007	5,074	3,136	1,790	36.34%
2008	3,977	3,491	2,532	42.04%
2009	3,111	3,855	3,034	44.04%
2010	1,920	4,407	3,673	45.46%

Table 3: Evolution of the number of domains with same and new remote JavaScript inclusions for the Alexa top 10,000

year, we consider those as *same inclusion*. Unfortunately, *archive.org* does not cover all the pages we examined completely, and thus we have cases where *no data* could be retrieved for a specific domain for all of the requested years. Also, many popular web sites did not exist 10 years ago. There were 892 domains for which we did not find a single URL that we previously crawled in *archive.org*. A domain might not be found on *archive.org* because of one of the following reasons: the website restricts crawling from its robots.txt file (182 domains), the domain was never chosen to be crawled (320 domains) or the domain was crawled, but not the specific pages that we chose during our first crawl (390 domains). In Table 3, we show how many domains introduced new inclusions in absolute numbers. In our experiment, we find (not surprisingly) that as we get closer in time to the present, *archive.org* has available versions for more of the URLs that we query for and thus we can examine more inclusions. We discovered that every year, a significant amount of inclusions change. Every year there are additional URLs involved in the inclusions of a website compared to the previous years and there is a clear trend of including even more. Back in 2001, 24.48% of the studied domains had at least one new remote inclusion. But as the web evolves and becomes more dynamic, more web sites extend their functionality by including more JavaScript code. In 2010, 45.46% of the examined web sites introduced a new JavaScript inclusion since the last year. This means that almost half of the top 10,000 Alexa domains had at least one new remote JavaScript inclusion in 2010, when compared to 2009.

But introducing a new JavaScript inclusion does not automatically trans-

Year	Unique domains	Total remote inclusions	Average # of new domains
2001	428	1,447	1.41
2002	680	2,392	1.57
2003	759	2,732	1.67
2004	894	3,258	1.67
2005	941	3,576	1.64
2006	974	3,943	1.61
2007	1,168	5,765	1.67
2008	1,513	8,816	1.75
2009	1,728	11,439	1.86
2010	2,249	16,901	2.10

Table 4: Number of new domains that are introduced every year in remote inclusions.

late to a new dependency from a remote provider. In Table 4, we examine whether more inclusions translate to more top-level remote domains. We calculate the unique domains involved in the inclusions and the total number of remote inclusions. For every page examined, we keep the unique domains involved in its new inclusions, and we provide the average of that number for all available pages per year. There is a clear trend in Table 4 that more inclusions result into more external dependencies from new domains. In fact in 2010 we observed that on average each page expanded their inclusions by including JavaScript from 2.1 new domains on average compared to 2009. This trend shows that the circle of trust for each page is expanding every year and that the surface of attack against them increases.

2.3.2 Quality of Maintenance Metric

Whenever developers of a web application decide to include a library from a third-party provider, they allow the latter to execute code with the same level of privilege as their own code. Effectively, they are adding the third-party host to the *security perimeter* of the web application, that is the set of the hosts whose exploitation leads to controlling the code running on that web application. Attacking the third-party, and then using that foothold to compromise the web application, might be easier than a direct attack of the latter. The aforementioned incident of the malicious modification of the qTip2 plugin [2], shows that cybercriminals are aware of this and have already used indirect exploitation to infect more hosts and hosts with more secure perimeters.

To better understand how many web applications are exposed to this kind of indirect attack, we aim to identify third-party providers that could be a weak link in the security of popular web applications. To do so, we design a metric that evaluates how well a website is being maintained, and apply it to the web applications running on the hosts of library providers (that is co-located with the JavaScript library that is being remotely included). We indicate the low-scoring as potential weak links, on the assumption that unkempt websites seem easier targets to attackers, and therefore are attacked more often.

Note that this metric aims at characterizing how well websites are maintained, and how security-conscious are their developers and administrators. It is not meant to investigate if a URL could lead to malicious content (a la Google Safebrowsing, for example). Also, we designed this metric to look for the signs of low maintenance that an attacker, scouting for the weakest host to attack, might look for. We recognize that a white-box approach, where we have access to the host under scrutiny, would provide a much more precise metric, but this would require a level of access that attackers usually do not have. We identified the closest prior work in establishing such a metric in SSL Labs's SSL/TLS survey [3] and have included their findings in our metric.

Our *Quality of Maintenance* (QoM) metric is based on the following features:

- **Availability:** If the host has a DNS record associated with it, we check that its registration is not expired. Also, we resolve the host's IP address, and we verify that it is not in the ranges reserved for private networks (e.g., `192.168.0.0/16`). Both of these features are critical, because an attacker could impersonate a domain by either registering the domain name or claiming its IP address. By impersonating a domain, an attacker gains the trust of any web application that includes code hosted on the domain.

- **Cookies:** We check the presence of at least one cookie set as `HttpOnly` and, if SSL/TLS is available, at least one cookie set as `Secure`. Also, we check that at least one cookie has its `Path` and `Expiration` attributes set. All these attributes improve the privacy of session cookies, so they are a good indication that the domain administrators are concerned about security.

- **Anti-XSS and Anti-Clickjacking protocols:** We check for the presence of the `X-XSS-Protection` protocol, which was introduced with

Internet Explorer 8 [25] to prevent some categories of Cross-site Scripting (XSS) attacks [19]. Also, we check for the presence of Mozilla's Content Security Policy protocol, which prevents some XSS and Clickjacking attacks [6] in Firefox. Finally, we check for the presence of the `X-Frame-Options` protocol, which aims at preventing ClickJacking attacks and is supported by all major browsers.

- **Cache control:** If SSL/TLS is present, we check if some content is served with the headers `Cache-Control: private` and `Pragma:no-cache`. These headers indicate that the content is sensitive and should not be cached by the browser, so that local attacks are prevented.

- **SSL/TLS implementation:** For a thorough evaluation of the SSL/TLS implementation, we rely on the study conducted by SSL Labs in April 2011. In particular, we check that the domain's certificate is valid (unrevoked, current, unexpired, and matches the domain name) and that it is trusted by all major browsers. Also, we verify that current protocols (e.g, TLS 1.2, SSL 3.0) are implemented, that older ones (e.g., SSL 2.0) are not used, and if the protocols allow weak ciphers. In addition, we check if the implementation is PCI-DSS compliant [12], which is a security standard to which organizations that handle credit card information must comply, and adherence to it is certified yearly by the Payment Card Industry. Also, we check if the domain is vulnerable to the SSL insecure-renegotiation attack. We check if the key is weak due to a small key size, or the Debian OpenSSL flaw. Finally, we check if the site offers Strict Transport Security, which forces a browser to use secure connections only, like HTTPS.

  SSL Labs collected the features described in the previous paragraph nine months before we collected all the remaining features. We believe that this is acceptable, as certificates usually have a lifespan of a few years, and the Payment Card Industry checks PCI-DSS compliance yearly. Also, since these features have been collected in the same period for all the hosts, they do not give unfair advantages to some of them.

- **Outdated web servers:** Attackers can exploit known vulnerabilities in web servers to execute arbitrary code or access sensitive configuration files. For this reason, an obsolete web server is a weak link in the security of a domain. To establish which server versions (in the HTTP `Server` header) should be considered obsolete, we collected these HTTP Server header strings during our crawl and, after clustering them, we selected the most popular web servers and their versions. Consulting

Web server	Up-to-date version(s)
Apache	1.3.42, 2.0.65, 2.2.22
NGINX	1.1.10, 1.0.9, 0.8.55, 0.7.69, 0.6.39, 0.5.38
IIS	7.5, 7.0
Lighttpd	1.5 , 1.4.29
Zeus	4.3
Cherokee	1.2
CWS	3.0
LiteSpeed	4.1.3
0w	0.8d

Table 5: Up-to-date versions of popular web servers, at the time of our experiment

change-logs and CVE reports, we compiled a list of stable and up-to-date versions, which is shown in Table 5. While it is technically possible for a web server to report an arbitrary version number, we assume that if the version is modified it will be modified to pretend that the web server is more up-to-date rather than less, since the latter would attract more attacks. This feature is not consulted in the cases where a web server does not send a Server header or specifies it in a generic way (e.g., “Apache”).

The next step in building our QoM metric is to weigh these features. We cannot approach this problem from a supervised learning angle because we have no training set: We are not aware of any study that quantifies the QoM of domains on a large scale. Thus, while an automated approach through supervised learning would have been more precise, we had to assign the weights manually. Even so, we can verify that our QoM metric is realistic. To do so, we evaluated with our metric the websites in the following four datasets of domains in the Alexa Top $10,000$:

- **XSSed domains:** This dataset contains 1,702 domains that have been exploited through cross-site scripting in the past. That is, an attacker injected malicious JavaScript on at least one page of each domain. Using an XSS exploit, an attacker can steal the cookies or password as it is typed into a login form [19]. Recently, the Apache Foundation disclosed that their servers were attacked via an XSS vulnerability, and the attacker obtained administrative access to several servers [1]. To build this dataset, we used XSSed [30], a publicly available database of over $45,000$ reported XSS attacks.

- **Defaced domains:** This dataset contains 888 domains that have been defaced in the past. That is, an attacker changed the content of one or more pages on the domain. To build this dataset, we employed the Zone-H database [33]. This database contains more than six million reports of defacements, however, only 888 out of the 10,000 top Alexa domains have suffered a defacement.

- **Bank domains:** This dataset contains 141 domains belonging to banking institutions (online and brick and mortar) in the US.

- **Random domains:** This dataset contains 4,500 domains, randomly picked, that do not belong to the previous categories.

The cumulative distribution function of the metric on these datasets is shown in Figure 3. At score 60, we have 506 defaced domains, 698 XSSed domains, 765 domains belonging to the random set, and only 5 banks. At score 120, we have all the defaced and XSSed domains, 4,409 domains from the random set, and all but 5 of the banking sites. The maximum score recorded is 160, held by `paypal.com`. According to the metric, sites that have been defaced or XSSed in the past appear to be maintained less than our dataset of random domains. On the other hand, the majority of banking institutions are very concerned with the maintenance of their domains. These findings are reasonable, and empirically demonstrate that our metric is a good indicator of the quality of maintenance of a particular host. This is especially valid also because we will use this metric to classify hosts into three wide categories: high maintenance (metric greater than 150), medium, and low maintenance (metric lower than 70).

2.3.3 Risk of Including Third-Party Providers

We applied our QoM metric to the top 10,000 domains in Alexa and the domains providing their JavaScript inclusions. The top-ranking domain is `paypal.com`, which has also always been very concerned with security (e.g., it was one of the proposers of HTTP Strict Transport Security). The worst score goes to `cafemom.com`, because its SSL certificate is not valid for that domain (its `CommonName` is set to `mom.com`), and it is setting cookies non-`HTTPOnly`, and not `Secure`. Interestingly, it is possible to login to the site both in HTTPS, and in plain-text HTTP.

In Figure 4, we show the cumulative distribution function for the inclusions we recorded. We can see that low-maintenance domains often include JavaScript libraries from low-maintenance providers. High-maintenance domains, instead, tend to prefer high-maintenance providers, showing that they

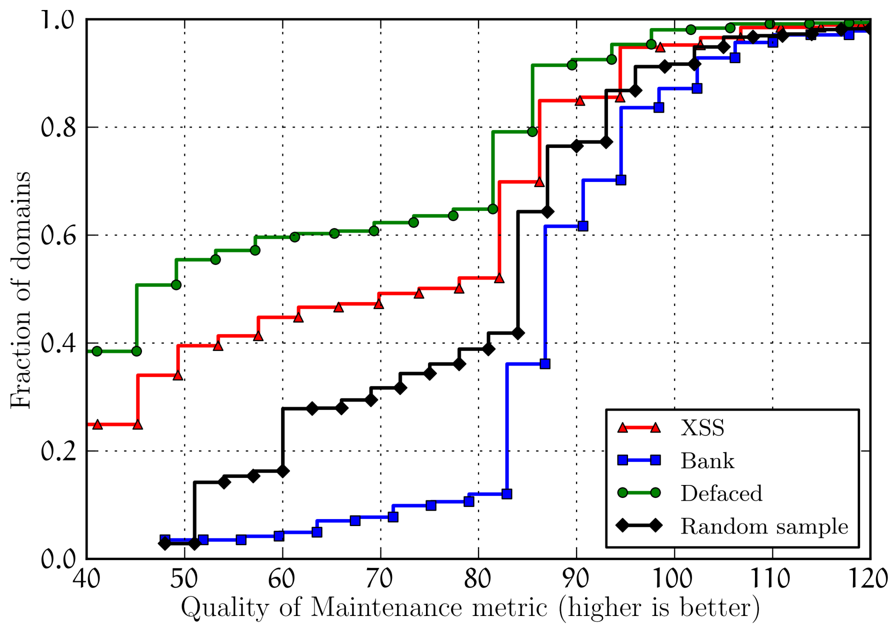

Figure 3: Cumulative distribution function of the maintenance metric, for different datasets

are indeed concerned about the providers they include. For instance, we can see that the JavaScript libraries provided by sites with the worst maintenance scores, are included by over 60% of the population of low-maintenance sites, versus less than 12% of the population of sites with high-maintenance scores. While this percentage is five times smaller than the one of low-maintenance sites, still, about one out of four of their inclusions come from providers with a low maintenance score, which are potential "‘weak spots"’ in their security perimeter. For example, criteo.com is an advertising platform that is remotely included in 117 of the top 10,000 Alexa domains, including ebay.de and sisal.it, the society that holds the state monopoly on bets and lottery in Italy. criteo.com has an implementation of SSL that supports weak ciphers, and a weak Diffie-Hellman ephemeral key exchange of 512 bits. Another example is levexis.com, a marketing platform, which is included in 15 of the top 10,000 Alexa websites, including lastminute.com, and has an invalid SSL certificate.

2.4 Attacks

In this section, we describe four types of vulnerabilities that are related to unsafe third-party inclusion practices, which we encountered in the analysis of the top 10,000 Alexa sites. Given the right conditions, these vulnerabilities

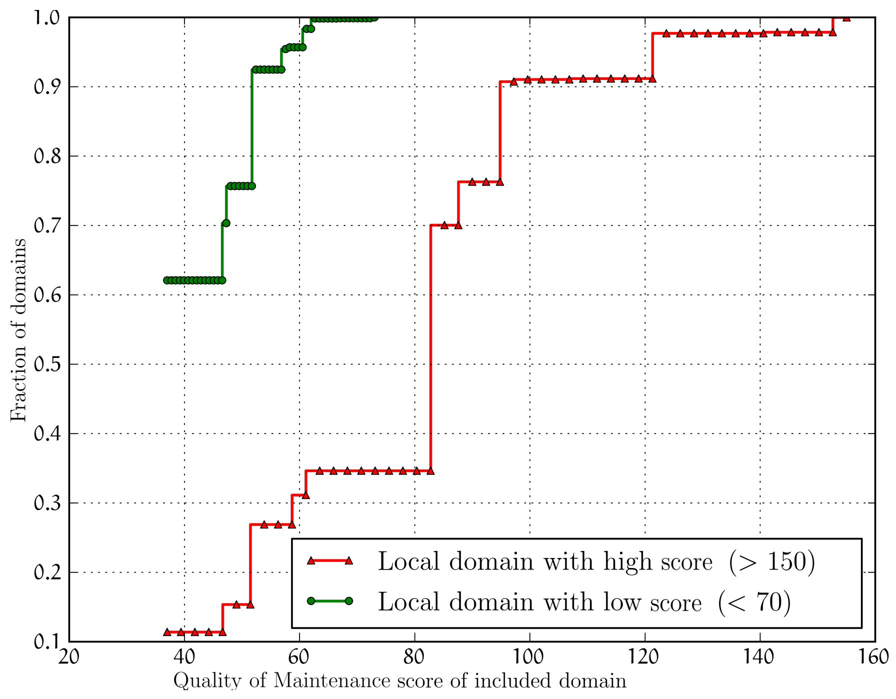

Figure 4: Risk of including third-party providers, included in high and low maintenance web applications.

enable an attacker to take over popular web sites and web applications.

2.4.1 Cross-user and Cross-network Scripting

In the set of remote script inclusions resulting from our large-scale crawling experiment, we discovered 133 script inclusions where the "src" attribute of the script tag was requesting a JavaScript file from `localhost` or from the `127.0.0.1` IP address. Since JavaScript is a client-side language, when a user's browser encounters such a script tag, it will request the JavaScript file from the user's machine. Interestingly, 131 out of the 133 localhost inclusions specified a port (e.g., `localhost:12345`), which was always greater than 1024 (i.e., a non-privileged port number). This means that, in a multiuser environment, a malicious user can set up a web server, let it listen to high port numbers, and serve malicious JavaScript whenever a script is requested from `localhost`. The high port number is important because it allows a user to attack other users without requiring administrator-level privileges.

In addition to connections to `localhost`, we found several instances where the source of a script tag was pointing to a private IP address (e.g., `192.168.2.2`). If a user visits a site with such a script inclusion, then her browser will search for the JavaScript file on the user's local network. If an attacker manages to get the referenced IP address assigned to his machine, he will be able to serve malicious JavaScript to the victim user.

We believe that both vulnerabilities result from a developer's erroneous understanding of the way in which JavaScript is fetched and executed. The error introduced is not immediately apparent because, often times, these scripts are developed and tested on the developer's local machine (or network), which also hosts the web server.

The set of domains hosting pages vulnerable to cross-user and cross-network scripting, included popular domains such as `virginmobileusa.com`, `akamai.com`, `callofduty.com` and `gc.ca`.

2.4.2 Stale Domain-name-based Inclusions

Whenever a domain name expires, its owner may choose not to renew it without necessarily broadcasting this decision to the site's user-base. This becomes problematic when such a site is providing remote JavaScript scripts to sites registered under different domains. If the administrators of the including sites do not routinely check their sites for errors, they will not realize that the script-providing site stopped responding. We call these inclusions "stale inclusions". Stale inclusions are a security vulnerability for a site, since an attacker can register the newly-available domain and start providing all stale JavaScript inclusion requests with malicious JavaScript. Since the vulnerable pages already contain the stale script inclusions, an attacker does not need to interact with the victims or convince them to visit a specific page, making the attack equivalent to a stored XSS.

To quantify the existence of stale JavaScript inclusions, we first compiled a list of all JavaScript-providing domains that were discovered through our large-scale crawling experiment. From that list, we first excluded all domains that were part of Alexa's top one million web sites list. The remaining 4,225 domains were queried for their IP address and the ones that did not resolve to an address were recorded. The recorded ones were then queried in an online WHOIS database. When results for a domain were not available, we attempted to register it on a popular domain-name registrar.

The final result of this process was the identification of 56 domain names, used for inclusion in 47 of the top 10,000 Internet web sites, that were, at the time of our experiments, available for registration. By manually reviewing these 56 domain names, we realized that in 6 cases, the developers mistyped the JavaScript-providing domain. These form an interesting security issue, which we consider separately in Section 2.4.4.

Attackers could register these domains to steal credentials or to serve

	blogtools.us	hbotapadmin.com
Visits	80,466	4,615
Including domains	24	4
Including pages	84	41

Table 6: Results from our experiment on expired remotely-included domains

malware to a large number of users, exploiting the trust that the target web application puts in the hijacked domain. To demonstrate how easy and effective this attack is, we registered two domains that appear as stale inclusions in popular web sites, and make them resolve to our server. We recorded the `Referer`, source IP address, and requested URL for every HTTP request received for 15 days. We minimized the inconvenience that our study might have caused by always replying to HTTP requests with a HTML-only `404 Not Found` error page, with a brief explanation of our experiment and how to contact us. Since our interaction with the users is limited to logging the three aforementioned pieces of data, we believe there are no ethical implications in this experiment. In particular, we registered blogtools.us, a domain included on goldprice.org, which is a web application that monitors the price of gold and that ranks 4,779th in the US (according to Alexa). Previously, blogtools.us was part of a platform to create RSS feeds. We also registered hbotapadmin.com, included in a low-traffic page on hbo.com, which is an American cable television network, ranking 1,411th in the US. hbotapadmin.com was once owned by the same company, and its registration expired in July 2010. The results of our experiment are shown in Table 6. While hbotapadmin.com is being included exclusively by HBO-owned domains, it is interesting to notice that blogtools.us is still included by several lower-ranking domains, such as happysurfer.com, even though the service is not available anymore.

2.4.3 Stale IP-address-based Inclusions

As described in Section 2.2, some administrators choose to include remote scripts by addressing the remote hosts, not through a domain name but directly through an IP address. While at first this decision seems suboptimal, it is as safe as a domain-name-based inclusion, as long as the IP address of the remote machine is static or the including page is automatically updated whenever the IP address of the remote server changes.

To assess whether one of these two conditions hold, we manually visited all 299 pages performing an IP address-based inclusion, three months after our initial crawl. In the majority of cases, we recorded one of the following three scenarios: a) the same scripts were included, but the host was now addressed through a domain name, b) the IP addresses had changed or the inclusions were removed or c) the IP addresses remained static. Unfortunately, in the last category, we found a total of 39 IP addresses (13.04%) that had not changed since our original crawl but at the same time, were not providing any JavaScript files to the requests. Even worse, for 35 of them (89.74%) we recorded a "Connection Timeout," attesting to the fact that there was not even a Web server available on the remote hosts. This fact reveals that the remote host providing the original scripts either became unavailable or changed its IP address, without an equivalent change in the including pages.

As in domain-name-based stale inclusions, these inclusions can be exploited by an attacker who manages to obtain the appropriate IP address. While this is definitely harder than registering a domain-name, it is still a vulnerability that could be exploited given an appropriate network configuration and possibly the use of the address as part of a DHCP address pool.

2.4.4 Typosquatting Cross-site Scripting (TXSS)

Typosquatting [17, 29] is the practice of registering domain names that are slight variations of the domains associated with popular web sites. For instance, an individual could register `wikiepdia.org` with the intent of capturing a part of the traffic originally meant to go toward the popular Wikipedia website. The user that mistypes Wikipedia, instead of getting a "Server not found" error, will now get a page that is under the control of the owner of the mistyped domain. The resulting page could be used for advertising, brand wars, phishing credentials, or triggering a drive-by download exploit against a vulnerable browser.

Traditionally, typosquatting always refers to a user mistyping a URL in her browser's address bar. However, web developers are also humans and can thus mistype a URL when typing it into their HTML pages or JavaScript code. Unfortunately, the damage of these mistakes is much greater than in the previous case, since every user visiting the page containing the typo will be exposed to data originating from the mistyped domain. In Table 7, we provide five examples of mistyped URLs found during our experiment for which we could identify the intended domain.

As in the case of stale domain-names, an attacker can simply register these sites and provide malicious JavaScript to all unintended requests. We observed this attack in the wild: according to Google's Safe Browsing, `worldofwaircraft.com` has spread malware in January 2012. To prove the efficacy of this attack, we registered `googlesyndicatio.com` (mistyped

Intended domain	Actual domain
googlesyndication.com	googlesyndicatio.com
purdue.edu	purude.edu
worldofwarcraft.com	worldofwaircraft.com
lesechos.fr	lessechos.fr
onegrp.com	onegrp.nl

Table 7: Examples of mistyped domains found in remote JavaScript inclusion tags

googlesyndication.com), and logged the incoming traffic. We found this domain because it is included in leonardo.it, an Italian online newspaper (Alexa global rank: 1,883, Italian rank: 56). Over the course of 15 days, we recorded 163,188 unique visitors. Interestingly, we discovered that this misspelling is widespread: we had visitors incoming from 1,185 different domains, for a total of 21,830 pages including this domain. 552 of the domains that include ours belong to blogs hosted on *.blogspot.com.br, and come from the same snippet of code: It seems that bloggers copied that code from one another. This mistype is also long living: We located a page containing the error, http://www.oocities.org/br/dicas.html/, that is a mirror of a Brazilian Geocities site made in October 2009.

2.5 Countermeasures

In this section, we review two techniques that a web application can utilize to protect itself from malicious remotely-included scripts. Specifically, we examine the effectiveness of using a coarse-grained JavaScript sandboxing system and the option of creating local copies of remote JavaScript libraries.

2.5.1 Sandboxing remote scripts

Recognizing the danger of including a remote script, researchers have proposed a plethora of client-side and server-side systems that aim to limit the functionality of remotely-included JavaScript libraries (see Section 2.6). The majority of these countermeasures apply the principle of least privilege to remotely-included JavaScript code. More precisely, these systems attempt to limit the actions that can be performed by a remotely-included script to the bare minimum.

The least-privilege technique requires, for each remotely-included JavaScript file, a profile describing which functionality is needed when the script is executed. This profile can be generated either through manual code inspection or by first allowing the included script to execute and then recording all functions and properties of the Document Object Model (DOM) and Browser Object Model (BOM) that the script accessed. Depending on the sandboxing mechanism, these profiles can be either coarse-grained or fine-grained.

In a coarse-grained sandboxing system, the profile-writer instructs the sandbox to either forbid or give full access to any given resource, such as forbidding a script to use `eval`. Contrastingly, in a fine-grained sandboxing system, the profile-writer is able to instruct the sandbox to give access to only parts of resources to a remotely included script. For instance, using ConScript [16], a profile-writer can allow the dynamic creation of all types of elements except iframes, or allow the use of `eval` but only for the unpacking of JSON data. While this approach provides significantly more control over each script than a coarse-grained profile, it also requires more effort to describe correct and exact profiles. Moreover, each profile would need to be updated, every time that a remote script *legitimately* changes in a way that affects its current profile.

Static and dynamic analysis have been proposed as ways of automatically constructing profiles for sandboxing systems, however, they both have limitations in the coverage and correctness of the profiles that they can create. Static analysis cannot account for dynamically-loaded content, and dynamic analysis cannot account for code paths that were not followed in the training phase of the analysis. Moreover, even assuming a perfect code-coverage during training, it is non-trivial to automatically identify the particular use of each requested resource in order to transit from coarse-grained sandboxing to fine-grained.

Given this complex, error-prone and time-consuming nature of constructing fine-grained profiles, we wanted to assess whether coarse-grained profiles would sufficiently constrain popular scripts. To this end, we automatically generated profiles for the 100 most included JavaScript files, discovered through our crawl. If the privileges/resources required by legitimate scripts include everything that an attacker needs to launch an attack, then a coarse-grained sandboxing mechanism would not be an effective solution.

The actions performed by an included JavaScript file were discovered using the following setup: A proxy was placed in between a browser and the Internet. All traffic from the web browser was routed through the web proxy [11], which we modified to intercept HTTP traffic and inject instrumentation code into the passing HTML pages. This instrumentation code uses JavaScript's `setters` and `getters` to add wrappers to certain sensitive JavaScript functions and DOM/BOM properties, allowing us to monitor their use. The browser-provided on-demand stack-tracing functionality, al-

JS Action	**# of Top scripts**
Reading Cookies	41
`document.write()`	36
Writing Cookies	30
`eval()`	28
`XHR`	14
Accessing LocalStorage	3
Accessing SessionStorage	0
Geolocation	0

Table 8: JavaScript functionality used by the 100 most popularly included remote JavaScript files

lowed us to determine, at the time of execution of our wrappers, the chain of function calls that resulted in a specific access of a monitored resource. If a function, executed by a remote script, was part of this chain, then we safely deduce that the script was responsible for the activity, either by directly accessing our monitored resources or by assisting the access of other scripts.

For instance, suppose that a web page loads `a.js` and `b.js` as follows:

```
/* a.js */
function myalert(msg) {
    window.alert(msg);
}
```

```
/* b.js */
myalert("hello");
```

```
/* stack trace */
b.js:1:myalert(...)
a.js:2:window.alert(...)
```

In `a.js`, a function `myalert` is defined, which passes its arguments to the `window.alert()` function. Suppose `b.js` then calls `myalert()`. At the time this function is executed, the *wrapped* `window.alert()` function is executed. At this point, the stack trace contains both `a.js` and `b.js`, indicating that both are involved in the call to `window.alert()` (a potentially-sensitive function) and thus both can be held responsible. These accesses can be straightforwardly transformed into profiles, which can then be utilized by coarse-grained sandboxing systems.

Using the aforementioned setup, we visited web pages that included the top 100 most-included JavaScript files and monitored the access to sensitive JavaScript methods, DOM/BOM properties. The results of this experiment, presented in Table 8, indicate that the bulk of the most included JavaScript files read and write cookies, make calls to `document.write()`, and dynamically evaluate code from strings. Newer APIs on the other hand, like `localStorage`, `sessionStorage` and `Geolocation`, are hardly ever used, most likely due to their relatively recent implementation in modern web browsers.

The results show that, for a large part of the included scripts, it would be impossible for a coarse-grained sandboxing system to differentiate between benign and malicious scripts solely on their usage of cookie functionality. For instance, a remotely-included benign script that needs to access cookies to read and write identifiers for user-tracking can be substituted for a malicious script that leaks the including site's session identifiers. Both of these scripts access the same set of resources, yet the second one has the possibility of fully compromising the script-including site. It is also important to note that, due to the use of dynamic analysis and the fact that some code-paths of the executed scripts may not have been followed, our results are lower bounds of the scripts' access to resources, i.e., the tracked scripts may need access to more resources to fully execute.

Overall, our results highlight the fact that even in the presence of a coarse-grained sandboxing system that forbids unexpected accesses to JavaScript and browser resources, an attacker could still abuse the access already whitelisted in the attacked script's profile. This means that regardless of their complexity, fine-grained profiles would be required in the majority of cases. We believe that this result motivates further research in fine-grained sandboxing and specifically in the automatic generation of correct script profiles.

2.5.2 Using local copies

Another way that web sites can avoid the risk of malicious script inclusions is by simply not including any remote scripts. To this end, a site needs to create local copies of remote JavaScript resources and then use these copies in their script inclusions. The creation of a local copy separates the security of the remote site from the script-including one, allowing the latter to be unaffected by a future compromise of the former. At the same time, however, this shifts the burden of updates to the developer of the script-including site who must verify and create a new local copy of the remote JavaScript library whenever a new version is made available.

To quantify the overhead of this manual procedure on the developer of a script-including web application, we decided to track the updates of the top 1,000 most-included scripts over the period of one week. This experiment was conducted four months after our large-scale crawling experiment, thus some URLs were no longer pointing to valid scripts. More precisely, from the top 1,000 scripts we were able to successfully download 803. We started by downloading this set three consecutive times within the same hour and comparing the three versions of each script. If a downloaded script was different all three times then we assume that the changes are not due to actual updates of the library, such as bug fixes or the addition of new functionality, but due to the embedding of constantly-changing data, such as random tokens, dates, and execution times. From this experiment, we found that 3.99% of our set of JavaScript scripts, seem to embed such data and thus appear to be constantly modified. For the rest of the experiment, we stopped tracking these scripts and focused on the ones that were identical all three times.

Over a period of one week, 10.21% of the monitored scripts were modified. From the modified scripts, 6.97% were modified once, 1.86% were modified twice, and 1.36% were modified three or more times. This shows that while some scripts undergo modifications more than once a week, 96.76% are modified at most once. We believe that the weekly manual inspection of a script's modified code is an acceptable tradeoff between increased maintenance time and increased security of the script-including web application. At the same time, a developer who currently utilizes frequently-modified remote JavaScript libraries, might consider substituting these libraries for others of comparable functionality and less frequent modifications.

2.6 Related Work

Measurement Studies To the best of our knowledge, there has been no study of remote JavaScript inclusions and their implications that is of comparable breadth to our work. Yue and Wang conducted the first measurement study of insecure JavaScript practices on the web [31]. Using a set of 6,805 homepages of popular sites, they counted the sites that include remote JavaScript files, use the `eval` function, and add more information to the DOM of a page using `document.write`. Contrastingly, in our study, we crawled more than 3 million pages of the top 10,000 popular web sites, allowing us to capture five hundred times more inclusions and record behavior that is not necessarily present on a site's homepage. Moreover, instead of treating all remote inclusions as uniformly dangerous, we attempt to characterize the quality of their providers so that more trustworthy JavaScript providers can be utilized when a remote inclusion is unavoidable.

Richards et al. [24] and Ratanaworabhan et al. [21] study the dynamic behavior of popular JavaScript libraries and compare their findings with common usage assumptions of the JavaScript language and the functionality tested by common benchmarks. However, this is done without particular focus on the security features of the language. Richarts et al. [23] have also separately studied the use of `eval` in popular web sites.

Ocariza et al. [13] performed an empirical study of JavaScript errors in the top 100 Alexa sites. Seeking to quantify the reliability of JavaScript code in popular web applications, they recorded errors falling into four categories: "Permission Denied," "Null Exception," "Undefined Symbol" and "Syntax Error." Additionally, the authors showed that in some cases the errors were non-deterministic and depended on factors such as the speed of a user's interaction with the web application. The authors did not encounter any of the new types of vulnerabilities we described in Section 2.4, probably due to the limited size of their study.

Limiting available JavaScript functionality Based on the characterization of used functionality, included JavaScript files could be executed in a restricted environment that only offers the required subset of functionality. As we showed in Section 2.5.1, a fine-grained sandboxing system is necessary because of the inability of coarse-grained sandboxes to differentiate between legitimate and malicious access to resources.

BrowserShield [22] is a server-side rewriting technique that replaces certain JavaScript functions to use safe equivalents. These safe equivalents are implemented in the "bshield" object that is introduced through the BrowserShield JavaScript libraries and injected into each page. BrowserShield makes use of a proxy to inject its code into a web page. Self-protecting JavaScript [20, 15] is a client-side wrapping technique that applies wrappers around JavaScript functions, without requiring any browser modifications. The wrapping code and policies are provided by the server and are executed first, ensuring a clean environment to start from.

ConScript [16] allows the enforcement of fine-grained security policies for JavaScript in the browser. The approach is similar to self-protecting JavaScript, except that ConScript modifies the browser to ensure that an attacker cannot abuse the browser-specific DOM implementation to find an unprotected access path. WebJail [28] is a client-side security architecture that enforces secure composition policies specified by a web-mashup integrator on third-party web-mashup components. Inspired by ConScript, WebJail modifies the Mozilla Firefox browser and JavaScript engine, to enforce these secure composition policies inside the browser. The new "sandbox" attribute of the iframe element in HTML5 [10] provides a way to limit JavaScript functionality, but it is very coarse-grained. It only supports limited restrictions, and as far as JavaScript APIs are concerned, it only supports to completely enable or disable JavaScript.

ADJail [27] is geared toward securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page to which the web developer wishes the ad to have access. Changes to the shadow page are replicated to the hosting page if those changes conform to the specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy.

FlowFox [7] uses the related technique of secure multi-execution [8] to execute arbitrary included scripts with strong guarantees that these scripts can not break a specified confidentiality policy.

Content Security Policy (CSP) [26] is a mechanism that allows web application developers to specify from which locations their web application is allowed to load additional resources. Using CSP, a web application could be limited to only load JavaScript files from a specific set of third-party locations. In the case of typos in the URL, a CSP policy not containing that same typo will prevent a JavaScript file from being loaded from that mistyped URL. Cases where a JavaScript-hosting site has been compromised and is serving malicious JavaScript however, will not be stopped by CSP.

AdSentry [9] is a confinement solution for JavaScript-based advertisement scripts. It consists of a shadow JavaScript engine which is used to execute untrusted JavaScript advertisements. Instead of having direct access to the DOM and sensitive functions, access from the shadow JavaScript engine is mediated by an access control policy enforcement subsystem.

2.7 Conclusion

Web sites that include JavaScript from remote sources in different administrative domains open themselves to attacks in which malicious JavaScript is sent to unsuspecting users, possibly with severe consequences. In this paper, we extensively evaluated the JavaScript remote inclusion phenomenon, analyzing it from different points of view. We first determined how common it is to include remote JavaScript code among the most popular web sites on the Internet. We then provided an empirical evaluation of the quality-of-maintenance of these "code providers," according to a number of indicators. The results of our experiments show that indeed there is a considerable number of high-profile web sites that include JavaScript code from external sources that are not taking all the necessary security-related precautions and thus could be compromised by a determined attacker. As a by-product of our experiments, we identified several attacks that can be carried out by exploiting failures in the configuration and provision of JavaScript code inclusions. Our findings shed some light into the JavaScript code provider infrastructure and the risks associated with trusting external parties in implementing web applications.

References

[1] Apache.org. https://blogs.apache.org/infra/entry/apache_org_04_09_2010.

[2] Qtip compromised. https://github.com/Craga89/qTip2/issues/286.

[3] SSL Labs Server Rating Guide. https://www.ssllabs.com/downloads/SSL_Server_Rating_Guide_2009.pdf.

[4] Wayback Machine. http://archive.org.

[5] Alexa - Top sites on the Web. http://www.alexa.com/topsites.

[6] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 135–144, 2010.

[7] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[8] Dominique Devriese and Frank Piessens. Noninterference Through Secure Multi-Execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[9] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 297–306, New York, NY, USA, 2011. ACM.

[10] I. Hickson and D. Hyatt. HTML 5 Working Draft - The sandbox Attribute. http://www.w3.org/TR/html5/the-iframe-element.html#attr-iframe-sandbox, June 2010.

[11] Suzuki Hisao. Tiny HTTP Proxy in Python. http://www.okisoft.co.jp/esc/python/proxy/.

[12] Payment Card Industry. (Approved Scanning Vendor) Program Guide. https://www.pcisecuritystandards.org/pdfs/asv_program_guide_v1.0.pdf.

[13] Frolin Ocariza Jr., Karthik Pattabiraman, and Benjamin Zorn. Javascript errors in the wild: An empirical study. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 100 –109, 2011.

[14] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *IEEE Symposium on Security and Privacy*, May 2012.

[15] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *The 15th Nordic Conf. in Secure IT Systems. Springer Verlag*, 2010.

[16] Leo Meyerovich and Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[17] Tyler Moore and Benjamin Edelman. Measuring the perpetrators and funders of typosquatting. In *Proceedings of the 14th international conference on Financial Cryptography and Data Security*, FC'10, pages 175–191, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] 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 *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*, pages 736–747, October 2012.

[19] OWASP. "cross-site scripting (xss)". https://www.owasp.org/index.php/XSS.

[20] 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*, ASI-ACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[21] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. JS-Meter: comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[22] 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.

[23] 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 *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[25] David Ross. IE8 Security Part IV: The XSS Filter. http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx.

[26] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 921–930, New York, NY, USA, 2010. ACM.

[27] Mike Ter Louw, Karthik Thotta Ganesh, and V.N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the 19th USENIX Security Symposium*, August 2010.

[28] 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 *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM.

[29] Yi-Min Wang, Doug Beck, Jeffrey Wang, Chad Verbowski, and Brad Daniels. Strider typo-patrol: discovery and analysis of systematic typo-squatting. In *Proceedings of the 2nd conference on Steps to Reducing Unwanted Traffic on the Internet - Volume 2*, SRUTI'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.

[30] XSSed | Cross Site Scripting (XSS) attacks information and archive.

[31] Chuan Yue and Haining Wang. Characterizing insecure JavaScript practices on the web. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 961–970, New York, NY, USA, 2009. ACM.

[32] Kim Zetter. Google Hack Attack Was Ultra Sophisticated, New Details Show. http://www.wired.com/threatlevel/2010/01/operation-aurora/.

[33] Zone-H: Unrestricted information. http://zone-h.org/.

3 A Two-Tier Sandbox Architecture for Untrusted JavaScript[4][5]

3.1 Introduction

Embedding external content into web pages is becoming more and more popular. A recent report [1] shows that 97% of Fortune 500 web sites display content from external partners using e.g. JavaScript widget providers, ad networks, or packaged software providers. Typically, a web master of a hosting page needs to trust the external JavaScript code before inserting it to the page since the external JavaScript (mashup) code run in the context of the hosting page. However sometimes this trust can be misplaced. In September 2009, readers of the New York Times website faced a fake virus infection pop-up which directs the readers to a web page that claims to offer anti-virus software. The attack happened because the external content, e.g. the ad, is normally fetched dynamically from an external source by the user's browser [25], which is not under the control of the hosting page.

There have been a number of solutions proposed dealing with untrusted JavaScript code, such as Google Caja [17], Facebook JavaScript [4], ADsafe [2] and Conscript [16], and so on. However, we identified some important limitations with the current state of the art as will be discussed in more detail in Section 3.2. In particular, most of the approaches require (1) browser modifications or server-side pre-processing e.g. static validation, filtering, or transformation of the untrusted JavaScript, limiting the deployment capabilities, or (2) restrict the expressiveness of the enforcement to coarse-grained access control policies.

To achieve both the expressiveness of security policies as well as easy of deployment of our secure integration architecture, we propose a client-side security architecture that enforces fine-grained, stateful security policies for untrusted JavaScript code but does not require pre-processing of the code nor browser modification. This client-side security architecture is realized by means of a two-tier sandbox architecture: a generic, coarse-grained sand-

[4]This paper has been published as [19]: Phu H. Phung, Lieven Desmet, A two-tier sandbox architecture for untrusted JavaScript, Proceedings of the Workshop on JavaScript Tools (JSTools '12), pages 1-10, Beijing, China, 13 June 2012

[5]This research is funded by the EU FP7 WebSand project. Thanks to David Sands and Andrei Sabelfeld for their feedback and helpful comments. Part of this work was performed while the first author was visiting Stanford hosted by John Mitchell. The first author would like to thank John Mitchell, Ankur Taly, and Mark Miller for their discussions. For the second author, this work was also partially funded by the EU FP7 NESSoS project, the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund KU Leuven.

box provides strong baseline isolation guarantees, whereas a second sandbox enables fine-grained, stateful policy enforcement specific to a particular untrusted application. The application-specific policy enforcement code is executed within an outer sandbox environment which guarantees that it – even if subverted or badly written – will still adhere to some general security policies provided by the baseline isolation of the outer sandbox, and e.g. does not give unauthorized access to sensitive resources or unintentionally leak unprotected references.

We have developed a prototype implementation of the proposed two-tier sandbox architecture, leveraging on recent developments in ECMAScript 5, a new JavaScript specification. For the sandboxing technology, we have built upon an existing open-source JavaScript library [3]. This library applies the strict mode of ECMAScript 5, and allows to load and execute untrusted code dynamically in a sandbox environment. The fined-grained policy enforcement is a modified version of our lightweight self-protecting JavaScript mechanism [20, 14], and improves upon the earlier work by achieving strong security guarantees and being able to deal with untrusted code. The policies express access control restrictions for both method calls as well as property accesses, and are expressed in pure JavaScript so that a policy writer can easily express stateful and fine-grained policies.

Our proposed architecture improves upon the state-of-the-art since it does not depend on browser modification nor transformation of client-side code, and allows the secure enforcement of fine-grained, stateful access control policies for untrusted JavaScript. The solution enables application-specific policy enforcement, even if multiple third-party applications are loaded on the same page. We have applied the prototype implementation to a set of mashup components, and a representative online advertisement case study to validate the feasibility and security of the proposed architecture.

Organization The rest of this paper is written as follows. The next section briefly sketches the challenges for securely integrating third-party JavaScript, and expresses the requirements for a client-side security architecture. Section 3.3 proposes the two-tier enforcement architecture for policy enforcement of untrusted code on an API in sandbox compartments. We present the prototype implementation of the architecture in Section 3.4. In Section 3.5 we present our design for specifying fine-grained policies. The validation of the prototype are presented and discussed in Section 3.6. Discussion and future work are given in Section 3.8.

3.2 Problem statement

The state-of-practice technique to integrate third-party scripts in webpages is via script inclusion [23]. By doing so, the browser will execute the code as if it part of the original webpage, without any restrictions of the Same-Origin Policy. As a side effect, the third-party code executes in the same JavaScript context, and has access to all the code and data of the integrating webpage.

This is clearly not desirable in case the third-party JavaScript is malicious or can not be trusted. But even if the external party is trustworthy at the moment of website construction, the partner can become malicious over time, or be the victim of an attacker by itself. As a result, if an attacker controls the integrated script, he has full control over the hosting website and can perform unwanted actions on behalf of the website owner as well as website visitors.

Therefore, several countermeasures have been proposed [22, 13, 11, 20, 16, 12] and some of them are widely adopted [17, 4, 2]. Section 3.7 discusses various solutions in more detail, but in this section we briefly highlight two important shortcomings in the current state-of-the-art: the ease of deployment and the expressiveness of security policies.

Ease of deployment There are several modes of deployment, each with their own characteristics. The most invasive set of solutions require client-side modifications [18, 16, 26], which strongly limits the adoption of the solution, and requires web users to install additional extensions or custom-build browsers. Other solutions requires server-side pre-processing (such as filtering, transforming or wrapping untrusted code) [11] or impose restrictions to the third-party code to adhere to safe subset of JavaScript [4, 2, 12].

Both browser modification and pre-processing approaches make the deployment more difficult, and differ from the state-of-practice setting in which a legacy browser fetches JavaScript code directly from the external partner.

Expressiveness Most of available solutions only allow coarse-grained access control policies to be specified. Facebook JavaScript [4], ADsafe [2], and Google Caja [17] for instance only allow to enforce general coarse-grained policies for untrusted JavaScript. Alternative solutions providing fine-grained policy enforcement rely however on client-side modifications [18, 16, 26] or server-side pre-processing [11].

Another key issue in policy enforcement is that writing fine-grained, stateful policies can be hard and error-prone, and attackers can subvert defective policies to bypass the enforcement [16, 14]. One approach to tackling this issue is to constrain the way that policies are written, however it limits the expressiveness of policies. For instance, WebJail [26] constrains policies to whitelists to avoid '*inverse sandbox*' attacks. Designing a specific policy language which can construct a suitable static type system to ensure the correctness of policies [16] but this typically also requires modifications to the browser.

Requirements

As stated before, the current set of secure integration techniques for third-party JavaScript either rely on browser modification or code pre-processing, or strongly limits the expressiveness of the security policies. To achieve an expressive and easy-deployable client-side security architecture, we therefore propose the following requirements:

R1. The client-side security architecture should be able to cope with fine-grained security and/or stateful security policies.

For instance, a policy developer should be able to express that (1) only a subset of security-sensitive operations can be used, (2) that certain functions only can be called if the arguments satisfy additional constraints (e.g. appear in a whitelist), and (3) that no cross-origin requests may be sent out after user-supplied input has been received (e.g. as part of a form).

R2. To ease the deployment of the security architecture, the solution should not require any modification of the browser, nor should depend on server-side pre-processing of the untrusted JavaScript code.

By targeting mainstream browsers, the solution can be deployed without additional effort of the end-user (such as downloading a specific browser extension or custom-build browser). Without the need for code pre-processing, the state-of-practice integration technique can be preserved to directly embed external scripts in the browser.

R3. The client-side security architecture provides complete mediation in accessing security-sensitive operations. Irrespectively of how the security-sensitive operation gets called, the security policies is always applied.

R4. The client-side security architecture must be robust to potential flaws in security policies.

Since the process of writing fine-grained and/or stateful security policies can be error-prone [15, 16, 14], the security architecture must be able to limit the elevation of privileges in case a security check can be circumvented due to a policy flaw.

3.3 Two-tier sandbox architecture

The main purpose of the client-side security architecture is to build a sandbox environment, in which untrusted scripts can run securely. In such a sandbox environment, all accesses to security-sensitive operations are mediated, and are controlled via a security policy.

Two-tier approach

Since writing fine-grained and/or stateful security policies can be hard and error-prone, especially for rich environment such as client-side web scripting, we propose a two-tier sandbox architecture. The architecture protects security-sensitive operations by nesting two sandbox environments, as depicted in Fig. 5: a generic, coarse-grained outer sandbox provides strong baseline isolation guarantees, whereas a second sandbox enables fine-grained, stateful policy enforcement specific to a particular untrusted application.

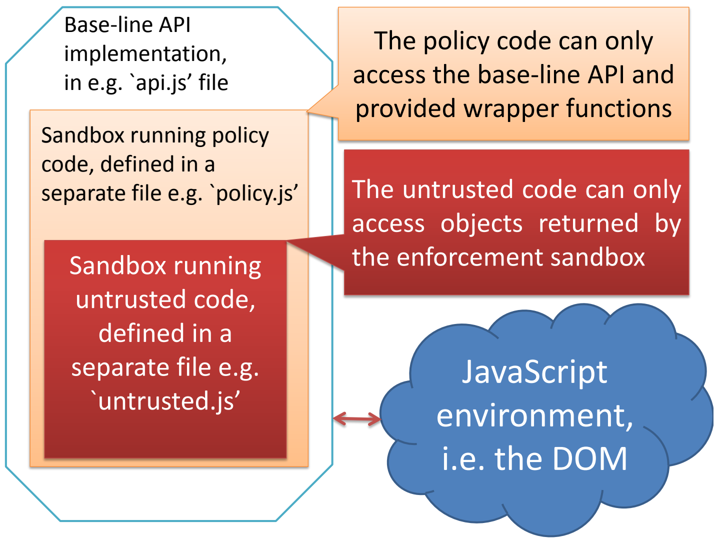
Base-line API implementation, in e.g. `api.js' file
Sandbox running policy code, defined in a separate file e.g. `policy.js'
Sandbox running untrusted code, defined in a separate file e.g. `untrusted.js'
The policy code can only access the base-line API and provided wrapper functions
The untrusted code can only access objects returned by the enforcement sandbox
JavaScript environment, i.e. the DOM

Figure 5: The two-tier sandbox architecture

Our motivation for the architecture is that giving a limited set of allowed operations as coarse-grained policy, a policy developer can define additional, fine-grained restrictions. Hereby, the policy developer can focus on the application-specific policy, rather that coping with the technicalities of achieving the basic set of restrictions and complete mediation.

Outer sandbox The outer sandbox provides a reusable sandboxing layer that enforces coarse-grained security policies, i.e. the outer sandbox limits the set of security-sensitive operations and properties available to the inner-sandbox.

By separating this functionality as a separate and reusable sandbox, the two-tier architecture is able to provide strong baseline guarantees, irrespectively of eventual policy flaws in the inner sandbox.

Also the quite challenging effort of achieving complete mediation in a JavaScript environment [20, 14] can be easily reused over multiple instantiations. Once the outer sandbox achieves complete mediation to a particular security-sensitive operation, this complete mediation is automatically inherited by any inner sandbox.

Inner sandbox The inner sandbox enforces additional application-specific constraints upon access to security-sensitive operations or properties. Since the baseline guarantees are already provided by the outer sandbox, the inner sandbox can provide an expressive policy environment to the policy writer, allowing him to express fine-grained and/or stateful security policies specific to an untrusted application.

In the remainder of this paper, we will focus on fine-grained access control policies, but the proposed security architecture can also be interesting to enforce usage control policies or information flow control policies.

Loading the security architecture Loading the security architecture works as follows. First, the outer sandbox, mediating access between the inner sandbox and the hosting page, loads itself and makes a limited set of security-sensitive operations available to the inner sandbox according to the coarse-grained policy. Next, the inner sandbox loads the untrusted JavaScript code into the second sandbox environment, in which accesses to security-sensitive operations are mediated by the application-specific security policy.

By doing so, accesses from untrusted JavaScript to security-sensitive operations first pass the application-specific policy enforcement of the inner sandbox. Upon approval of the application-specific policy, the call is delegated to the coarse-grained policy enforcement of the outer sandbox. This defense-in-depth approach ensures that the two-tier sandbox architecture, even if application-specific policy get subverted, can always guarantee coarse-grained isolation provided by the reusable outer sandbox, and therefore cannot unintentionally provide accesses to security-sensitive operations or properties on the hosting page.

3.4 Prototype implementation

The prototype implementation of the two-tier sandbox architecture consists of two parts: the security architecture realizing the sandboxing of untrusted code, and the fine-grained policy enforcement mechanism. For realizing the sandbox architecture purely in JavaScript, we employ a sandbox library namely Secure ECMAScript (SES) [3], developed in ECMAScript 5 by the Google Caja Team.

In this section, we briefly introduce the SES sandbox library and then present our prototype implementation of the architecture which is built on top of the SES library. The fine-grained policy enforcement mechanism is discussed in more detail in Section 3.5.

3.4.1 The Secure ECMAScript 5 sandbox library (SES)

Features of the current JavaScript language conspire to make sandboxing nontrivial, and sandboxing untrusted JavaScript code normally requires complex filtering, transforming and wrapping untrusted code to restrict the code to a manageable subset [12]. The ECMAScript 5 specification, released by the ECMA committee in December 2009, has been modified to make sandboxing easier and more widely applicable. ECMAScript 5 (ES5) is a new standard specification of JavaScript language which represents, from a security perspective, a huge improvement over the previous (current) specification, ECMAScript 3. Besides new features such as providing a way (`Object.defineProperty` method) to emulate platform objects, or providing new APIs, ES5 provides more robust programming to write *secure* JavaScript. Firstly, objects in ES5 can be frozen such that the frozen objects are tamper-proof. Secondly, isolation problems in ES3 are solved in ES5 *strict mode*, a restricted subset of ES5. The strict mode creates a restriction on ES5 language to archive two isolation properties: *static lexical scope*, and *no encapsulation leak*. Strict mode provides complete and static lexical scoping by disallowing deletes on variable names, no prototype chain for scope objects, disallowing `with` statements, which provide a mechanism to insert scope objects. In ES3 and most current browsers, there are several channels that untrusted code running within a restricted closure can access the global object by referring to the implicit `this` parameter, or access critical resources by abusing caller-chain. ES5's strict mode repairs these leaks by disallowing these channels to ensure *no encapsulation leaks in a closure*. These restrictions can plug encapsulation leaks that happen in ES3 so that make it possible to implement safe closure-based encapsulation.

Secure ECMAScript 5 (SES) is a sandbox library, developed in ES5 strict mode, that enables the construction of a sandbox without the need for server-side transformation or pre-processing. The main goal of the library is to make untrusted code run inside an isolated environment so that the untrusted code cannot access global variables and the global object but can only have access to the whitelist built-in objects, a provided API (which is essentially mediating access to security-sensitive operations) and the objects created by itself. We refer the readers to [24] for the full detail of the library together with its semantics and soundness. Giving a piece of untrusted code represented as a string variable `untrustedCodeSrc` with an API `api`, once the SES library is initiated in a web page (frame), a sandbox environment can be constructed in the frame as illustrated as follows.

```
var api = {...}; //constructing
var makeSandbox =
      cajaVM.compileModule(untrustedCodeSrc);
var sandboxed = makeSandbox(api);
```

3.4.2 The two-tier sandbox architecture prototype

Built on top of the SES sandbox library, our two-tier sandbox architecture consists of two nested sandbox compartments. As shown in Fig. 5, we assume that the baseline API is developed in a separated file 'api.js', the policy code is specified in 'policy.js' file, and the untrusted code is retrieved from a third-party site and stored at the hosting server in 'untrusted.js' file[6]. Listing 4 illustrates the deployment of the architecture. In this listing, the `api` variable is the baseline API, constructed in the 'api.js' file (included in the hosting page by a `<script>` tag as normal); the 'policy.js' and 'untrusted.js' files are loaded (using `XMLHtmlRequest`) into the `policyCode` and `untrustedCode` variables, respectively.

Creating the baseline API The outer sandbox relies on a baseline API being provided, and we assume that such a baseline API is available (and verified as in e.g. [24, 21]). Given such a baseline API, our architecture realizes modular and fine-grained policy enforcement on top of such a coarse-grained API in a secure manner. Constructing such an API is not a simple

[6]We use this method to get untrusted code in order to load at runtime using *XMLHtmlRequest*. An alternative method is to use the Uniform Messaging Policy (see: http://www.w3.org/TR/UMP/) to request the code directly from the third-party site (cross domain).

```
var outerSandbox = cajaVM.compileModule(policyCode);
var enforcedAPI = outerSandbox(api);
load_untrustedCode(enforcedAPI);
function load_untrustedCode(api){
    var innerSandbox = cajaVM.compileModule(untrustedCode);
    innerSandbox(api);
}
```

Listing 4: The structure of two-tier sandbox architecture

task and out of scope for this paper, but we like to refer to ongoing efforts such as the DOMADO library as part of the Google Caja project.

To validate the results in this paper, we opted to construct a proof-of-concept API which mediates selected accesses to the DOM. This API has been realized as follows.

Firstly, we virtualize a critical object by creating a constructor function and store the original critical object in a reference map pointing to the constructor itself (virtualization is a known technique which has been employed in e.g. Domita [17] or ADsafe [2]). The map object is out of the scope of the constructor function, therefore it is inaccessible from the untrusted code. This can avoid a transformation or static validation of untrusted code as performed in e.g. Domita or ADsafe to prevent untrusted code access special variables storing original critical objects. We use the WeakMap implementation in the SES library [3] to keep such references.

Secondly, we have built the prototype of the constructors with methods and properties having the same name of those of critical objects. In each method or property, we can check the arguments and enforce a static policy to ensure some security properties before invoking or returning the original method or property retrieved from the reference map. The returned object by the original method is also mediated to ensure the complete mediation. Arguments of a method call are also wrapped so that no side-effects can happen.

3.4.3 Tamper-proofing Arguments

Our architecture is based on sandbox compartments in a SES environment of which the soundness and confinement have been proved [24]. The capability of untrusted code, therefore, is confined by the API provided by the enforced objects and the sandbox environment. It means that the untrusted code within the sandbox cannot access arbitrary references except the enforced API.

In a SES environment, built-in objects are frozen so that untrusted code cannot modify a built-in prototype to launch *prototype poisoning* attack on policy enforcement code. *Prototype poisoning* is an attack vector in which the attacker can compromise trusted code by modifying a global prototype that is inherited by the trusted code [15, 14]. Our enforced objects are protected by using `Object.seal(obj)` in ES5 so that existing properties of the object become *non-configurable*, i.e. no property descriptors can be changed, and no properties can be deleted. This is an important improvement since in Mozilla, deleting a wrapped object recovers the original object [20].

3.5 Fine-grained Policy Definition and Enforcement

The main goal of our two-tier sandbox architecture is to define and enforce stateful, fine-grained security policies specifically to a piece of untrusted JavaScript code. As mentioned briefly in introduction, we adopt the lightweight self-protecting JavaScript proposed in [20] for policy enforcement in the inner sandbox in our two-tier sandbox architecture. Security policies in this mechanism are defined in pure JavaScript language in aspect-oriented programming (AOP) style so that they can express stateful and fine-grained policies. Although the self-protecting JavaScript method provides a way to specify and enforce fine-grained policies, it does consider the whole page as untrusted code, except for the policy code itself[7], and therefore cannot be used to define modular policies for portions of untrusted code within a page. In this work we adapt this enforcement mechanism to fit on our two-tier sandbox architecture. Moreover, the implementation of [20] is in current JavaScript specification and faced some vulnerabilities which have been patched in later work [14]. We revisit and revise the issues addressed in [14] to fit in the implementation in the new context of ES5.

Similar to [20], policy enforcement mechanism in this work mediate access to security-sensitive methods and fields. A policy for such a mediator defines if the access is allowed, rejected or modified according to a further policy. Within policy code, a policy writer can define helper functions and variables as security states to keep some execution history of the code, or as some sensitive information such as whitelists. The basic idea of the enforcement is first to keep the reference to the method or property to be mediated, and then execute the policy which decides whether to allow access on the original method or property. Differing from [20], this enforcement is executed within a sandbox environment, therefore local variables and functions are protected.

[7]The policy code is injected into the header of the page to ensure that the policy code is executed first in order to wrap the security critical methods before the untrusted (attacker) code can get a handle on them

Moreover, the enforced object is sealed in ES5[8] so that it cannot be deleted. We present in detail the enforcement for method invocation and property access.

3.5.1 Policy Definition

A policy for method invocation defines whether or not the invocation may proceed depending on some conditions. In our enforcement model, a condition could be based on security states, patterns such as whitelists, and the value of the arguments. We propose two types of policy definition: (1) property access policy and (2) method invocation policy since an object in an API contains properties and methods proxying accesses to the real corresponding object.

```
var document_policy={
 getElementById : {
   method:function(args,proceed){
     var id = args[0];
     if(id === 'main'){
       return proceed(div_Main_policy);
     }
     //.. more cases
   },
   args:['string']
 }
//other properties and methods' definition
}
var div_Main_policy = {
 style: {
   property:{
    read: function(){return div_M_style;},
    write: function(value){return false;}
   },
   args:['*']
 }
//properties and methods' policy definition
}
var div_M_style=..//further policy
```

Listing 5: A policy example for an API object

A *property access policy* includes *read* and *write* policy defined in cor-

[8]In ES5, sealing an object by e.g. *Object.seal(obj)* can set all existing properties of the *obj* object to *non-configurable*, i.e. all property descriptors cannot be changed and all the properties cannot be deleted.

responding functions which returns a boolean value or an object indicating access permission. In the *write* function, the *value* argument is the real value assigning to the property at runtime, which the policy can inspect before writing. Note that these values might be further constrained by some general policies e.g. sanitizing HTML content, in the baseline API, i.e. by the outer sandbox. In the *read* function, we can define further restrictions on the returned value by returning a policy predefined in an object variable which is enforced further on the value. These policies are stateful in which *security states* can be defined and updated runtime to be used by the policies.

A *method invocation policy* of a specific object is defined in a function with two parameters `function(args,proceed){..}`, where *args* contains the arguments of the invocation, and *proceed* is the function to control the execution of the original method. Calling the *proceed*(..) function will allow the original method to be executed. Our policy definition proposal provides a systematic way to write fine-grained and stateful policies depending on invocation arguments (first parameter in the policy function) and security states (can be encoded in variables) at runtime. If the original method returns an object, the object must be enforced by a predefined policy to ensure full mediation. Based on the above assumption on a return object of API call is safe and the fact that the policy writer knows exactly the type of the returned object and which policy should be enforced on the returned object, we provide a way to define this recursive enforcement by calling the *proceed*(..) function with one parameter as the desired policy. This implementation feature is different from [20] because we enforce policies on API objects while the implementation in [20] enforces policies on built-in methods. Listing 5 illustrates a policy example on an API object including a *method invocation policy* (`getElementById` in `document_policy`) that recursively enforces a further policy (`div_Main_policy`) on the return value, and a *property access policy* (`style` in `div_Main_policy`).

Inspecting arguments As mentioned, a policy may need to inspect the invocation arguments, *security states*, and/or patterns such as whitelists. As pointed out in our recent work [20, 14], arguments in JavaScript are *non-declarative*, thus could be the source of attacks [20, 14, 11, 16, 15] because of implicit type conversion in JavaScript when a policy inspects arguments provided by untrusted code. In our previous work [14], we proposed a way to define and enforce declarative arguments by coercing each argument value based on a declared type to ensure that the value when inspecting is the same value when using the argument. This declarative argument approach is applied in this work: the types of the arguments are declared in the `args` field in policy code as e.g. in Listing 5.

Only argument elements declared by the type array can be inspected by the policy and the value is explicitly coerced to the defined type. However, we do not have a type for the return value as in [14] since our policy enforcement is on API objects which are assumed to be safe. Instead, we propose more fine-grained enforcement for the returned object as argued above, whereby it is possible to specify a policy for a returned object in order to recursively enforce full mediation.

3.5.2 Enforcement Method

Our enforcement method is implemented in a whitelist manner, i.e. only methods and properties defined in the policy are accessible, the other are absent from the enforced object. We provide an interface to enforce a policy on an object. The key functionality of the interface is to traverse the policy to get all the names of methods and properties together with the policy in order to enforce the policy on the same name of the object. The methods and properties of the object not defined in the policy are redefined as an empty function or null value so that they are not accessible from untrusted code. Listing 6 illustrates part of the implementation of the interface.

```
function enforceWhitelistPolicies(object, policies){
  Object.keys(policies).forEach(
    function(name) {
    //inspect if the element is a method,
    //get the corresponding policy and types
      if(method)
        wrapMethod(object,name,policy,types);
      else //property
        wrapProperty(object,name,readPolicy, writePolicy,types);
  });
  //iterate the object to make the methods
  //and properties that are not defined in
  //policies unaccessible
  ...
  return Object.freeze(object);
};
```

Listing 6: Policy enforcement

A method call policy and a property access policy are enforced slightly different. We explain in detail the two enforcement mechanism as below.

```
function wrapMethod(object, method, policy,types){
 //..find function for possible aliases
 var original = object[method];
 object[method] = function() {
   var polArgs=//..clone arguments by the defined types
   var proceed= function(policies) {
     var result=//execute original func
     if(!policies) return result;
   return enforceWhitelistPolicies(result, policies);
   }
   return policy(polArgs, proceed);
 }
 return object[method];
}
```

Listing 7: Enforcing a method invocation

Enforcing method call policies We adapt our previous enforcement implementation [14] to handle the enforcement for returned object. In summary, the enforcement for a method invocation policy is a wrapper that keeps the reference to the original method of the object to be wrapped, and redefines the method by invoking a policy function which can control the execution of the original method. As described, a policy is defined as a function with two arguments: the first argument is the parameters of the invocation, the second argument is the `proceed` function, representing the reference to the original function; calling the `proceed` function will execute the original method. We modify the `proceed` function (from [14]) to take one argument as a policy for the returned object of the original method. If this policy is defined (from the policy to be enforced), the returned object will be recursively enforced by the provided policy. The simplified snippet of this interface is illustrated in Listing 7.

Enforcing property access policies Our enforcement on property access policies relies on the `Object.defineProperty(..)` method in ES5 to enforce desired policies. We first get the current getter-setter functions of the property of the object (using `Object.getOwnPropertyDescriptor(..)`) to be enforced. We then define a new descriptor with getter and setter functions to execute *read* and *write* policy functions from the policy so that these policy functions are always invoked whenever the property is accessed. Depending on runtime policy results from the *read* and *write* policy functions, the original *getter* and *setter* functions may be called to run in the context of the object. The returned value will be further enforced if there is a policy

```
function wrapProperty(object, property, read, write, type){
  var desc = Object.getOwnPropertyDescriptor(object, property);
  //..assert desc object
  var newdesc = {
   get: function() {
    var readPolicy = read();
    var value = desc.get.call(object);
    if (readPolicy===true) return value;
    if (typeof readPolicy==='object')
      return enforceWhitelistPolicies(value, readPolicy);
   },
   set: function(v) {
    var cloneValue = coerceByType(type,v);
    var writePolicy = write(cloneValue);
    if(typeof cloneValue==='object')
      cloneValue = combine(cloneValue,v);
    if(writePolicy===true)
      return desc.set.call(object,[cloneValue]);
  },
  configurable: false,
  enumerable: true
  }
  Object.defineProperty(object,property, newdesc);
};
```

Listing 8: Enforcing a property access with a policy

defined in *read* policy function. Listing 8 shows this enforcement mechanism.

3.6 Validation

To validate the feasibility and security of our two-tier architecture, we have applied our prototype implementation in various application scenarios, in which untrusted third-party JavaScript code get integrated (e.g. gadget integration in web mashups). In this section, we will report on one particular case study, the integration of online advertisements. This case study captures the representative characteristics of context-sensitive text advertisement services such as AdBrite and Google Adsense.

The policy we want to enforce on the untrusted advertisement code is the following. First, we want to restrict the untrusted code to only write to a particular subset of the page (i.e. one particular div element, where the ad will be displayed). Next, we want to restrict the DOM read access to a particular subset of the page, so that only that part of the page is used in the context-sensitive analysis of the untrusted code. Moreover, we want to disable this reading access as soon as the user enters data into the page (e.g. by filling in an input form). These are just simple policies but represent application-specific and stateful policies. For example, we can define specific elements that the ad can read or write, or define a security state to monitor if the user enters the data.

To enforce these policies on context-sensitive advertisement scripts, we first select the basic API of the outer sandbox to which the application-specific policies and untrusted code will be confined. Next, we define fine-grained, stateful policies to enforce on the untrusted code.

The baseline API enables in a coarse-grained and application-independent way the set of features that are accessible within the outer sandbox (i.e. that can be used by the application-specific policy as well as the untrusted code). In our online advertisement validation experiment, we have chosen to only provide a very limited API, namely access to DOM operations. The baseline API is constructed via virtualization as briefly mentioned in Section 3.4.2. The technique constructs mediator objects by creating virtual objects which provide predefined methods and properties to mediate accesses to the DOM.

Next, we define the application-specific policy described informally above. Similar to the baseline API, we construct mediator objects to mediate read and write access to the DOM. The policy code also subscribes to the keyboard events e.g. *keydown* to capture any user input. The mediator object for read access grants access based on the requested DOM element and whether a user input event was captured, write access is granted solely on the DOM element specific to the ad. The policy allows the ad to set the style, width, height of the ad area and also restricts the maximum value of width, height so that the ad can not display an oversize area. Listing 9 illustrates some of these policies.

```
var data_read = false;//a security state
var div_Main_policy = {
  innerHTML: {
    read: function(){
      if(dataread)
        return fasle;
      return true;
    },
    //... other definitions
};
var user_input = {
  addEventListener : {
    method:function(args,proceed){
      var eventStr = args[0];
      if(eventStr==='keydown'){
        dataread = true;
        return proceed();
      }
    },
    args:['string','function','boolean']
  },
  //... other definitions
};
```

Listing 9: Application-specific and stateful policy examples for untrusted ad

The context-sensitive advertisement script and the application-specific policy are deployed in two separate code files, and loaded into the two sandbox environments as described in Section 3.3. This case study has been successfully tested on Mozilla Firefox 4.0.1 on a Windows 7 platform.

In addition, various security tests have been performed to assess the security guarantees by our proposed architecture. Based on known attack vectors and vulnerabilities in previous solutions (as described in e.g. [15, 16, 14]), we assessed whether untrusted code could break out of the sandbox environment.

For reference, we have first executed the attack vectors in the hosting page to ensure that the attacks are successful. We then have deployed the attack vectors into a sandbox environment with an API without any enforcement to ensure the API provides adequate functionalities and the attacks are successful. Finally, we have deploy the malicious script into the sandbox environment with an API enforced by above defined policies.

We did not success to break out of our proposed two-tier security architecture; the malicious script execution is prevented by two-tier sandbox enforcement. This result was to be expected, since our two-tier architecture relies on the same foundations and security guarantees of the Secure ECMAScript library (SES) [3].

3.7 Related work

Solving security issues for untrusted JavaScript has recently received wide attention both in industry and in the research community. However, most of the recent work concern the context of current version of JavaScript (ECMAScript 3). Proxy [27] is a recent approach in ECMAScript 5 to construct robust APIs. Although this approach does not allow to specify modular and flexible policies, it can be used to construct a robust API as a baseline API library for our approach, providing a complete framework for the DOM access for untrusted code. To the best of our knowledge, our work is the first study in enforcing *fine-grained security policies* for untrusted JavaScript in ECMAScript 5. In [23], the authors have reviewed current security mechanisms for untrusted JavaScript in the literature. In this section, we only review recent work related to fine-grained policy enforcement and sandboxing mechanisms. We divide the related work based on whether it requires browser modification.

Browser-level implementations Browser-level implementations have access to the lower-level implementation of the JavaScript interpreter, and therefore have the possibility to modify or extend the semantics of JavaScript to provide greater security. However, this approach also has down side from an immediate practical perspective. It requires the browser users to be proactive to protect themselves. From a technical point of view, modifying a browser requires much effort. Moreover, the implementation is likely to change more frequently since the codebase of browsers e.g. Firefox normally change rapidly.

JCShadow [18] is a recent work (and closest to our work) that also motivates for fine-grained policy enforcement for untrusted JavaScript, and proposes a reference monitor within a JavaScript engine to enforce policies. The mechanism is implemented by modifying the JavaScript engine in Firefox 3.5.

ConScript [16] modifies Internet Explorer 8 to provide aspect-oriented programming constructs for JavaScript in order to enforce fine-grained security policies. ConScript can enforce edit automata [10] policies which is essentially the same class covered by our policies.

Inspired by ConScript, WebJail [26] applies the deep aspect weaving technique to FireFox browser, and introduces a least-privilege composition policy on top of this security architecture. This secure composition policy is based on an analysis of security-sensitive operations in the upcoming HTML5 specification, and provides a whitelist-based approach to nine disjoint categories of sensitive operations (such as external communication and client-side storage).

There are also several other approaches such as [5, 7] using browser modification to enforce policies. However, these methods can only enforce coarse-grained access control policies which are not applicable to untrusted script scenarios.

On the other hand, approaches to enforcing security policies without modifying browser have advantage in themselves. The enforcement can be provided as a library by a server or a proxy and the policies are enforced at runtime at the browser. One branch in this area is to modify the original program or restrict untrusted code in a safe subset while the other deploys *non-invasive* approach to original code.

Code transformation and safe subsets BrowserShield [22], Caja [17], and Facebook JavaScript [4] are examples of the approaches using code modification or filtering. BrowserShield [22] is an approach using code transformation dynamically to enforce security policies. The idea of BrowserShield is further developed at Microsoft Live Labs as a Web Sandbox framework [9] which rewrites untrusted JavaScript to run it inside an isolated virtual machine which mediates access to the real JavaScript environment.

Google Caja [17] is another approach to enforcing policies of a web page on the client side. Caja defines a safe JavaScript subset based on *object-capability* model. Untrusted JavaScript code is transformed into a safe version with isolated modules by a rewriting process. The transformed code is provided APIs by libraries such as Domita to have indirect access to the DOM. However, Caja does not support fine-grained policies enforcement as we investigate in this work.

Similar to Caja, Facebook JavaScript (FBJS) [4] is an another industrial approach to sandboxing untrusted JavaScript application embedded into Facebook. Untrusted code written in FBJS is also transformed in a separate namespace so that it is isolated to the other.

Maffeis *et al* proposed another approach [11] for untrusted JavaScript which uses filtering, rewriting, and wrapping to isolate the untrusted code. Although these mechanisms have proved the soundness by semantics or automated tools [12, 24], they limit untrusted code into a subset of JavaScript and do not allow developers to specify application-specific and fine-grained policies as we investigate in this work.

ADsafe [2] is another safe subset of JavaScript to allow untrusted advertisements executing on a trusted hosting page. The safe subset is an interface that mediates access to the DOM and other global variables to ensure that the untrusted code cannot perform malicious behaviors. Before placing an untrusted ad code into a hosting page, the ad code must be validated by an static analysis tool called JSLint to ensure that the untrusted code only has access to the interface provided by the ADsafe library. The soundness of API confinement of ADsafe has been shown in [21, 24]. Although this mechanism do not allow to define fine-grained policies, the ADsafe subset could be provided as a baseline API in our architecture so that the untrusted code can be loaded and executed dynamically without code validation by an off-line tool.

Non-invasive approaches Non-invasive approach to enforcing security policies is exemplified by the lightweight self-protecting JavaScript method [20]. This method defines a wrapper library in aspect-oriented programming [8] style to intercept built-in functions with a security policy. The library is placed at the header of a page so that it can execute first to wrap sensitive function calls and property accesses, and therefore to make the web page self-protecting. The implementation of this method faces several challenges which have been addressed in a later work [14]. However, the implementations in [20, 14] focus on enforcing policies on built-in objects which is at page-level while our architecture is to enforce policies on API objects for untrusted code. In our work, we revisit the implementation in the context of ECMAScript 5, and adapt and revise the implementation by the new advantage features of ES5.

ObjectViews [15] is a similar approach to our work which provides wrappers as a library in JavaScript to share objects among principals in the browser. In untrusted code context, ObjectViews [15] focuses on safe sharing of objects (in ES3) between privileged code and untrusted code. However did not discuss how to load and execute untrusted code as we investigate in this paper.

Similar to our case study of context-sensitive advertisement application, AdJail [25] is an approach to isolating an ad script into a hidden iframe (shadow page) which is enforced by the same-origin policy. The ad script interact with the hosting page through tunnel scripts in both frames which can enforce to confidentiality and integrity policies. However, the framework only supports limited coarse-grained access control policies.

3.8 Discussion and future work

Our two-tier sandbox architecture is built on the specification of ECMAScript 5 and its *"strict mode"* to provide modular and fine-grained security policies for untrusted JavaScript code. The implementation of the architecture is based on an existing technique [20], and a library [3], however, improving on both of these. In particular, the two-tier sandbox architecture allows application-specific and fine-grained security policies be defined and enforced modularly, which is lacking in [3], and allow the enforcement on untrusted code, which is missing in [20]. Moreover, the policy enforcement mechanism is also executed within a sandbox so that the policy code cannot expose unprotected resources. This improves on both [3] and [20] by providing a fail-safe for badly written policies without need for complex policy language development. To the best of our knowledge, our security architecture is the first study in enforcing fine-grained security policies for untrusted JavaScript in ECMAScript 5[9]. In summary, the architecture is unique in the sense that it enforces application-specific and stateful fine-grained security policies for untrusted JavaScript code without browser modification or pre-processing of the code. In addition, the baseline API in the outer sandbox ensures a failsafe fallback in case of badly written policies.

Not all third-party code available in the wild supports yet the strict mode of ECMAScript 5, but we believe that this will be shifting quite rapidly. As ECMAScript 5 becomes available in all major browsers, and multiple content providers (such as Google and Yahoo) already favor the strict mode. Moreover, the API of the SES sandbox library is currently under proposal by the ECMA committee (TC 39) to be included as built-in features in a future version of ECMAScript [24], our work demonstrates the use of SES and provides a step towards a mechanism for executing untrusted code with application-specific and fine-grained security policy enforcement.

In future work we will further contribute to a robust baseline API, and we plan to validate our two-tier sandbox architecture on a broader range of real-life applications, by automatically injecting the client-side architecture for existing untrusted third-party code.

[9]IceShield [6] is a very recent ECMAScript 5 library inlined to a page to detect and prevent malicious behaviour of the page. However, similar to our lightweight self-protecting JavaScript approach, this library does not separate between trusted and untrusted code

References

[1] Dasient Blog. Q1'10 web-based malware data and trends. http://blog.dasient.com/2010/05/. May 10, 2010.

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

[3] Mark S. Miller et al. Secure EcmaScript 5. http://code.google.com/p/es-lab/wiki/SecureEcmaScript. Accessed in September 2011.

[4] Facebook. Facebook JavaScript. http://developers.facebook.com/docs/fbjs.

[5] Oystein Hallaraker and Giovanni Vigna. Detecting Malicious JavaScript Code in Mozilla. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Mario Heiderich, Tilman Frosch, and Thorsten Holz. IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, RAID'11, 2011.

[7] 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.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[9] Microsoft Live Labs. Web Sandbox. http://www.websandbox.org/. Accessed in May 2011.

[10] Jay Ligatti, Lujo Bauer, and David Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[11] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *ESORICS*, pages 505–522, 2009.

[12] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *Proc of IEEE Security and Privacy'10*. IEEE, 2010.

[13] Sergio Maffeis and Ankur Taly. Language-Based Isolation of Untrusted JavaScript. In *CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 77–91, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *T. Aura, K. Jarvinen, and K. Nyberg (Eds.): The 15th Nordic Conference in Secure IT Systems (NordSec 2010), LNCS 7127*, pages 239–255. Springer-Verlag, 2012. (Selected papers from OWASP AppSec Research 2010, June 2010, Stockholm, Sweden).

[15] Leo Meyerovich, Adrienne Porter Felt, and Mark Miller. Object Views: FineGrained Sharing in Browsers. In *WWW2010: Proceedings of the 16th international conference on World Wide Web*, New York, NY, USA, 2010. ACM.

[16] Leo Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *SP '10: Proceedings of the 2010 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010.

[17] Mark S. Miller, Mike Samuel, Ben Laurie, and Ihab Awad Mike Stay. Caja: Safe active content in sanitized JavaScript. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

[18] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards Fine-Grained Access Control in JavaScript Contexts. In *Proceedings of the 31st IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2011.

[19] Phu H. Phung and Lieven Desmet. A two-tier sandbox architecture for untrusted javascript. In *Proceedings of the Workshop on JavaScript Tools (JSTools '12)*, pages 1–10. ACM, June 2012.

[20] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight Self-Protecting JavaScript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, Sydney, Australia, 10 - 12 March 2009. ACM.

[21] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-Based Verification of JavaScript Sandboxing. In *20th USENIX Security Symposium*, 2011.

[22] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.

[23] Philippe De Ryck, Maarten Decat, Lieven Desmet, Frank Piessens, and Wouter Joosen. Security of Web Mashups: a Survey. In *T. Aura, K. Jarvinen, and K. Nyberg (Eds.): Information Security Technology for Applications, The 15th Nordic Conference in Secure IT Systems (NordSec 2010), LNCS 7127*, pages 223–238. Springer-Verlag, 2012.

[24] Ankur Taly, John C. Mitchell, Ulfar Erlingsson, Jasvir Nagra, and Mark S. Miller. Automated analysis of security-critical javascript apis. In *Proc of IEEE Security and Privacy'11*. IEEE, 2011.

[25] 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*, 2010.

[26] 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 *ACSAC*, December 2011.

[27] Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 59–72, New York, NY, USA, 2010. ACM.

4 JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications[10][11]

4.1 Introduction

In the last decade, the web platform has become the number one platform on the Internet. There is a clear paradigm shift from desktop applications and proprietary client-server solutions towards web-enabled services. An important catalyst for this paradigm shift has been the power of JavaScript as well as the advent of HTML5, giving web developers the tools to build rich and interactive websites.

As a consequence of this enormous growth in popularity, the web has also become the primary attack platform: SANS [31] reported in 2009 that more than 60% of all cyber attacks are aimed at web applications, and more than 80% of discovered vulnerabilities are web-related. A whole range of web attacks exists in the wild, ranging from Cross-Site Scripting, Cross-Site Request Forgery and SQL injection to the exploitation of broken authorization and session management. This paper focuses on one particular and important class of web attacks, namely attacks due to the insecure integration of JavaScript.

To enrich the functionality and interactivity of a website, a common and wide-spread approach is to integrate JavaScript from third-party script providers. Recent studies [41, 24] have shown that 96.9% of websites include scripts from external sources, and on average each website includes scripts from 3.1 external sources. For example, websites integrate among others JavaScript-enabled advertisements (such as Google AdSense and adBrite), Web analytics frameworks (such as Google Analytics, Yahoo! Web Analytics and Tynt), web widgets and buttons (such as Google Maps, addToAny button and Google +1 button), and JavaScript programming libraries (such as jQuery and Dojo).

[10]This paper has been published as [1]: Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, Frank Piessens, JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications, Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC 2012), pages 1-10, Orlando, Florida, USA, 3-7 December 2012

[11]This research is partially funded by the Research Fund KU Leuven, the EU-funded FP7 projects NESSoS and WebSand and by the IWT-SBO project SPION. Pieter Agten holds a Ph.D. fellowship of the Research Foundation - Flanders (FWO). With the financial support from the Prevention of and Fight against Crime Programme of the European Union European Commission - Directorate-General Home Affairs.

The de facto browser security model today is defined by the Same-Origin Policy (SOP). The SOP restricts access of client-side scripts to resources belonging to the same origin[12]. For instance, the SOP ensures that document data and cookies from one origin cannot be read by scripts belonging to another origin. However, the SOP includes some important relaxations with respect to navigation and content inclusion (e.g. embedded images and scripts) [42]. In particular, if a page from one origin includes a script from another origin, the included script is treated as if it belongs to the including origin, and hence it inherits all the capabilities and permissions of the hosting page. This makes malicious script inclusion a very powerful attack vector.

Several countermeasures have been proposed to limit the capabilities of third-party JavaScript, including (1) the introduction of safe subsets of JavaScript [34, 4, 16], (2) client-side reference monitors [18, 35], and (3) server-side transformations of the script to be included [21, 33]. However, all of these have at least one of the following limitations.

First, some approaches [18, 35] require intrusive *browser modifications*, in particular to the JavaScript engine and the binding between browser and JavaScript engine. Such modifications hinder short-term deployment of the countermeasure.

Second, some approaches do *not support client-side script inclusion*: in order to perform server-side pre-processing (e.g. source-to-source translation or filtering) of the scripts, the scripts have to pass through the web server [21, 34, 4]. This effectively changes the architectural model of client-side script inclusion to server-side script inclusion.

Third, some approaches do *not provide complete mediation* between different scripts on the same page, or to all resources exposed in the browser. Self-Protecting JavaScript (SPJS) [27, 17] assumes that all scripts included on a hosting page need identical security constraints. It does not differentiate between different external scripts nor between local and remote inclusions. AdJail [33] successfully isolates untrusted advertisements from the Document Object Model (DOM) of the hosting page, but since it uses iframes as isolation units, it cannot fully protect security-sensitive APIs such as XHR, Geolocation and local storage.

Inspired by recent advances in achieving object-capability guarantees for JavaScript [21, 15, 22, 26, 9], this paper presents JSand, a novel security architecture to securely integrate third-party JavaScript. We improve upon the state-of-the-art with the following contributions:

1. JSand is the first JavaScript sandbox that (1) does not need browser

[12] An origin is a (protocol, domain name, port) tuple.

modifications, (2) supports client-side script inclusion and (3) completely mediates different scripts and the browser APIs.

2. We show evidence that JSand is secure, compatible with complex and widely used scripts (such as Google Maps, Google Analytics and jQuery) and performs sufficiently well.

The rest of this paper is structured as follows. Section 4.2 introduces the necessary background and defines the problem statement. In Section 4.3, the JSand architecture is presented, and Section 4.4 discusses several relevant implementation aspects. Section 4.5 evaluates the security, compatibility and performance of JSand. Finally, Section 4.6 discusses related work, and we conclude in Section 4.7.

4.2 Problem statement

4.2.1 Integrating third-party JavaScript

To enrich the functionality and interactivity of a website, a common and wide-spread approach is to integrate JavaScript from third-party script providers. The two most wide-spread techniques to integrate third-party JavaScript in web pages are through script inclusion and via iframe integration [7].

Script inclusion HTML script tags are used to execute JavaScript as part of a web page. If the JavaScript code is integrated from an external source, the browser will still execute the code within the security context of the web page, without any restrictions of the SOP.

Iframe integration HTML iframe tags allow a web developer to include one document inside another. The advantage of iframe integration is that the integrated document is loaded in its own security context: integrated content from another origin is isolated from the integrating web page by the SOP.

Script inclusion is the de facto script integration technique on the web, both for local scripts as well as for external scripts. The iframe integration technique is used for web gadgets that don't have strong integration needs with the embedding web page, or have an out-of-band service-to-service communication channel (such as the Facebook Like button or Facebook Apps). In the remainder of this paper, we focus on third-party JavaScript integration through script inclusion.

4.2.2 Malicious script inclusion

The browser security model for integrating third-party JavaScript is problematic. Once included in a website, a malicious script cannot only access all the document data and cookies, but with the advent of HTML5, the malicious script has also access to local storage data (e.g. Web Storage, IndexedDB), intra-window communication (Web Messaging), remote resource fetching via XHR and user-consented privileges (such as Geolocation, media capture, access to System Information API, and many more). This makes malicious script inclusion a very powerful attack vector. One can distinguish between two types of attackers.

Malicious script provider The script provider has malicious intentions (but covers up by providing appealing functionality to potential customers), or becomes malicious over time (e.g. intentionally, or by selling out or quitting his business [24]).

Benign script provider under attack The script provider has no malicious intentions, but the scripts delivered to its clients become under control of an attacker. This can be due to the inclusion of other untrusted resources (e.g. in advertisement networks), due to a bug in the delivered script (e.g. a DOM-based XSS vulnerability [13]), due to a server-side take-over (e.g. via SQL injection) or due to in-transit tampering with the scripts by a network attacker.

In both cases, the attacker controls the scripts included by the hosting page, and by default gains full access to the execution environment of the web page.

4.2.3 Requirements

Given the wide spread of script inclusion and the increasing impact of malicious script inclusion, there is a clear need for a novel security architecture to **securely integrate third-party JavaScript**, but **without introducing disruptive changes**. Preserving backwards compatibility is crucial in the web context. We therefore identify the following requirements:

R1 Complete mediation All access to security-sensitive functionality should be completely mediated by the security mechanism. This includes access to the DOM, as well as security-sensitive JavaScript APIs (such as Geolocation and local storage). The attacker must be unable to circumvent the security mechanisms in place.

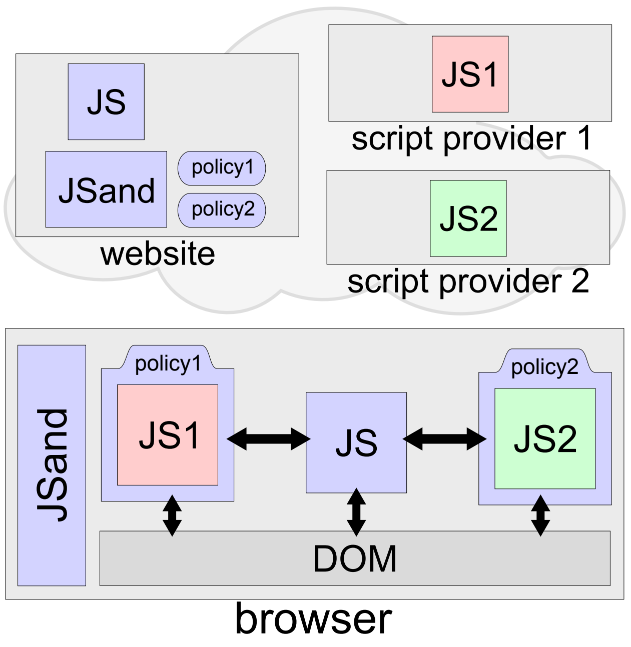

Figure 6: the JSand architecture. Inside the browser, all access from JSand sandboxes to the JavaScript environment is mediated according to server-supplied policies.

R2 Backwards compatible The security mechanism should seamlessly operate in the current web ecosystem, i.e. it should not rely on browser modifications or disable the direct delivery of scripts from the script provider to the browser. In addition, the security mechanism should support the integration of legacy scripts.

R3 Performance The security mechanism should introduce only a minimal performance penalty, unnoticeable to the end-user.

4.3 JSand security architecture

The JSand architecture enables the owner of a website to securely integrate third-party scripts, without needing disruptive changes to either server-side or client-side infrastructure. We first give a high-level overview of the architecture and then discuss the architectural choices under the hood.

4.3.1 Architectural overview

Figure 6 depicts the JSand architecture. A website owner deploys JSand by including the JSand JavaScript library in his webpages. When one of these pages is loaded in a visitor's browser, the third-party scripts to be sandboxed are fetched directly from the servers of the script provider. The JSand library confines each script to its own secure sandbox, which isolates the script from other scripts and from the DOM.

4.3.2 Under the hood

The JSand architecture is based upon the secure confinement of third-party JavaScript. JSand realizes this through the use of an object-capability environment. Such environment provides an appropriate device for isolating untrusted JavaScript: without an explicit and unforgeable reference to a security-sensitive resource (i.e. an object or a function), a script is unable to access the resource or make use of its capabilities. The object-capability model is at the basis of Caja [21], and many other safe subsets of JavaScript [15].

The JSand library invokes third-party scripts, initially giving them only a minimal set of unforgeable references. To maintain control over all references acquired by a sandboxed script, JSand applies the Membrane pattern proposed by Miller [23]. Our implementation of this pattern consists of placing policy-enforcing wrappers around objects that provide potentially security-sensitive operations. Whenever one of these objects returns a reference to another object, the membrane is extended to cover that object as well. This ensures a sandboxed script never has direct access to a security-sensitive operation.

The membrane's wrappers intercept all operations performed on the objects they wrap and hence implement the security policy enforcement points. On each enforcement point, the wrapper consults the security policy to determine whether or not the corresponding operation is permitted. If not, this will be indicated by the security policy and the operation will be blocked. The architecture is not bound to any specific type of security policy, which gives website owners the freedom to enforce arbitrarily complex policies.

Since all interactions between a script and the browser are performed by calling DOM methods, it suffices to place a wrapper around each DOM object in order to enforce a policy on all security-sensitive operations. These include not only operations to read or modify content of the hosting page, but also to communicate with other scripts and to use browser-provided JavaScript APIs.

In conclusion, the JSand architecture provides an end-to-end solution for securely integrating third-party JavaScript scripts on a website. The website owner is able to define and enforce security policies on scripts, which puts him back into the driver's seat. JSand does not require disruptive changes to the architecture of the web: it does not break direct script delivery towards the browser, and can be deployed without additional server-side or client-side infrastructure. The combination of the object-capability model and the Membrane pattern ensures that all access from a sandboxed script to security-sensitive operations passes through a membrane's wrappers, which enforce the security policy.

4.4 Prototype implementation

This section reports on the development of a mature JSand prototype, which is designed to work in ECMAScript 5 (ES5) compatible browsers with support for the proxy features of the upcoming ES Harmony standard. The current prototype runs seamlessly in Google Chrome v20.0.1132.21.

In Subsections 4.4.1 and 4.4.2, we present the client-side technology for executing third-party JavaScript in a confined sandbox. Subsection 4.4.3 describes the type of security policies that are enforced. Next, Subsection 4.4.4 illustrates how access to security-sensitive operations is completely mediated. Subsection 4.4.5 discusses how our prototype deals with dynamic script loading and Subsection 4.4.6 describes a set of automatic script transformations to improve compatibility with legacy scripts.

4.4.1 Object-capability system

As described in Section 4.3.2, the JSand architecture relies on an object-capability environment to provide complete mediation. The ECMAScript language does not qualify as an object-capability language by itself. For instance, any script has access to all global variables by default, and consequently has capabilities that are not under control of any security framework. However, in 2008, Miller et al. [21] identified a subset of ES3 which forms a true object-capability language. More recently, the Google Caja team has identified a subset of ES5 strict, named Secure ECMAScript (SES), that also provides such an object-capability language. Moreover, they have developed a JavaScript library that enables the execution of SES on ES5-compatible browsers [22]. This library provides methods for safely evaluating SES-compliant code in an isolated environment. A key feature of the library is that it can execute completely at the client side and hence doesn't rely on any custom server-side architecture. JSand uses the SES library to realize its underlying object-capability environment.

However, since SES is a subset of ES5 strict, which in turn is a subset of ES5 non-strict, not all currently deployed JavaScript scripts are SES-compliant. Furthermore, the language supported by the SES library differs from true SES in several minor ways, further reducing compatibility with legacy scripts. Two important incompatibilities between ES5 and the SES-like language supported by the SES library are described below.

Global variables In ES5, the global `window` object can have arbitrary properties and for each of these properties there is a corresponding global variable with the same name. Conversely, for any global variable, a corresponding property with the same name is defined on the global object. In SES, this is no longer the case: global variables are not aliased by properties on the global object or vice versa.

Strict mode SES enforces strict mode for all scripts. Hence, ES5 non-strict code might be incompatible with SES. For instance, strict mode drops support for the `with` keyword, prevents the introduction of new variables into the outer scope by an `eval` and no longer binds `this` to the global object in a function call.

SES was designed to support (only) recognized ES5 best practices. Therefore, scripts that adhere to these best practice standards are SES-compliant and hence we expect the number of fully SES-compliant scripts to increase progressively as these best practices become more widespread. Although not all legacy scripts run without errors under the SES library, the secure confinement of these scripts is never at stake. Nevertheless, we have developed a support layer to improve compatibility with legacy scripts. This layer is described in detail in Section 4.4.6.

To enforce the object-capability model and to provide support for legacy scripts, the SES library and the support layer need access to the source code of scripts to be sandboxed. Our prototype fetches this code using the XMLHttpRequest (XHR) API. By default, this API is subject to the SOP, but recently added web features have facilitated cross-domain interactions, namely Cross-Origin Resource Sharing (CORS) [38] and the Uniform Messaging Policy (UMP) [39]. In case CORS or UMP are not supported by the script provider, our solution can fall back to a server-side JavaScript proxy [40].

4.4.2 Policy-enforcing membranes

The Proxy API To implement the Membrane pattern in an efficient and transparent way, our prototype uses the Harmony Proxy API, which is scheduled to be standardized in the next version of ECMAScript [36]. This API enables us to create wrappers that generically intercept all property accesses and assignments on specific objects, as shown in the code below.

```
function wrap(target, policy) {
  var handler = {
    get: function(proxy, propertyName) {
      if (policy.isGetAllowed(propertyName)) {
        return target[name];
      }
      return null;
    }
    set: function(proxy, propertyName, value) {
      if (policy.isSetAllowed(propertyName)) {
        target[name] = value;
        return true;
      }
      return false;
    }
  }
  return Proxy.create(handler, Object.getPrototypeOf(target));
}
```

The `wrap` function creates a simple policy-enforcing wrapper around a specific `target` object. All property accesses and assignments on this wrapper are intercepted by the `get` and `set` traps of the handler object, which uses the `policy` object to determine whether or not the access or assignment is allowed.

Membrane implementation To implement the Membrane pattern, the handlers used in JSand transitively wrap all objects they return from the `get` trap and unwrap the objects they receive in the `set` trap. The entire prototype chain of a wrapper must be wrapped as well, to prevent an attacker from piercing the membrane by accessing an unwrapped prototype.

If an object to be returned from the `get` trap is a function, a function proxy that wraps the original function is returned. This function proxy first unwraps all its arguments, then calls the original function using these unwrapped arguments and finally wraps the return value before returning it to the caller, thereby further expanding the metaphorical membrane. Some methods, such as `window.addEventListener`, take a callback function as an argument; like all other arguments, this callback must be wrapped appropriately to uphold the membrane. Because a callback function is executed in the context of a sandbox, its wrapper must wrap each of its arguments and must unwrap the return value after calling the original function with the wrapped arguments.

Each sandbox keeps a mapping from its wrappers to the target objects they wrap and vice versa. This makes it possible to unwrap previously wrapped objects and to ensure that there is at most one wrapper (per sandbox) corresponding to each target object, making the membrane identity-preserving [5]. The mapping from wrappers to their corresponding targets is only accessible from outside the sandbox, for otherwise an attacker could use it to escape from the sandboxed environment.

Whereas sandboxed code should always be confined to the bounds of its own sandbox, many use cases require an operation to introduce code from outside a sandbox into an existing sandbox. Such operations enable a website owner to extend or interact with a sandboxed script. JSand sandboxes provide two functions for introducing new code into them: `innerEval(code)` and `innerLoadScript(url)`. The first function evaluates a literal code string, while the second loads a script at a given URL.

In conclusion, the Membrane pattern transparently isolates a sandbox from code running outside of it or in other sandboxes. Since the handlers intercept each property access and assignment made on a wrapper, they contain the enforcement points which consult the security policy to determine whether or not an operation is permitted.

4.4.3 Security policies

Defining good security policies is important for ensuring the secure confinement of sandboxed scripts. To avoid needing a *known-good* version of a script to be sandboxed, a policy should be based on the claimed functionality of a script, as opposed to being based on actions performed by any specific version of the script. Generic templates can be provided to support website owners in defining good security policies.

Since the JSand architecture is independent of the specific type of security policy to enforce, policies can range from simple stateless policies, to arbitrarily complex policies. In both cases, the security policy can be specified as a JavaScript function that takes information about the operation to be performed as input and returns a boolean indicating whether or not the operation is allowed. We discuss three types of policies in more detail below.

Stateless policies Stateless policies determine whether or not an operation is permitted based on information associated with that operation alone. For instance, a stateless security policy could specify that a specific function call performed on a specific object is only allowed when the value of the first argument is on a predefined whitelist.

WebJail [35] is an example of such a stateless policy for securely integrating third-party JavaScript. It classifies security-sensitive operations into nine categories, including DOM access, cookies, external communication, device access, etc., which can be permitted or blocked individually. A WebJail policy is based on a static whitelist of each of these categories, and could easily be implemented with JSand.

Stateful policies Stateful policies can accumulate internal security state over multiple calls and use this global state as part of the policy, in addition to the local information made available on each operation request. For instance, a stateful security policy could specify that the use of XHR is allowed as long as no cookies have been read. This type of policy is more expressive than its stateless counterpart, but it is also more complex to specify and more prone to mistakes.

The shadow page in AdJail [33] is another example of internal state that could be accumulated over multiple calls. This page represents a ghost DOM, which is not directly rendered to the user, but allows an advertisement to execute various DOM operations in a confined environment.

Advanced policies More complex policies can be used to enforce more advanced security properties, such as information flow security. One example of this is a set of policies to implement *noninterference* through *secure multi-execution* (SME) [8, 6]. For any script, SME can classify each input and each output channel as either H (high security, confidential) or L (low security, public). A script is noninterferent if its low-level outputs are not influenced by high-level inputs. Consider for instance the following script on a webserver at `mydomain.com`.

```
1 var cookies = document.cookie;
2 document.getElementById('some-img').src =
      'http://attacker.com/img.jpg?c=' + escape(cookies);
```

The first line can be classified as H input, since cookie values are security sensitive. The second line can be classified as L output, since this triggers an HTTP request to a different domain. This program is interferent, because the low-level output statement at line 2 is clearly influenced by the high-level input statement at line 1.

Under secure multi-execution, a script is run multiple times, once for each security level. Outputs of a given security level are only generated in the execution belonging to that security level and inputs of a given security level are replaced by `undefined` in all executions of a lower level. Hence, high-level, security-sensitive input can never leak to low-level, public output channels, or even have an influence on them.

To multi-execute a script using JSand, that script must be executed once for the low security level and once for the high security level, each time in a different sandbox, with a different security policy. The low-level policy should disable all high-level inputs and ignore high-level outputs, while the high-level policy should simply ignore low-level outputs. Since each output statement is executed in only one of the executions, the net effect of a noninterferent script under secure multi-execution will be the same as the net effect of executing the same script without multi-execution.

4.4.4 Wrapping the DOM

All interactions between a script and the browser are performed through the DOM. Hence, to control access to all security-sensitive operations, JSand needs to control access to all facets of the DOM. To implement this, each sandboxed script is initially only given a single reference to a wrapper of the `window` object, which is the root of the DOM tree. As described in Section 4.4.2, all property accesses, property assignments and function calls on this wrapper or on any object transitively reached from it are intercepted by a handler. These handlers can thus enforce an arbitrary policy on the entire DOM, and hence effectively control access to all security-sensitive operations.

For any DOM object wrapper, a distinction can be made between two categories of properties. The first category consists of *standard DOM properties*, i.e. properties that are part of the DOM as defined by the ECMAScript standard (or implementation-specific properties provided by the browser). The second category consists of *custom properties* that have been added to a DOM object wrapper by a sandboxed script. For instance, `window.document` belongs to the first category, while `window.googlemaps` could belong to the second. Properties from these two categories need to be handled differently. Assignments to standard DOM properties should be propagated outside the sandbox to the corresponding target property on the real DOM object (if allowed by the security policy), since this is the only way a sandboxed script can interact with the browser. Custom properties should however be confined to the bounds of the sandbox, to prevent sandboxed code from polluting the global namespace and from reading or modifying properties defined outside the sandbox.

To make the distinction between standard DOM properties and custom properties, JSand uses a statically defined *DOM description*, derived from the W3C DOM specification [37]. This description consists of an array of property descriptors, indexed by a DOM object name (e.g. `Window`) and a property name (e.g. `alert`). Since each descriptor corresponds to a standard DOM property, they enable the handlers to determine whether or not a given property is a standard DOM property

4.4.5 Dynamic script loading support

From experience, we have learned that many scripts dynamically load additional scripts during their execution. This is typically accomplished by inserting a new script tag with a `src` attribute in the document, because this method is not under restriction of the same origin policy. However, when a script is included this way, it is executed in the global context. Hence, if we would allow sandboxed scripts to simply add new script tags to the document, they could trivially break out of their sandbox. Any script included by a sandboxed script should execute within that same sandbox.

For this reason, JSand uses special handlers to intercept methods that allow script tags to be added to the document, including `node.appendChild`, `node.insertBefore`, `node.replaceChild`, `node.insertAfter` and `document.write`. The first four of these take a (partial) DOM tree as argument and append or insert it at a certain place in the DOM. Our handlers for these methods search the given DOM tree for script tags, extract the value of the `src` attribute and execute the corresponding scripts in the sandbox that included them, using the `innerLoadScript` function described in Section 4.4.2. The `document.write` method is similar but takes an HTML string as argument and appends that string verbatim to the document. The handler for this method parses the given HTML string, extracts script tags out of it and loads them as described above.

We have considered two different techniques for parsing a given HTML string in JavaScript. The first technique consists of creating an iframe and setting its `srcdoc` attribute [2] to the given HTML. To prevent the iframe from fetching and executing scripts included in the HTML, its `sandbox` attribute [2] must be set as well. The second technique consists of using a pure JavaScript library to parse the HTML [12]. The iframe-based technique has a potential performance benefit, since the parsing is done by native code in the browser instead of in JavaScript. Moreover, using the first technique ensures that the HTML is parsed exactly as the browser will interpret it. However, one of the problems of this approach, is that the parsing is performed asynchronously. That is, we can only access the iframe's fully populated DOM tree from its `onload` callback, which is triggered some time after setting the `srcdoc` attribute. Consequently, scripts that immediately make use of the HTML written by `document.write` could fail, since the HTML might not yet have been processed. Performing a continuation-passing style transformation on these scripts could solve this problem, but this is a complex transformation which we leave for future work. Our prototype uses the second technique, since it does not suffer from this problem.

4.4.6 Support for legacy scripts

Although the SES library natively supports scripts adhering to recognized ES5 best practices, as described in Section 4.4.1 not all currently deployed JavaScript scripts do so. Although the secure confinement of legacy scripts is never at stake, not all of them run without errors under the SES library. Therefore, we have developed a support layer to further improve the compatibility with these legacy scripts, based on three abstract syntax tree (AST) transformations.

T1 Adding a property to the global `window` object normally introduces that property as a global variable, but this does not hold in a SES environment. This transformation introduces a global alias variable for each property of `window`. The variable is updated whenever an assignment is made to its corresponding property.

T2 Conversely, declaring a global variable normally creates an alias property on the `window` object, but this doesn't hold in a SES environment. This transformation adds a property on `window` for each global variable. The property is updated whenever an assignment is made to its corresponding global variable.

T3 Since SES enforces strict mode for all scripts, ES5 non-strict code might be incompatible with SES. The most common incompatibility we have encountered is the lack of `this`-coercion. That is, `this` is no longer bound to the global `window` object in a function call. This transformation replaces `this` by the expression `(this === undefined ? window : this)`.

We have implemented a client-side component for applying these transformations, using the UglifyJS JavaScript parser [20]. These transformations do not provide a full translation from ES5 to SES, but they are sufficient to make many legacy scripts work with our prototype.

4.5 Evaluation

In this section we evaluate to what extent JSand satisfies the requirements set forth in Section 4.2.3.

4.5.1 Complete mediation

All sandboxed scripts are executed in an object-capability environment, set up by the SES library. Our implementation of the Membrane pattern ensures that each DOM access and JavaScript API call made by a sandboxed script is assessed by the security policy. Based on the theory of object-capability systems, this provides complete mediation.

Note that JSand provides a one-way isolation and hence makes no attempt to protect a sandboxed script from its environment. That is, code running in the global security context, such as browser plugins and unsandboxed scripts, have the power to modify a sandbox's security policy or to inject a DOM proxy that allows access to any DOM object. However, since malicious global code has already full power over the web page, we consider protecting against such scenarios out of scope for our solution.

4.5.2 Backwards compatibility

We have extensively and successfully tested our prototype on a variety of JavaScript scripts. In this section we report and discuss in detail three of the most widespread included scripts around: Google Analytics, Google Maps and the jQuery library. Google Analytics is included from more than 68% of all domains from the Alexa Top 10 000, making it the most included script on this list [24]. Google Maps is the most included web mashup API according to [29], being used in 17.41% of registered mashups. jQuery is the most popular JavaScript library in use today, included in more than 57% of the top 10 000 websites to date [3]. As future work, we would like to extend our evaluation to more legacy scripts.

Google Analytics Google Analytics (GA) is a web analytics service that generates statistics about visitors to a website. The GA API allows web administrators to collect custom visitor properties, in addition to the standard properties that are collected by default (such as referrer and geographical location). The collected statistics can be monitored using a dashboard interface on the GA website.

To enable GA, the website owner must add a small JavaScript code template provided by Google to the header of the page to track. This template sets up an array of options to pass to the GA service and dynamically adds a new script tag to the page to include the main GA script. Any script included like this has unrestricted access to the DOM, making the page vulnerable to malicious script inclusions.

Manual inspection of the GA script is practically impossible, since the code is minified. Moreover, since the main GA script is loaded dynamically from the Google servers, any static, offline security analysis would fail to detect malicious changes introduced to the script after the initial analysis. However, by running GA in a JSand sandbox with a policy that permits only the operations necessary for a benign web analytics script, the impact of a malicious action on behalf of the GA script can be reduced to a minimum. The code snippet below shows how this can be implemented.

```
// main page:
var sb = new jsand.Sandbox('ganalytics.js',policy);
sb.load();
```

This code snippet creates a new sandbox and initializes it with the `ganalytics.js` script, which is shown below and consists of the code template provided by Google.

```
// ganalytics.js:
var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-xxxxxxxx-x']);
_gaq.push(['_trackPageview']);

(function() {
  var ga = document.createElement('script');
  ga.src = 'http://www.google-analytics.com/ga.js';
  var s = document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(ga, s);
})();
```

Both the `ganalytics.js` script and the main `ga.js` script (which is loaded from the code above) are executed in the same sandbox and are patched-up automatically, based on the AST transformations described in Section 4.4.6. The following code fragment shows the first two lines of the patched-up `ganalytics.js`.

```
// patched-up ganalytics.js:
var _gaq = _gaq || [];
window._gaq = _gaq;
[...]
```

The global variable `_gaq` is explicitly aliased as a property on `window`. This transformation is necessary because the `ga.js` script frequently refers to the `_gaq` array as `window._gaq`. Such references would fail without the patch shown here.

The `_gaq` array exposes an API to interact with GA after it has been initialized, for instance to add a custom property to collect or to track the click of a button. The website owner can access this array using the `innerEval` method described in Section 4.4.2. To facilitate these interactions and to make abstraction of the fact that GA is running in a sandbox, the website owner could implement an object that automatically forwards its calls to the `_gaq` array inside the sandbox.

Clearly, the effort required to run GA in a JSand sandbox is minimal and introduces no disruptive changes whatsoever. Nevertheless, the power of the GA script is reduced to a safe minimum, dramatically reducing the impact of a malicious script inclusion attack.

Google Maps The Google Maps (GM) API enables website owners to embed a Google Maps gadget on their website. The standard way to add this gadget to a page is to (1) place a div element somewhere in the body where the map should be displayed, (2) add a script tag to the head of the page, which loads the GM library from the Google servers and (3) add a small piece of JavaScript code to the page, to create a new GM instance in the div element.

As with Google Analytics, the default way of including the GM script lets it have unrestricted access to the DOM and JavaScript APIs, putting the confidentiality and integrity of the entire web page at risk. JSand enables the website owner to confine the GM gadget to a sandbox with the minimal privileges required for legitimate operation.

The steps required to run GM in a JSand sandbox are very similar to the standard steps described above. In step (1), in addition to placing a div element somewhere in the body, the integrator must include the JSand library and the libraries it depends on. In step (2), instead of adding a script tag to directly load the GM library in the global page context, a new sandbox must be created for the GM script to run in. In step (3), the website owner can use the `innerEval` method to create a new GM instance in the sandbox. These steps are depicted in the following code fragment.

```
var sb = new jsand.Sandbox(
 'http://maps.googleapis.com/maps/api/js?sensor=false', policy);
sb.load();
sb.innerEval(
 "var m = window.google.maps;
  var options = {
    center: new m.LatLng(-34.397, 150.644),
    zoom: 8, mapTypeId: m.MapTypeId.ROADMAP
  };
  var map = new m.Map(document.getElementById('map_div'),
      options);"
);
```

When the main GM script is loaded, a complex process of dynamically loading and patching other scripts is performed in the background. Figure 7 depicts the sequence of scripts that are dynamically loaded from the main `js` script initially loaded in step (2). In addition to the scripts shown in this figure, more scripts are loaded and patched whenever the user changes

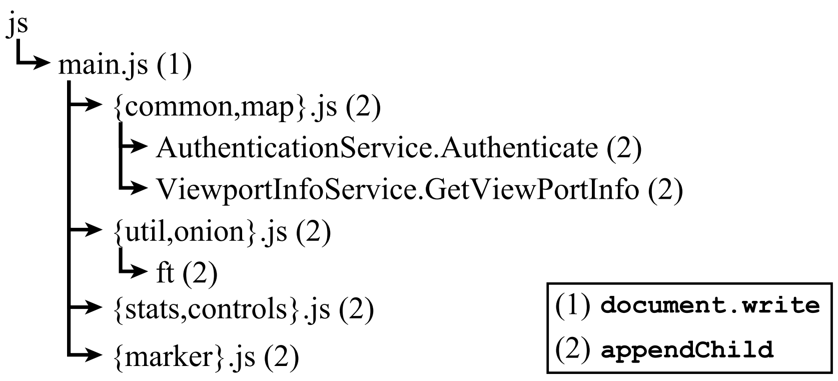

Figure 7: Tree of scripts dynamically loaded by Google Maps.

the map's viewport (by dragging it or changing the zoom level). All three translations described in Section 4.4.6 are required for the GM gadget to work.

The GM API provides extensive support for customization, to support feature-rich web mashups built around the GM gadget. For instance, website owners can provide custom map overlays, place markers, register callbacks for mouse events, etc. As with GA, a website owner can use the `innerEval` method to interact with the sandboxed GM gadget.

The fact that JSand can successfully execute this gadget in a sandbox without any compatibility issues, illustrates that our solution is able to sandbox complex JavaScript gadgets that depend on dynamic script inclusions and that feature advanced DOM interactions.

jQuery The jQuery library aims to provide a simple cross-browser API for performing common JavaScript operations, such as creating and selecting DOM elements, handling events, invoking Ajax interactions, etc. While jQuery can be used as an abstraction layer on top of an extensive set of JavaScript APIs, a website owner typically uses only a limited subset of what the library has to offer. By running jQuery in a sandbox with tight restrictions on the permitted JavaScript API and DOM operations, the risk and impact of a malicious script inclusion attack are reduced dramatically.

For our jQuery evaluation scenario, we executed jQuery together with the jQuery-geolocation plugin [25] in a sandbox, using a fine-grained security policy that allows us to toggle access to the JavaScript Geolocation API. Disabling the Geolocation API in the policy effectively prevents jQuery from using it in the sandbox. The following code fragment shows how this scenario is implemented.

```
var sb = new jsand.Sandbox('jquery-1.7.2.js',policy);
```

```
sb.load();
sb.innerLoadScript('jquery-geolocation-0.1.js');
sb.innerEval(
 "if (jQuery.geolocation.support()) {
    jQuery.geolocation.find(function(loc) {
       alert(loc.latitude+\", \"+loc.longitude);
    });
  } else { alert('Geolocation not supported'); }");
```

This scenario illustrates that, with minimal effort, a website owner can create a secure JSand sandbox around an extensible JavaScript library, while still being able to interact with it from outside the sandbox.

4.5.3 Performance benchmarks

To evaluate the runtime overhead of our prototype, we have conducted micro- and macro-benchmarks. All benchmarks were run using Google Chrome v20.0.1132.21 on Ubuntu 11.04 x86-64, running on an Intel Core 2 Duo T8300 2.4GHz processor with 4 GiB of RAM.

Micro benchmarks

JSand framework load time To measure the load time of the JSand framework, a page was created that loads the framework but doesn't use it. This page was reloaded 1000 times and the elapsed time was recorded. The average load time measured in this way was 71.5±1.8 ms. The same experiment was run with all JavaScript code commented out, so the same network load time would be maintained, but the code would not be executed. The load time in this case was 23.0±0.2 ms. This means that once loaded from the network, the framework takes on average 48.5 ms to deploy on the client side.

Third-party library load time Similar experiments were performed to measure the overhead of loading and parsing a third-party JavaScript library into a JSand sandbox. We chose jQuery as a representative JavaScript library and loaded it in a JSand sandbox, as well as a regular, unsandboxed JavaScript environment, using XHR and eval(). In both cases, we supplied a real JavaScript library as well as a commented-out version to factor out network overhead.

In a regular JavaScript environment, the code loads in 53.0±0.8 ms and 26.8±0.2 ms for normal and commented-out code respectively. Inside a JSand sandbox, the code loads in 1458.2±16.0 ms and 107.6±1.4 ms respectively, so that the overhead of parsing the library code is about 1350.6 ms.

A large portion of this overhead is due to the script rewriter of the legacy support layer described in Section 4.4.6. Since jQuery is SES-compliant, this rewriting step is not required. Disabling it lowers the average load time from 1458.2 ms to 705.8±1.1 ms, and the average overhead from 1350.6 ms to 598.2 ms. This means that 44.3% of the overhead can be contributed to our efforts for making legacy code SES-compliant.

Membrane transition cost To verify the runtime overhead of a function call crossing the membrane, a function was executed both inside and outside a JSand sandbox 1 million times and the elapsed time is recorded. We chose the `window.clearTimeout` function as a representative function, because intuitively it should return quickly when no timer is registered. When called from inside the sandbox, the `window.clearTimeout` call must cross the membrane separating the sandbox from the real JavaScript environment. Outside the sandbox, the average execution time is 0.9±0.0 μs, while inside the sandbox it is 8.0±0.1 μs.

Macro benchmarks The most important metric that counts when executing JavaScript in a browser, is the user experience. Ideally, the user should not notice that JSand is being used at all. To measure how much overhead the user experiences, we created a typical web application using Google Maps and measured two things: the total load time of the web application, and the delay a user experiences when interacting with it.

The load time of the web application was measured from the time the page is loaded until the Google Maps API emits a 'tilesloaded' event, signaling that the application is ready to be used. Running outside of the JSand sandbox, this load time is 308.0±13.7 ms, and 1432.8±24.2 ms inside of it. Keeping in mind that a large portion of this overhead is due to script-rewriting for legacy code, the total overhead without the legacy support layer can be estimated to be about 626.5 ms.

To measure the delay experienced when interacting with the application, we waited until the application was loaded, and then panned 400 px to the right, 100 times. The average time elapsed between two pans was considered as a reasonable approximation of the user-experienced delay. This delay is 320.2±0.8 ms outside and 420.0±2.7 ms inside the sandbox.

The overall performance of a JSand sandbox is acceptable. The overhead when loading a reasonably-sized SES-compliant JavaScript library inside the sandbox, is about 203%. For legacy scripts, JSand requires a code transformation step that results in a total overhead of about 365%, but it is expected that this step can be removed or at least sped up significantly for future JavaScript code in future browsers. Furthermore, the tendency of users to keep certain websites open using persistent tabs, makes the load time overhead less important. Additionally, despite the nine-fold execution time of a function-call traversing the sandbox membrane, the delay experienced by a user when using a realistic web application inside a JSand sandbox, is an acceptable 31.2%, corresponding to an absolute delay on the order of 100 ms.

4.6 Related work

Server-side processing of scripts A common technique for preventing undesired script behavior is to restrict the untrusted code (i.e. the third-party component) to a safe subset of JavaScript [16]. Compliance to the subset is verified at the server side. The allowed operations within the subset prevent the untrusted code from obtaining elevated privileges, unless explicitly allowed by the integrator. ADSafe [4], ADsafety [28] and FBJS [34] are examples of techniques where third-party JavaScript must conform to a certain JavaScript subset. Techniques such as Caja [21], Jacaranda [10] and Live Labs' Websandbox [19] on the other hand, statically analyze and rewrite the third-party JavaScript on the server side into a safe version.

Instead of forcing the use of a JavaScript subset, the JavaScript code can also be instrumented with extra checks that mediate access to certain functionality. BrowserShield [30] and Browser-Enforced Embedded Policies (BEEP) [11] are examples of such instrumentation on the server-side.

While safe subsets, code rewriting and server-side code instrumentation can restrict third-party code at the source, their adoption by mashup integrators is problematic. These techniques require either access to code running on the server-side, or require the website owner to implicitly trust the JavaScript provider to deliver safe JavaScript code. In real-world scenarios, it is infeasible to impose any such restrictions on third-party code providers. In contrast, JSand requires no server-side processing of the third-party code and imposes no fundamental restrictions on included code.

Extending the browser with a reference monitor A second class of techniques extends the browser to enforce code restrictions. Systems like ConScript [18], WebJail [35] and Contego [14] require modifications to the JavaScript engine to enforce policies on third-party code, while AdSentry requires the installation of a Firefox extension to restrict the functionality available to advertisements.

Browser modifications to restrict third-party JavaScript can be implemented very efficiently and can guarantee that enforcement cannot be circumvented. The major disadvantage of this approach however, is that the browser must be modified. Unless all the users of a web application are using a browser which implements the desired modification, there is little or no incentive for the website owner to make use of it. Because of the large variety of active browser vendors and versions on the internet, it is unrealistic to assume that a certain modification will ever be implemented in all browsers. For this reason, JSand does not depend on any special browser-side features except for what is available in the web standards.

Leveraging existing browser security features Finally, some approaches leverage recent browser security extensions to contain scripts. The new `sandbox` attribute of the iframe element in HTML5 [2] can restrict third-party JavaScript in a very coarse-grained way: it only supports to completely enable or disable JavaScript.

The Content Security Policy (CSP) [32] allows the insertion of a security policy through HTTP response headers and meta tags, which must be enforced in the browser. This policy can restrict the locations a web application loads its content from, thus preventing some forms of content-injection. However, CSP does not provide any fine-grained control over which JavaScript functionality is available to a script.

AdJail [33] is geared towards securely isolating ads from a hosting page for confidentiality and integrity purposes, while maintaining usability. The ad is loaded on a shadow page that contains only those elements of the hosting page that the web developer wishes the ad to have access to, and it relies on the SOP to isolate the shadow page. Changes to the shadow page are replicated to the hosting page if those changes conform to a specified policy. Likewise, user actions on the hosting page are mimicked to the shadow page if allowed by the policy. AdJail is a good approach to restrict access to the DOM, but cannot enforce a policy on the other JavaScript APIs like JSand does.

Self-protecting JavaScript (SPJS) [27, 17] is a client-side wrapping technique that applies advice around JavaScript functions, without requiring browser modifications (unlike [18] or [35]). It builds on standard aspect-oriented libraries for JavaScript. The wrapping code and advice are provided by the server and are executed first, ensuring a clean environment to start from. SPJS does not guarantee that all access-paths to certain JavaScript functionality can be restricted, because the aspect library it relies on was not designed with security in mind. JSand uses the Membrane pattern instead, which was designed to provide complete mediation.

Secure ECMAScript (SES) [22] is a subset of ES5 strict which provides an object-capability language. Unlike Caja, from which it originated, SES runs completely on the client-side without any browser modifications. To the best of our knowledge, JSand is the first fully functional JavaScript integration technique built on SES, capable of handling legacy scripts such as Google Maps and Google Analytics.

4.7 Conclusion

This paper introduced JSand, a server-driven but client-side JavaScript sandboxing framework that does not rely on any browser modifications. We have implemented a prototype of this framework and evaluated it on the most widespread JavaScript scripts around. Although there has been a lot of activity in this research area, we are the first to deliver a solution that provides complete mediation, backwards compatibility and an acceptable performance overhead.

References

[1] 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 (ACSAC 2012)*, pages 1–10. ACM, December 2012.

[2] Robin Berjon. W3C HTML5 Working Draft. http://www.w3.org/TR/html5/, September 2012.

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

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

[5] Tom Van Cutsem and Mark S. Miller. On the Design of the ECMAScript Reflection API. Technical Report VUB-SOFT-TR-12-03, Department of Computer Science, Vrije Universiteit Brussel, February 2012.

[6] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In *Proc. of CCS'12*. ACM, 2012.

[7] Philippe De Ryck, Maarten Decat, Lieven Desmet, Frank Piessens, and Wouter Joosen. Security of web mashups: a survey. In *Proc. of Nord-Sec'10*. Springer, 2011.

[8] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proc of SP'10*, IEEE, pages 109–124, Washington, DC, USA, 2010.

[9] Mario Heiderich. Locking the Throne Room - How ES5+ will change XSS and Client Side Security. http://www.slideshare.net/x00mario/locking-the-throneroom-20, November 2011.

[10] Jacaranda. Jacaranda. http://jacaranda.org.

[11] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proc. of WWW'07*, pages 601–610, New York, NY, USA, 2007. ACM.

[12] John Resig. Pure JavaScript HTML Parser. http://ejohn.org/blog/pure-javascript-html-parser/.

[13] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. http://www.webappsec.org/projects/articles/071105.shtml, April 2005.

[14] Tongbo Luo and Wenliang Du. Contego: capability-based access control for web browsers. TRUST'11, pages 231–238, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] S. Maffeis, J.C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc. of SP'10*. IEEE, 2010.

[16] S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *Proc. of CSF'09*, IEEE, 2009.

[17] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *Proc. of Nordsec'10*, 2010.

[18] Leo Meyerovich and Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *Proc. of SP'10*, 2010.

[19] Microsoft Live Labs. Live Labs Websandbox. http://websandbox.org.

[20] Mihai Bazon. UglifyJS. https://github.com/mishoo/UglifyJS/.

[21] 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.

[22] Mark Samuel Miller. Secure EcmaScript 5. `http://code.google.com/p/es-lab/wiki/SecureEcmaScript`.

[23] 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.

[24] 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 *Proc. of CCS'12*, October 2012.

[25] NoMoreSleep. jquery-geolocation. `http://code.google.com/p/jquery-geolocation/`.

[26] Phu H. Phung and Lieven Desmet. A two-tier sandbox architecture for untrusted javascript. In *Proc. of JSTools'12*, pages 1–10, New York, NY, 2012. ACM.

[27] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[28] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: type-based verification of JavaScript Sandboxing. In *Proc. of USENIX'11*, SEC'11, pages 12–12, Berkeley, CA, USA, 2011.

[29] Programmable Web. Keeping you up to date with APIs, mashups and the Web as platform. `http://www.programmableweb.com/`.

[30] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *Proc. of OSDI'06*, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[31] SANS Institute. SANS: Top Cyber Security Risks. http://www.sans.org/top-cyber-security-risks/, 2009.

[32] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proc. of WWW'10*, pages 921–930, New York, NY, 2010. ACM.

[33] 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*, August 2010.

[34] The FaceBook Team. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

[35] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. WebJail: least-privilege integration of third-party components in web mashups. ACSAC '11, pages 307–316, New York, NY, USA, 2011. ACM.

[36] Tom Van Cutsem and Mark S. Miller. Proxies: design principles for robust object-oriented intercession APIs. *SIGPLAN Not.*, 45(12):59–72, October 2010.

[37] W3C. Document Object Model (DOM) Technical Reports. http://www.w3.org/DOM/DOMTR.

[38] W3C. W3C Standards and drafts - Cross-Origin Resource Sharing. http://www.w3.org/TR/cors/.

[39] W3C. W3C Standards and drafts - Uniform Messaging Policy, Level One. http://www.w3.org/TR/UMP/.

[40] Yahoo! Developer Network. JavaScript: Use a Web Proxy for Cross-Domain XMLHttpRequest Calls. http://developer.yahoo.com/javascript/howto-proxy.html.

[41] Chuan Yue and Haining Wang. Characterizing Insecure JavaScript Practices on the Web. In *Proc. of WWW'09*, pages 961–961, April 2009.

[42] M. Zalewski. Browser Security Handbook. http://code.google.com/p/browsersec/wiki/Main.

5 PreparedJS: Secure Script-Templates for JavaScript[13]

5.1 Introduction

5.1.1 Motivation

Cross-site Scripting (XSS) is one of the most prevalent security problems of the Web. It is listed at the second place in the OWASP Top Ten list of the most critical Web application security vulnerabilities [19]. Even though the basic problem has been known since at least 2000 [4], XSS still occurs frequently, even on high-profile Web sites and mature applications [25]. The primary defense against XSS is secure coding on the server-side through careful and context-aware sanitization of attacker provided data [20]. However, the apparent difficulties to master the problem on the server-side have let to investigations of client-side mitigation techniques.

A very promising approach in this area is the Content Security Policy (CSP) mechanism, which is currently under active development and has already been implemented by the Chrome and Firefox Web browsers. CSP provides powerful tools to mitigate the vast majority of XSS exploits.

However, in order to properly benefit from CSP's protection capabilities, site owners are required to conduct significant changes in respect to how JavaScript is used within their Web application, namely getting rid of inline JavaScript, such as event handlers in HTML attributes, and string-to-code transformations, which are provided by `eval()` and similar functions (see Sec. 5.2.2 for further details). Unfortunately, as we will discus in Section 5.3, all this effort does not result in complete protection against XSS attacks. Some potential loopholes remain, which cannot be closed by the current version of CSP.

Contribution We explore the remaining weaknesses of CSP (see Sec. 5.3) and examine which steps are necessary to fill the identified gaps for completing CSP's protection capabilities. Based on our results, we propose PreparedJS, an extension of the CSP mechanism (see Sec. 5.5). PreparedJS is built on two pillars: A templating format for JavaScript which follows SQL's prepared statement model (see Sec. 5.5.1) and a light-weight script checksumming scheme, which allows fine-grained control over permitted script code (see Sec. 5.5.2). In combination with the base-line protection provided

[13]This paper has been published as [9]: Martin Johns: PreparedJS: Secure Script-Templates for JavaScript, Proceedings of the 10th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '13), Berlin, Germany, July 2013

by CSP, PreparedJS is able to prevent the full spectrum of potential XSS attacks. We outline how PreparedJS can be realized as a native browser component while providing backwards compatibility with legacy browsers that cannot handle PreparedJS's script format. Furthermore, we report on a prototypical implementation in the form of a browser extension for Google Chrome (see Sec. 5.6).

5.2 Technical background

5.2.1 Cross-site Scripting (XSS)

The term *Cross-site Scripting (XSS)* [27] summarizes a set of attacks on Web applications that allow an adversary to alter the syntactic structure of the application's Web content via code or mark-up injection.

Even though XSS, in most cases, also enables the attacker to inject HTML or CSS into the vulnerable application, the main concern with this class of attacks is the injection of JavaScript. JavaScript injection actively circumvents all protective isolation measures which are provided by the same-origin policy [24], and empowers the adversary to conduct a wide range of potential attacks, ranging from session hijacking [18], over stealing of sensitive data [29] and passwords [28], up to the creation of self-propagating JavaScript worms.

To combat XSS vulnerabilities, it is recommended to implement a careful and robust combination of input validation (only allow data into the application if it matches its specification) and output sanitation (encode all potential syntactic content of untrusted data before inserting it into an HTTP response). However, a recent study [25] has shown, that this protective approach is still error prone and the quantitative occurrence of XSS problems is not declining significantly.

5.2.2 Content Security Policies (CSP)

Due to the fact, that even after several years of increased attention to the XSS problem, the number of vulnerabilities remains high, several reactive approaches have been proposed, which mitigate the attacks, even if a potential XSS vulnerability exists in a Web application.

Content Security Policies (CSP) [26] is such an approach: A Web application can set a policy that specifies the characteristics of JavaScript code which is allowed to be executed[14]. CSP policies are added to a Web docu-

[14]CSP also provides further features in respect to other HTML elements, such as images or iframe. However, these features do not affect JavaScript execution and, hence, are omitted in the CSP description for brevity reasons.

```
Content-Security-Policy: default-src 'self'; img-src *;
                         object-src media.example.com;
                         script-src trusted.example.com;
```

Figure 8: CSP example

ment through an HTTP header or a `Meta`-tag (see Lst. 8 for an example). More specifically, a CSP policy can:

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

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

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

If used in combination, these three capabilities lead to an effective thwarting of the vast majority of XSS attacks: The forbidding of inline scripts renders direct injection of script code into HTML documents impossible. Furthermore, the prevention of interpreting string data as code removes the danger of DOM-based XSS [11]. And, finally, only allowing code from whitelisted hosts to run deprives the adversary from the capability to load attack code from Web locations that are under his control.

In summary, strict CSP policies enforce a simple yet highly effective protection approach: Clean separation of HTML-markup and JavaScript code in connection with forbidding string-to-code transformations via `eval()`. The future of CSP appears to be promising. The mechanism is pushed into major Web browsers, with recent versions of Firefox (since version 4.0) and Chrome (since version 13) already supporting it. Furthermore, CSP is currently under active standardization by the W3C [30].

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

5.3 CSP's remaining weaknesses

In general, CSP is a powerful mitigation for XSS attacks. If a site issues a strong policy, which forbids inline scripts and unsafe string-to-code transforms, the vast majority of all potential exploits will be robustly prevented, even in the presence of HTML injection vulnerabilities.

However, as we will show in this section, three potential attack variants remain feasible under the currently standardized version 1.0 of CSP [30]. Furthermore, in Section 5.3.4, we will discuss to which degree the proposed enhancements of CSP 1.1 affect these identified weaknesses.

5.3.1 Weakness 1: Insecure server-side assembly of JavaScript code

As described above, CSP can effectively prevent the execution of JavaScript which has been dynamically assembled on the client-side. This is done by forbidding all functions that convert string data to JavaScript code, such as `eval()` or `setTimeout()`. However, if a site's operator implements dynamic script assembly on the server-side, this directive is powerless.

Server-side generated JavaScript is utilized to fill values in scripts with data that is retrieved at runtime. If such data can be controlled by the attacker, he might be able to inject further JavaScript.

Take for instance the scenario that is outlined in Listings 9 and 10: A script-loader JavaScript (`loader.js`, Lst. 9), is used to dynamically outfit further JavaScript resources with runtime data via URL parameters[15]. The referenced script (`ga.php`, Lst. 10) is assembled dynamically on the server-side, including the dynamic data in the source code without any sanitization.

If the attacker is able to control the `document.location` property, he can break out of the variable assignment in line 5 and inject arbitrary JavaScript code. Thus, he can effectively circumvent CSP's protection features: The attack uses no string-to-code conversion on the client-side. All the browser retrieves is apparently static JavaScript. In addition, the attack does not rely on inline scripts, as the injected script is included externally. Finally, the vulnerable script is part of the actual application and, hence, the script's hosting domain is included in the policy's whitelist.

[15]The depicted code was consciously designed in a naive fashion to make the issue easily understandable. In more realistic conditions, the attacker controlled data could find its way into the script assembly in more subtle fashions, e.g., through existing data in the user's session.

```
(function() {
  var ga = document.createElement('script');
  ga.src =
     'http://serv.com/ga.php?source='+document.location;
  var s = document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(ga, s);
})();
```

Figure 9: JavaScript for dynamic script loading (`loader.js`)

```
// JS code to set a global variable with the
// request's call context
<?php
$s = '$_GET["source"]';
echo "var callSource='".$s."';";
?>
// [...rest of the JavaScript]
```

Figure 10: Variable setting script (`ga.php`)

5.3.2 Weakness 2: Full control over external, whitelisted scripts

It is common practice to include external JavaScript components from third party hosts into Web applications. This is done to consume third party services (such as Web analytics), enhance the Web application with additional functionality (e.g., via integrating external mapping services), or for monetary reasons (i.e, to include advertisements).

Recently Nikiforakis et al. conducted a wide scale analysis on the current state of cross-domain inclusion of third party JavaScripts [17]. Their survey showed that 88.45% of the Alexa top 10,000 Web sites included at least one remote JavaScript. If the attacker is able to control the script's content, which is provided by the external provider, he is able to execute JavaScript in the context of the targeted Web application.

A straight forward scenario for such an attack is a full compromise of one of the external script providers for the targeted site. In such a case, the adversary is able to inject and execute arbitrary JavaScript in the context of targeted application. To examine this potential threat, Nikiforakis et al. created a security metric for script providers, which is based on indicators for maintenance quality of the hosts. Subsequently, they compared the security score of the including sites to the score of the consumed script providers: In approximately 25% of all cases, the security score of the script provider was lower than the score of the consumer, suggesting that a compromise of the script provider was more likely than a compromise of the targeted Web application.

As alternatives to a full compromise of the script provider, Nikiforakis et al. list four further, more subtle attacks which enable the same class of script inclusion attacks and show their practical applicability (see [17] for details).

CSP is not able to protect against such cases: To utilize external JavaScript components, a CSP-protected site has to whitelist the script provider's domain in the CSP policy. However, as the adversary is able to control the contents of the whitelisted host, he is able to circumvent CSP's protection mechanism.

5.3.3 Weakness 3: Injection of further script-tags

This class of potential CSP circumvention was first observed by Michael Zalewski [32]: Given an HTML-injection vulnerability, a strict CSP policy will effectively prevent the direct injection of attacker-provided script code. However, he still is be able to inject HTML markup including further `script`-tags pointing to the whitelisted domains.

This way an attacker is able to control the URLs and order from which the scripts in a Web page are retrieved. Thus, he might be able to combine existing scripts in an unforeseen fashion. All scripts in a Web page run in the same execution context. JavaScript provides no native isolation or scoping, e.g., via library specific name-spaces. Hence, all side-effects that a script causes on the global state directly affect all scripts that are executed subsequently. Given the growing quantity and complexity of script code hosted by Web sites, a non-trivial site might provide an attacker with a well equipped toolbox for this purpose. Also, the adversary is not restricted to the application's original site. Scripts from all domains that are whitelisted in the CSP-policy can be combined freely.

Only little research has been conducted to validate this class of attacks. Nonetheless, such attacks are theoretically possible. Furthermore, with the ever-growing reliance on client-side functionality and the rising number of available JavaScripts their likelihood can be expected to increase.

5.3.4 CSP 1.1's script-nonce directive

The 1.0 version of CSP currently holds the status of a W3C "Candidate Recommendation". This means the significant features of the standard are mostly locked and are very unlikely to change in the further standardization process. Hence, major changes and new features of CSP will happen in the

```
Content-Security-Policy: script-src 'self';
                         script-nonce A3F1G1H41299834E;
```

Figure 11: CSP 1.1 policy requiring script-nonce

```
<script nonce="A3F1G1H41299834E">
  alert("I execute! Hooray!");
</script>
<script> alert("I don't execute. Boo!"); </script>
```

Figure 12: Exemplified usage of script-nonce

subsequent versions of CSP. The next iteration of the standard is CSP version 1.1, which is currently under active discussion [31].

Among other changes, that primarily focus on the data exfiltration aspect of CSP, the next version of the standard introduces a new directive called `script-nonce`. This directive directly relates to a subset of the identified weaknesses of CSP 1.0. In case, that a site's CSP utilizes the `script-nonce` directive (see Lst. 11), the policy specifies a random value that is required to be contained in all `script`-tags of the site. Only JavaScript in the context of a `script`-tag that carries the nonce value as an attribute value is permitted to be executed (see Lst. 12). For apparent reasons, a site is required to renew the value of the nonce for every request. Please note, that the nonce is not a signature or hash of the script nor has it other relations to the actual script content. This characteristic allow the usage of the directive to reenable inline scripts (as depicted in Lst. 12) without significant security degradation.

Effect on the identified weaknesses: The `script-nonce` directive effectively prevents the attacker from injecting additional `script`-tags into a page, as he won't be able to insert the correct nonce value into the tag. In this section, we examine to which degree the directive is able to mitigate the identified weaknesses:

Unsafe script assembly: To exploit this weakness, an attacker is not necessarily required to inject additional `script`-tags into the page. The unsafe script assembly can also happen in legitimate scripts due to attacker controlled data which was transported through session data or query parameters set by the vulnerable application itself.

Adversary controlled scripts: In such cases, the directive has no effect. The script import from the external host is intended from the vulnerable application. Hence, the corresponding `script`-tag will carry the nonce and, thus, is permitted to be executed.

Adversary controlled script tags: This weakness can be successfully mitigated through the directive. As the attacker is not able to guess the correct nonce value, he cannot execute his attack through injecting additional `script`-tags.

Only the third weakness can be fully mitigated through the usage of script-nonces. The reason for the persistence of the other two problems, lies in the missing relationship between the nonce and the script content. A further potential downside of the `script-nonce` directive is that it requires dynamic creation of the CSP policy for each request. Hence, a rollout of a well audited, static policy is not possible.

5.3.5 Analysis

The discussed CSP weaknesses are caused by two characteristics of the policy mechanism:

1. A site can only specify the origins which are allowed to provide script content, but not the actually allowed scripts.

2. Even if a site would be able to provide more fine-grained policies on a per-script-URL level, at the moment there are no client-side capabilities to reason about the validity of the actual script content.

The first characteristic is most likely a design decision which aims to make CSP more easily accessible and maintainable to site-owners. It could be resolved through making the CSP policy format more expressive. However, the second problem is non-trivial to address, especially in the presence of dynamically assembled scripts.

5.4 Goal: Stable Cryptographic Checksums for Scripts

As deducted above, all existing loopholes which allow the circumvention of CSP can be reduced to the fact that no reliable link exists between the policy and the actual script code. Hence, a mechanism is needed that allows site owners to clearly define which exact scripts are allowed to be executed. And, as seen in Sec. 5.3.1, this specification mechanism should not only rely on a script's URL. It should also take the script's content into consideration.

A straight forward approach to solve this problem is utilizing script signatures or cryptographic checksums, that are calculated over the scripts' source code: On deploy-time the checksums of all legitimate JavaScripts are generated and are included in an extended CSP policy. At runtime, this policy is communicated to the browser which in turn only allows the execution of scripts with correct checksums. This technique works well as long as only static scripts are utilized.

Unfortunately, this approach is too restrictive. As soon as the need for dynamic data values during script assembly occurs, the mechanism cannot be applied anymore: The source code of the scripts is non-static and, hence, creating source code checksums on deploy-time is infeasible. However, creating these checksums at runtime defeats their purpose, as in such cases in-script injection XSS (see Sec. 5.3.1) will be included in the checksum and, thus, the browser will allow the script to be executed.

Therefore, a secure mechanism is needed which allows the creation of stable cryptographic checksums of script code while still allowing a certain degree of flexibility in respect to run-time script creation.

5.5 PreparedJS

In this section, we present PreparedJS - our approach to fill the identified weaknesses of CSP. PreparedJS is built on two pillars:

- A *templating mechanism*, that enables developers to separate dynamic data values from script code, thus, allowing the usage of purely static scripts without losing needed flexibility,
- and a *script checksumming scheme*, that allows the server to non-ambiguously communicate to the browser which scripts are allowed to run.

As the name of our mechanism suggests, the templating mechanism is inspired by SQL's prepared statements: In a prepared statement, the query syntax is separated from the data values, using placeholders. At runtime, this statement is passed to the database together with a set of values which are to be used within the query at the placeholders' position. This way, the statement can be outfitted with dynamic values. As the syntactic structure of the statement has already been processed by the database engine, before the placeholders are exchanged with the data values, code injection attacks are impossible.

Following the prepared statement's model, PreparedJS defines a JavaScript variant which allows placeholder for data values, which will be filled at runtime in a fashion that is unsusceptible to code injection vulnerabilities (see Sec. 5.5.1 for details). This way, developers can create completely static script source code, for which the calculation of stable cryptographic checksums on

```
// JS code to set a global variable with the
// request's call context
var callSource= ?source?;
// [...rest of the JavaScript]
```

Figure 13: PreparedJS variable setting script (`ga.js`)

deploy-time is feasible. While the Web application is accessed, only scripts which have a valid checksum are allowed to run: If the checksum checking terminates successfully, the data values, which are retrieved along with the script code, are inserted into the respective placeholders, thus, creating a valid JavaScript, that can be executed by the Web browser.

5.5.1 JavaScript templates for static server-side scripts

In this section, we give details on the PreparedJS templating mechanism. The mechanism consists of two components: The *script template* and the *value list*.

The PreparedJS *script template* format supports using insertion marks in place of data values. These placeholders are named using the syntactic convention of framing the placeholders identifier with question marks, e.g., `?name?`. Such placeholders can be utilized in the script code, wherever the JavaScript grammar allows the injection of data values. See Listing 13 for a template which represents the dynamic script of Listing 10.

The PreparedJS *value list* contains the data values, which are to be applied during script execution in the browser. The list consists of identifier/-value pairs, in which the identifier links the value to the respective placeholder within the script template. The values can be either basic datatypes, i.e., strings, booleans, numbers, or JSON (JavaScript Object Notation [5]) formatted complex JavaScript data objects. The latter option allows the insertion of non-trivial data values, such as arrays or dictionaries.

Also, the value list itself follows the JSON format, which is very well suited for this purpose: The top level structure represents a key/value dictionary. By using the placeholder identifiers as the keys in the dictionary, a straight forward mapping of the values to insertion points is given. Furthermore, JSON is a well established format with good tool, language, and library support for creation and verification of JSON syntax. See Listing 14 for a PHP-script which creates the value list for Lst. 13 according to the dynamic JavaScript assembly in Lst. 10.

In the communication with the Web browser, the script template and the

```
<?php
$source = $_GET["source"];
$vals = array('callSource' => $source);
echo json_encode($vals);
?>
```

Figure 14: Creating value list for Lst. 13 (`ga_values.php`)

value list are sent in the same HTTP response, using an HTTP multipart response (see Lst. 15).

5.5.2 Code legitimacy checking via script checksums

As discussed in Section 5.3, parts of the existing shortcomings of CSP result from the mechanism's inability to specify which exact scripts are allowed to run in the context of a given Web page. Within PreparedJS we fill the gap by unambiguously identifying whitelisted scripts through their *script checksums*.

A script's PreparedJS-checksum is a cryptographic hash calculated over the corresponding PreparedJS script template. The script's value list is not included in the calculation. This allows a script's values to change on run time without affecting the checksum.

To whitelist a specific scripts, a policy lists the script's checksums in the policy declaration (see Sec. 5.5.3). For each script that is received by the browser, the browser calculates the checksum of the corresponding script template and verifies that it indeed is contained in the policy's set of allowed script checksums. If this is the case, the script is permitted to execute. If not, the script is discarded.

This approach is well aligned with the applicable attacker type. The sole capability of the XSS Web attacker consists of altering the syntactic structure of the application's HTML content. The XSS attacker is not able to alter the application's CSP policy, which is generally transported via HTTP header (if the attacker is able to compromise the site's CSP itself, all provided protection is void anyway). Hence, if the application's server-side can unambiguously communicate to the browser which exact scripts are whitelisted, altering the syntactic structure of the document has no effect.

For this purpose, cryptographic checksums are well suited: The checksum is sufficient to robustly identify the script, as long as a strong cryptographic hash function algorithm, such as SHA256, was used. Due to the algorithm's security properties, is it a reasonable assumption that the attacker is not able to produce a second script which both carries his malicious intend and

```
HTTP/1.1 200 OK
Date: Thu, 23 Jan 2012 10:03:25 GMT
Server: Foo/1.0
Content-Type: multipart/form-data;boundary=xYzZY

--xYzZY
Content-Type: application/pjavascript;
              charset=UTF-8
Content-Disposition: form-data;name="preparedJS"

// JS code to set a global variable with the
// request's call context
var callSource = ?callSource?;
--xYzZY
Content-Type: application/json
Content-Disposition: form-data;name="valueList"

{"callSource": "http://serv.com?this=that#attackerData"}
--xYzZY--
```

Figure 15: PreparedJS HTTP multipart response

produces the same checksum.

5.5.3 Extended CSP Syntax

For the PreparedJS scheme to function, we require a simple extension of the CSP syntax. In addition to the list of allowed script hosts, also the list of allowed script checksums has to be included in a policy. This can be achieved, for instance, using a comma delimited list of script checksums following directly a whitelisted script host (see Lst. 16 for an example).

5.5.4 PreparedJS-aware script tags

CSP was carefully designed with backward compatibility in mind: If a legacy browser, that does not yet implement CSP, renders a CSP-enabled Web page, the CSP header is simply ignored and the page's functionality is unaffected.

We intend to follow this example as closely as possible. However, as the PreparedJS-format differs from the regular JavaScript syntax (see Lst. 15), the server-side explicitly has to provide backwards compatible versions of the script code. A PreparedJS-aware HTML document utilizes a slightly extended syntax for the `script`-tag. The reference to the PreparedJS-script

```
X-Content-Security-Policy: script-src 'self'
                           (135c1ac6fa6194bab8e6c5d1e7e98cd9,
                            2de1cd339756e131e873f3114d807e83)
```

Figure 16: Extended CSP syntax, whitelisting two script checksums

```
<script src="[path to legacy script]"
    pjs-src="[path to preparedJS script]">
```

Figure 17: Extended PreparedJS `script`-tag syntax

is given in a dedicated `pjs-src`-attribute. If an application also wants to provide a standard JavaScript for legacy fallback, this script can be referenced in the same tag using the standard `src`-attribute (see Lst. 17). This approach provides transparent backwards compatibility on the client-side: PreparedJS-aware browsers only consider the `pjs-src`-attribute and handle it according to the process outlined above. The legacy script is never touched by such browsers. Older browsers ignore the `pjs-src`-attribute, as it is unknown to them, and retrieve the fallback script referenced by `src`-attribute.

Please note: If naively implemented, this approach causes additional implementation effort on the server-side, as all scripts have to be maintained in two versions. However, in Section 5.6.2 we show, how applications can provide backwards compatibility support for legacy browser automatically.

5.5.5 Summary: The three stages of PreparedJS

PreparedJS affects three stages in an application's lifecycle: The development phase, the deployment phase, and the execution phase:

During development: If the Web application requires JavaScript, with dynamic, run-time generated data values, PreparedJS templates are created for these scripts and methods are implemented to generate matching value lists.

On deployment: For all JavaScripts and PreparedJS templates, which are authorized to run in the context of the Web application, cryptographic checksums are calculated. On application deployment these checksums are added to the site's extended CSP policy.

During execution: Before the execution of regular script code, the CSP policy is checked, if the script's host is whitelisted in the policy and if for this host a list of allowed script checksums is given. If both is the case, the cryptographic checksum for the received script code is calculated and compared with the policy's whitelisted script checksums. Only if the calculated checksum can be found in the policy, the script is allowed to execute.

For scripts in the PreparedJS format, first the script template is retrieved from the multipart response (see Lst. 15). Then, the checksum is calculated over the template. If the checksum test succeeds, the value list is retrieved from the HTTP response and the placeholders in the script are substituted with the actual values. After this step, the script is executed.

5.6 Implementation and enforcement

In this section, we show how the PreparedJS scheme can be practically realized. In this context, we propose a native, browser-based implementation (see Sec. 5.6.1) and discuss how backwards compatibility can be provided for browsers that are not able to handle PreparedJS's template format natively (see Sec. 5.6.2).

5.6.1 Native, browser-based implementation

As mentioned earlier, the main motivation behind PreparedJS is to fill the last loopholes that the current CSP approach still leaves for adversaries to inject JavaScript into vulnerable Web applications. For this reason, we envision a native, browser-based implementation of PreparedJS as an extension of CSP.

To execute JavaScript and enforce standard CSP, a Web browser already implements the vast majority of processes which are needed to realize our scheme, namely HTML/script parsing and checking CSP compliance of the encountered scripts. Hence, an extension to support our scheme is straight forward:

Whenever during the parsing process a `script`-tag is encountered, the script's URL is tested, if it complies with the site's CSP policy. Furthermore, if the policy contains script checksums for the URL's host, the checksum for the script's source code is calculated and it is verified, that the checksum is included in the list of legitimate scripts.

In case of PreparedJS templates, first the template code is parsed by the browser's JavaScript parser, treating the placeholders as regular data tokens. Only after the parse tree of the script is established, the placeholders are exchanged with the actual data values contained in the value list. This way, regardless of their content, these values are unable to alter the script's syntactic structure, hence, no code injection attacks are possible.

Prototypical implementation for Google Chrome: To gain insight in practically using PreparedJS's protection mechanism and experiment with the templating format, we conducted a prototypical implementation of the approach in the form of a browser extension for Google Chrome.

Chrome's extension model does not allow direct altering of the browser's HTML parsing or JavaScript execution behavior. Hence, to implement PreparedJS we utilized two capabilities that are offered by the extension model: The network request interception API, to examine all incoming external JavaScripts, and the extension's interface to Chrome's JavaScript debugger, to insert the compiled PreparedJS-code into the respective `script`-tags.

When active, the extension monitors all incoming HTTP responses for CSP headers. If such a header is identified, the extension extracts all contained PreparedJS-checksums and intercepts all further network requests that are initiated because of `src`-attribute in `script` tags in the corresponding HTML document. Whenever such a request is encountered, the extension conducts two actions: First, the actual request is redirected to a specific JavaScript, that causes the corresponding JavaScript threat to trap into Chromes's JavaScript debugger via the `debugger` statement, causing the JavaScript execution to briefly pause until the script legitimacy checking has concluded. Furthermore, the request's original URL is used to retrieve the external JavaScript's source code, or, in in the presence of a `pjs-src`-attribute, the PreparedJS-template and value list the extension.

For the retrieved source code or the PreparedJS-template the cryptographic checksum is calculated using the SHA256 implementation of the Stanford JavaScript Crypto Library[16]. If the resulting checksum was not contained in the site's CSP policy, the process is terminated and the script's source code is blanked out. If the checksum was found in the policy, the script is allowed to be executed. In case of a PreparedJS-template, the template is parsed and the items of the value-list are inserted in the marked positions. To re-insert the resulting script code into the Web page, the extension uses Chrome's JavaScript debugger and the Javascript execution is resumed.

Performance measurements: Using our prototypical implementation, we conducted measurements to gain first insight into the runtime characteristics of the proposed mechanism. For several reasons, the obtained results can be regarded as a worst case measurement: For one, the full implementation, including the template parsing and the checksum calculation, is done in JavaScript instead of native code, resulting in implementations with inferior performance compared to native code. Furthermore, the Chrome extension model made it necessary to repeatedly conduct costly context-switches into

[16]Stanford JavaScript Crypto Library: http://crypto.stanford.edu/sjcl/

Site	#[a]	**LoC**[b]	**Default**[c]	**Debug**[d]	**PJS**[e]	**Delta**
local testpage[f]	2	3624	67.9 ms	230.6 ms	309.8 ms	79 ms
gmail.com	5	16132	2184.5 ms	2542.8 ms	2691.4 ms	148.6 ms
twitter.com	2	9195	1686.0 ms	2058.8 ms	2112.8 ms	54 ms
facebook.com	18	31701	2583.8 ms	4067.5 ms	4189.0 ms	121.5 ms

a: Number of external scripts contained in the page, *b*: Total lines of JS code after de-minimizing, *c*: loadtime without extension, *d*: loadtime with extension (debugger only, no script processing),

e: load time with full PreparedJS functionality on all external scripts. *f*: Testpage with PreparedJS template, served from the same machine as the test browser

Table 9: Performance of the browser extension, mean values over 10 iterations

Chrome's debugger.

As it can be seen in Table 9, we conducted three separate measurements of page load times: Without the extension, with the PreparedJS extension, and with an "empty" extension that neither processes the script code nor calculates checksums but traps into the debugger and conducts the network interception steps. This was done to be able to distinguish between the performance cost that is caused by the limitations of Chrome's extension model, i.e., the script redirection and context-switches into the debugger, and the effort that is caused by the actual PreparedJS functionality, namely the calculation of the script checksum and the parsing of the JavaScript code. As the former only occurs because of the implementation method's limitations and won't occur in a native integration in the browser's CSP implementation, only the additional performance overhead of the latter measurement is relevant in estimating PreparedJS's actual cost (as reflected in the table). To conduct the actual measurements we utilized the *Page Benchmarker*[17] extension, using mean values of ten page load iterations over a standard German household DSL line. During the tests, all encountered external JavaScripts were treated, as if they were PreparedJS-templates and, thus, fully parsed and checksummed.

In general, we do not expect the PreparedJS approach to cause noticeable performance overhead (an estimate that is backed by the performance evaluation): PreparedJS only takes effect during the initial script parsing steps. Here three new steps are introduced, that do currently not exist. The cryptographic checksum has to be calculated, value list has to be parsed, and the obtained values have to be inserted for the placeholders. Non of these steps requires considerable computing effort: Modern hash-functions are highly optimized to perform very well, the browser's JavaScript engine has already

[17]Page Benchmarker: https://chrome.google.com/webstore/detail/page-benchmarker/channimfdomahekjcahlbpccbgaopjll

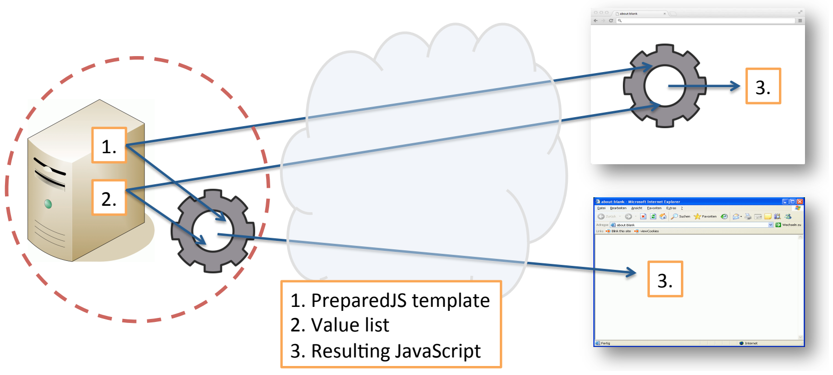

Figure 18: Native browser support (top), backwards compatibility via server-side composition service (bottom)

native capabilities for parsing the JSON-formatted value list, and inserting the data values after the script parser's tokenization step is straight foreword and does not require sophisticated implementation logic. From here on, the browser's actual JavaScript execution functionality remains unchanged. After script parsing, a PreparedJS script is indistinguishable from a regular JavaScript and all recent performance increases of modern JavaScript engines apply unmodified.

5.6.2 Transparently providing legacy support

As mentioned in Section 5.5.4, providing a second, backwards compatible version of all scripts can cause considerable additional development and maintenance effort. This in turn might hinder developer acceptance of the measure.

However, providing a backwards compatible version of scripts that only exist in the PreparedJS format can be conveniently achieved with a server-side composition service: Such a service compiles the script-template together with the value list on the fly, before sending the resulting JavaScript to the browser. For this purpose, the service conducts the exact same steps as the browser in the native case (see Fig. 18): It retrieves the template, the value-list, and the list of whitelisted checksums from the Web server. Then it calculates the templates checksum and verifies that the script is indeed in the whitelist. Then it parses the value list and inserts the resulting values into the template in place of the corresponding value identifiers.

Please note: The actual script compiling process has to be carefully implemented to avoid the reintroduction of injection vulnerabilities. For this, the data values have to be properly sanitized, such that they don't carry syntactic content which could alter the semantics of the resulting JavaScript.

Taking advantage of the composition service, the `script`-tags of the application can reference the script in its PreparedJS form directly (via the `pjs-src`-attribute) and utilize a specific URL-format for the legacy `src`-attribute, which causes the server-side to route request through the composition service. For instance, this can be achieved through a reserved URL-parameter which is added to the scripts URL, such as `?pjs-prerender=true`. All requests carrying this parameter automatically go through the composition service.

5.7 Discussion

5.7.1 Security evaluation

In this section, we verify that PreparedJS indeed closes CSP's existing protection gaps, as identified in Section 5.3.

(1) Insecure server-side assembly of JavaScript code: Vulnerabilities, such as discussed in Section 5.3.1 and shown in Lst. 9 and 10, cannot occur if PreparedJS is in use. The cryptographic checksum of dynamically assembled scripts vary for every iteration, hence, the checksumming validation step will fail, as the script's checksum won't be included in the site's CSP policy (see below for a potential limitation, in case the scheme is used wrongly).

The introduction of the PreparedJS templates offers a reliably secure alternative to insecure server-side script assembly via string concatenation. As the script's syntactic structure is robustly maintained through preparsing in the browser, before the potentially untrusted data values are inserted, XSS vulnerabilities are rendered impossible, even in cases in which the attacker controls the dynamic values.

(2) Full control over external, whitelisted script-sources: The mechanism's fine-grained checksum whitelisting reliably prevents this attack. Due to the checksum checking step, the attacker cannot leverage a compromised external host or related weaknesses. If he attempts to serve altered script code from the compromised origin this code's checksum won't appear in the policy's list of permitted scripts. Hence, the browser will refuse to execute the adversary's attack attempt.

(3) Attacker provided src-attributes in script-tags: Our proposed CSP syntax allows for finer-grained control, which scripts are allowed to run in the context of a given Web page. Hence, each page can exactly specify which scripts it really requires, leaving the adversary only minimal opportunities to combine script side effects to his liking. This is especially powerful, when it comes to script inclusion from large scale external service providers, such as Facebook or Google, from which, in most cases, only dedicated scripts are needed for the site to function. Take for example analytics services: If a site utilizes the product *Google Analytics*[18], currently *all* scripts hosted on Google's domain have to be allowed by the CSP policy. This provides the attacker with a lot of potential options under the scenario outlined in Sec. 5.3.3. Using our extended policy mechanism, it is ensured that only the required analytics script will be executed by the browser.

Limitation – Checksumming of insecurely assembled code: Apparently, if a developer creates an application which first insecurely creates dynamic script code and only after this step creates the checksums and CSP policies, the introduced protection measure can be circumvented. However, it is easy to enforce development and deployment processes that prevent such a scenario: The CSP policy generation (which requires a full set of calculated checksums) has to be decoupled from the parts of the application that handles potentially untrusted data. For instance, a requirement that decrees that all script checksums are calculated on deploy-time of the application and remain stable during execution would resolve the issue.

5.7.2 Cost of adoption

Before the introduction of CSP, a mechanism like PreparedJS would have been infeasible, due to the highly flexible nature of the Web: JavaScript can be inserted on many places within a Web page's markup, e.g., through numerous inline event handlers or `JavaScript`-URLs. Creating templates and code checksums for each of these mini-scripts would cause very high development and maintenance overhead, which in turn would hinder the mechanisms acceptance.

However, CSP policies already impose considerable restrictions on how JavaScript is used within Web applications. Thus, to adopt the PreparedJS mechanism on top, is only a small further step and the needed effort appears to be manageable: Strong CSP policies requires all JavaScript to be delivered by dedicated HTTP responses. Hence, script code is already cleanly separated from HTML markup. In result, the total number of to be handled scripts for CSP-enabled sites will be much smaller. Also, this clean separation of the script-code from the markup eases the task of identifying

[18]Google Analytics: http://www.google.com/intl/de/analytics/

the to-be signed code and creating the actual code checksums considerably. We expect for a sanely designed Web site that the majority of its JavaScript sources are contained in a limited number of dedicated places within the application structure (such as a `/js`-path).

Starting with an enumerable set of dedicated paths in which the scripts reside, the task to separate the script's dynamic code insertion routines from the main static script content is straight forward.

5.8 Related work

Server-side XSS prevention: Preventing and mitigating Cross-site Scripting attacks has received considerable attention. Most documented methods aim to fight XSS through preventing the actual code injection. They approach the problem, for instance, via tracking untrusted data during execution [21, 16, 3], enforcing type safety [22, 8, 10], or providing integrity guarantees over the document structure [12, 15]. As a general observation, it can be stated, that these approaches have to address a wide range of potential attack variants and injection vectors, thus, requiring extensive browser/server infrastructure or significant changes on the server-side. In comparison, the scope of PreparedJS's templating mechanism is much more focused on one specific problem, hence, allowing for a concise solution that effectively can leverage the existing CSP infrastructure.

Client-side techniques: Furthermore, conceptional close to out approach is BEEP [7], which proposes whitelisting of static scripts using cryptographic checksums. Similar to our approach, a JavaScript's checksum is calculated and verified, before the script is executed. In comparison to our approach, BEEP does not consider server-side script assembly. Instead, they propose runtime calculation of the server-side checksums. Hence, the protection characteristics of BEEP do not significantly surpass CSP's capabilities while requiring a considerably different enforcement architecture. Our approach only requires a extension to the browser's CSP handling. Furthermore, several approaches exist that aim to restrict JavaScript execution in general, through applying fine-grained security policies that enforce least privilege measures on script code [14, 1]. In certain cases, such techniques can be utilized to soften the effect of successful XSS attacks. However, their primary focus is at runtime control over third party JavaScript components. Due to this focus, the provided means of these techniques are not sufficient to reach the protection coverage of CSP (and, thus, of PreparedJS). Finally, more techniques exist, that explicitly aim to prevent the execution of XSS payloads. Most prominent in this area are browser-based XSS filters, which are currently provided by Webkit-based browsers [2], Internet Explorer [23], and the Firefox extension NoScript [13].

Script-less attacks: In [6] Heiderich et al. discuss XSS payloads that do not rely on JavaScript execution. Instead, the presented attacks function via the injection of HTML markup and CSS. The primary goal of these attacks is data exfiltration, i.e., transmitting sensitive information, such as credit card numbers or passwords, to the adversary. While CSP's `unsafe-inline` also restricts inline CSS declarations, such attacks are generally out of reach for our proposed technique. PreparedJS sole focus is on secure JavaScript generation and tight control over which scripts are allowed to be executed. A generalization towards HTML markup or CSS is neither planned nor realistic.

5.9 Conclusion

The Content Security Policy mechanism is a big step forward to mitigate XSS attacks on the client-side. Unfortunately, CSP is not bulletproof. In this section, we identified three distinct scenarios in which a successful XSS attack can occur even in the presence of a strong CSP. Based on this motivation, we presented PreparedJS, an extension to CSP which addresses the identified weaknesses: Through safe script templates, PreparedJS removes the requirement of unsafe server-side JavaScript assembly. Furthermore, using script checksums, PreparedJS allows fine grained control via whitelisting specific scripts. The combination of these two capabilities with the baseline protection provided by CSP, full protection against XSS attacks can be achieved in a robust fashion.

References

[1] 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 *Proceedings of the ACSAC 2011 conference*, 2011.

[2] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.

[3] Prithvi Bisht and V. N. Venkatakrishnan. Xss-guard: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA*, pages 23–43, 2008.

[4] CERT/CC. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. [online], http://www.cert.org/advisories/CA-2000-02.html (01/30/06), February 2000.

[5] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, http://www.ietf.org/rfc/rfc4627.txt, July 2006.

[6] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless attacks: stealing the pie without touching the sill. In *ACM Conference on Computer and Communications Security*, 2012.

[7] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW2007*, May 2007.

[8] Martin Johns. *Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting*. PhD thesis, University of Passau, 2009.

[9] 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.

[10] Martin Johns, Christian Beyerlein, Rosemaria Giesecke, and Joachim Posegga. Secure Code Generation for Web Applications. In *2nd International Symposium on Engineering Secure Software and Systems (ESSoS '10)*. Springer, 2010.

[11] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. [online], http://www.webappsec.org/projects/articles/071105.shtml, (05/05/07), Sebtember 2005.

[12] 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 (Oakland'09)*, May 2009.

[13] Giorgio Maone. NoScript Firefox Extension. [software], http://www.noscript.net/whats, 2006.

[14] Leo A. Meyerovich and V. Benjamin Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *IEEE Symposium on Security and Privacy*, pages 481–496. IEEE Computer Society, 2010.

[15] Yacin Nadji, Prateek Saxena, and Dawn Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS 2009*, 2009.

[16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, May 2005.

[17] 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 *CCS 2012*, 2012.

[18] Nick Nikiforakis, Wannes Meert, Yves Younan, Martin Johns, and Wouter Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *ESSoS 2011*, February 2011.

[19] Open Web Application Project (OWASP). OWASP Top 10 for 2010 (The Top Ten Most Critical Web Application Security Vulnerabilities). [online], http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2010.

[20] Open Web Application Project (OWASP). XSS (Cross Site Scripting) Prevention Cheat Sheet. [online], https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet, last accessed 12/03/12, 2012.

[21] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID2005)*, 2005.

[22] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.

[23] David Ross. IE 8 XSS Filter Architecture / Implementation. [online], http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx, last accessed 05/05/12, August 2008.

[24] Jesse Ruderman. The Same Origin Policy. [online], http://www.mozilla.org/projects/security/components/same-origin.html (01/10/06), August 2001.

[25] Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Have things changed now? an empirical study on input validation vulnerabilities in web applications. *Computers & Security*, 31(3):344–356, 2012.

[26] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *WWW*, 2010.

[27] The webappsec mailing list. The Cross Site Scripting (XSS) FAQ. [online], http://www.cgisecurity.com/articles/xss-faq.shtml, May 2002.

[28] Ben Toews. Abusing Password Managers with XSS. [online], http://labs.neohapsis.com/2012/04/25/abusing-password-managers-with-xss/, last accessed 05/05/2012, April 2012.

[29] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS 2007*, 2007.

[30] W3C. Content Security Policy 1.0. W3C Candidate Recommendation, http://www.w3.org/TR/2011/WD-CSP-20111129/, November 2012.

[31] W3C. Content Security Policy 1.1. W3C Editor's Draft 02, https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html, December 2012.

[32] Michal Zalewski. Postcards from the post-XSS world. [online], http://lcamtuf.coredump.cx/postxss/, December 2011.

6 Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting[19][20]

6.1 Introduction

In 1994, Lou Montulli, while working for Netscape Communications, introduced the idea of cookies in the context of a web browser [33]. The cookie mechanism allows a web server to store a small amount of data on the computers of visiting users, which is then sent back to the web server upon subsequent requests. Using this mechanism, a website can build and maintain state over the otherwise stateless HTTP protocol. Cookies were quickly embraced by browser vendors and web developers. Today, they are one of the core technologies on which complex, stateful web applications are built.

Shortly after the introduction of cookies, abuses of their stateful nature were observed. Web pages are usually comprised of many different resources, such as HTML, images, JavaScript, and CSS, which can be located both on the web server hosting the main page as well as other third-party web servers. With every request toward a third-party website, that website has the ability to set and read previously-set cookies on a user's browser. For instance, suppose that a user browses to *travel.com*, whose homepage includes a remote image from *tracking.com*. Therefore, as part of the process of rendering *travel.com*'s homepage, the user's browser will request the image from *tracking.com*. The web server of *tracking.com* sends the image along with an HTTP Set-Cookie header, setting a cookie on the user's machine, under the *tracking.com* domain. Later, when the user browses to other websites affiliated with *tracking.com*, e.g., *buy.com*, the tracking website receives its previously-set cookies, recognizes the user, and creates a profile of the user's browsing habits. These *third-party cookies*, due to the adverse effects on a user's privacy and their direct connection with online behavioral advertising, captured the attention of both the research community [18, 19, 30] and the popular media outlets [34] and, ever since, cause the public's discomfort [36, 37].

[19]This paper has been published as [27]: Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, Giovanni Vigna, Cookieless monster: Exploring the ecosystem of web-based device fingerprinting, IEEE Security and Privacy, San Francisco, 19-22 May 2013

[20]For KU Leuven, this research was performed with the financial support of the Prevention against Crime Programme of the European Union (B-CCENTRE), the Research Fund KU Leuven, the EU FP7 projects NESSoS and WebSand, as well as the IWT project SPION. For UCSB, this work was supported by the Office of Naval Research (ONR) under grant N000140911042, and by the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and in part by Secure Business Austria.

The user community responded to this privacy threat in multiple ways. A recent cookie-retention study by comScore [7] showed that approximately one in three users delete both first-party and third-party cookies within a month after their visit to a website. Multiple browser-extensions are available that reveal third-party tracking [13], as well as the "hidden" third-party affiliations between sites [6]. In addition, modern browsers now have native support for the rejection of all third-party cookies and some even enable it by default. Lastly, a browser's "Private Mode" is also available to assist users to visit a set of sites without leaving traces of their visit on their machine.

This general unavailability of cookies motivated advertisers and trackers to find new ways of linking users to their browsing histories. Mayer in 2009 [21] and Eckersley in 2010 [10] both showed that the features of a browser and its plugins can be fingerprinted and used to track users without the need of cookies. Today, there is a small number of commercial companies that use such methods to provide *device identification* through web-based fingerprinting. Following the classification of Mowery et al. [24], fingerprinting can be used either constructively or destructively. Constructively, a correctly identified device can be used to combat fraud, e.g., by detecting that a user who is trying to login to a site is likely an attacker who stole a user's credentials or cookies, rather than the legitimate user. Destructively, device identification through fingerprinting can be used to track users between sites, without their knowledge and without a simple way of opting-out. Additionally, device identification can be used by attackers in order to deliver exploits, tailored for specific combinations of browsers, plugins and operating systems [16]. The line between the constructive and destructive use is, however, largely artificial, because the same technology is used in both cases.

Interestingly, companies were offering fingerprinting services as early as 2009, and experts were already voicing concerns over their impact on user privacy [23]. Even when fingerprinting companies honor the recently-proposed "Do Not Track" (DNT) header, the user is still fingerprinted for fraud detection, but the companies *promise* not to use the information for advertising purposes [1]. Note that since the fingerprinting scripts will execute regardless of the DNT value, the verification of this promise is much harder than verifying the effect of DNT on stateful tracking, where the effects are visible at the client-side, in a user's cookies [20].

In this paper, we perform a four-pronged analysis of device identification through web-based fingerprinting. First, we analyze the fingerprinting code of three large, commercial companies. We focus on the differences of their code in comparison to Panopticlick [10], Eckersley's "open-source" implementation of browser fingerprinting. We identify the heavy use of Adobe Flash as a way of retrieving more sensitive information from a client, including the ability to detect HTTP proxies, and the existence of intrusive fingerprinting plugins that users may unknowingly host in their browsers. Second, we measure the adoption of fingerprinting on the Internet and show that, in many cases, sites of dubious nature fingerprint their users, for a variety of purposes. Third, we investigate special JavaScript-accessible browser objects, such as `navigator` and `screen`, and describe novel fingerprinting techniques that can accurately identify a browser even down to its minor version. These techniques involve the ordering of methods and properties, detection of vendor-specific methods, HTML/CSS functionality as well as minor but fingerprintable implementation choices. Lastly, we examine and test browser extensions that are available for users who wish to spoof the identity of their browser and show that, unfortunately *all* fail to completely hide the browser's true identity. This incomplete coverage not only voids the extensions but, ironically, also allows fingerprinting companies to detect the fact that user is attempting to hide, adding extra fingerprintable information.

Our main contributions are:

- We shed light into the current practices of device identification through web-based fingerprinting and propose a taxonomy of fingerprintable information.
- We measure the adoption of fingerprinting on the web.
- We introduce novel browser-fingerprinting techniques that can, in milliseconds, uncover a browser's family and version.
- We demonstrate how over 800,000 users, who are currently utilizing user-agent-spoofing extensions, are more fingerprintable than users who do not attempt to hide their browser's identity, and challenge the advice given by prior research on the use of such extensions as a way of increasing one's privacy [42].

6.1.1 Commercial Fingerprinting

While Eckersley showed the principle possibility of fingerprinting a user's browser in order to track users without the need of client-side stateful identifiers [10], we wanted to investigate popular, real-world implementations of fingerprinting and explore their workings. To this end, we analyzed the fingerprinting libraries of three large, commercial companies: BlueCava[21], Iovation[22] and ThreatMetrix[23]. Two of these companies were chosen due to

[21] http://www.bluecava.com
[22] http://www.iovation.com
[23] http://www.threatmetrix.com

Fingerprinting Category	**Panopticlick**	**BlueCava**	**Iovation ReputationManager**	**ThreatMetrix**
Browser customizations	Plugin enumeration$_{\text{(JS)}}$	Plugin enumeration$_{\text{(JS)}}$		Plugin enumeration$_{\text{(JS)}}$
	Mime-type enumeration$_{\text{(JS)}}$	▨ ActiveX + 53 CLSIDs$_{\text{(JS)}}$		Mime-type enumeration$_{\text{(JS)}}$
	ActiveX + 8 CLSIDs$_{\text{(JS)}}$	▨ Google Gears Detection$_{\text{(JS)}}$		ActiveX + 6 CLSIDs$_{\text{(JS)}}$
				▨ Flash Manufacturer$_{\text{(FLASH)}}$
Browser-level user configurations	Cookies enabled$_{\text{(HTTP)}}$	▨ System/Browser/User Language$_{\text{(JS)}}$	▨ Browser Language$_{\text{(HTTP, JS)}}$	▨ Browser Language$_{\text{(FLASH)}}$
	Timezone$_{\text{(JS)}}$	Timezone$_{\text{(JS)}}$	Timezone$_{\text{(JS)}}$	Timezone$_{\text{(JS, FLASH)}}$
	Flash enabled$_{\text{(JS)}}$	Flash enabled$_{\text{(JS)}}$	Flash enabled$_{\text{(JS)}}$	Flash enabled$_{\text{(JS)}}$
		▨ Do-Not-Track User Choice$_{\text{(JS)}}$	▨ Date & time$_{\text{(JS)}}$	▨ Proxy Detection$_{\text{(FLASH)}}$
		▨ MSIE Security Policy$_{\text{(JS)}}$	▨ Proxy Detection$_{\text{(FLASH)}}$	
Browser family & version	User-agent$_{\text{(HTTP)}}$	▨ User-agent$_{\text{(JS)}}$	▨ User-agent$_{\text{(HTTP, JS)}}$	▨ User-agent$_{\text{(JS)}}$
	ACCEPT-Header$_{\text{(HTTP)}}$	▨ Math constants$_{\text{(JS)}}$		
	Partial S.Cookie test$_{\text{(JS)}}$	▨ AJAX Implementation$_{\text{(JS)}}$		
Operating System & Applications	User-agent$_{\text{(HTTP)}}$	User-agent$_{\text{(JS)}}$	User-agent$_{\text{(HTTP, JS)}}$	User-agent$_{\text{(JS)}}$
	Font Detection$_{\text{(FLASH, JAVA)}}$	▨ Font Detection$_{\text{(JS, FLASH)}}$	▨ Windows Registry$_{\text{(SFP)}}$	Font Detection$_{\text{(FLASH)}}$
		▨ Windows Registry$_{\text{(SFP)}}$	▨ MSIE Product key$_{\text{(SFP)}}$	▨ OS+Kernel version$_{\text{(FLASH)}}$
Hardware & Network	Screen Resolution$_{\text{(JS)}}$	Screen Resolution$_{\text{(JS)}}$	Screen Resolution$_{\text{(JS)}}$	▨ Screen Resolution$_{\text{(JS, FLASH)}}$
		▨ Driver Enumeration$_{\text{(SFP)}}$	▨ Device Identifiers$_{\text{(SFP)}}$	
		▨ IP Address$_{\text{(HTTP)}}$	▨ TCP/IP Parameters$_{\text{(SFP)}}$	
		▨ TCP/IP Parameters$_{\text{(SFP)}}$		

Table 10: Taxonomy of all features used by Panopticlick and the studied fingerprinting providers - shaded features are, in comparison to Panopticlick, either sufficiently extended, or acquired through a different method, or entirely new

them being mentioned in the web-tracking survey of Mayer and Mitchell [22], while the third one was chosen due to its high ranking on a popular search engine. Given the commercial nature of the companies, in order to analyze the fingerprinting scripts we first needed to discover websites that make use of them. We used Ghostery [13], a browser-extension which lists known third-party tracking libraries on websites, to obtain the list of domains which the three code providers use to serve their fingerprinting scripts. Subsequently, we crawled popular Internet websites, in search for code inclusions, originating from these fingerprinting-owned domains. Once these web sites were discovered, we isolated the fingerprinting code, extracted all individual features, and grouped similar features of each company together.

In this section, we present the results of our analysis, in the form of a taxonomy of possible features that can be acquired through a fingerprinting library. This taxonomy covers all the features described in Panopticlick [10] as well as the features used by the three studied fingerprinting companies. Table 10 lists all our categories and discovered features, together with the method used to acquire each feature. The categories proposed in our taxonomy resulted by viewing a user's fingerprintable surface as belonging to a layered system, where the "application layer" is the browser and any fingerprintable in-browser information. At the top of this taxonomy, scripts seek to fingerprint and identify any browser customizations that the user has directly or indirectly performed. In lower levels, the scripts target user-specific information around the browser, the operating system and even the hardware and network of a user's machine. In the rest of this section, we focus on all the non-trivial techniques used by the studied fingerprinting providers that were not previously described in Eckersley's Panopticlick [10].

6.1.2 Fingerprinting through popular plugins

As one can see in Table 10, all companies use Flash, in addition to JavaScript, to fingerprint a user's environment. Adobe Flash is a proprietary browser plug-in that has enjoyed wide adoption among users, since it provided ways of delivering rich media content that could not traditionally be displayed using HTML. Despite the fact that Flash has been criticized for poor performance, lack of stability, and that newer technologies, like HTML5, can potentially deliver what used to be possible only through Flash, it is still available on the vast majority of desktops.

We were surprised to discover that although Flash reimplements certain APIs existing in the browser and accessible through JavaScript, its APIs do not always provide the same results compared to the browser-equivalent functions. For instance, for a Linux user running Firefox on a 64-bit machine, when querying a browser about the platform of execution, Firefox reports “Linux x86_64”. Flash, on the other hand, provides the full kernel version, e.g., Linux 3.2.0-26-generic. This additional information is not only undesirable from a privacy perspective, but also from a security perspective, since a malicious web-server could launch an attack tailored not only to a browser and architecture but to a specific kernel. Another API call that behaves differently is the one that reports the user’s screen resolution. In the Linux implementations of the Flash plugin (both Adobe’s and Google’s), when a user utilizes a dual-monitor setup, Flash reports as the width of a screen the sum of the two individual screens. This value, when combined with the browser’s response (which lists the resolution of the monitor were the browser-window is located), allows a fingerprinting service to detect the presence of multiple-monitor setups.

Somewhat surprisingly, none of the three studied fingerprinting companies utilized Java. One of them had some dead code that revealed that in the past it probably did make use of Java, however, the function was not called anymore and the applet was no longer present on the hard-coded location listed in the script. This is an interesting deviation from Panopticlick, which did use Java as an alternate way of obtaining system fonts. We consider it likely that the companies abandoned Java due to its low market penetration in browsers. This, in turn, is most likely caused by the fact that many have advised the removal of the Java plugin from a user’s browser [5, 17] due to the plethora of serious Java vulnerabilities that were discovered and exploited over the last few years.

6.1.3 Vendor-specific fingerprinting

Another significant difference between the code we analyzed and Panopticlick is that, the fingerprinting companies were not trying to operate in the same way across all browsers. For instance, when recognizing a browser as Internet Explorer, they would extensively fingerprint Internet-Explorer-specific properties, such as `navigator.securityPolicy` and `navigator.systemLanguage`. At the same time, the code accounted for the browser’s “short-comings,” such as using a lengthy list of predefined CLSIDs for Browser-Helper-Objects (BHOs) due to Internet Explorer’s unwillingness to enumerate its plugins.

6.1.4 Detection of fonts

The system’s list of fonts can serve as part of a user’s unique fingerprint [10]. While a browser does not directly provide that list, one can acquire it using either a browser plugin that willingly provides this information or using a side-channel that indirectly reveals the presence or absence of any given font.

Plugin-based detection ActionScript, the scripting language of Flash, provides APIs that include methods for discovering the list of fonts installed on a running system. While this traditionally was meant to be used as a way of ensuring the correct appearance of text by the plugin, it can also be used to fingerprint the system. Two out of the three studied companies were utilizing Flash as a way of discovering which fonts were installed on a user's computer. Interestingly, only one of the companies was preserving the order of the font-list, which points, most likely, to the fact that the other is unaware that the order of fonts is stable and machine-specific (and can thus be used as an extra fingerprinting feature).

```
function get_text_dimensions(font){

  h = document.getElementsByTagName("BODY")[0];
  d = document.createElement("DIV");
  s = document.createElement("SPAN");

  d.appendChild(s);
  d.style.fontFamily = font;
  s.style.fontFamily = font;
  s.style.fontSize = "72px";
  s.innerHTML = "font_detection";
  h.appendChild(d);

  textWidth = s.offsetWidth;
  textHeight = s.offsetHeight;
  h.removeChild(d);

  return [textWidth, textHeight];
}
```

Listing 10: Side-channel inference of the presence or absence of a font

Side-channel inference The JavaScript code of one of the three fingerprinting companies included a fall-back method for font-detection, in the cases where the Flash plugin was unavailable. By analyzing that method, we discovered that they were using a technique, similar to the CSS history stealing technique [14], to identify the presence or absence of any given font - see Listing 10.

Font Family	String	Width x Height
Sans	font_detection	519x84
Arial	font_detection	452x83
Calibri	font_detection	416x83

Figure 19: The same string, rendered with different fonts, and its effects on the string's width and height, as reported by the Google Chrome browser

More precisely, the code first creates a `<div>` element. Inside this element, the code then creates a `<span>` element with a predetermined text string and size, using a provided font family. Using the `offsetWidth` and `offsetHeight` methods of HTML elements, the script discovers the layout width and height of the element. This code is first called with a "sans" parameter, the font typically used by browsers as a fall-back, when another requested font is unavailable on a user's system. Once the height and text for "sans" are discovered, another script goes over a predefined list of fonts, calling the `get_text_dimensions` function for each one. For any given font, if the current width or height values are different from the ones obtained through the original "sans" measurement, this means that the font does exist and was used to render the predefined text. The text and its size are always kept constant, so that if its width or height change, this change will only be due to the different font. Figure 19 shows three renderings of the same text, with the same font-size but different font faces in Google Chrome. In order to capitalize as much as possible on small differences between fonts, the font-size is always large, so that even the smallest of details in each individual letter will add up to measurable total difference in the text's height and width. If the height and width are identical to the original measurement, this means that the requested font did not exist on the current system and thus, the browser has selected the sans fall-back font. All of the above process, happens in an invisible iframe created and controlled by the fingerprinting script and thus completely hidden from the user.

Using this method, a fingerprinting script can rapidly discover, even for a long list of fonts, those that are present on the operating system. The downside of this approach is that less popular fonts may not be detected, and that the font-order is no longer a fingerprintable feature.

6.1.5 Detection of HTTP Proxies

One of the features that are the hardest to spoof for a client is its IP address. Given the nature of the TCP protocol, a host cannot pretend to be listening at an IP address from which it cannot reliably send and receive packets. Thus, to hide a user's IP address, another networked machine (a proxy) is typically employed that relays packets between the user that wishes to remain hidden and a third-party. In the context of browsers, the most common type of proxies are HTTP proxies, through which users configure their browsers to send all requests. In addition to manual configuration, browser plugins are also available that allow for a more controlled use of remote proxies, such as the automatic routing of different requests to different proxies based on pattern matching of each request[24], or the cycling of proxies from a proxy list at user-defined intervals[25].

From the point of view of device identification through fingerprinting, a specific IP address is an important feature. Assuming the use of fingerprinting for the detection of fraudulent activities, the distinction between a user who is situated in a specific country and one that *pretends* to be situated in that country, is crucial. Thus, it is in the interest of the fingerprint provider to detect a user's real IP address or, at least, discover that the user is utilizing a proxy server.

When analyzing the ActionScript code embedded in the SWF files of two of the three fingerprinting companies, we found evidence that the code was circumventing the user-set proxies at the level of the browser, i.e., the loaded Flash application was contacting a remote host directly, disregarding any browser-set HTTP proxies. We verified this behavior by employing both an HTTP proxy and a packet-capturing application, and noticing that certain requests were captured by the latter but were never received by the former. In the code of both of the fingerprinting companies, certain long alphanumerical tokens were exchanged between JavaScript and Flash and then used in their communication to the server. While we do not have access to the server-side code of the fingerprinting providers, we assume that the identifiers are used to correlate two possibly different IP addresses. In essence, as shown in Figure 20, if a JavaScript-originating request contains the same token as a Flash-originating request from a different source IP address, the server can be certain that the user is utilizing an HTTP proxy.

Flash's ability to circumvent HTTP proxies is a somewhat known issue among privacy-conscious users that has lead to the disabling of Flash in anonymity-providing applications, like TorButton [35]. Our analysis shows

[24]FoxyProxy - `http://getfoxyproxy.org/`
[25]ProxySwitcher - `http://www.proxyswitcher.com/`

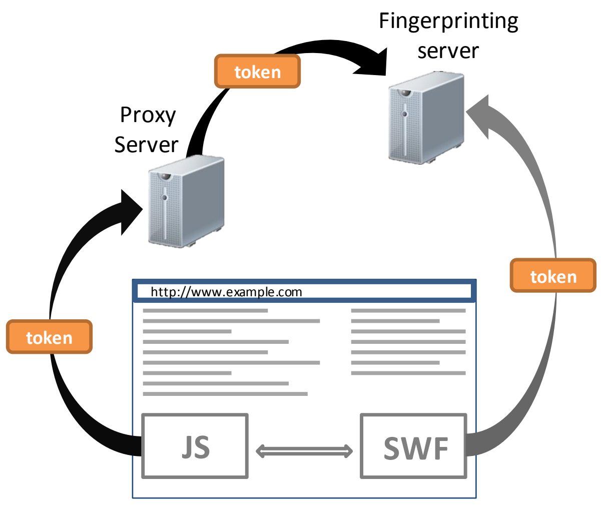

Figure 20: Fingerprinting libraries take advantage of Flash's ability to ignore browser-defined HTTP proxies to detect the real IP address of a user

that it is actively exploited to identify and bypass web proxies.

6.1.6 System-fingerprinting plugins

Previous research on fingerprinting a user's browser focused on the use of popular browser plugins, such as Flash and Java, and utilized as much of their API surface as possible to obtain user-specific data [21, 10]. However, while analyzing the plugin-detection code of the studied fingerprinting providers, we noticed that two out of the three were searching a user's browser for the presence of a special plugin, which, if detected, would be loaded and then invoked. We were able to identify that the plugins were essentially native fingerprinting libraries, which are distributed as CAB files for Internet Explorer and eventually load as DLLs inside the browser. These plugins can reach a user's system, either by a user accepting their installation through an ActiveX dialogue, or bundled with applications that users download on their machines. DLLs are triggered by JavaScript through ActiveX, but they run natively on the user's machine, and thus can gather as much information as the Internet Explorer process.

We downloaded both plugins, wrapped each DLL into an executable that simply hands-off control to the main routine in the DLL and submitted both executables to Anubis [4], a dynamic malware analysis platform that executes submitted binaries in a controlled environment. We focused on the Windows registry values that were read by the plugin, since the registry is a rich environment for fingerprinting. The submitted fingerprinting DLLs were reading a plethora of system-specific values, such as the hard disk's identifier, TCP/IP parameters, the computer's name, Internet Explorer's product identifier, the installation date of Windows, the Windows Digital Product Id and the installed system drivers – entries marked with SFP in Table 10.

All of these values combined provide a much stronger fingerprint than what JavaScript or Flash could ever construct. It is also worthwhile mentioning that one of the two plugins was misleadingly identifying itself as "ReputationShield" when asking the user whether she wants to accept its installation. Moreover, none of 44 antivirus engines of VirusTotal [38] identified the two DLLs as malicious, even though they clearly belong to the *spyware* category. Using identifiers found within one DLL, we were also able to locate a Patent Application for Iovation's fingerprinting plugin that provides further information on the fingerprinting process and the gathered data [29].

6.1.7 Fingerprint Delivery Mechanism

In the fingerprinting experiments of Mayer [21] and Eckersley [10], there was a 1-to-1 relationship between the page conducting the fingerprinting and the backend storing the results. For commercial fingerprinting, however, there is a N-to-1 relationship, since each company provides fingerprinting services to many websites (through the inclusion of third-party scripts) and needs to obtain user fingerprints from each of these sites. Thus, the way that the fingerprint and the information about it are delivered is inherently different from the two aforementioned experiments.

Through our code analysis, we found two different scenarios of fingerprinting. In the first scenario, the first-party site was not involved in the fingerprinting process. The fingerprinting code was delivered by an advertising syndicator, and the resulting fingerprint was sent back to the fingerprinting company. This was most likely done to combat click-fraud, and it is unclear whether the first-party site is even aware of the fact that its users are being fingerprinted.

In the second scenario, where the first-party website is the one requesting the fingerprint, we saw that two out of the three companies were adding the final fingerprint of the user into the DOM of the hosting page. For instance, `www.imvu.com` is using BlueCava for device fingerprinting by including remote scripts hosted on BlueCava's servers. When BlueCava's scripts combine all features into a single fingerprint, the fingerprint is DES-encrypted (DES keys generated on the fly and then encrypted with a public key), concatenated with the encrypted keys and finally converted to Base64 encoding. The resulting string is added into the DOM of `www.imvu.com`; more precisely, as a new hidden input element in IMVU's login form. In this way, when the user submits her username and password, the fingerprint is also sent to IMVU's web servers. Note, however, that IMVU cannot decrypt the fingerprint and must thus submit it back to BlueCava, which will then reply with a "trustworthiness" score and other device information. This architecture allows BlueCava to hide the implementation details from its clients and to correlate user profiles across its entire client-base. Iovation's fingerprinting scripts operate in a similar manner.

Contrastingly, ThreatMetrix delivers information about users in a different way. The including site, i.e., a customer of ThreatMetrix, creates a session identifier that it places into a `<div>` element with a predefined identifier. ThreatMetrix's scripts, upon loading, read this session identifier and append it to all requests towards the ThreatMetrix servers. This means that the including site never gets access to a user's fingerprint, but only information about the user by querying ThreatMetrix for specific session identifiers.

6.1.8 Analysis Limitations

In the previous sections we analyzed the workings of the fingerprinting libraries of three popular commercial companies. The analysis was a mostly manual, time-consuming process, where each piece of code was gradually deobfuscated until the purpose of all functions was clear. Given the time required to fully reverse-engineer each library, we had to limit ourselves to analyze the script of each fingerprinting company as it was seen through two different sites (that is, two different clients of each company). However, we cannot exclude the possibility of additional scripts that are present on the companies' web servers that would perform more operations than the ones we encountered.

6.2 Adoption of fingerprinting

In Section 6.1.1, we analyzed the workings of three commercial fingerprinting companies and focused on the differences of their implementations when compared to Panopticlick [10]. In this section, we study the fingerprinting ecosystem, from the point of view of websites that leverage fingerprinting.

6.2.1 Adoption on the popular web

To quantify the use of web-based fingerprinting on popular websites, we crawled up to 20 pages for each of the Alexa top 10,000 sites, searching for script inclusions and iframes originating from the domains that the three studied companies utilize to serve their fingerprinting code. To categorize the discovered domains, we made use of the publicly-available domain categorization service of TrendMicro [26], a popular anti-virus vendor.

Through this process, we discovered 40 sites (0.4% of the Alexa top 10,000) utilizing fingerprinting code from the three commercial providers. The most popular site making use of fingerprinting is *skype.com*, while the two most popular categories of sites are: “Pornography” (15%) and “Personals/Dating” (12.5%). For pornographic sites, a reasonable explanation is that fingerprinting is used to detect shared or stolen credentials of paying members, while for dating sites to ensure that attackers do not create multiple profiles for social-engineering purposes. Our findings show that fingerprinting is already part of some of the most popular sites of the Internet, and thus the hundreds of thousands of their visitors are fingerprinted on a daily basis.

Note that the aforementioned adoption numbers are lower bounds since our results do not include pages of the 10,000 sites that were not crawled, either because they were behind a registration wall, or because they were not in the set of 20 URLs for each crawled website. Moreover, some popular sites may be using their own fingerprinting algorithms for performing device identification and not rely on the three studied fingerprinting companies.

6.2.2 Adoption by other sites

To discover less popular sites making use of fingerprinting, we used a list of 3,804 domains of sites that, when analyzed by Wepawet [8], requested the previously identified fingerprinting scripts.

Each domain was submitted to TrendMicro’s and McAfee’s categorization services [27] which provided as output the domain’s category and “safety” score. We used two categorizing services in an effort to reduce, as much as possible, the number of “untested” results, i.e., the number of websites not analyzed and not categorized. By examining the results, we extracted as many popular categories as possible and created aliases for names that were referring to the same category, such as “News / Media” versus “General News” and “Disease Vector” versus “Malicious Site”. If a domain was characterized as “dangerous” by one, and “not dangerous” by the other, we accepted the categorization of the latter, so as to give the benefit of the doubt to legitimate websites that could have been compromised, when the former service categorized it.

[26]TrendMicro - `http://global.sitesafety.trendmicro.com/`
[27]McAfee -`http://mcafee.com/threat-intelligence/domain/`

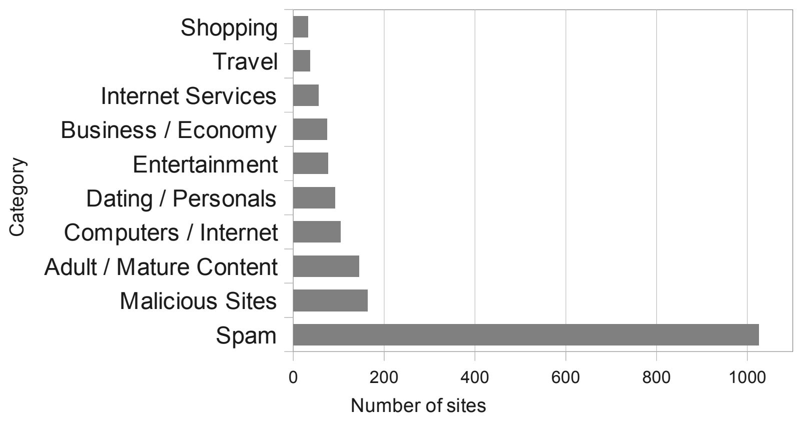

Figure 21: The top 10 categories of websites utilizing fingerprinting

Given the use of two domain-categorization services, a small number of domains (7.9%) was assigned conflicting categories, such as "Dating" versus "Adult/Mature" and "Business/Economy" versus "Software/Hardware." For these domains, we accepted the characterization of McAfee, which we observed to be more precise than TrendMicro's for less popular domains. Excluding 40.8% of domains which were reported as "untested" by both services, the results of this categorization are shown in Figure 21.

First, one can observe that eight out of the ten categories, include sites which operate with user subscriptions, many of which contain personal and possibly financial information. These sites are usually interested in identifying fraudulent activities and the hijacking of user accounts. The Adult/Mature category seems to make the most use of fingerprinting as was the case with the Alexa top 10,000 sites.

The top two categories are also the ones that were the least expected. 163 websites were identified as malicious, such as using exploits for vulnerable browsers, conducting phishing attacks or extracting private data from users, whereas 1,063 sites were categorized as "Spam" by the two categorizing engines. By visiting some sites belonging to these categories, we noticed that many of them are parked webpages, i.e., they do not hold any content except advertising the availability of the domain name, and thus do not currently include fingerprinting code. We were however able to locate many "quiz/survey" sites that are, at the time of this writing, including fingerprinting code from one of the three studied companies. Visitors of these sites are greeted with a "Congratulations" message, which informs them that they have won and asks them to proceed to receive their prize. At some later step, these sites extract a user's personal details and try to subscribe the user to expensive mobile services.

While our data-set is inherently skewed towards “maliciousness” due to its source, it is important to point out that all of these sites were found to include, at some point in time, fingerprinting code provided by the three studied providers. This observation, coupled with the fact that for all three companies, an interested client must set an appointment with a sales representative in order to acquire fingerprinting services, point to the possibility of fingerprinting companies working together with sites of dubious nature, possibly for the expansion of their fingerprint databases and the acquisition of more user data.

6.3 Fingerprinting the behavior of special objects

In Section 6.1.1, we studied how commercial companies perform their fingerprinting and created a taxonomy of fingerprintable information accessible through a user’s browser. In Table 10, one can notice that, while fingerprinting companies go to great lengths to discover information about a browser’s plugins and the machine hosting the browser, they mostly rely on the browser to willingly reveal its true identity (as revealed through the `navigator.userAgent` property and the User-Agent HTTP header). A browser’s user-agent is an important part of a system’s fingerprint [42], and thus it may seem reasonable to assume that if users modify these default values, they will increase their privacy by hiding more effectively from these companies.

In this section, however, we demonstrate how fragile the browser ecosystem is against fingerprinting. Fundamental design choices and differences between browser types are used in an effort to show how difficult it can be to limit the exposure of a browser to fingerprinting. Even different versions of the same browser can have differences in the scripting environment that identify the browser’s real family, version, and, occasionally, even the operating system. In the rest of this section we describe several novel browser-identifying techniques that: a) can complement current fingerprinting, and b) are difficult to eliminate given the current architecture of web browsers.

6.3.1 Experimental Fingerprinting Setup

Our novel fingerprinting techniques focus on the special, browser-populated JavaScript objects; more precisely, the `navigator` and `screen` objects. Contrary to objects created and queried by a page’s JavaScript code, these objects contain vendor- and environment-specific methods and properties, and are thus the best candidates for uncovering vendor-specific behaviors.

To identify differences between browser-vendors and to explore whether these differences are consistent among installations of the same browser on multiple systems, we constructed a fingerprinting script that performed a series of “everyday” operations on these two special objects (such as adding a new property to an object, or modifying an existing one) and reported the results to a server. In this and the following section, we describe the operations of our fingerprinting script and our results. Our constructed page included a JavaScript program that performed the following operations:

1. Enumerated the `navigator` and `screen` object, i.e., request the listing of all properties of the aforementioned objects.

2. Enumerated the `navigator` object again, to ensure that the order of enumeration does not change.

3. Created a custom object, populated it, and enumerated it. A custom, JavaScript-created object, allows us to compare the behavior of browser-populated objects (such as `navigator`) with the behavior of “classic” JavaScript objects.

4. Attempted to delete a property of the `navigator` object, the `screen` object, and the custom object.

5. Add the possibly-deleted properties back to their objects.

6. Attempted to modify an existing property of the `navigator` and `screen` objects.

7. If `Object.defineProperty` is implemented in the current browser, utilize it to make an existing property in the `navigator`, `screen`, and custom object non-enumerable.

8. Attempt to delete the `navigator` and `screen` objects.

9. Attempt to assign new custom objects to the `navigator` and `screen` variable names.

At each step, the objects involved were re-enumerated, and the resulting data was Base64-encoded and sent to our server for later processing. Thus, at the server side, we could detect whether a property was deleted or modified, by comparing the results of the original enumeration with the current one. The enumeration of each object was conducted through code that made use of the *prop in obj* construct, to avoid forcing a specific order of enumeration of the objects, allowing the engine to list object properties in the way of its choosing.

6.3.2 Results

By sharing the link to our fingerprinting site with friends and colleagues, we were able, within a week, to gather data from 68 different browsers installations, of popular browsers on all modern operating systems. While our data is small in comparison to previous studies [21, 10], we are not using it to draw conclusions that have statistical relevance but rather, as explained in the following sections, to find deviations between browsers and to establish the consistency of these deviations. We were able to identify the following novel ways of distinguishing between browsers:

Order of enumeration Through the analysis of the output from the first three steps of our fingerprinting algorithm (Sec. 6.3.1), we discovered that the order of property-enumeration of special browser objects, like the `navigator` and `screen` objects, is consistently different between browser families, versions of each browser, and, in some cases, among deployments of the same version on different operating systems. While in the rest of this section we focus to the `navigator` object, the same principles apply to the `screen` object.

Our analysis was conducted in the following manner. After grouping the `navigator` objects and their enumerated properties based on browser families, we located the `navigator` object with the least number of properties. This version was consistently belonging to the oldest version of a browser, since newer versions add new properties which correspond to new browser features, such as the `navigator.doNotTrack` property in the newer versions of Mozilla Firefox. The order of the properties of this object, became our baseline to which we compared the `navigator` objects of all subsequent versions of the same browser family. To account for ordering changes due to the introduction of new properties in the `navigator` object, we simply excluded all properties that were not part of our original baseline object, without however changing the relative order of the rest of the properties. For instance, assume an ordered set of features B, where $B_0 = \{a, b, c, d\}$ and $B_1 = \{a, b, \underline{e}, c, d, \underline{f}\}$. B_1 has two new elements in comparison with B_0, namely `e` and `f` which, however, can be removed from the set without disrupting the relative order of the rest. For every browser version within the same browser-family, we compared the `navigator` object to the baseline, by first recording and removing new features and then noting whether the order of the remaining features was different from the order of the baseline.

The results of this procedure are summarized in Table 11. For each browser family, we compare the ordering of the `navigator` object among up

Browser	$\mathbf{V_{c\text{-}4}}$	$\mathbf{V_{c\text{-}3}}$	$\mathbf{V_{c\text{-}2}}$	$\mathbf{V_{c\text{-}1}}$	$\mathbf{V_c}$
Mozilla Firefox	W	W+1	W+4	W+5	W+7
Microsoft IE	-	-	X	X	X+1
Opera	Y	Y+1	Y+1	Y+3	Y+5
Google Chrome	Z	Z	Z'+1	Z''+1	Z'''+1

Table 11: Differences in the order of `navigator` objects between versions of the same browser

to five different versions. The most current version is denoted as V_c. The first observation is that in almost 20 versions of browsers, no two were ever sharing the same order of properties in the `navigator` object. This feature by itself, is sufficient to categorize a browser to its correct family, regardless of any property-spoofing that the browser may be employing. Second, all browsers except Chrome maintain the ordering of `navigator` elements between versions. Even when new properties were introduced, these do not alter the relative order of all other properties. For instance, even though the newest version of Mozilla Firefox (V_c) has 7 extra features when compared to the oldest version ($V_{c\text{-}4}$), if we ignore these features then the ordering is the same with the original ordering (W).

Google Chrome was the only browser that did not exhibit this behavior. By analyzing our dataset, we discovered that Chrome not only changed the order between subsequent versions of the browser, but also between deployments of the same browser on different operating systems. For instance, Google Chrome v.20.0.1132.57 installed on Mac OSX has a different order of elements than the same version installed on a Linux operating system. In Table 11, we compare the order of properties of the `navigator` object when the underlying OS is Windows XP. While this changing order may initially appear to be less-problematic than the stable order of other browsers, in reality, the different orderings can be leveraged to detect a specific version of Google Chrome, and, in addition, the operating system on which the browser is running.

Overall, we discovered that the property ordering of special objects, such as the `navigator` object, is consistent among runs of the same browser and runs of the same version of browsers on different operating systems. Contrastingly, the order of properties of a custom script-created object (Step 3 in Section 6.3.1) was identical among all the studied browsers even though, according to the ECMAScript specification, objects are *unordered* collections of properties [11] and thus the exact ordering can be implementation-specific. More precisely, the property ordering of the custom objects was always the

Browser	**Unique methods & properties**
Mozilla Firefox	screen.mozBrightness screen.mozEnabled navigator.mozSms + 10
Google Chrome	navigator.webkitStartActivity navigator.getStorageUpdates
Opera	navigator.browserLanguage navigator.getUserMedia
Microsoft IE	screen.logicalXDPI screen.fontSmoothingEnabled navigator.appMinorVersion +11

Table 12: Unique methods and properties of the `navigator` and `screen` objects of the four major browser-families

same with the order of property creation.

In general, the browser-specific, distinct property ordering of special objects can be directly used to create models of browsers and, thus, unmask the real identity of a browser. Our findings are in par with the "order-matters" observation made by previous research: Mayer discovered that the list of plugins as reported by browsers was ordered based on the installation time of each individual plugin [21]. Eckersley noticed that the list of fonts, as reported by Adobe Flash and Sun's Java VM, remained stable across visits of the same user [10].

Unique features During the first browser wars in the mid-90s, browser vendors were constantly adding new features to their products, with the hope that developers would start using them. As a result, users would have to use a specific browser, effectively creating a browser lock-in [43]. The features ranged from new HTML tags to embedded scripting languages and third-party plugins. Signs of this "browser battle" are still visible in the contents of the user-agent string of modern browsers [3].

Today, even though the HTML standard is governed by the W3C committee and JavaScript by Ecma International, browser vendors still add new features that do not belong to any specific standard. While these features can be leveraged by web developers to provide users with a richer experience, they can also be used to differentiate a browser from another. Using the data gathered by our fingerprinting script, we isolated features that were available in only one family of browsers, but not in any other. These unique features are summarized in Table 12. All browser families had at least two such features that were not shared by any other browser. In many cases, the names of the new features were starting with a vendor-specific prefix, such as `screen.mozBrightness` for Mozilla Firefox and `navigator.msDoNotTrack` for Microsoft Internet Explorer. This is because browser-vendors are typically allowed to use prefixes for features not belonging to a standard or not yet standardized [39]. In the context of fingerprinting, a script can query for the presence or absence of these unique features (e.g., typeof screen.mozBrightness != "undefined") to be certain of the identity of any given browser.

An interesting side note is that these unique features can be used to expose the real version of Mozilla Firefox browser, even when the user is using the Torbutton extension. Torbutton replaces the `navigator` and `screen` objects with its own versions, spoofing the values of certain properties, so as to protect the privacy of the user [9]. We installed Torbutton on Mozilla Firefox version 14 and, by enumerating the `navigator` object, we observed that, among others, the Torbutton had replaced the `navigator.userAgent` property with the equivalent of Mozilla Firefox version 10, and it was claiming that our platform was Windows instead of Linux. At the same time, however, special Firefox-specific properties that Mozilla introduced in versions 11 to 14 of Firefox (such as `navigator.mozBattery` and `navigator.mozSms`) were still available in the `navigator` object. These discrepancies, combined with other weaknesses found in less thorough user-agent-spoofing extensions (see Section 6.4), can uncover not only that the user is trying to hide, but also that she is using Torbutton to do so.

Mutability of special objects In the two previous sections, we discussed the ability to exploit the enumeration-order and unique features of browsers for fingerprinting. In this section, we investigate whether each browser treats the `navigator` and `screen` objects like regular JavaScript objects. More precisely, we investigate whether these objects are mutable, i.e., whether a script can delete a specific property from them, replace a property with a new one, or delete the whole object. By comparing the outputs of steps four to nine from our fingerprinting algorithm, we made the following observations.

Among the four browser families, only Google Chrome allows a script to delete a property from the `navigator` object. In all other cases, while the "delete" call returns successfully and no exceptions are thrown, the properties remain present in the special object. When our script attempted to modify the value of a property of `navigator`, Google Chrome and Opera allowed it, while Mozilla Firefox and Internet Explorer ignored the request. In the same way, these two families were the only ones allowing a script to reassign `navigator` and `screen` to new objects. Interestingly, no browser allowed the script to simply delete the `navigator` or `screen` object. Finally, Mozilla Firefox behaved in a unique way when requested to make a certain property of the `navigator` object non-enumerable. Specifically, instead of just hiding the property, Firefox behaved as if it had actually deleted it, i.e., it was no longer accessible even when requested by name.

Evolution of functionality Recently, we have seen a tremendous innovation in Web technologies. The competition is fierce in the browsers' scene, and vendors are trying hard to adopt new technologies and provide a better platform for web applications. Based on that observation, in this section, we examine if we can determine a browser's version based on the new functionality that it introduces. We chose Google Chrome as our testing browser and created a library in JavaScript that tests if specific functionality is implemented by the browser. The features that we selected to capture different functionality were inspired by web design compatibility tests (where web developers verify if their web application is compatible with a specific browser). In total, we chose 187 features to test in 202 different versions of Google Chrome, spanning from version *1.0.154.59* up to *22.0.1229.8*, which we downloaded from *oldapps.com* and which covered all 22 major versions of Chrome. We found that not all of the 187 features were useful; only 109 actually changed during Google Chrome's evolution. These browser versions covered not only releases from the stable channel of Google Chrome, but also from Beta and Dev channels. We refer to a major version as the first number of Google Chrome's versioning system, and to minor version as the full number of the version. We used a virtual machine with Windows XP to setup all browser versions, and used all versions to visit our functionality-fingerprinting page.

In total, we found 71 sets of features that can be used to identify a specific version of Google Chrome. Each feature set could identify versions that range from a single Google Chrome version up to 14 different versions. The 14 Chrome versions that were sharing the same feature set were all part of the *12.0.742.** releases. Among all 71 sets, there were only four cases where the same feature set was identifying more than a single major version of the browser. In all of these cases, the features overlapped with the first

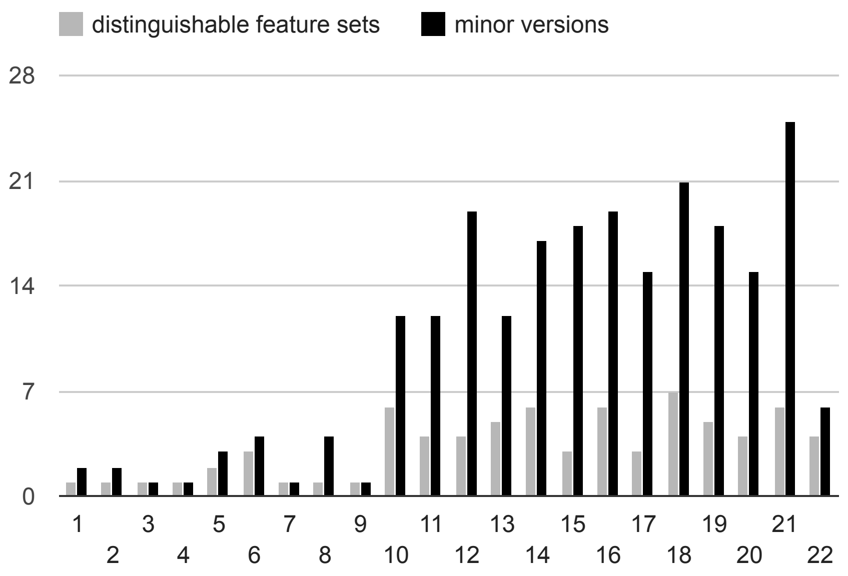

Figure 22: A comparison between how many distinguishable feature sets and minor Google Chrome versions we have per Google Chrome's major versions.

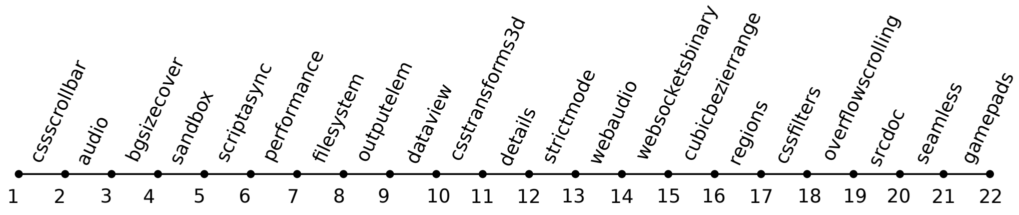

Figure 23: Feature-based fingerprinting to distinguish between Google Chrome major versions

Dev release of the next major version, while subsequent releases from that point on had different features implemented. In Figure 22, we show how many minor versions of Chrome we examined per major version and how many distinct feature sets we found for each major version. The results show that we can not only identify the major version, but in most cases, we have several different feature sets on the same major version. This makes the identification of the exact browser version even more fine-grained.

In Figure 23, we show how one can distinguish all Google Chrome's major versions by checking for specific features. Every pair of major versions is separated by a feature that was introduced into the newer version and did not exist in the previous one. Thus, if anyone wants to distinguish between two consecutive versions, a check of a single feature is sufficient to do so. Notice that our results indicate that we can perform even more fine-grained version detection than the major version of Google Chrome (we had 71 distinct sets of enabled features compared to 22 versions of Chrome), but for simplicity we examined only the major version feature changes in detail.

Miscellaneous In this section, we list additional browser-specific behaviors that were uncovered through our experiment but that do not fall in the previous categories.

Our enumeration of object-properties indirectly uses the method `toString()` for the examined objects. By comparing the formatted output of some specific properties and methods, we noticed that different browsers treated them in slightly different ways. For instance, when calling `toString()` on the natively implemented `navigator.javaEnabled` method, browsers simply state that it is a "native function." Although all the examined browser families print "function javaEnabled() { [native code] }," Firefox uses newline characters after the opening curly-bracket and before the closing one. Interestingly, Internet Explorer does not list the `navigator.javaEnabled` when requested to enumerate the `navigator` object, but still provides the "native function" print-out when asked specifically about the `javaEnabled` method. In the same spirit, when our scripts invoked the `toString()` method on the `navigator.plugins` object, Google Chrome reported "[object DOMPluginArray]," Internet Explorer reported "[object]," while both Mozilla Firefox and Opera reported "[object PluginArray]."

Lastly, while trying out our fingerprinting page with various browsers, we discovered that Internet Explorer lacks native support for Base64 encoding and decoding (`atob` and `btoa`, respectively) which our script used to encode data before sending them to the server.

6.3.3 Summary

Overall, one can see how various implementation choices, either major ones, such as the traversal algorithms for JavaScript objects and the development of new features, or minor ones, such as the presence or absence of a newline character, can reveal the true nature of a browser and its JavaScript engine.

6.4 Analysis of User-Agent-Spoofing Extensions

With the advent of browser add-ons, many developers have created extensions that can increase the security of users (e.g., extensions showing HTML forms with non-secure destinations) or their privacy (e.g., blocking known ads and web-tracking scripts).

Extension	**#Installations**	**User Rating**
Mozilla Firefox		
UserAgent Switcher	604,349	4/5
UserAgent RG	23,245	4/5
UAControl	11,044	4/5
UserAgentUpdater	5,648	3/5
Masking Agent	2,262	4/5
User Agent Quick Switch	2,157	5/5
randomUserAgent	1,657	4/5
Override User Agent	1,138	3/5
Google Chrome		
User-Agent Switcher for Chrome	123,133	4/5
User-Agent Switcher	21,108	3.5/5
Ultimate User Agent Switcher, URL sniffer	28,623	4/5

Table 13: List of user-agent-spoofing browser extensions

In the context of this paper, we were interested in studying the completeness and robustness of extensions that attempt to hide the true nature of a browser from an inspecting website. As shown in Table 10, while the studied companies do attempt to fingerprint a user's browser customizations, they currently focus only on browser-plugins and do not attempt to discover any installed browser-extensions. Given however the sustained popularity of browser-extensions [31], we consider it likely that fingerprinting extensions will be the logical next step. Note that, unlike browser plugins, extensions are not enumerable through JavaScript and, thus, can only be detected through their side-effects. For instance, some sites currently detect the use of Adblock Plus [2] by searching for the absence of specific iframes and DOM elements that are normally created by advertising scripts.

Since a browser exposes its identity through the user-agent field (available both as an HTTP header and as a property of the JavaScript-accessible `navigator` object), we focused on extensions that advertised themselves as capable of spoofing a browser's user agent. These extensions usually serve two purposes. First, they allow users to surf to websites that impose strict browser requirements onto their visitors, without fulfilling these requirements. For instance, some sites are developed and tested using one specific browser and, due to the importance of the content loading correctly, refuse to load on other browsers. Using a user-agent-spoofing extension, a user can visit such a site, by pretending to use one of the white-listed browsers.

Another reason for using these extensions is to protect the privacy of a

	Google Chrome	**Mozilla Firefox**	**MSIE**	**Opera**
navigator.product	Gecko	Gecko	N/A	N/A
navigator.appCodeName	Mozilla	Mozilla	Mozilla	Mozilla
navigator.appName	Netscape	Netscape	Microsoft Internet Explorer	Opera
navigator.platform	Linux i686	Linux x86_64	Win32	Linux
navigator.vendor	Google Inc.	(empty string)	N/A	N/A

Table 14: Standard properties of the navigator object and their values across different browser families

user. Eckeresly, while gathering data for the Panopticlick project, discovered that there were users whose browsers were reporting impossible configurations, for instance, a device was pretending to be an iPhone, but at the same time had Adobe Flash support. In that case, these were users who were obviously trying to get a non-unique browser fingerprint by Panopticlick. Since Eckersley's study showed the viability of using common browser features as parts of a unique fingerprint, it is reasonable to expect that legitimate users utilize such extensions to reduce the trackability of their online activities, even if the extensions' authors never anticipated such a use. Recently, Trusteer discovered in an "underground" forum a spoofing-guide that provided step-by-step instructions for cybercriminals who wished to fool fraud-detection mechanisms that used device-fingerprinting [15]. Among other advice, the reader was instructed to download an extension that changes the User-Agent of their browser to make their sessions appear as if they were originating by different computers with different browsers and operating systems.

Table 13 shows the Mozilla Firefox and Google Chrome extensions that we downloaded and tested, together with their user base (measured in July 2012) and the rating that their users had provided. The extensions were discovered by visiting each market, searching for "user-agent" and then downloading all the relevant extensions with a sufficiently large user base and an above-average rating. A high rating is important because it indicates the user's satisfaction in the extension fulfilling its purpose. Our testing consisted of listing the `navigator` and `screen` objects through JavaScript and inspecting the HTTP headers sent with browser requests, while the extensions were actively spoofing the identity of the browser. As in Section 6.3, we chose to focus on these two objects since they are the ones that are the most vendor-specific as well as the most probed by the fingerprinting libraries. Through our analysis, we discovered that, unfortunately, in *all* cases, the extensions were inadequately hiding the real identity of the browser, which could still be straightforwardly exposed through JavaScript. Apart from being vulnerable to every fingerprinting technique that we introduced in Section 6.3, each extension had one or more of the following issues:

- **Incomplete coverage of the navigator object.** In many cases, while an extension was modifying the `navigator.userAgent` property, it would leave intact other revealing properties of the navigator object, such as `appName`, `appVersion` and `vendor` - Table 14. Moreover, the extensions usually left the `navigator.platform` property intact, which allowed for improbable scenarios, like a Microsoft Internet Explorer browser running on Linux.

- **Impossible configurations.** None of the studied extensions attempted to alter the `screen` object. Thus, users who were utilizing laptops or normal workstations and pretended to be mobile devices, were reporting impossible screen width and height (e.g., a reported 1920x1080 resolution for an iPhone).

- **Mismatch between User-agent values.** As discussed earlier, the user-agent of any given browser is accessible through the HTTP headers of a browser request and through the userAgent property of the `navigator` object. We found that some extensions would change the HTTP headers of the browser, but not of the `navigator` object. Two out of three Chrome extensions were presenting this behavior.

We want to stress that these extensions are not malicious in nature. They are legitimately-written software that unfortunately did not account for all possible ways of discovering the true identity of the browsers on which they are installed. The downside here is that, not only fingerprinting libraries can potentially detect the actual identity of a browser, thus, undermining the goals of the extension, but also that they can discover the discrepancies between the values reported by the extensions and the values reported by the browser, and then use these differences as extra features of their fingerprints. The discrepancies of each specific extension can be modeled and thus, as with Adblock Plus, used to uncover the presence of specific extensions, through their side-effects.

The presence of any user-agent-spoofing extension is a discriminatory feature, under the assumption that the majority of browsing users are not familiar enough with privacy threats (with the possible exception of cookies) to install such spoofing extensions. As a rough metric, consider that the most popular extension for Mozilla Firefox is Adblock Plus [2] that, at the time of this writing, is installed by fifteen million users, 25 times more users than UserAgent Switcher, the most popular extension in Table 13.

We characterize the extension-problem as an *iatrogenic* [28] one. The users who install these extensions in an effort to hide themselves in a crowd of popular browsers, install software that actually makes them more visible and more distinguishable from the rest of the users, who are using their browsers without modifications. As a result, we advice against the use of user-agent-spoofing extensions as a way of increasing one's privacy. Our findings come in direct antithesis with the advice given by Yen et al. [42], who suggest that user-agent-spoofing extensions *can* be used, as a way of making tracking harder. Even though their study focuses on common identifiers as reported by client-side HTTP headers and the client's IP address, a server capable of viewing these can respond with JavaScript code that will uncover the user-agent-spoofing extension, using any of the aforementioned techniques.

6.5 Discussion

Given the intrusive nature of web-based device fingerprinting and the current inability of browser extensions to actually enhance a user's privacy, in this section, we first discuss possible ways of reducing a user's fingerprintable surface and then briefly describe alternative uses of fingerprinting which may become more prevalent in the future.

6.5.1 Reducing the fingerprintable surface

Flash. As described in Section 6.1.1, Adobe Flash was utilized by all three fingerprinting libraries that we studied, due to its rich API that allow SWF files to access information not traditionally available through a browser's API. In all cases, the SWF file responsible for gathering information from the host was hidden from the user, by either setting the width and height of the `<object>` tag to zero, or placed into an iframe of zero height and width. In other words, there was no visible change on the web page that included the fingerprinting SWF files. This observation can be used as a first line of defense. All modern browsers have extensions that disallow Flash and Silverlight to be loaded until explicitly requested by the user (e.g., through a click on the object itself). These hidden files cannot be clicked on and thus, will never execute. While this is a straightforward solution that would effectively stop the Flash-part of the fingerprint of all three studied companies, a circumvention of this countermeasure is possible. By wrapping their finger-

[28] *iatrogenic* - Of or relating to illness caused by medical examination or treatment.

printing code into an object of the first-party site and making that object desirable or necessary for the page's functionality, the fingerprinting companies can still execute their code. This, however, requires much more integration between a first-party website and a third-party fingerprinting company than the current model of "one-size-fits-all" JavaScript and Flash.

In the long run, the best solution against fingerprinting through Flash should come directly from Flash. In the past, researchers discovered that Flash's Local Shared Objects, i.e., Flash's equivalent of browser cookies, were not deleted when a user exited her browser's private mode or even when she used the "Clear Private Data" option of her browser's UI [32]. As a result, in the latest version of Flash, LSOs are not stored to disk but simply kept in memory when the browser's private mode is utilized [41]. Similarly, when a browser enters private mode, Flash could provide less system information, respect any browser-set HTTP proxies and possibly report only a standard subset of a system's fonts, to protect a user's environment from fingerprinting.

JavaScript. There are multiple vendors involved in the development of JavaScript engines, and every major browser is equipped with a different engine. To unify the behavior of JavaScript under different browsers, all vendors would need to agree not only on a single set of API calls to expose to the web applications, but also to internal implementation specifics. For example, hash table implementations may affect the order of objects in the exposed data structures of JavaScript, something that can be used to fingerprint the engine's type and version. Such a consensus is difficult to achieve among all browser vendors, and we have seen diversions in the exposed APIs of JavaScript even in the names of functions that offer the same functionality, e.g., `execScript` and `eval`. Also, based on the fact that the vendors *battle* for best performance of their JavaScript engines, they might be reluctant to follow specific design choices that might affect performance.

At the same time, however, browsers could agree to sacrifice performance when "private-mode" is enabled, where there could be an attempt to expose a unified interface.

6.5.2 Alternative uses of fingerprinting

Although, in this paper, we have mostly focused on fingerprinting as a fraud-detection and web-tracking mechanism, there is another aspect that requires attention. Drive-by downloads and web attacks in general use fingerprinting to understand if the browser that they are executing on is vulnerable to one of the multiple available exploits. This way, the attackers can decide, at the server-side, which exploit to *reveal* to the client, exposing as little as they can of their attack capabilities. There are three different architectures to detect drive-by downloads: low-interaction honeypots, high-interaction honeypots and honeyclients. In all three cases, the browser is either a specially crafted one, so that it can instrument the pages visited, or a browser installation that was never used by a real user. Given the precise, browser-revealing, fingerprinting techniques that we described in this paper, it is possible to see in the future these mechanisms being used by attackers to detect monitoring environments and circumvent detection.

6.6 Related Work

To the best of our knowledge, this paper is the first that attempts to study the problem of web-based fingerprinting from the perspectives of all the players involved, i.e., from the perspective of the fingerprinting providers and their fingerprinting methods, the sites utilizing fingerprinting, the users who employ privacy-preserving extensions to combat fingerprinting, and the browser's internals and how they relate to its identity.

Eckersley conducted the first large-scale study showing that various properties of a user's browser and plugins can be combined to form a unique fingerprint [10]. More precisely, Eckersley found that from about 500,000 users who visited `panopticlick.eff.org` and had Flash or Java enabled, 94.2% could be uniquely identified, i.e., there was no other user whose environment produced the same fingerprint. His study, and surprisingly accurate identification results, prompted us to investigate commercial fingerprinting companies and their approach. Yen et al. [42] performed a fingerprinting study, similar to Eckersley's, by analyzing month-long logs of Bing and Hotmail. Interestingly, the authors utilize a client's IP address as part of their tracking mechanism, which Eckersley explicitly avoids dismissing it as "not sufficiently stable." As a way of protecting oneself, the authors advocated the use of user-agent-spoofing extensions. As we discussed in Section 6.4, this is actually counter-productive since it allows for more fingerprinting rather than less.

Mowery et al. [24] proposed the use of benchmark execution time as a way of fingerprinting JavaScript implementations, under the assumption that specific versions of JavaScript engines will perform in a consistent way. Each browser executes a set of predefined JavaScript benchmarks, and the completion-time of each benchmark forms a part of the browser's performance signature. While their method correctly detects a browser-family (e.g., Chrome) 98.2% of the time, it requires over three minutes to fully execute. According to a study conducted by Alenty [12], the average view-time of a web page is 33 seconds. This means that, with high likelihood, the benchmarks will not be able to completely execute and thus, a browser may be misclassified. Moreover, the reported detection rate of more specific attributes, such as the browser-version, operating system and architecture, is significantly less accurate.

Mowery and Shacham later proposed the use of rendering text and WebGL scenes to a `<canvas>` element as another way of fingerprinting browsers [25]. Different browsers will display text and graphics in a different way, which, however small, can be used to differentiate and track users between page loads. While this method is significantly faster than the execution of browser benchmarks, these technologies are only available in the latest versions of modern browsers, thus they cannot be used to track users with older versions. Contrastingly, the fingerprinting techniques introduced in Section 6.3 can be used to differentiate browsers and their versions for any past version.

Olejnik et al. [28] show that web history can also be used as a way of fingerprinting without the need of additional client-side state. The authors make this observation by analyzing a corpus of data from when the CSS-visited history bug was still present in browsers. Today, however, all modern browsers have corrected this issue and thus, extraction of a user's history is not as straightforward, especially without user interaction [40]. Olejnik et al. claim that large script providers, like Google, can use their near-ubiquitous presence to extract a user's history. While this is true [26], most users have first-party relationships with Google, meaning that they can be tracked accurately, without the need of resorting to history-based fingerprinting.

6.7 Conclusion

In this paper, we first investigated the real-life implementations of fingerprinting libraries, as deployed by three popular commercial companies. We focused on their differences when compared to Panopticlick and discovered increased use of Flash, backup solutions for when Flash is absent, broad use of Internet Explorer's special features, and the existence of intrusive system-fingerprinting plugins.

Second, we created our own fingerprinting script, using multiple novel features that mainly focused on the differences between special objects, like the `navigator` and `screen`, as implemented and handled by different browsers. We identified that each browser deviated from all the rest in a consistent and measurable way, allowing scripts to almost instantaneously discover the true nature of a browser, regardless of a browser's attempts to hide it. To this end, we also analyzed eleven popular user-agent spoofing extensions and showed that, even without our newly proposed fingerprinting techniques, all of them fall short of properly hiding a browser's identity.

The purpose of our research was to demonstrate that when considering device identification through fingerprinting, user-privacy is currently on the losing side. Given the complexity of fully hiding the true nature of a browser, we believe that this can be efficiently done only by the browser vendors. Regardless of their complexity and sophistication, browser-plugins and extensions will never be able to control everything that a browser vendor can. At the same time, it is currently unclear whether browser vendors would desire to hide the nature of their browsers, thus the discussion of web-based device fingerprinting, its implications and possible countermeasures against it, must start at a policy-making level in the same way that stateful user-tracking is currently discussed.

References

[1] Opt out of being tracked. http://www.bluecava.com/preferences/.

[2] Adblock plus - for annoyance-free web surfing. http://adblockplus.org.

[3] Aaron Andersen. History of the browser user-agent string. http://webaim.org/blog/user-agent-string-history.

[4] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[5] Graham Cluley. How to turn off Java on your browser - and why you should do it now. http://nakedsecurity.sophos.com/2012/08/30/how-turn-off-java-browser/.

[6] Collusion: Discover who's tracking you online. http://www.mozilla.org/en-US/collusion/.

[7] comScore. The Impact of Cookie Deletion on Site-Server and Ad-Server Metrics in Australia, January 2011.

[8] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 281–290, 2010.

[9] Peter Eckersley. Panopticlick | Self-Defense. https://panopticlick.eff.org/self-defense.php.

[10] Peter Eckersley. How Unique Is Your Browser? In *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS)*, 2010.

[11] ECMAScript Language Specification, Standard ECMA-262, Third edition.

[12] Jean-Louis Gassée and Frederic Filloux. Measuring Time Spent On A Web Page. http://www.cbsnews.com/2100-215_162-5037448.html.

[13] Ghostery. http:wwww.ghostery.com.

[14] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript Web applications. In *Proceedings of CCS 2010*, October 2010.

[15] Amit Klein. How Fraudsters are Disguising PCs to Fool Device Fingerprinting. http://www.trusteer.com/blog/how-fraudsters-are-disguising-pcs-fool-device-fingerprinting.

[16] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *IEEE Symposium on Security and Privacy*, May 2012.

[17] Brian Krebs. How to Unplug Java from the Browser. http://krebsonsecurity.com/how-to-unplug-java-from-the-browser.

[18] Balachander Krishnamurthy. Privacy leakage on the Internet. presented at IETF 77, March 2010.

[19] Balachander Krishnamurthy and Craig E. Wills. Generating a privacy footprint on the Internet. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, IMC '06, pages 65–70, New York, NY, USA, 2006.

[20] Jonathan R. Mayer. Tracking the Trackers: Early Results | Center for Internet and Society. http://cyberlaw.stanford.edu/node/6694.

[21] Jonathan R. Mayer. Any person... a pamphleteer. Senior Thesis, Stanford University, 2009.

[22] Jonathan R. Mayer and John C. Mitchell. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy*, pages 413–427, 2012.

[23] Elinor Mills. Device identification in online banking is privacy threat, expert says. CNET News (April 2009).

[24] Keaton Mowery, Dillon Bogenreif, Scott Yilek, and Hovav Shacham. Fingerprinting information in JavaScript implementations. In Helen Wang, editor, *Proceedings of W2SP 2011*. IEEE Computer Society, May 2011.

[25] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In Matt Fredrikson, editor, *Proceedings of W2SP 2012*. IEEE Computer Society, May 2012.

[26] 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 *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[27] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Security and Privacy*, 2013.

[28] Łukasz Olejnik, Claude Castelluccia, and Artur Janc. Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns. In *the 5th workshop on Hot Topics in Privacy Enhancing Technologies (HOTPETS 2012)*.

[29] Greg Pierson and Jason DeHaan. Patent US20080040802 - NETWORK SECURITY AND FRAUD DETECTION SYSTEM AND METHOD.

[30] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *NSDI'12: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[31] Justin Scott. How many Firefox users have add-ons installed? 85%! https://blog.mozilla.org/addons/2011/06/21/firefox-4-add-on-users/.

[32] Ashkan Soltani, Shannon Canty, Quentin Mayo, Lauren Thomas, and Chris Jay Hoofnagle. Flash Cookies and Privacy. In *SSRN preprint (August 2009)*.

[33] The New York Times - John Schwartz. Giving the Web a Memory Cost Its Users Privacy. http://www.nytimes.com/2001/09/04/technology/04COOK.html.

[34] The Wall Street Journal. What They Know. http://blogs.wsj.com/wtk/.

[35] Torbutton: I can't view videos on YouTube and other flash-based sites. Why? https://www.torproject.org/torbutton/torbutton-faq.html.en#noflash.

[36] Joseph Turow, Jennifer King, Chris Jay Hoofnagle, Amy Bleakley, and Michael Hennessy. Americans Reject Tailored Advertising and Three Activities that Enable It, 2009.

[37] Blase Ur, Pedro Giovanni Leon, Lorrie Faith Cranor, Richard Shay, and Yang Wang. Smart, useful, scary, creepy: perceptions of online behavioral advertising. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 4:1–4:15, New York, NY, USA, 2012. ACM.

[38] VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/.

[39] Web Tracking Protection. http://www.w3.org/Submission/2011/SUBM-web-tracking-protection-20110224/.

[40] Zachary Weinberg, Eric Y. Chen, Pavithra Ramesh Jayaraman, and Collin Jackson. I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 147–161, 2011.

[41] Jimson Xu and Tom Nguyen. Private browsing and Flash Player 10.1. http://www.adobe.com/devnet/flashplayer/articles/privacy_mode_fp10_1.html.

[42] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martin Abadi. Host Fingerprinting and Tracking on the Web: Privacy and Security Implications. In *Proceddings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[43] Michael Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.

7 Conclusion

This deliverable has reported on how to express secure composition policies, how to securely integrated third-party JavaScript in web applications, and how to achieve this in a server-driven manner.

In particular, we explored the risks involved in integrating third-party JavaScript, and discussed the server-driven enforcement mechanisms developed in work package 4 of the WebSand project.

First, the web security model was briefly discussed, as well as the impact of this security model on integrating third-party JavaScript in a website. The assessment of script inclusion on the top 10,000 most popular websites illustrated the widespreadness of this type of code assembly on the web, and the urge to securely compose JavaScript (Section 2).

We identified that 88.45% of the 10,000 web sites included at least one remote JavaScript library, and there are even sites in the top Alexa list that trust up to 295 remote hosts. Moreover, we discovered that one out of four sites with high-maintenance scores include scripts from providers with a low maintenance score, which are potential “weak spots” in their security perimeter.

Next, we briefly summarized the set of security-sensitive operations, and bundled them in nine logical categories as part of the least-privilege secure composition policies. Finally, this deliverable also investigated how the richness of JavaScript APIs not only affects the security but also the privacy of end-users in web fingerprinting frameworks (Section 6).

A variety of enforcement techniques have been developed within WebSand to securely compose JavaScript. These enforcement techniques range from a security-enhanced browser (WebJail) to a JavaScript security architectures that runs on top of mainstream browsers (Two-tier sandbox , JSand and PreparedJS). All implementation strategies have reported successful isolation of third-party JavaScript. They mainly vary in their mode of deployment, but also make different trade-offs in terms of legacy support, precision, efficiency and maintainability.

WebJail (reported in Deliverable D4.1 and D4.2) focuses on the secure integration of third-party frames and realizes this enforcement by mediating access to security-sensitive operations in the browser core. The other three prototype focuses on the secure integration of scripts, and test the feasibility to realize this enforcement without (major) client-side modifications.

The two-tier sandbox architecture (Section 3) allows to apply application-specific, stateful fine-grained policies. By combining the fine-grained enforcement mechanism with a more coarse-grained outer sandbox, the technique ensure a baseline protection in case the policy writer mistakenly introduces vulnerabilities while expressing the fine-grained policies.

The JSand approach (Section 4) is more focused towards the enforcement of the least-privilege composition policy, which tends to be a good balance between the fine-grained policies of the two-tier sandbox approach and the very coarse-grained policies of the Same Origin Policy and the Content Security Policy.

PreparedJS (Section 5) enriches the Content Security Policy model, both in terms of usability for the web developer, and security towards white-listing the intended third-party scripts. This latter technique provides an additional layer of protection on top of the mechanisms specified above, to protect websites against trusted scripts and script providers that get compromised over time.

We selected JSand as most promising technique for the server-driven enforcement. To this extent, we have further matured the JSand prototype implementation to support a representative selection of DOM operations, and applied it to the most frequently used third-party scripts.

The server-driven enforcement is successfully realized and works as follows. First a set of secure composition policies is expressed by the website owner or the security officer in charge. The least-privilege composition policy expresses for each of the nine categories whether or not scripts should have access to these security-relevant operations. Next, a sandbox environment is configured as part of the web application, by selecting the appropriate secure composition policy and the code that needs to be executed as part of the sandbox. Both of these steps are part of the development or deployment of the web application.

During execution, the client-side code to set up the security architecture and the secure composition policy are pushed towards the browser as part of the web page. Before loading the third-party JavaScript code, the JavaScript security architecture is set up, and a script-specific sandbox environment is created based on the provided secure composition policy. Once fully configured, the third-party JavaScript code is loaded in the sandbox environment and executed.