
# The Unexpected Dangers of Dynamic JavaScript

**Sebastian Lekies,** ***Ruhr-University Bochum;*** **Ben Stock,** ***Friedrich-Alexander-Universität Erlangen-Nürnberg;*** **Martin Wentzel and Martin Johns,** ***SAP SE***

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies

**This paper is included in the Proceedings of the 24th USENIX Security Symposium**

**August 12–14, 2015 • Washington, D.C.**

ISBN 978-1-931971-232

**Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX**

# The Unexpected Dangers of Dynamic JavaScript

Sebastian Lekies
*Ruhr-University Bochum*
*paper@sebastian-lekies.de*

Ben Stock
*FAU Erlangen-Nuremberg*
*ben.stock@fau.de*

Martin Wentzel
*SAP SE*
*martin.wentzel@sap.com*

Martin Johns
*SAP SE*
*martin.johns@sap.com*

## Abstract

Modern Web sites frequently generate JavaScript on-the-fly via server-side scripting, incorporating personalized user data in the process. In general, cross-domain access to such sensitive resources is prevented by the Same-Origin Policy. The inclusion of remote scripts via the HTML script tag, however, is exempt from this policy. This exemption allows an adversary to import and execute dynamically generated scripts while a user visits an attacker-controlled Web site. By observing the execution behavior and the side effects the inclusion of the dynamic script causes, the attacker is able to leak private user data leading to severe consequences ranging from privacy violations up to full compromise of user accounts.

Although this issues has been known for several years under the term Cross-Site Script Inclusion, it has not been analyzed in-depth on the Web. Therefore, to systematically investigate the issue, we conduct a study on its prevalence in a set of 150 top-ranked domains. We observe that a third of the surveyed sites utilize dynamic JavaScript. After evaluating the effectiveness of the deployed countermeasures, we show that more than 80% of the sites are susceptible to attacks via remote script inclusion. Given the results of our study, we provide a secure and functionally equivalent alternative to the use of dynamic scripts.

## 1 Introduction

Since its beginning in the early nineties, the Web evolved from a mechanism to publish and link static documents into a sophisticated platform for distributed Web applications. This rapid transformation was driven by two technical cornerstones:

*1. Server-side generation of code:* For one, on the server side, static HTML content was quickly replaced by scripting to dynamically compose the Web server's HTTP responses and the contained HTML/JavaScript on the fly. In turn, this enabled the transformation of the Web's initial document-centric nature into the versatile platform that we know today.

*2. Browser-driven Web front-ends:* Furthermore, the Web browser has proven to be a highly capable container for server-provided user interfaces and application logic. Thanks to the flexible nature of the underlying HTML model and the power of client-side scripting via JavaScript, the server can push arbitrary user interfaces to the browser that rival their counterparts of desktop application. In addition, and unlike monolithic desktop applications, however, browser-based UIs enable easy incorporation of content from multiple parties using HTML's inherent hypertext capabilities.

Based on this foundation, the recent years have shown an ongoing shift from Web applications that host the majority of their application logic on the server side towards rich client-side applications, which use JavaScript to realize a significant portion of their functionality within the user's browser.

With the increase of the functionality implemented on the client side, the necessity for the JavaScript code to gain access to additional user data rises natrually. In this paper, we explore a specific technique that is frequently used to pull such data from the server to the client-side: Dynamic JavaScript generation.

Similar to HTML, which is often generated dynamically, JavaScript may also be composed on the fly through server-side code. In this composition process, user-specific data is often included in the resulting script code, e.g., within the value of a variable. After delivering the script to the browser, this data is immediately available to the client-side logic for further processing and presentation. This practice is potentially dangerous as the inclusion of script files is exempt from the Same-Origin Policy [23]. Therefore, an attacker-controlled Web page is able to import such a dynamically generated script and observe the side effects of the execution, since all included scripts share the global object

of the embedding web document. Thus, if the script contains user-specific data, this data might be accessible to other attacker-controlled JavaScript. Although this attack, dubbed *Cross-Site Script Inclusion* (XSSI), has been mentioned within the literature [31], the prevalence of flaws which allow for this attack vector has not been studied on real-world Web sites.

In this paper we therefore present the first, systematic analysis of this vulnerability class and provide empirical evidence on its severeness. First, we outline the general attack patterns and vectors that can be used to conduct such an attack. Furthermore, we present the results of an empirical study on several high-profile domains, showing how these domains incorporate dynamic scripts into their applications. Thereby, we find evidence that many of these scripts are not or only inadequately protected against XSSI attacks. We demonstrate the sever consequences of these data leaks by reporting on real-world exploitation scenarios ranging from de-anonymization, to targeted phishing attacks up to complete compromise of a victim's account.

To summarize, we make the following contributions:

- We elaborate on different ways an attacker is capable of leaking sensitive data via dynamically generated scripts, enabled by the object scoping and dynamic nature of JavaScript.
- We report on the results of an empirical study on several high-ranked domains to investigate the prevalence of dynamic scripts.
- Using the data collected during our empirical study, we show that many dynamic scripts are not properly protected against XSSI attacks. To demonstrate the severity of the outlined vulnerabilities, we present different exploitation scenarios ranging from de-anonymization to complete hijacking of a victim's account.
- Based on the observed purposes of the dynamic scripts encountered in our study, we discuss secure ways of utilizing such data without the use of dynamically generated scripts.

The remainder of the paper is structured as follows: In Section 2, we explain the technical foundations needed for rest of the paper. Section 3 then covers the general attack patterns and techniques to exploit cross-domain data leakage vulnerabilities. In Section 4, we report on the results of our empirical study and analyze the underlying purposes of dynamic scripts. Furthermore, in Section 5, we provide a scheme that is functionally equivalent, but is not prone to the attacks described in this paper. Section 6 covers related work, Section 7 gives an outlook and Section 8 concludes the paper.

# 2 Technical Background

In this section, we cover the technical background relevant for this work.

## 2.1 The Same-Origin Policy

The *Same-Origin Policy* (SOP) is the principal security policy in Web browsers. The SOP strongly separates mutually distrusting Web content within the Web browser through origin-based compartmentalization [23]. More precisely, the SOP allows a given JavaScript access only to resources that have the same *origin*. The origin is defined as the triple consisting of scheme, host, and port of the involved resources. Thus, for instance, a script executed under the origin of `attacker.org` is not able to access a user's personal information rendered under `webmail.com`.

While JavaScript execution is subject to the SOP, the same does not hold true for cross-domain inclusion of Web content using HTML tags. Following the initial hypertext vision of the WWW HTML-tags, such as `image`, may reference resources from foreign origins and include them into the current Web document.

Using this mechanism, the HTML `script` tag can point to external script resources, using the tag's `src` attribute. When the browser encounters such a tag, it issues a request to the foreign domain to retrieve the referenced script. Important to note in this instance is the fact that the request also carries authentication credentials in the form of cookies which the browser might have stored for the remote host. When the response arrives, the script code inherits the origin of the *including* document and is executed in the context of the hosting page. This mechanism is used widely in the Web, for instance to consume third party JavaScript services, such as traffic analysis or advertisement reselling [24].

## 2.2 JavaScript Language Features

In the following, we cover the most important JavaScript concepts necessary for the rest of the paper.

**Scoping** In JavaScript, a scope is "a lexical environment in which a function object is executed" [6]. From a developer's point of view, a scope is the region in which an identifier is defined. While C++ or Java make use of block scoping, JavaScript utilizes so-called function scoping. This means that the JavaScript engine creates a new scope for each new function it encounters. As a consequence, an identifier that is locally defined in such a function is associated with the corresponding scope. Only code that is defined within the same function is thus able to access such a variable residing in the *local scope*,

whereas global variables are associated with the *global scope*.

Listing 1 shows an example for local and global variables. A local variable in JavaScript can be created by utilizing the `var` keyword. All variables defined outside of a function are associated with the global scope, whereas code within a function can define variables in the global scope by either omitting the `var` keyword or explicitly assigning to `window.varname`.

**Listing 1** Example for global and local variables

```
// A global variable
var globalVariable1 = 5;

function globalFunction(){
  // A local variable
  var localVariable = 2;

  // Another global variable
  globalVariable2 = 3;

  // Yet another global variable
  window.globalVariable3 = 4;
}
```

**The Prototype Chain** As opposed to classical programming languages such as C++ or Java, JavaScript is a prototype-based language. This means that JavaScript's inheritance is not based on classes but directly on other objects, whereas "each object has a link to another object called its prototype" [21]. On creation of an object, it either automatically inherits from `Object.prototype` or if a prototype object is explicitly provided, the prototype property will point to this object. On access to an object's property, the JavaScript runtime checks whether the current object contains a so-called *own property* with the corresponding name.

If no such property exists, the object's prototype is queried for the same property and if lookup fails again, the process is recursively repeated for the object's prototypes. Hence, objects in JavaScript form a so-called *prototype chain*. Listing 2 gives a commented example for this behavior.

**Listing 2** The prototype chain

```
var object1 = {a: 1};
// object1 ---> Object.prototype ---> null

var object2 = Object.create(object1);
// object2 ---> object1
//         ---> Object.prototype ---> null
console.log(object2.a); // 1 (inherited)
```

# 3 Cross-Domain Data Leakages

In this section, we show how an adversary can utilize an external JavaScript file, which is dynamically generated at runtime, to leak security sensitive data. After first covering the different types of these dynamic scripts, we elaborate on the attacker model and then demonstrate different attack vectors that can be leveraged to leak sensitive data from such a script.

## 3.1 Dynamic Scripts

As discussed in Section 2.1, Web pages can utilize `script`-tags to import further JavaScript resources. For the remainder of this paper, we define the term *dynamic script* to describe such a JavaScript resource in case it is generated by the Web server on the fly via server-side code.

As opposed to static scripts, the contents of dynamic scripts may vary depending on factors such as input parameters or session state. In the context of this paper, the latter type is of special interest: If a dynamic JavaScript is generated within a user's authenticated Web session, the contents of this script may contain privacy or security sensitive data that is bound to the user's session data. Thus, an execution of the script can potentially lead to side effects which leak information about this data.

## 3.2 Attack Method

HTML `script` tags are not subject to the Same-Origin Policy (see Section 2.1). Hence, script resources can be embedded into cross-domain Web pages. Although such cross-domain Web pages cannot access the source code of the script directly, this inclusion process causes the browser to load and execute the script code in the context of the cross-domain Web page, allowing the importing page to observe the script's behavior. If a dynamic script exposes side effects dependent on sensitive data in the script code, the execution of such a script may leak the secret data.

Figure 1 depicts an example attack. A user is authenticated to his mail provider at `webmail.com`, thus his browser automatically attaches the corresponding session cookies to all requests targeting `webmail.com`, which utilizes session-state dependent dynamic scripts. Thus, whenever a user is logged in, the script at `webmail.com/script.js` creates a global variable containing the current user's email address. In the same browser, the user now navigates to an attacker-controlled Web site at `attacker.org`. The attacker includes the dynamic script in his own Web page and subsequently, the browser requests the script with attached authentication cookies. Although the script originates

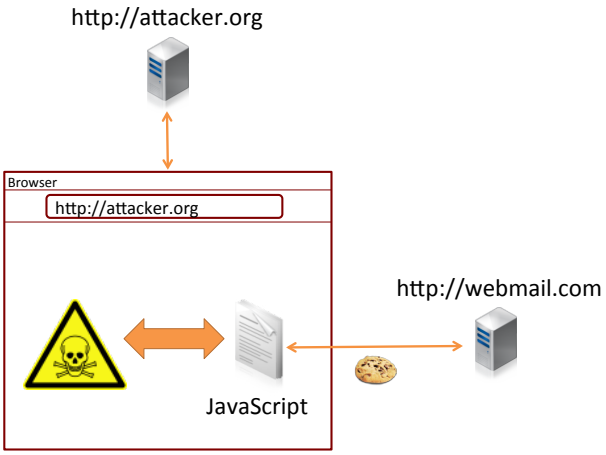


Figure 1: Attacker Model

from `webmail.com`, it is now executed in the context of `attacker.org`, creating the global variable with the user's email in the corresponding context. The global variable is now accessible to any other script executed by `attacker.org`. Hence, the attacker can simply access the value of this global variable, effectively leaking the user's email address.

## 3.3 Attack Vectors

As previously explained, an attacker is able to leak sensitive user data by including a script from a different domain and observing the results of the execution. In this section we outline different situations in which sensitive data is accessible to an attacker after the included script has been executed.

### 3.3.1 Global Variables

As noted in the previous section, global variables created by a dynamic script can be accessed by any other script executed on the same Web document. Hence, whenever sensitive user data is assigned to such a global variable inside a script, an attacker can gain access to the corresponding data. In order to do so, he simply includes the script and waits for the global variable to be created. As soon as the value assignment has occurred, the attacker's code can read the sensitive data and leak it back to his backend.

### 3.3.2 Redefinition of Global APIs

Due to JavaScript's dynamic nature, (almost) any function can be overwritten by an attacker, including a number of globally available APIs. If a dynamic script passes a security-sensitive value to such a function, the attacker may overwrite it beforehand and hence retrieve the secret value. Listing 3 demonstrates how an attacker can, for example, change the behavior of the global function `JSON.stringify`. In order to conduct an attack, the attacker overrides the function first and then includes a dynamic script which passes a sensitive data value to the function. When the user visits the attacker's Web site, his browser retrieves and executes the dynamic script. Rather than invoking the native `JSON.stringify` function, the contained code invokes the attacker-controlled function. In this case, instead of serializing the object, the function sends the user's data back to the attacker's server.

**Listing 3** Passing a variable to a global function

```
// Attacker's script overwriting a global function
JSON.stringify = function(data){
  sendToAttackerBackend(data);
}

------------------------------------------------
//Within the dynamic script
function myFunction() {
  var myVar = { secret: "secret value"};

  // Calling a predefined global function
  return JSON.stringify(myVar);
}
```

### 3.3.3 Prototype Tampering

As outlined in the previous section, variables are available in the scope in which they were defined unless the `var` keyword is omitted. Listing 4 shows an example of code making use of this paradigm. The function allocates an array with three secret values using the `var` keyword and therefore, as it seems, protects the array from access by outside code. As discussed in Section 2.2, JavaScript is a prototype-based language. Hence, when requesting a property of an object, the JavaScript interpreter walks up the prototype chain until it finds a matching property. In our example shown in Listing 4, the function `slice` is called on the array named `arr`. By default, an array object does not provide the slice function itself. Therefore, the call is made to the function in the array's prototype, which points to the object `Array.prototype`. In a scenario where the script is included without any malicious intent, the programmer may assume that the call will eventually trigger invocation of the `slice` method for arrays.

This behavior may, however, be changed by an attacker. Listing 5 depicts a small snippet of code that is provided by the attacker. Similar to what we discussed earlier with respect to overwriting global functions, the snippet overwrites the `slice` method in the array's pro-

**Listing 4** Variable protected in a closure

```
(function(){
  var arr = ["secret1","secret2","secret3"];
  // intents to slice out first entry
  var x = arr.slice(1);
  ...
})();
```

totype. Since by default all arrays in JavaScript share the same prototype, the call to `slice` in Listing 4 is passed to the attacker-provided function. Since the function is called on the `arr` object, the attacker can use the `this` keyword to gain a reference to the object. Therefore, rather than exhibiting the intended behavior of slicing out a part of the array, the attacker's code now sends the otherwise properly protected information back to the attacker. This attack works for any object that has a globally accessible prototype, i.e., it is feasible on any built-in objects such as Strings or Functions.

**Listing 5** Leaking data via the this keyword

```
Array.prototype.slice = function(){
  //leaks ["secret1","secret2","secret3"]
  sendToAttackerBackend(this);
};
```

## 3.4 Distinction towards CSRF

On first view, the described attack method is related to Cross-site Request Forgery (CSRF) [1], as it follows a similar attack pattern.

In fact, leaking sensitive information via cross-domain script includes belongs to a larger class of Web attacks which function via creating authenticated requests in the context of an authenticated Web user (including CSRF, ClickJacking [12] and reflected Cross-site Scripting [2]).

However, the goal and consequences of the attack differ significantly from other attack variants: CSRF is an attack in which an attacker generates requests to cause state-changing actions in the name of the user. Thereby the attacker is by no means able to read content from a response to a CSRF request. To prevent CSRF developers are advised to conduct state-changing actions only via HTTP POST requests and to protect all these post requests with CSRF tokens.

As opposed to this, dynamic scripts are neither designed to conduct state-changing actions on the server-side nor are these scripts ever fetched via POST requests. Those scripts are stateless and are fetched via GET requests through script tags and, hence, are not classified as a critical endpoint in the context of CSRF, i.e., not contained in the application's CSRF protection surface.

# 4 Empirical Study

In this section we report on the results of an empirical study designed to gain insights into the prevalence and exploitability of data leakages due to the use of dynamic script generation on the server side. We first discuss the methodology of our study and report on the general prevalence of dynamically generated JavaScript files in the wild. Based on the gathered data, we analyze the underlying purposes of these scripts, discuss the types of security-sensitive data contained in the scripts and highlight who these can be leaked, allowing us specific exploits against a number of sites. We end the section with a discussion of situations in which we could not exploit a dynamic script due to the use of adequate protection measures.

## 4.1 Methodology

In the following, we cover our research questions, explain our detection methodology and describe our data set.

### 4.1.1 Research Questions

This study provides an in-depth analysis of dynamic script includes. Before diving into the security aspects of these scripts, we aim at collecting data on this technique in general. Hence, we are first interested in the general prevalence of dynamically generate scripts. More specifically, the goal is to find out how common dynamic script generation is in today's Web and how often these dynamic scripts are dependent on a user's session state. The study sheds light on the purpose of these scripts and the contained data. Finally, we investigate the security aspects by investigating the exploitability and discussing potential countermeasures.

### 4.1.2 Detecting State-dependent Scripts

As a basis for our empirical study, we needed a means to easily detect state-dependent dynamic scripts. Therefore, we implemented a Chrome browser extension that fulfills two separate tasks:

1. **Collecting scripts:** The first step towards analyzing Web pages for dynamic scripts is the collection of all external script resources included by the Web page. For this purpose, we created a browser extension that collects all included scripts of a page by using a so-called *Mutation Observer* [22]. As soon as a new script node is found, it is immediately passed on to our analysis module.

2. **Detecting dynamic code generation based on authentication credentials:** Whenever the analysis is

invoked, our extension requests the script file twice: once with authentication credentials attached to the request, and once without authentication credentials. After the responses have been received, the extension compares both and if they differ, stores them in a central database for later analysis.

In order to allow for valid credentials to be sent along with the request, a necessary prerequisite are valid session cookies. To obtain these, the user needs to manually log in to the application under investigation beforehand.

The final step in this phase is the manual analysis of the gathered data to precisely determine which scripts have a dynamic nature depending on the user's session state rather than randomness (such as banner rotation scripts).

#### 4.1.3 Data Set

Unlike general vulnerabilities, the detection of potential data leakages through dynamic JavaScript generation requires an active user account (or a similar stateful relationship) at the tested site, so that the scripts are generated in the context of an authenticated Web session.

Since this requires initial manual registration and account set up on sites we want to test, the size and the nature of our data set is limited. We therefore chose the 150 highest ranking (according to Alexa) Web pages matching the following criteria:

1. Account registration and login is freely available for anyone. This excludes, services that have only paid subscription models or require country-dependent prerequisites (such as a mobile phone number).

2. Available in either German, English or a Web site which can be translated using Google Translate. If this is not given, the set up of meaningful user accounts was not feasible.

3. Not a duplicate or localized variant of an already investigated site (e.g. google.com vs. google.co.in)

After manually registering accounts on these sites, we investigated the site employing the methodology and techniques previously explained, thoroughly interacting with the complete functionality of the sites by adding, processing and viewing plausible data within the different Web applications.

### 4.2 Prevalence of Dynamic Scripts

The first goal of our study was to count the number of Web sites that make use of dynamic script generation. In the course of this study, using our aforementioned methodology, we gathered a total of 9,059 script files spread across 334 domains and their subdomains. Although our data set only consists of 150 different domains, we gathered scripts from such a large number of domains due to the fact that the investigated Web sites include third-party frames pointing to, e.g., advertisement providers. In a first step, we therefore filtered out scripts from all sites not directly related to the domains under investigation.

Out of these, we found that over half of the sites—81 out of the 150 analyzed domains—utilized some form of dynamic script generation. In a subsequent manual examination step we removed dynamic scripts which only exposed changes in apparently random token values (see below for details), resulting in 209 unique scripts on 49 domains, that were dependent on a user's session state. In relation to our initial data set of 150 domains, this shows that the usage of state-dependent dynamic scripts is widespread, namely one third of the investigated domains.

### 4.3 Purposes of Dynamic Scripts

We analyzed the applications to ascertain the underlying purpose motivating the utilization of the dynamic scripts. In doing so, we found three categories of use cases as well as a few purposes which could not be categorized. Since these were only single use cases specific to one application, we do not outline these any further but instead put them in the *Others* category. The results of our categorization are depicted in Table 1, showing the total amount of domains per category as well as the highest Alexa rank.

The most commonly applied use case was **retrieval of user-specific data**, such as the name, email address or preferences for the logged-in user. This information was used both to greet users on the start page as well as to retrieve user-provided settings and profile data on the corresponding edit pages. We observed that a number of Web applications utilized modal dialogs to present the profile data forms to the user, whereas the HTML code of said form was embedded into the document already and all currently stored values were retrieved by including a dynamic script.

The second category of scripts we found was **service bootstrapping**, i.e., setting up variables necessary for a rich client-side application to work. One example of such a bootstrapping process was observed in a popular free-mail service's file storage system in which the UI was implemented completely in JavaScript. When initially loading the page, the dynamic script we found provided a secret token which was later used by the application to interact with the server using XMLHttpRequests.

| Category | # domains | Highest rank |
|---|---|---|
| Retrieval of user-specific data | 16 | 7 |
| Service bootstrapping | 15 | 5 |
| Cross-service data sharing | 5 | 8 |
| Others | 13 | 1 |

Table 1: Amounts and highest Alexa rank of domains with respect to their use case

| Data | domains | exploitable | highest rank |
|---|---|---|---|
| Login state | 49 | 40 | 1 |
| Unique Identifiers | 34 | 28 | 5 |
| Personal data | 15 | 11 | 11 |
| Tokens & Session IDs | 7 | 4 | 107 |

Table 2: Sensitive data contained in dynamic scripts

The third widely witnessed use case was **cross-service data sharing**, which was often applied to allow for single sign-on solutions across multiple services of the same provider or for tracking of users on different domains through a single tracking service. The latter was evidenced by the same script being included across a multitude of domains from different service providers.

## 4.4 Types of Security Sensitive Data

In a next step, we conducted a manual analysis of the scripts' data that changed its value, depending on the authentication state of the script request. Within our data, we identified four categories of potentially security-critical data:

- **Login state**: The first type of data that could be extracted from dynamic scripts was a user's login state to a certain application. We found that this happened either explicitly, i.e., assign a variable differently if a user is logged in – or implicitly, e.g. in cases where a script did not contain any code if a user was not logged in.

- **Unique identifiers**: The second category we discovered was the leakage of data that uniquely identified the user. Among these values are customer or user IDs as well as email addresses with which a user was registered to a specific application.

- **Personal data**: In this category we classified all those pieces of data which do not necessarily uniquely identify a user, but provide additional information on him, such as his real name, his location or his date of birth.

- **Tokens & Session IDs**: The last category we encountered were tokens and session identifiers for an authenticated user. These tokens potentially provide an attacker with the necessary information to interact with the application in the name of the user.

Table 2 depicts our study's results with respect to the occurrences of each category. Please note, that a given domain may carry more than one script containing security sensitive information and that a given script may fit into more than one of the four categories.

The following sections give a more detailed insight into these numbers. The final column shows the highest rank of any domain on which we could successfully extract the corresponding data, i.e., on which we could bypass encountered protection mechanisms.

## 4.5 Exploitation

In the following, we discuss several attacks which leverage the leakage of sensitive user information. After outlining potential attack scenarios, we discuss several concrete examples of attacks we successfully conducted against our own test accounts.

### 4.5.1 Utilizing Login Oracles

In the previous section, we discussed that 49 domains had scripts which returned somewhat different content if the cookies for the logged in user were removed. In our notion, we call these scripts *login oracles* since they provide an attacker with either explicit or implicit information on whether a user is currently logged into an account on a given website or not. However, out of these domains, nine domains had scripts with unguessable tokens in the URL, therefore these cannot be utilized as login oracles unless the tokens are known, leaving 40 domains with login oracles.

The most prominent script we found to show such a behavior is hosted by Google and is part of the API for Google Plus. This script, which has a seemingly static address, shows differences in three different variables, namely `isLoggedIn`, `isPlusUser` and `useFirstPartyAuthV2` and hence enables an attacker to ascertain a user's login status with Google.

The information obtained from the oracles can be utilized to provide additional bits to fingerprinting approaches [7]. It may however also be used by an attacker to perform a service-specific phishing attack against his victim. Oftentimes, spam emails try to phish user credentials from banks or services the receiving user does not even have an account on. If, however, the attacker knows with certainty that the user currently visiting his

website is logged in to, e.g., google.com, he can display a phishing form specifically aimed at users of Google. This attack can also be improved if additional information about the user is known – we will discuss this attack later in this section.

#### 4.5.2 Tracking Users

Out of the 40 domains which provided a login oracle, 28 also provided some pieces of data which uniquely identify a user. Among these features, the most common identifier was the email address used to register for the corresponding service, followed by some form of user ID (such as login name or customer ID). These features can be used to track users even across device platforms, given that they log in to a service leaking this information. The highest-rated service leaking this kind of unique identifier was a top-ranked Chinese search engine. Following that, we found that a highly-frequented page which features a calendar function also contained a script leaking the email address of the currently logged in user. Since the owning company also owns other domains which all use a single sign-on, logging in to any of these sites also enabled the attack.

#### 4.5.3 Personalized Social Engineering

In many applications, we found that email addresses were being leaked to an attacker. This information can be leveraged to construct highly-personalized phishing attacks against users. As Downs et al. [5] discovered, users tend to react on phishing emails in more of the cases if they have a standing business relationship with the sending entity, i.e. have an account on a given site, or the email appears to be for them personally.

Hence, gathering information on sites a user has an account on as well as retrieving additional information such as his name can aid an attacker in a personalized attack. An attacker may choose to abuse this in two ways – first and foremost, trying to send phishing mails to users based on the services they have accounts. However, by learning the email address and hence email provider of the user, an attacker may also try to phish the user's mail account. In our study, we found that 14 different domains leak email addresses and out of these, ten domains also revealed (at least) the first name of the logged in user.

In addition, two domains leaked the date of birth and one script, hosted on a Chinese Web site, even contained the (verified) mobile phone number of the victim. We believe that, especially considering the discoveries by Downs et al., all this information can be leveraged towards creating highly-personalized phishing attacks.

Another form of personalized social engineering attacks enabled by our findings is targeted advertisement. We found that two online shopping platforms utilize a dynamic script which provides the application with the user's wish list. This information can be leveraged by an attacker to either provide targeted advertisements aimed at profiting (e.g. linking to the products on Amazon, using the attacker's affiliate ID) or to sell fake products matching the user's wishes.

**Application-Specific Attacks** Alongside the theoretical attack scenarios we discussed so far, we found multiple applications with issues related to the analyzed leaking scripts as well as several domains with CSRF flaws. In the following, we discuss these attacks briefly.

**Extracting Calendar Entries:** One of the most prominent Web sites we could successfully exploit was a mail service which offers a multitude of additional functionality such as management of contacts and a calendar. The latter is implemented mostly in JavaScript and retrieves the necessary bootstrap information when the calendar is loaded. This script, in the form a function call to a custom JavaScript API, provides the application with all of the user's calendars as well as the corresponding entries. This script was not protected against inclusion by third-party hosts and hence, leaks this sensitive information to an attacker. Alongside the calendar's and entries, the script also leaks the e-mail address of the victim, therefore allowing the attacker to associate the appointments to their owner.

**Reading Email Senders and Subjects:** When logging in to the portal for a big Chinese Web service provider, we found that the main page shows the last five emails for the currently logged in user. Our browser extension determined that this information was provided by an external script, solely using cookies to authenticate the user. The script contained the username, amount of unread emails and additionally the senders and subjects as well as the received dates for the last five emails of the victim. An abbreviated excerpt is shown in Listing 6. Although this attack does not allow for an actual extraction of the content of an email, at the very least contacts and topics of current discussions of the victim are leaked which we believe to be a major privacy issue.

**Listing 6** Excerpt of the script leaking mail information

```
var mailinfo = {
  "email": "user@domain.com",
  ...,
  "maillist": [{
    "mid": "0253FE71.....001",
     "mailfrom":"First Last <firstlast@gmail.com>",
    "subject":"Top secret insider information",
    "ctime": "2014-05-02 21:11:46"}]
  ..}
```

**Session Hijacking Vulnerabilities:** During the course of our study, we found that two German file storage services contained session hijacking vulnerabilitie. Both these services are implemented as a JavaScript application, which utilizes XMLHttpRequest to retrieve directory listings and manage files in the storage. To avoid unauthorized access to the system, both applications require a session key to be present within a cookie as well as in an additional HTTP header. When first visiting the file storage service, the application loads an external script called `userdata.js` which contains the two necessary secrets to access the service: the username and the aforementioned session key. We found that this script is not properly protected against cross-domain data leakage, allowing an attacker to leak the secret information. With this information at hand, we were able to list and access any file in the victim's file storage. Furthermore, it enabled us to invoke arbitrary actions in the name of the user such as creating new files or deleting existing ones.

One minor drawback in this attack is the need for the attacker to know the victim's username in advance, since the dynamic script requires a GET parameter with the username. Regardless, we believe that by either targeted phishing emails or retrieving the email address through another service (as discussed earlier) this attack is still quite feasible.

**Circumventing CSRF Protection:** One way of preventing cross-domain attacks is the use of CSRF tokens, namely secrets that are either part of the URL (as a GET parameter) or need to be posted in a form and can then be verified by the server. Although CSRF tokens are a well-understood means of preventing these attacks and provide adequate security, the proper implementation is a key factor. In our analysis, we found that two domains contained scripts which leaked just these critical tokens.

The first one was present on a new domain, which required the knowledge of two secrets in order to change profile data of the user – a 25 byte long token as well as the numerical user ID. While browsing the Web site, our extension detected a state-dependent dynamic script that exactly contained these two values. As a consequence, we were able to leak this data and use it to send a properly authenticated profile change request to the corresponding API. As a consequence, we were able to arbitrarily change a user's profile data. Interestingly, one field that was only visible to the user himself contained a stored XSS vulnerability. Hence, we were able to send a Cross-Site Scripting payload within this field to exploit the, otherwise unexploitable, XSS flaw.

Apart from the obvious issues an XSS attack could cause, for a user logged in via the Facebook Social Login, we could retrieve the Facebook API access token and hence interact with the Facebook API in the name of the user, accessing profile information and even make posts in the name of the user.

Similar to the first finding, we found an issue on the highly-ranked domain of a weather service. The application provides an API for changing a user's profile as well as the password, whereas the old password does not need to be entered to set a new one. Nevertheless, the API requires knowledge of the email address of the currently logged in user, thereby employing at least a variant of a CSRF token. Similar to the previously outlined flaw, we found a script that provides information on the user – among which also the email address is contained. Hence, we could successfully automate the attack by first retrieving the necessary token (email) from the leaking script and subsequently sending a password change request to the API. Afterwards, we sent both the email address (which is also used as the login name) and the new password back to our servers, essentially taking over the user's account in a fully automated manner.

#### 4.5.4 Notification of Vulnerable Sites

In order to allow affected pages to fix the vulnerabilities before they can be exploited, we notified the security teams of all domains for which we could successfully craft exploits. To allow for a better understanding of the general vulnerability as well as the specifics of each domain, we created a Web site detailing the problem associated with cross-domain includes of JavaScript and the attack pattern. In addition, we created proof-of-concept exploits for each flaw and shared this information, augmented by a description of the problem and its impact, e.g., the potential to hijack a user's session, with the domains owners.

As of this writing, we received only three replies stating that the flaw was either being dealt with or had been fixed already. However, none of the affected sites agreed to be mentioned in the paper, therefore we opted to anonymize all the vulnerable services we discovered.

#### 4.5.5 Summary of Our Findings

In total, we found that out of the 49 domains which are dependent on the user's login state, 40 lack adequate protection and can therefore be used to deduce if a user is logged into a certain application. On 28 of these domains, dynamic scripts allowed for unique identification of the current user through various means like customer IDs or email addresses.

Additionally and partly overlapping with the aforementioned scripts, we found that personal data (such as the name or location) was contained in scripts on 13 domains. Last but not least, we encountered four domains which allow for extraction of tokens that could in turn be

used to control the target application in the name of the victimized user. An overview of these results is depicted in Table 2.

## 4.6 Non-exploitable Situations

As shown in Table 2, we were not able to leak data from all of the dynamic scripts we found. In general, we identified two different reasons for this: Either the URL of the script was not guessable by an attacker or the Web site utilized referrer checking to avoid the inclusion of resources by third parties. While these mechanisms protected some Web sites from being exploitable, we believe that the corresponding countermeasures were not placed intentionally against the described attack, but were rather in place because of the used application framework (Referrer checking) or because of the application's design (unguessable URLs). In this section, we briefly discuss and analyze these situations.

### 4.6.1 Unguessable URLs

A prerequisite for the attack described in this paper is that an attacker is able to include a certain script file into his page during a user's visit. For this, the attacker needs to know the exact URL under which a certain dynamic script is available.

Some of the scripts we found required a session ID or another unguessable token to be present in a GET parameter of the URL. As the attacker is in general not able to obtain such a session ID, the script cannot be included by the attacker and hence sensitive data cannot be leaked.

### 4.6.2 Referrer Checking

Another technique that prevented us from exploiting a script leakage vulnerability was referrer checking. When a browser generates an HTTP request for an embedded script, it adds the `Referer` header containing the URL of the embedding site. Many Web pages tend to misuse this header as a security feature [31]. By checking the domain of the referrer, a Web site is in theory able to ascertain the origin of the page requesting a resource.

In 2006, however, Johns showed that referrer checking has several pitfalls [17]. As the `Referer` header was never intended to serve as a security feature, it should not be used as a reliable source of information. So, for example, many proxies and middle boxes remove the `Referer` header due to privacy concerns. Furthermore, several situations exist in which a browser does not attach a `Referer` header to a request and as discussed by Kotowicz, an attacker can intentionally remove the header from requests [19].

As a consequence, servers should not rely on the presence of the `Referer` header. Hence, if a server receives a request for a dynamic script that does not provide a `Referer` header, it needs to decide whether to allow the request or whether to block it. If the request is allowed, the attacker may force the removal of the referrer as discussed before. On the other hand, if the server blocks the request (strict referrer checking), it might break the application for users behind privacy-aware proxies.

We found several domains that implemented referrer checking. However, of seven pages that conducted such a check, only two conducted strict referrer checking. As a consequence, the other five Web sites were still exploitable by intentionally removing the `Referer` header. Listing 7 shows the attack we utilized aiming at stripping the `Referer` header. In this example, we use a data URI assigned to an iframe to embed the leaking script.

**Listing 7** Using a data URL within a frame to send a request without a Referer header

```
var url = "data:text/html,"
    + "<script src='"
    + "http://example.org/dynamic_script.js"
    + "'></script>"

    + "<script>"
    +  "function leakData(){ ... }; "
    +  "leakData();"
    + "</script>";

// create a new iframe
var frame = document.createElement('iframe');
// assign the previously created data url
frame.src = url;
body.appendChild(frame);
```

## 5 Protection Approach

In our study, we observed a surprisingly high number of popular Web sites utilizing the dangerous pattern of using external, dynamically-generated scripts to provide user-specific data to an application. It seems that developers are not aware of the severe consequences this practice has. In order to improve this situation, we provide a secure and functionally-equivalent solution. The main problem of dynamically generated script includes is the incorporation of sensitive user data into files that are not completely protected by the Same-Origin Policy. We discourage this practice and advise developers to strictly separate JavaScript code from sensitive user data.

Figure 2 depicts our design proposal. In this proposal script code is never generated on the fly, but always pulled from a static file. Sensitive and dynamic data values should be kept in a separate file, which cannot be interpreted by the browser as JavaScript. When the static JavaScript gets executed, it sends an XMLHttpRequest to the file containing the data. By default access

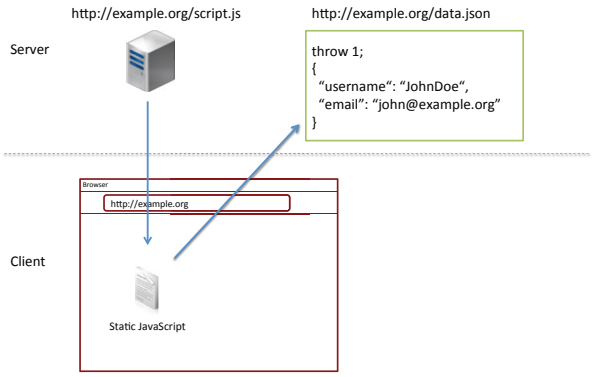


Figure 2: Protection scheme

to the response of an XMLHttpRequest is governed by the Same-Origin Policy. Hence, third-party sites cannot request the file within the user's authentication context. As a consequence, attackers cannot access the data contained within this file. By using Cross-Origin Resource Sharing (CORS) [28], Web developers are able to selectively grant access to the file to any third party service that requires access.

While the attacker is able to include and execute the static JavaScript file within his page, the corresponding code will be executed in the origin of the attacker's Web site. Hence, when the script code requests the data file, which resides in the origin of the legitimate Web site, the two origins do not match and hence the Same-Origin Policy protects the file's content from being accessed by the attacker. If, however, the legitimate site requests the data files, the two origins match and thus access is granted.

As the data file does not contain valid JavaScript code, it cannot be included and executed by the attacker via the HTML `script` tag. To completely avoid this risk, Web developers can either include a so-called unparseable cruft to the beginning of file which causes a compile time failure or add valid JavaScript that effectively stops execution during run time, such as an uncatchable exception (cp. Figure 2) [31].

## 6 Related Work

Conceptually closest to the attacks presented in Section 4.5 is *JSON Hijacking*, an exploitation technique initially presented by Grossman in 2006 [9]. In his attack he utilized a cross-domain script include pointing to a JSON-array resource, which originally was intended as an end-point for an XMLHttpRequest. Via using a non-standard redefinition of JavaScript's object constructor, he was able obtain the content of the user's GMail address book. Grossman reported the issue to Google, where the term Cross-Site Script Inclusion (XSSI) was coined by Christoph Kern. Kern later mentioned the term publicly for the first time in his book from 2007 [18]. Several other authors later on picked up this term to refer to slight variations of the attack [27, 31].

At the same time Chess et al. [3] picked up Grossman's technique, slightly generalized it and coined the term *JavaScript Hijacking*. Unlike the vulnerabilities in this paper, these attacks do not target dynamic JavaScript resources. Instead they use `script`-tags in combination with a non-standard JavaScript quirk (that has been removed from all major browsers in the meantime) to leak data that is encoded in the JSON-array format.

Furthermore, in 2013, Grossman [11] discussed the idea of utilizing resources which are only accessible by users that are logged in to determine the logon status of a user. He also proposed to employ click-jacking attacks on the user to force him to like the attacker's Facebook or Google+ site. In doing so and in comparing the latest addition to his followers, an attacker could thereby deduce the identity of the user currently visiting his website. The idea of determining a user logon status was picked up by Evans [8], who demonstrated a login oracle on `myspace.com` by including a Cascading Style Sheet file from the service which changed certain properties based on whether the user was logged in or not.

In 2015, Takeshi Terada presented another variation of the attack that he called Identifier-based XSSI [27]. Terada used script tags to reference CSV files from third-party domains. A CSV file usually consists of a comma separated list of alphanumeric words. Under certain circumstances this list also represents a syntactically correct list of JavaScript variable declarations. Hence, by referencing such a file the JavaScript engine will create a set of global variables named like the values in the CSV file. By enumerating all globally accessible variables, Terada was able leak the contents of the file.

Other related work has focused on CSS-based history leakage [13, 10, 14]. Analogously to login state leakage, retrieval of a user's history allows an attacker to deduce that a victim has an account on a given site, hence enabling him to start target phishing attacks similar to the ones we outlined in Section 4.5.3.

Another means of utilizing history leakage was discussed in 2010 by Wondracek et al. [30], who proposed a scheme capable of de-anonymizing users based on their group membership in OSNs. To do so, they utilized the stolen history of a user to determine the group sites the user had previously visited. Comparing these to a list of the members of the corresponding groups allowed the authors to determine the user's identity. Recently, for a poster, Jia et al. [16] discussed the notion of utilizing

timing side-channels on the browser cache to ascertain a user's geo location.

In 2012, Nikiforakis et al [24] conducted a large-scale analysis of remote JavaScript, focusing mainly on the potential security issues from including third-party code. For W2SP 2011, two groups [20, 15] conducted an analysis of cross-domain policies for Flash, aiming specifically at determining those domains which allow access from any domain. Since Flash attaches the cookies for the *target* domain to said requests, they discussed attack scenarios in which a malicious Flash applet is used to retrieve proprietary information.

In addition to these attacks, Paul Stone demonstrated another means of stealing sensitive information across origin boundaries. To do so, he leveraged a timing side channel, allowing him to leak a framed document pixel by pixel [26].

## 7 Outlook

The goal of this paper was to conduct an initial study into the usage and potential pitfalls of dynamic scripts in real world applications. Our data set of 150 highly ranked domains gives a good glimpse into the problems caused by such scripts. Nevertheless, we believe that a large-scale study could provide additional key insights into the severity of the issue. To enable such a study, an important problem to solve is the automation of the analysis—starting from fully automated account registration and ranging to meaningful interaction with the application. Therefore, implementing such a generic, yet intelligent crawler and investigating how well it can imitate user interaction is a challenging task we leave for future work. Along with such a broader study, enhancements have to be made to cope with the increased amount of data. As an example, our Chrome extension could use advanced comparisons based on syntactical and semantical differences of the JavaScript code rather than based on content. Since our data set was limited by the fact that our analysis required manual interaction with the investigated applications, the need to automate the secondary analysis steps, i.e., examination of the differences and verification of a vulnerability, did not arise.

Recently, the W3C has proposed a new security mechanism called *Content Security Policy* (CSP), which is a "declarative policy that lets authors of a web application inform the client from where the application expects to load resources" [25]. In its default setting, CSP forbids the usage of inline scripts and hence, programmers are compelled to put the code into external scripts. During our study we noticed that many of these inline scripts are also generated dynamically and incorporate sensitive user data. If all these current inline scripts are naively transformed into dynamic, external script resources, it is highly likely that the attack surface of this paper's attacks will grow considerably.

For instance, Doupé et al. [4] developed a tool called *deDacota* which automatically rewrites applications to adhere to the CSP paradigms by moving all inline script code to external scripts. As our work has shown, these external scripts – if not protected properly – may be included by any third-party application and hence might leak secret data. Therefore, we believe that it is imperative that measures are taken to ensure the secure, yet flexible client-side access to sensitive data and that the changing application landscape caused by CSP adoption is closely monitored. As discussed by Weissbacher et al., however, CSP is not yet widely deployed and significantly lags behind other security measures [29].

Furthermore, in this paper, we exclusively focused on dynamic JavaScript that is pulled into the browser via `script`-tags. This is not necessarily the only method, how server generated script content is communicated. An alternative to `script` tags is to transport the code via XMLHttpRequests bodies, which are subsequently passed to the `eval()` API. In future work, we plan to investigate such XMLHttpRequests endpoints in respect to their susceptibility to attack variants related to this paper's topic.

Finally, as related work has indicated, internal application information, such as the login state of a user, may also be leaked via images or style sheets. In this case, the observed effects of a cross-domain element inclusion manifest themselves through side effects on the DOM level, as opposed to a footprint in the global script object. Hence, a systematical further analysis on other classes of server-side content generation that might enable related attacks would be a coherent extension of our work.

## 8 Summary & Conclusion

In this paper, we conducted a study into the prevalence of a class of vulnerabilities dubbed Cross-Site Script Inclusion. Whenever a script is generated on the fly and incorporates user-specific data in the process, an attacker is able to include the script to observe its execution behavior. By doing so, the attacker can potentially extract the user-specific data to learn information which he otherwise wouldn't be able to know.

To investigate this class of security vulnerabilities, we developed a browser extension capable of detecting such scripts. Utilizing this extension, we conducted an empirical study of 150 domains in the Alexa Top 500, aimed at gaining insights into prevalence and purpose of these scripts as well as security issues related to the contained sensitive information.

Our analysis showed that out of these 150 domains, 49 domains utilize server-side JavaScript generation. On 40

domains we were able to leak user-specific data leading to attacks such as deanonymizing up to full account hijacking. Our practical experiments show that even high-profile sites are vulnerable to this kind of attacks.

After having demonstrated the severe impact these flaws can incur, we proposed a secure alternative using well-known security concepts, namely the Same-Origin Policy and Cross-Origin Resource Sharing, to thwart the identified security issues.

## Acknowledgements

The authors would like to thank the anonymous reviewers for their valued feedback. More over, we want to thank our shepherd Joseph Bonneau for the support in getting our paper ready for publication. This work was in parts supported by the EU Project STREWS (FP7-318097). The support is gratefully acknowledged.

## References

[1] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 75–88.

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

[3] CHESS, B., O'NEIL, Y. T., AND WEST, J. JavaScript Hijacking. [whitepaper], Fortify Software, `http://www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf`, March 2007.

[4] DOUPÉ, A., CUI, W., JAKUBOWSKI, M. H., PEINADO, M., KRUEGEL, C., AND VIGNA, G. dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 1205–1216.

[5] DOWNS, J. S., HOLBROOK, M. B., AND CRANOR, L. F. Decision strategies and susceptibility to phishing. In *Proceedings of the second symposium on Usable privacy and security* (2006), ACM, pp. 79–90.

[6] ECMASCRIPT, E., ASSOCIATION, E. C. M., ET AL. Ecmascript language specification, 2011.

[7] ELECTRONIC FRONTIER FOUNDATION. Panopticlick – how unique – and trackable – is your browser? online, `https://panopticlick.eff.org/about.php`, last accessed 2014/05/10.

[8] EVANS, C. Cross-domain leaks of site logins. online, `http://bit.ly/1lz1HPl`, last accessed 2014/05/10.

[9] GROSSMAN, J. Advanced Web Attack Techniques using GMail. [online], `http://jeremiahgrossman.blogspot.de/2006/01/advanced-web-attack-techniques-using.html`, January 2006.

[10] GROSSMAN, J. I know where you've been. [online], `http://jeremiahgrossman.blogspot.com/2006/08/i-know-where-youve-been.html`, August 2006.

[11] GROSSMAN, J. The web won't be safe or secure until we break it. *Communications of the ACM 56*, 1 (January 2013), 68–72.

[12] HANSEN, R., AND GROSSMAN, J. Clickjacking. *Sec Theory, Internet Security* (2008).

[13] JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. C. Protecting Browser State from Web Privacy Attacks. In *Proceedings of the 15th ACM World Wide Web Conference (WWW 2006)* (2006).

[14] JAKOBSSON, M., AND STAMM, S. Invasive Browser Sniffing and Countermeasures. In *Proceedings of The 15th annual World Wide Web Conference (WWW2006)* (2006).

[15] JANG, D., VENKATARAMAN, A., SAWKA, G. M., AND SHACHAM, H. Analyzing the cross-domain policies of flash applications. In *Proceedings of the 5th Workshop on Web* (2011), vol. 2.

[16] JIA, Y., DONGY, X., LIANG, Z., AND SAXENA, P. I know where you've been: Geo-inference attacks via the browser cache. IEEE Security&Privacy 2014, `http://www.ieee-security.org/TC/SP2014/posters/JIAYA.pdf`, last accessed 2014/05/17.

[17] JOHNS, M., AND WINTER, J. Requestrodeo: Client side protection against session riding. *Proceedings of the OWASP Europe 2006 Conference* (2006).

[18] KERN, C., KESAVAN, A., AND DASWANI, N. *Foundations of security: what every programmer needs to know*. Apress, 2007.

[19] KOTOWICZ, K. Stripping the referrer for fun and profit. online, `http://blog.kotowicz.net/2011/10/stripping-referrer-for-fun-and-profit.html`, last accessed 2014/05/10.

[20] LEKIES, S., JOHNS, M., AND TIGHZERT, W. The state of the cross-domain nation. In *Proceedings of the 5th Workshop on Web* (2011), vol. 2.

[21] MOZILLA. Inheritance and the prototype chain. online, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain`, last accessed 2014/05/10.

[22] MOZILLA. Mutationobserver. online, `https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver`, last accessed 2014/05/10.

[23] MOZILLA DEVELOPER NETWORK, AND RUDERMAN, J. Same-origin policy. online, `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`.

[24] NIKIFORAKIS, N., INVERNIZZI, L., KAPRAVELOS, A., ACKER, S. V., JOOSEN, W., KRUEGEL, C., PIESSENS, F., AND VIGNA, G. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *19th ACM Conference on Computer and Communications Security (CCS 2012)* (2012).

[25] STERNE, B., AND BARTH, A. Content security policy 1.0. online, `http://www.w3.org/TR/2012/CR-CSP-20121115/`. last accessed 2014/05/10.

[26] STONE, P. Pixel perfect timing attacks with html5.

[27] TERADA, T. Identifier based xssi attacks, 2015.

[28] VAN KESTEREN, A., ET AL. Cross-origin resource sharing. *W3C Working Draft WD-cors-20100727* (2010).

[29] WEISSBACHER, M., LAUINGER, T., AND ROBERTSON, W. Why is csp failing? trends and challenges in csp adoption. In *Research in Attacks, Intrusions and Defenses*. Springer, 2014, pp. 212–233.

[30] WONDRACEK, G., HOLZ, T., KIRDA, E., AND KRUEGEL, C. A practical attack to de-anonymize social network users. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 223–238.

[31] ZALEWSKI, M. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.