Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control

Eric Y. Chen, Sergey Gorbaty, Astha Singhal and Collin Jackson
Carnegie Mellon University
{*eric.chen, sergey.gorbaty, collin.jackson*}*@sv.cmu.edu, astha.singhal@andrew.cmu.edu*

***Abstract*—Since the early days of Netscape, browser vendors and web security researchers have restricted out-going data based on its destination. The security argument accompanying these mechanisms is that they prevent sensitive user data from being sent to the attacker's domain. However, in this paper, we show that regulating web information flow based on its destination server is an inherently flawed security practice. It is vulnerable to self-exfiltration attacks, where an adversary stashes stolen information in the database of a whitelisted site, then later independently connects to the whitelisted site to retrieve the information. We describe eight existing browser security mechanisms that are vulnerable to these "self-exfiltration" attacks. Furthermore, we discovered at least one exfiltration channel for each of the Alexa top 100 websites. None of the existing information flow control mechanisms we surveyed are sufficient to protect data from being leaked to the attacker. Our goal is to prevent browser vendors and researchers from falling into this trap by designing more systems that are vulnerable to self-exfiltration.**

I. Introduction

As the World Wide Web matures into a ubiquitous computing platform, people are growing comfortable with sharing their personal information with web applications they trust. However, this casual sharing of information is accompanied by serious privacy and security implications. Vulnerabilities in web applications can lead to compromise of users' sensitive data, resulting in embarrassment, inconvenience, and financial loss.

Some of the most prominent attacks that exist on the web today are code injection attacks. In a code injection attack, the adversary injects malicious JavaScript or HTML code into a benign web page the user is viewing, allowing the attacker to either perform sensitive actions on behalf of the user or steal user's sensitive information. In this paper, we focus on the scenario where a code injection attack leads to the exfiltration of users' personal information, which we refer to as a *data-exfiltration attack*.

Many browser vendors and researchers have come up with solutions to protect users from data-exfiltration attacks. However, much of the existing work is based on prohibiting information flow to unauthorized web servers. This paper presents a new class of data-exfiltration attacks that circumvent these existing defenses for data-exfiltration. To launch this attack, the adversary first stores users' information in the database of a whitelisted site, then later independently connects to the whitelisted site to retrieve the information. Because the attack relies on exfiltrating users' sensitive information through either the victim website itself or another whitelisted website, we call this attack a *self-exfiltration attack*. We demonstrate that an adversary can launch self-exfiltration attacks with or without executing JavaScript.

To successfully launch a self-exfiltration attack, the adversary must utilize an existing channel on the whitelisted website to store stolen information. To confirm whether these exfiltration channels exist in real world websites, we surveyed 100 websites and discovered at least one exfiltration channel for each of these websites. We conclude that none of these existing data-exfiltration defenses can protect real world websites from self-exfiltration attacks.

Organization

Section II introduces data-exfiltration attacks, presents existing defenses, and outlines the threat model assumed for the rest of this paper. Section III introduces self exfiltration attacks and details the steps required to launch such an attack. Section IV presents the results of our survey. Section V discusses possible solutions, and finally Section VI concludes.

II. Data-Exfiltration

The desire to safeguard users' information from untrusted websites led to the creation of the most important security feature in browsers – the *same origin policy*. The same origin policy states that JavaScript from one origin should not be able to read the private documents loaded from another origin. Although this policy prevents adversaries from trivially obtaining users' information, it is by no means a panacea. Countless attacks have been discovered that compromise users' data despite the same origin policy's restrictions.

One way for an adversary to circumvent the same origin policy is to steal sensitive information using cross site scripting (XSS) attacks. Cross site scripting attacks happen when an adversary is able to inject JavaScript code onto a victim site's page and lure the user into visiting this page. When the user visits this malicious page, the attacker's script will execute in the context of the victim's origin. At this point, the attacker can exfiltrate any private data on that page and can often use certain private data on that page (such as CSRF tokens) to perform further unauthorized actions (e.g. transfer money to the attacker's account). For the rest of this paper, we will focus on the first step of the attack where the adversary simply wishes to obtain access to the sensitive

Data type	Script attacker	Non-script attacker
Cookie	✓	
Password	✓	✓
CSRF Token	✓	✓
Capability-bearing URL	✓	✓
Personal data	✓	✓

Table I
PERSONAL DATA VULNERABLE TO A DATA-EXFILTRATION ATTACKS.

Data exfiltration defense	Year of release
Navigator 3.0	1996
Noxes	2006
Dynamic tainting (Vogt *et al.*)	2007
SOMA	2008
HTTP Fences	2008
Content Security Policy	2010
Security Style Sheet	2011
Dynamic tainting (Jang *et al.*)	2011

Table II
EXISTING DEFENSES FOR DATA EXFILTRATION AND THEIR YEAR OF RELEASE.

data. Table I lists examples of private data that can be stolen by this attack. We proceed by describing these data types in detail.

- **Cookies** – Cookies are commonly used as authentication tokens. If an attacker manages to compromise a user's session cookie, she can subsequently impersonate the user and initiate unauthorized transactions within the same site. To protect session cookies from being compromised by the adversary, many websites label their cookies as "HttpOnly." HttpOnly cookies cannot be obtained using Javascript, hence a web attacker can only steal a cookie if the HttpOnly flag is absent.
- **Password** – Most modern browsers offer password managers that store users' passwords for websites they visit. These saved passwords will be automatically filled in for future visits. An adversary could inject fraudulent HTML elements into the victim's website to trick the browser's password manager into exposing the user's login credentials.
- **CSRF Tokens** – Many websites use cross site request forgery (CSRF) tokens to distinguish legitimate requests from forgeries. In a nutshell, CSRF tokens are shared secrets between the client and the server. The server will only accept requests made from the client if the client presents a valid CSRF token. This effectively prevents malicious websites from tricking the victim's browser into issuing a bogus request. However, if the attacker is able to obtain a CSRF token, she can use it to forge requests, bypassing the existing defense.
- **Capability-bearing URLs** – Michal Zalewski suggested in his blog post [1] that capability-bearing URLs are also vulnerable to data theft. These capability-bearing URLs are utilized by many web applications to issue invitations, enable sharing of private user content, and to implement single sign-on flows. If an adversary is able to obtain a capability-bearing URL, she can gain access to the user's private data, or even worse, full access to the user's account.
- **Personal Data** – Many websites include users' personal information, such as their addresses, phone numbers, content of their private emails, and even the contact information of all their friends. This information is extremely valuable to spammers and phishers.

We define a *data exfiltration attack* to be an attack where the adversary exports user's private data to a server controlled by the attacker, possibly using a code injection vulnerability. It has to be noted that although cross site scripting vulnerabilities can lead to data exfiltration attacks, an adversary doesn't necessarily need JavaScript to exfiltrate data. We will illustrate in Section III how an adversary can exfiltrate data without executing any JavaScript.

A. *Existing Defenses*

Various defenses have been proposed to mitigate data-exfiltration attacks by regulating outgoing data based on its *destination address*. When the victim page makes a request containing sensitive data to an unknown origin, the defense mechanism will assume the request to be malicious and deny it. Unfortunately, this does not solve the core of the problem; the attacker can still export users' confidential information to a white-listed origin. For the rest of this section, we describe the current mitigations for data-exfiltration (summarized in Table II) and explain why they are insufficient to stop real world adversaries.

Dynamic tainting is a technique that tracks the information flow of sensitive data. The goal is to ensure sensitive data is never sent to an unauthorized source. Dynamic tainting of web-based data was first seen in Netscape Navigator 3.0 [2]. Users of Navigator 3.0 may enable the taint feature to enforce that data loaded from one website will never be exported to another website. Besides Netscape, many academic researchers also proposed to use dynamic tainting to mitigate web vulnerabilities. In a paper published in NDSS 2007 [3], Vogt *et al.* proposed to use dynamic tainting to mitigate cross site scripting attacks. Most recently, Jang *et al.* also suggested using dynamic taint tracking to protect browser plugins from leaking sensitive user information [4]. Dynamic tainting requires the browser to differentiate between trustworthy and non-trustworthy sites, because it needs to decide whether or not sensitive information should be sent. However, the boundary of trustworthiness is not so clear; the adversary can export the data to a trusted site, then later independently connect to it to retrieve the information.

HTTP Fences [5] uses the term "information leak attack" to describe data-exfiltration. It attempts to mitigate data-exfiltration using a whitelist to enforce that sensitive data is only sent to authorized servers. Similar to dynamic tainting, HTTP fences has an implicit notion of which sites should be trusted. This can be exploited by the attacker to first smuggle data to a trusted website, then fetch the data through a separate channel.

SOMA, Security Style Sheets, and Noxes [6]–[8] attempt to prevent cross site scripting attacks using a whitelist to determine which server the user is allowed to send data to. SOMA and Security Style Sheets require site operators to approve external domains for sending or receiving information. In contrast, Noxes acts as a web proxy and uses both manual and automatically generated rules to filter outgoing traffic. Unfortunately, all three papers make the same assumption that cross site scripting attacks necessarily engage in cross-domain communications, and by restricting malicious outgoing requests, one can block cross site scripting attacks. However, this assumption is not correct, because mitigating data-exfiltration attacks is different from mitigating cross site scripting attacks – a cross site scripting attacker can still use JavaScript to perform unauthorized actions (e.g., using JavaScript function calls to transfer money to the attacker's account), without leaking any sensitive data. In addition to not being able to protect users from unauthorized actions, neither of these systems can effectively defend users against data-exfiltration attacks as well. Similar to previous examples, the adversary can send data to a whitelisted origin and extract the data independently using a channel visible to the attacker.

Content Security Policy (CSP) [9] was originally proposed by Mozilla as a mitigation for cross site scripting attacks. Since then, it has been adopted by Chrome and Safari and has become a W3C specification. Websites can opt-in to CSP by attaching a custom HTTP header, `X-Content-Security-Policy`, with pages they serve. The browser will enforce CSP policies indicated in the custom header. One of the most important functionalities of CSP is that it disables inline scripts by default, and only allows JavaScript from authorized external script files to execute. Although this mitigates the majority of cross site scripting attacks, we show in Section III-A2 that the attacker can still export users' sensitive data using "HTML form injection attack", without running any JavaScript code. Recently, there have been discussions on the W3 security mailing list to create a CSP form post directive. The form post directive enables websites to specify which origin they are allowed to post the content of HTML forms to, in an attempt to prevent form injection attacks. Unfortunately, we demonstrate in Section III that the form directive is not sufficient to stop form injection attacks. An attacker can post the form to a white listed domain to bypass the CSP form directive.

B. Threat Model

This section describes the capability of the adversary that we assume for the rest of this paper. A *data-exfiltration attacker* is similar to a standard web attacker threat model [10], where the adversary has a web server under her control, and has the ability to lure the user into visiting her web page. However, in addition to the traditional web attacker model, a data-exfiltration attacker also has the ability to inject her own code into the victim's web page. This is a realistic assumption, because most data-exfiltration defenses are designed as alternatives to traditional XSS defenses.

Several data-exfiltration mitigations (e.g., Content Security Policy) prevent injected JavaScript code from executing in the victim's page. Therefore, a data-exfiltration attacker may or may not have the ability to execute JavaScript. We proceed by dividing the adversary into two categories:

- **Script attackers** are adversaries who are able to inject JavaScript code onto the victim's page.
- **Non-script attackers** are adversaries who cannot inject any JavaScript code, but can still inject non-JavaScript code such as HTML elements or Cascading Style Sheets.

III. Self-exfiltration Attack

In this section, we investigate the shortcomings of existing defenses. For each type of attacker described in Section II-B, we present methods they can use to extract users' sensitive data. Furthermore, we outline a list of vulnerable channels an adversary can use to export the collected information, bypassing each of the systems discussed in Section II-A.

Data-exfiltration, like its name suggests, requires the adversary to export the data out of the victim's domain. Hence, an obvious but naive approach to mitigate data-exfiltration attacks would be to detect and block these data exports. In fact, all previously discussed defenses for data-exfiltration rely on preventing these data exports from happening. Unfortunately, none of these defenses have considered the scenario where the attacker chooses not to export the data out of the victim's domain, but instead, she stores the sensitive data in a region of the victim's domain that is visible to the attacker.

To see how an adversary can launch this attack on a real website, consider the following example:

1) Alice visits auction.com, an online auction website that deploys a data-exfiltration protection discussed in Section II-A.
2) Due to a code injection vulnerability, Mallory is able to inject JavaScript code onto the page Alice is viewing, and read Alice's session cookie. However, due to the data-exfiltration protection, Mallory is unable to send Alice's cookie to her own server using JavaScript.
3) Instead of exporting Alice's cookie out of the auction site, Mallory posts the cookie as a user comment to one of her own auction items.
4) Mallory can now extract Alice's cookie by viewing the comments to her auction.

In this scenario, the adversary is able to exfiltrate user's data despite the existing data-exfiltration defense. This is because all of the current protections for data-exfiltration failed to consider that most websites contain pages visible to the attacker; an attacker can extract user's information without explicitly sending the information to the attacker's server. Since this type of attack involves transferring user's private data to another region of the same website, we refer to it as a *self-exfiltration attack*.

A. *Obtaining the data*

Before the adversary can export user's sensitive data, she must first obtain this data. In this section, we investigate the techniques an attacker can use to obtain the data mentioned in Table I. More specifically, we will focus on the two types of adversaries described in section II-B, script attackers and non-script attackers.

1) Script attacker: A script attacker has the ability to execute JavaScript code in the victim's origin. Because the attacker and the victim share the same origin, the same origin policy would permit the attacker to request and read all contents of the victim's origin. Using JavaScript, the attacker can obtain all data described in Table I, including the user's session cookie (if the HttpOnly flag is absent), CSRF tokens, sensitive URLs, profile information, and the user's password.

2) Non-script attacker: At first glance, it is not clear what data a non-script attacker can obtain. Without JavaScript, the adversary would not be able to read user's cookie, or access arbitrary content on the user's page. However, a clever non-script adversary can still obtain a considerable amount of sensitive data even without executing JavaScript code. In fact, a non-script attacker has the ability to gain access to all of the data shown in Table I, with exception of cookies.

Before going into any details, we would like to make a small disclaimer. After the initial distribution of this paper, Gareth Heyes pointed out in his blog post [11] that the form element injection attacks we discovered were, in fact, not new. The attacks were openly discussed on *sla.ckers.org* many years ago [12]. We humbly thank Gareth Heyes for informing us about the history of these attacks. In the section below, we present a more complete version of our original non-script based attacks, incorporating ideas from the *sla.ckers.org* community.

Base target overwrite – The HTML *base* element's target attribute can be used to set the window.name property of all documents opened from hyperlinks. For instance, if the webpage from http://cmu.edu sets its base target to the string "CMU", and if a user clicks on a hyperlink of this page, the resulting page would have its windows.name property set to "CMU".

Unfortunately, one minor caveat of the *base* element makes it a viable vector for non-script adversaries to steal sensitive information. The caveat works as follows, when parsing the target string, browsers will accept any characters, including newline, that comes between two `"` character. Consider the following code snippet:

```
<!--line 2 is the attacker's code-->     1
<base target="                           2
<script>                                 3
var secret=PRIVATE_INFO;                 4
var name="Eric";                         5
```

In this example, the attacker injects the malicious code in line 2. This target string consumes all HTML content before the first `"` character in line 5. If a user clicks on an attacker controlled hyperlink, the newly opened webpage will be able to extract the value of the `secret` variable by calling `window.name`.

Abusing HTML forms – Another common way for a non-script attacker to extract users' confidential information is by using HTML forms. For readers who are unfamiliar with the subject, an HTML form is a portion of an HTML document that contains regular HTML content as well as user adjustable fields called "controls". Controls are HTML elements that typically require user input, such as text fields, check boxes, radio buttons, text areas, and password fields. Users complete HTML forms by modifying the controls(e.g., fill in their passwords), then submit the completed forms to web servers via HTTP GET or POST requests. We demonstrate that an adversary can steal users' confidential information by injecting or modifying HTML forms on the victim page; then subsequently trick the user's browser into leaking sensitive information through form controls.

- **Button formaction overwrite** – According to the HTML5 specification [13], the *formaction* attribute of an HTML button can be used to overwrite the action attribute of its parent form. Once overwritten, clicking of the button would result the form data to be posted to the location specified by the attacker. The code snippet below demonstrates this attack:

```
<form action="update_info.php"                  1
          method="post">                        2
<input type="text" id="name" />                 3
<input type="text" id="addr" />                 4
<input type="text" id="creditcard" />           5
                                                6
<!--Beginning of attacker's code -->            7
<button formaction="http://evil.com">           8
        Button </button>                        9
<style> #submit{visibility:hidden;}             10
</style>                                        11
<!--End of attacker's code -->                  12
                                                13
<input type="submit"                            14
       value="Real Button" />                   15
```

In this example, the attacker injects a bogus submit button in line 8, and hides the real submit button using the style sheet in line 10. When the user clicks on the bogus submit button, all of her personal information will be submitted to evil.com.

- **Attack on text area** – Another way to steal HTML form data is by using HTML text area elements. This attack is further enhanced by the relaxed policies browsers employ when handling malformed HTML content. When a browser detects a syntactically incorrect HTML statement, instead of rejecting it as an error, it will sometimes attempt to infer the correct syntax. A non-script attacker can utilize this feature to lure the user into exfiltrating their data with the following code:

```
<!--Beginning of attacker's code -->
<form action="comment.php"
          method="post">
<input type="submit"
       value="Click to continue" />
<textarea style="visibility:hidden;">
<!--End of attacker's code -->
...
<!--User's sensitive data -->
...
```

  In this example, the attacker first creates a form that would post to a vulnerable region of the victim's domain. Inside the form tag, the attacker uses a half-opened HTML text area tag to enclose user's data; it is half-open because the standard text area syntax requires a closing tag (i.e., </textarea>) at the end of the enclosed content. When the browser observes a half-open text area, it will attempt to infer where the text area terminates. For most browsers, the termination point is assumed to be at the next text area closing tag, or at the end of the HTML document. This indicates that unless the attacker's code is followed by another text area, all of the user's private data appearing after the attacker's code will be contained in text area. When the user clicks on the submit button, all data inside this text area will be posted to the location indicated by the attacker. We have confirmed this attack for Chrome 16, Firefox 10, Safari 5, and IE 8.
  It must be noted that this attack may work even if browsers do not infer the end of a text area. Similar to cross site style sheet attacks discovered by Huang *et al.* [14], if the attacker is able to inject code before and after the sensitive data, she can terminate the tag herself.
- **Attack on drop-down menu** – Similar to the previous attack on text areas, the *option* tag for HTML drop down menus can also be used to consume sensitive data. To launch the attack, the adversary would use a half-open <option>tag to surround the sensitive HTML text that comes after, as opposed to using a <textarea>tag in the previous example.
- **Attack on Password Manager** – Most browsers have password manager features that allow users to save their passwords into the browser. The next time users visit the same site, the browser will autofill the password fields with the saved passwords. Different browsers implement password managers differently, but this attack is applicable to all password managers that autofill passwords based on the URL of the page containing the password. For example, Chrome's password manager would autofill a password if the current page shares the same origin as the page where the password is stored [15]. This implies that an adversary who is able to inject a password field into an arbitrary page of the victim's domain can now exfiltrate user's password if she can lure the user into submitting the form.

A keen reader may notice that HTML form based attacks rely on a successful phishing attempt. That is, these attacks require users to trigger the malicious form submission. However, unlike a traditional phishing attack, where the user may detect the attack by carefully observing the browser's security indicators (e.g., the URL bar or the SSL lock icon), form injection attacks are more difficult to detect because the adversary resides on the same page as the victim. Furthermore, the adversary can change the opacity and the size of the button to make it completely transparent and cover the entire page, triggering the malicious form submission regardless of where the user clicks.

B. Exfiltrating the data

As the last step of a self-exfiltration attack, the adversary must be able to store the user's information to a location that can be later read by the attacker. We will now describe several channels in real world web applications that can be utilized to bypass data-exfiltration protections:

- **Public comment** – Many popular websites such as YouTube, Amazon, and CNN include comment sections for content they serve. This allows users to discuss videos, review books, or even participate in heated debates about recent news events. Unfortunately, this opens up a venue for self-exfiltration attacks. Once an adversary obtains user's sensitive information, she can post this information as a publicly visible comment, and later extract the information by viewing the comment page in her own browser.
- **Public profile** – Many websites allow users to create profiles that can be viewed by other users. This feature is especially common among social networks and online dating sites, where users have the option to disclose their age, physical location, or even personal interests. The adversary can abuse this feature to export stolen data by posting it on the user's public profile, then later collect the data by viewing the profile.
- **Private message/email invitation** – Most websites with user profiles also offer a private message feature, where one user may directly contact another user by sending them an in-site message that is only visible to the two parties involved. Once the adversary obtains a user's confidential information, she can simply direct the user (via JavaScript or HTML form post) to send a private

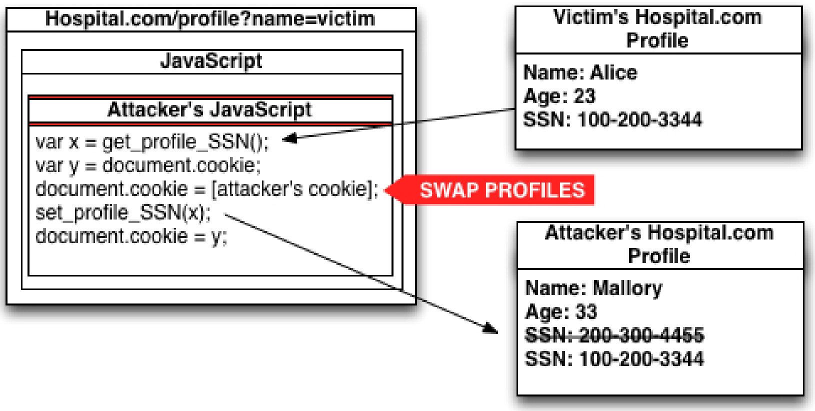

Figure 1. JavaScript based self-exfiltration attack used to steal victim's Social Security Number (SSN).

message containing the sensitive information to the attacker. Besides the private message feature, many sites also offer the option to send an email invitation to your friends. Similar to private messaging, an attacker can also use the email feature to export sensitive data to herself, bypassing any data-exfiltration defense.

- **Attacker's profile** – Some websites contain user profiles, but these profiles are not visible to public; users may only view their own profiles by logging into their account. On the surface, it is not obvious how an attacker can exfiltrate data through these private profiles. However, a clever *script attacker* can override the user's session with her own to exfiltrate the sensitive data. Figure 1 demonstrates an example of such an attack, the attack can be done in the following steps:
  1) User views a victim page containing the attacker's JavaScript code.
  2) Attacker reads user's sensitive data using JavaScript.
  3) Attacker overwrites the user's session cookie with an existing session cookie of the attacker. At this point, any changes made to the user's profile will be made to the attacker's profile instead.
  4) Attacker stores (in her own profile) the sensitive information she obtained earlier.
  5) Attacker logs in to her own profile later and obtains the user's sensitive information.

  This attack is possible due to the stateless nature of HTTP, that is, the identity of the user is entirely linked to her cookie. A script attacker can swap sessions with any user by overwriting the user's cookie with her own session cookie.
- **Search history** – Many websites offer search features similar to popular search engines. It is a common practice for these sites to link each user's search history with their accounts, so users can view their search history in a drop down menu by clicking on the search box. However, similar to the previous attack, an adversary could overwrite the user's session with her own, then post sensitive information as search queries. These search queries can later be accessed by the attacker by simply logging into her account and clicking on the search box to display the most recent search queries.

Types of websites surveyed	Number of websites
Sites with JavaScript exfiltration channels	57
Sites with Non-JavaScript exfiltration channels	43
Sites without exfiltration channels	0

Table III
NUMBER OF SURVEYED WEBSITES CONTAINING EXFILTRATION CHANNELS.

IV. EVALUATION

To determine if most real world web applications contain exfiltration channels for attackers to store stolen information, we surveyed a total of 100 websites. These websites were gathered from the Alexa top 100 global sites list. Since this study required us to create an account on each website to conduct manual analysis; we replaced all non-english sites, adult sites, and sites that we cannot obtain an account for (e.g., banks) with sites further down the Alexa's top website list.

For each website in our study, we assume that the attacker has obtained the user's private information through a code injection attack. This is a realistic assumption because many data-exfiltration defenses act as a replacement for traditional JavaScript/HTML injection defenses deployed by modern websites. Therefore, we must evaluate these defenses individually in the absence of existing security mechanisms. We proceed by analyzing if each site contains a channel for the attacker to store stolen information, and whether exploiting this channel requires JavaScript execution capabilities. The results are shown in Table III. We managed to discover a JavaScript based exfiltration channel for **all** of the websites in our study. Additionally, 43% of these channels can be exploited by a non-JavaScript attacker. We discuss below some of the challenges we faced during our study and how we resolved them.

- **CSRF token** – CSRF tokens are user-specific, session-specific secrets that are sent with form submissions to assist the server in verifying the user. An adversary located on another website will not be able to make unauthorized requests to the victim's domain using the secret token, because browsers' same origin policy prevents attacker's scripts from obtaining the secret token. A Self-exfiltration attack is somewhat similar to a CSRF attack, because a self-exfiltration attacker also attempts to make unauthorized state changes to the victim's web server. However, although CSRF tokens can be used to mitigate cross site request forgery attacks, they are ineffective against self-exfiltration attacks. This is because unlike most CSRF attackers, a self-exfiltration attacker is located on the same origin as the victim, enabling them to obtain the secret token trivially using JavaScript.
- **CAPTCHA** – Many websites require users to solve CAPTCHAs before performing highly sensitive actions, for the purpose of reducing spams and bogus user accounts. CAPTCHAs are challenges posed to the user to verify if they are human. By definition, an ideal CAPTCHA cannot be solved by an automated JavaScript program. This is problematic for a self-exfiltration adversary because if the exfiltration channel (e.g., a comment box) contains a CAPTCHA, then she would not be able to export the data. Fortunately for the attacker, CAPTCHAs have a major usability drawback, that is, the conversion rate of web forms decrease dramatically when users have to solve CAPTCHAs [16]. This discourages websites from deploying a large number of CAPTCHAs. In our study, we were able to discover one non-CAPTCHA exfiltration channel for every website in our survey.
- **AJAX** – Many websites use Asynchronous JavaScript and XML (AJAX) to commit changes to the server. It may appear that JavaScript is required to exfiltrate data for these AJAX-based sites. However, most AJAX requests can be simulated with HTML form posts. In our study, the only case where we failed to simulate AJAX requests is when AJAX requests are accompanied by CSRF tokens.

V. Solutions

We have demonstrated in previous sections that data-exfiltration attacks cannot be mitigated by filtering outbound requests based on their destination servers, because most whitelisted web applications contain exfiltration channels the attacker can use to smuggle stolen user data. In this section, we turn to defense and discuss what a website has to do in order to protect itself from data-exfiltration attacks.

One weakness in existing data-exfiltration defenses is that the whitelist used to filter outbound requests is too coarse-grained. That is, websites are only allowed to specify the origin of outbound requests, not individual URLs. One may argue that, by allowing websites to specify individual URLs on the whitelist, an attacker would be prohibited from transferring stolen data to a vulnerable region of the website. Unfortunately, this finer-grained whitelist suffers from two fatal drawbacks – First, web developers are required to identify all valid URLs their web pages are allowed to send requests to, including forms, images, or even user supplied hyperlinks. Additionally, all whitelisted exfiltration channels must be removed, such as comment sections, forums, private messages, or profile pages. Because fulfilling both of these requirements is highly impractical for any real world web application, we urge web developers to make use of other mitigations for data-exfiltration attacks, such as the ones we describe below.

Instead of protecting users' data from being stolen after the occurrence of a code injection attack, websites can attempt to prevent the code injection attack from occurring in the first place. Content Security Policy [9], BEEP [17], App Isolation [18] and XSS filters [19] can be used to prohibit the attacker from injecting JavaScript into the victim's page. It is crucial to combine these systems with an HTML sanitizer in order to address non-script based form injection attacks. Alternatively, one may use context-sensitive sanitization tools such as ScriptGard [20] or CSAS [21] to automatically sanitize user input and eliminate any code injection vulnerabilities.

The HTML5 iframe sandbox attribute [22] gives websites the ability to isolate untrusted documents into iframes with unique origins and with JavaScript disabled. Similar to other cross site scripting mitigations, iframe sandbox by itself is not sufficient to stop data-exfiltration attacks (e.g., the attacker can still use a form injection attack to read sensitive data in the sandboxed page). Therefore, other defenses (such as an HTML sanitizer) must be used in conjunction with the iframe sandbox to fully protect users from data-exfiltration attacks.

VI. Conclusion

We have surveyed a number of data-exfiltration mitigations that filter outbound requests based on their destination servers. An adversary can bypass these protections with relative ease, by stashing stolen data in a whitelisted database. To verify our claim, we surveyed popular websites and confirmed that self-exfiltration attacks are feasible to most real world web applications. We urge future researchers to beware of self-exfiltration attacks when designing defenses for data-exfiltration.

References

[1] M. Zalewski, "Notes from the post-XSS world," http://lcamtuf.coredump.cx/postxss/.

[2] D. Flanagan, *JavaScript: The Definitive Guide, 4th Edition*. O'Reilly Media, 2001, chapter 20.4.

[3] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *Proceedings of the Network and Distributed System Security Symposium (NDSS))*, 2007.

[4] D. Jang, A. Venkataraman, G. M. Sawka, and H. Shacham, "Analyzing the Cross-domain Policies of Flash Applications," in *Web 2.0 Security and Privacy (W2SP)*, 2011.

[5] S. Stamm, "HTTP fences: Immigration control for web pages," 2008 Technical Report.

[6] T. Oda, G. Wurster, P. V. Oorschot, and A. Somayaji, "SOMA: Mutual approval for included content in web pages," in *ACM Conference on Computer and Communications Security*, 2008.

[7] T. Oda and A. Somayaji, "Enhancing Web Page Security with Security Style Sheets," 2011 Technical Report.

[8] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing (SAC '06)*, 2006.

[9] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *International Conference on World Wide Web (WWW)*, 2010.

[10] A. Barth, C. Jackson, and J. C. Mitchell, "Securing browser frame communication," in *Proceedings of the 17th USENIX Security Symposium*, 2008.

[11] G. Heyes, "HTML scriptless attacks," http://www.thespanner.co.uk/2011/12/21/html-scriptless-attacks/.

[12] "Sla.ckers XSS forum," http://sla.ckers.org/forum/list.php?2/.

[13] "Attributes common to form controls," http://dev.w3.org/html5/spec/attributes-common-to-form-controls.html#attr-fs-formaction.

[14] L.-S. Huang, Z. Weinberg, C. Evans, and C. Jackson, "Protecting browsers from cross-origin CSS attacks," in *ACM Conference on Computer and Communications Security (CCS))*, 2010.

[15] A. Barth and T. Steele, "Security in depth: The password manager," http://blog.chromium.org/2008/12/security-in-depth-password-manager.html.

[16] C. Henry, "CAPTCHAs' effect on conversion rates," http://www.seomoz.org/blog/captchas-affect-on-conversion-rates.

[17] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *International Conference on World Wide Web (WWW)*, 2007.

[18] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: get the security of multiple browsers with just one," in *CCS '11: Proceedings of the 18th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2011, pp. 227–238.

[19] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side XSS filters," in *WWW '10: Proceedings of the 19th international conference on World Wide Web*. New York, NY, USA: ACM, 2010, pp. 91–100.

[20] P. Saxena, D. Molnar, and B. Livshits, "SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications," in *CCS '11: Proceedings of the 18th ACM conference on Computer and Communications security*. New York, NY, USA: ACM, 2011, pp. 601–614.

[21] M. Samuel, P. Saxena, and D. Song, "Context-sensitive auto-sanitization in web templating languages using type qualifiers," in *CCS '11: Proceedings of the 18th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2011, pp. 587–600.

[22] "The iframe element," http://dev.w3.org/html5/spec/the-iframe-element.html#attr-iframe-sandbox.