

Client-Side Detection of XSS Worms by Monitoring Payload Propagation

Fangqi Sun, Liang Xu, and Zhendong Su

Department of Computer Science
University of California, Davis
{sunf, xu, su}@cs.ucdavis.edu

Abstract. Cross-site scripting (XSS) vulnerabilities make it possible for worms to spread quickly to a broad range of users on popular Web sites. To date, the detection of XSS worms has been largely unexplored. This paper proposes the first purely client-side solution to detect XSS worms. Our insight is that an XSS worm must spread from one user to another by reconstructing and propagating its payload. Our approach prevents the propagation of XSS worms by monitoring outgoing requests that send self-replicating payloads. We intercept all HTTP requests on the client side and compare them with currently embedded scripts. We have implemented a cross-platform Firefox extension that is able to detect all existing self-replicating XSS worms that propagate on the client side. Our test results show that it incurs low performance overhead and reports no false positives when tested on popular Web sites.

Keywords: cross-site scripting worm, client-side detection, Web application security.

1 Introduction

Web applications have drawn the attention of attackers due to their ubiquity and the fact that they regulate access to sensitive user information. To provide users with a better browsing experience, a number of interactive Web applications take advantage of the JavaScript language. The support for JavaScript, however, provides a fertile ground for XSS attacks. According to a recent report from OWASP [22], XSS vulnerabilities are the most prevalent vulnerabilities in Web applications. They allow attackers to easily bypass the Same Origin Policy (SOP) [19] to steal victims' private information or act on behalf of the victims.

XSS vulnerabilities exist because of inappropriately validated user inputs. Mitigating all possible XSS attacks is infeasible due to the size and complexity of modern Web applications and the various ways that browsers invoke their JavaScript engines. Generally speaking, there are two types of XSS vulnerabilities. *Non-persistent XSS vulnerabilities*, also known as reflected XSS vulnerabilities, exist when user-provided data are dynamically included in pages immediately generated by Web servers; *persistent XSS vulnerabilities*, also referred to as stored XSS vulnerabilities, exist when insufficiently validated user inputs are persistently stored on the server side and later displayed in dynamically generated Web pages for others to read. Persistent XSS vulnerabilities

allow more powerful attacks than non-persistent XSS vulnerabilities as attackers do not need to trick users into clicking specially crafted links. The emergence of *XSS worms* worsens this situation since XSS worms can raise the influence level of persistent XSS attacks in community-driven Web applications. XSS worms are special cases of XSS attacks in that they replicate themselves to propagate, just like traditional worms do. Different from traditional XSS attacks, XSS worms can collect sensitive information from a greater number of users within a shorter period of time because of their self-propagating nature.

The threats that come from XSS worms are on the rise as attackers are switching their attention to major Web sites, especially social networking sites, to attack a broad user base [25]. Connections among different users within Web applications provide channels for worm propagation. In community-driven Web applications, XSS worms tend to spread rapidly — sometimes exponentially. For example, the first well-known XSS worm, the Samy worm [13], affected more than one million MySpace users in less than 20 hours in October 2005. MySpace, which had over 32 million users at that time, was forced to shut down to stop the worm from further propagation. In April 2009, during the outbreak of the StalkDaily XSS worm which hit twitter.com, users became infected when they simply viewed the infected profiles of other users. We show a list of XSS worms in Table 1 (Section 4.1). Common playgrounds for XSS worms include social networking sites, forums, blogs, and Web-based email services.

At present, although much research has been done to detect either traditional worms or XSS *vulnerabilities*, little research has been done to detect XSS *worms*. This is because XSS worms usually contain site-specific code which evades input filters. XSS worms can stealthily infect user accounts by sending asynchronous HTTP requests on behalf of users using the Asynchronous JavaScript and XML (AJAX) technology. Spectator [17] is the first JavaScript worm detection solution. It works by monitoring worm propagation traffic between browsers and a specific Web application. However, it can only detect JavaScript worms that have propagated far enough and is unable to stop an XSS worm in its initial stage. In addition, Spectator requires server cooperation and is not easily deployable.

In this paper, we present the first purely client-side solution to detect the propagation of self-replicating XSS worms. Clients are protected against XSS worms on all Web applications. We detect worm payload propagation on the client side by performing a string-based similarity calculation. We compare outgoing requests with scripts that are embedded in the currently loaded Web page. Our approach is similar in spirit to traditional worm detection techniques that are based on payload propagation monitoring [27, 28]; we have developed the first effective client-side solution to detect XSS *worms*.

This paper makes the following contributions:

- We propose the first client-side solution to detect XSS worms. Our approach is able to detect self-replicating XSS worms in a timely manner, at the very early stage of their propagation.
- We have developed a cross-platform Firefox extension that is able to detect all existing XSS worms that propagate on the client side.
- We evaluated our extension on the top 100 most visited Web sites in the United States [3]. Our results demonstrate that our extension produces no false positives and incurs low performance overhead, making it practical for everyday use.

The rest of the paper is organized as follows. Section 2 describes our worm detection approach in detail. Section 3 presents the implementation details of our Firefox extension. Section 4 evaluates the effectiveness of our approach and measures the performance overhead of our Firefox extension on popular Web sites. Finally, we survey related work (Section 5) and conclude (Section 6).

2 Our Approach

Section 2.1 describes the background of how an XSS worm usually propagates. Section 2.2 gives an overview of how we detect the propagation behavior of an XSS worm. We describe the details of our detection routine in Section 2.3.

2.1 Background of XSS Worm Propagation

The Document Object Model (DOM) is a cross-platform and language-independent interface for valid HTML and well-formed XML documents [26]. It closely resembles the tree-like logical structure of the document it models. Every node in a DOM tree represents an object in the corresponding document. With the DOM, programs and scripts can dynamically access and update the content, structure and style of documents.

In practice, building a flawless Web application is an extremely difficult task due to the challenges of sufficiently sanitizing user-supplied data. Site-specific XSS vulnerabilities become a growing concern as attackers discover that they can compromise more users by exploiting a single vulnerability within a popular Web site than by compromising numerous small Web sites [25]. Exploiting persistent XSS vulnerabilities, XSS worms are normally stored on the servers of vulnerable Web applications. The typical infection process of an XSS worm is as follows:

1. A user, Alice, is lured into viewing a malicious Web page that is dynamically generated by a compromised Web application. The Web page, for example, can be the profile page of Alice's trusted friend.
2. The XSS worm payload, which is embedded in the dynamically generated Web page, is interpreted by a JavaScript engine on the client side. During this interpretation process, an XSS worm usually replicates its payload and injects the replicated payload into an outgoing HTTP request.
3. The crafted malicious HTTP request is sent to the Web application server on Alice's behalf. By exploiting the server's trust in Alice, the XSS worm also compromises Alice's account. Later on, when Alice's friends visit her profile, their accounts will also be infected.

2.2 High-Level Overview

Our goal is to detect the self-replicating characteristics of XSS worms with no modification to existing Web applications or browser architecture. To this end, we compare outgoing HTTP requests with scripts in the currently loaded DOM tree. Figure 1 shows the architecture of our client-side XSS worm detection mechanism. Our solution captures the essential self-propagating characteristics of XSS worms and protects users

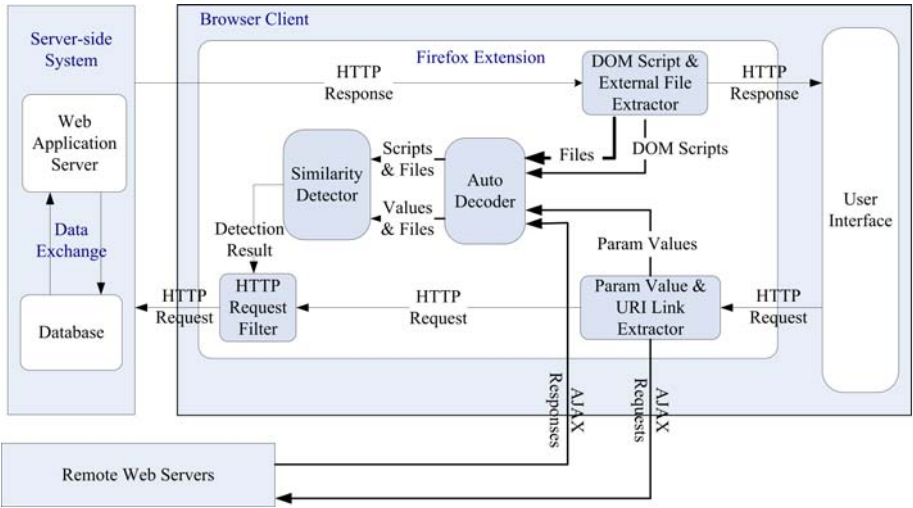


Fig. 1. Client architecture for XSS worm detection

from infection. We choose to detect XSS worms on the client side because the propagation process of an XSS worm is normally triggered during the script interpretation process on the client side. Moreover, a client-side solution is easily deployable.

Scripts can be directly embedded in Web pages or dynamically loaded from remote servers. Therefore, it is necessary to examine all external JavaScript files that are pointed to by Uniform Resource Identifier (URI) links in both outgoing HTTP requests and loaded DOM trees. The steps we take to detect an XSS worm are as follows:

1. We intercept each outgoing HTTP request that may contain the payload of an XSS worm. We extract parameter values from each intercepted request. From these parameter values, we then extract URI links which may point to malicious JavaScript files.
2. If there exist embedded URI links, we send asynchronous requests to retrieve external JavaScript files according to the extracted URI links. We do not begin our decoding process (Step 4) until we receive all responses or signals of timeout events from remote servers. We call the set of parameter values and retrieved external files set \mathcal{P} .
3. We extract scripts from the DOM tree of the current Web page. Next, we retrieve external JavaScript files, which are dynamically loaded into the current Web page, from cached HTTP responses. We call the set of extracted scripts and cached external files set \mathcal{D} .
4. We apply an automatic decoder on code from both set \mathcal{P} and set \mathcal{D} . We repeat this decoding process until we find no encoded text.
5. Finally, we use a similarity detector to compare decoded code from set \mathcal{P} with decoded code from set \mathcal{D} in search of similar code, which indicates the potential propagation behavior of an XSS worm. If we detect suspiciously similar code, we redirect the malicious HTTP request and alert the user.

2.3 Approach Details

This section describes the details of our XSS worm detection algorithm: how we extract parameter values and URI links; possible locations where scripts might appear in Web pages; our decoder, which can handle a number of encoding schemes; and the string similarity detection algorithm that we use.

Parameter Values and URI Links from HTTP Requests. The payload of an XSS worm could be sent in the form of plaintext or as a URI link pointing to an external file stored on a remote server. In either case, the plaintext or the URI link needs to be embedded in the parameter values of an outgoing HTTP request in order to propagate. The extracted parameter values and retrieved external files compose set \mathcal{P} .

As it is impossible to tell whether an HTTP request is sent by an XSS worm or a legitimate user, we intercept and process each outgoing HTTP request. We first extract parameter values, if there are any, from the `path` property of the requested URI. We then examine the request method of the outgoing HTTP request. If the `POST` method is used, we retrieve the request body and then extract additional parameter values from it.

Since attackers may store XSS worm payloads on remote servers and propagate URI links instead of plaintext, we extract URI links from parameter values of HTTP requests using JavaScript regular expressions. We send requests to retrieve external files according to the URI links.

DOM Scripts and External Files from Web pages. To enumerate possible locations where scripts may exist, we studied the source code of several XSS worms in the wild, the XSS Cheat Sheet [11], and some other documentation [26, 29]. We classify possible locations where scripts may reside into the following categories:

- *script elements.* A script can be defined within the `script` element of a DOM tree.
- *Event handlers.* W3C specifies eighteen intrinsic event handlers [26]. In addition, some browsers have implemented browser-specific event handlers.
- *HTML attributes.* Attackers sometimes exploit the attributes of standard DOM elements to dynamically load external files into a document.
- *Scripts specified by browser-specific attributes or tags.* Some browsers implement browser-specific attributes and tags.
- *javascript: URIs.* By declaring the JavaScript protocol, JavaScript code can be put into a place where a URI link is expected.

Based on the possible locations discussed above, we extract scripts directly embedded in the currently loaded DOM tree and retrieve cached external files pointed to by the attributes of DOM elements. Extracted scripts and cached external files compose set \mathcal{D} . If an XSS worm exists, its payload should be embedded in at least one of these locations in order to trigger script interpretation.

Automatic Decoder. Taking into consideration that the payload of an XSS worm may be encoded, we perform a decoding process before carrying out our similarity detection process. We decode all extracted parameter values, retrieved external JavaScript files,

extracted DOM scripts, and cached external JavaScript files. In order to automate the decoding process, we use a regular expression for each encoding scheme.

It is possible that a JavaScript obfuscator applies multiple layers of encoding routines. To handle such situations, we keep track of the total match count for all regular expressions in each decoding routine. If the total match count in a decoding routine reaches zero, it means that no encoded text is found. For this case, we stop our decoding routine; otherwise, we repeat the decoding routine.

Similarity Detection. We detect both suspicious URI links and similar strings.

An XSS worm may propagate by sending a URI link pointing to itself instead of directly sending its payload as plaintext. Therefore, before comparing the elements from set \mathcal{P} with the elements from set \mathcal{D} , we compare the URI links extracted from the parameter values of an outgoing HTTP request with the URI links embedded in the current DOM tree. If a match is found, we immediately redirect the current request and alert the user of the suspicious URI link; otherwise, taking into account the possibility that attackers may use different URI links for the same payload, we examine the contents of external files. Although we have not seen such attacks in real-world XSS worms, we conservatively examine file contents. We do not begin our URI detection process until the decoding process completes.

Once we have all the decoded code, we perform a similarity detection routine in search of a possible XSS worm payload. We use a string similarity detection algorithm based on trigrams [2] for its robustness in dealing with some JavaScript obfuscation techniques such as code shuffling and code nesting.

In our implementation, we use character-level trigrams, which are three character substrings of given strings, to detect the similarity between two strings. Note that a trigram is a special case of an n -gram where $n = 3$. We introduce formal definitions of the trigram algorithm as follows.

Definition 1. $\mathcal{T}(s)$ denotes the set of character-level trigrams of string s .

Definition 2. $\mathcal{S}(p, d)$ denotes the similarity between two strings p and d , where $p \in \mathcal{P}$, $d \in \mathcal{D}$, and both \mathcal{P} and \mathcal{D} are sets of strings.

We compute the similarity of p and d in the following way:

$$\mathcal{S}(p, d) = \frac{|\mathcal{T}(p) \cap \mathcal{T}(d)|}{|\mathcal{T}(p) \cup \mathcal{T}(d)|} \quad (1)$$

In our current settings, \mathcal{P} is the set of parameter values and contents of retrieved external files, and \mathcal{D} is the set of scripts extracted from the DOM tree and contents of cached external files. $\mathcal{S}(p, d)$ has a value between 0 and 1, inclusive.

To make our implementation scalable, we sort the elements in each set into ascending order before calculating their union or intersection. The average-case time complexity of the trigram algorithm is determined by the time complexity of the sorting algorithm, which is $\mathcal{O}(n \log n)$.

Definition 3. If $\exists p \in \mathcal{P}$, $d \in \mathcal{D}$ such that $\mathcal{S}(p, d)$ **exceeds** a customized threshold t , we say that an outgoing HTTP request may contain an XSS worm payload because it exhibits self-propagating behavior.

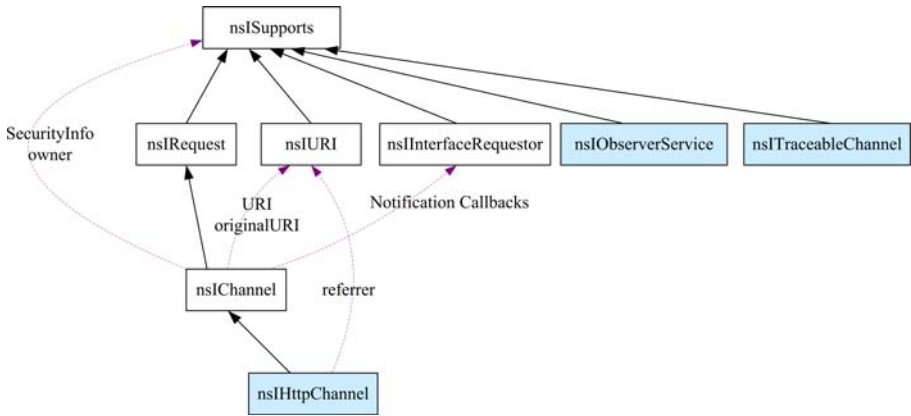


Fig. 2. The interface collaboration diagram

3 Implementation

We have developed a standard cross-platform Firefox 3.0 extension to detect XSS worms. Most Firefox extensions are written in JavaScript because the bindings between JavaScript and XPCOM are strong and well-defined. We wrote most components of our extension in JavaScript, except that we implemented the trigram algorithm using C++ for its better execution efficiency over JavaScript.

We show key user interfaces that we have used in our extension in Figure 2. This diagram is taken from Mozilla Cross-Reference to show the collaboration among the interfaces.

nsIObserverService. To monitor parameter values in each HTTP request, we first get a service from the XPCOM component `observer-service` through the interface `nsIObserverService`. We then register an `http-on-modify-request` observer and an `http-on-examine-response` observer in our extension with the observer service we just obtained.

nsIHttpChannel. Through an `nsIHttpChannel` object, we can obtain an `nsIURI` object to read its `asciiSpec` property for an ASCII representation of the requested URI. To get the body of a POST request from an `nsIHttpChannel` object, we gain access to the post data by obtaining a pointer to the `nsIUploadChannel` interface, and then rewind the stream of the post data with a pointer to the `nsISeekableStream` interface. If an XSS worm is detected, we obtain a pointer to the `nsIRequest` interface, and then call the `cancel` method in that interface to cancel the malicious HTTP request.

nsITraceableChannel. The `nsITraceableChannel` interface enables us to directly retrieve external files from cached HTTP responses rather than sending asynchronous requests and waiting for their responses. This interface was recently introduced in Firefox 3.0.4, which was released in November 2008. Through this interface, we can replace a channel's original listener with a new one, and collect all the data we need by

intercepting `OnDataAvailable` calls. We examine the Multipurpose Internet Mail Extensions (MIME) types of all HTTP responses to determine which responses might trigger script interpretation.

4 Empirical Evaluation

This section shows the effectiveness of our approach on real-world XSS worms, as well as obfuscated JavaScript code produced by some common JavaScript obfuscators. We then present the performance overhead results, reason about parameter settings, and discuss potential threats to the validity of our approach.

4.1 Real-World XSS Worms

The XSS worms examined in this section are all XSS worms released on popular real-world Web applications. All of these worms exploited persistent XSS vulnerabilities in different Web applications.

The Samy Worm. Based on the source code of the Samy worm [13], we wrote and tested an XSS worm on a small-scale Web application that we constructed. We named our Web application `SamySpace`. `SamySpace` mimics the necessary functionality of `MySpace` to allow the propagation of an XSS worm. We stored each user's profile in a backend `MySQL` database. We modified the original Samy worm code [13] as little as possible to make it work on `SamySpace`. As in the Samy worm, our worm sends five AJAX requests in total.

The original Samy worm is only slightly obfuscated using short variable names and few newline characters to fit itself into the limited space of the `interest` field in a user's profile. Our implementation is able to detect the XSS worm released on `SamySpace` by observing a high similarity of 91% between a parameter value sent in a request and a script extracted from the DOM tree.

The Orkut Worm. Different from the Samy worm, the Orkut worm is a heavily obfuscated XSS worm. During its outbreak, a user's account on Orkut was infected when the user simply read a `scrap` sent by the user's infected friend. The Orkut worm payload is contained in an external JavaScript file named `virus.js`, the URI of which is included in an `<embed>` element. The `<embed>` element is injected into the value of a parameter, which is used to store the message body of a `scrap`.

The Orkut worm works in three steps. To begin with, the worm payload embedded in `virus.js` reconstructs itself. It then propagates the worm payload to everyone present in the victim's friend list. Finally, it sends an asynchronous request to add the victim to a community, which tracks the total number of infected users, without the user's approval.

The parameter values sent by the Orkut worm include a malicious URI link. Since the URI link contained in the outgoing request is also embedded in the DOM, we are able to detect the propagation behavior of this XSS worm.

Table 1. Statistics of XSS worms in the wild

XSS worm	Propagation method	Triggering method	Payload location	Release date
Samy Worm	XHR	javascript: URIs	DOM	Oct. 2005
Xanga	XHR	javascript: URIs	DOM	Dec. 2005
Yamanner	XHR	onload event handler	server	Jun. 2006
SpaceFlash	XHR	javascript: URIs	URI link	Jul. 2006
MyYearBook	form submission	innerHTML	DOM	Jul. 2006
Gaia	XHR	src attribute	URI link	Jan. 2007
U-Dominion	XHR	src attribute	URI link	Jan. 2007
Orkut	XHR	src attribute	URI link	Dec. 2007
Hi5	form submission	-moz-binding / expression	URI link	Dec. 2007
Justin.tv	XHR	src attribute	URI link	Jun. 2008
Twitter	XHR	src attribute / expression	URI link	Apr. 2009

XSS Worms In the Wild. We carefully examined the available source code of XSS worms in the wild. Table 1 lists eleven of them. The first column of the table shows the names of XSS worms. We use the names of the infected Web sites to represent released worms when there is no ambiguity. The second column shows the propagation methods that were used. XMLHttpRequest (XHR) denotes asynchronous requests sent in the background, while form submission denotes HTTP requests which are sent when HTML forms are submitted or links are clicked. The third column of the table shows the triggering methods of worm payloads. The `expression` function, which takes a piece of JavaScript code as its parameter, is supported in both Internet Explorer and Netscape; the `-moz-binding` attribute, which binds JavaScript code to a DOM element, is supported in both the Firefox and the Netscape browsers. The fourth column shows where worm payloads were extracted for payload reconstruction. Finally, the last column shows the release dates.

Yamanner was released on Yahoo!Mail; the Xanga worm was released on a blog; the Gaia worm and the U-Dominion worm were released on gaming Web sites; the Justin.tv worm was released on a video hosting Web site; and all of the six remaining worms were released on social networking Web sites. MySpace, a popular social networking Web site, is one of the favorite targets of XSS worms. Both the Samy worm and the SpaceFlash worm were released on it.

With regard to worm propagation methods, most attackers chose to use XHR for its advantage over form submission: asynchronous requests sent quietly in the background often go unnoticed. There are various kinds of XSS vulnerabilities, and so are script triggering methods. Most methods work on major Web browsers, except `-moz-binding` and `expression`, which are browser specific.

For XSS worms that propagate by reconstructing their payload from the loaded DOM tree, our approach is able to detect high similarity between parameter values of outgoing HTTP requests and DOM scripts; for XSS worms which propagate by sending links to external files, our approach is able to detect the existence of identical URI links that appear in both parameter values of outgoing requests and HTML attributes of the current DOM tree.

The Yamanner worm is a special case of XSS worms because its propagation process is actually performed on the server side rather than on the client side. Back in 2005 when the Yamanner worm was unleashed, the Yahoo!Mail system provided two ways to forward an email: either as inline text or as an attachment. For forwarded email with inline text, the message body of the original email was embedded in the message body of the forwarded email. For forwarded email with an attachment, only the message ID of the original email was embedded in the forwarded email; the message body of the original email was later retrieved when the attachment was opened or downloaded. The Yamanner worm used the attachment method to forward malicious emails; therefore, outgoing requests sent by Yamanner to forward emails only contained the message IDs of original emails. To obtain the message body with the actual worm payload on the client side, we would need to write application-specific code to retrieve message bodies from the application server.

4.2 Effectiveness of Our Approach for Obfuscated JavaScript Code

When JavaScript code obfuscation was introduced a few years ago, it was mainly used to provide control over intellectual property theft. With the advent of XSS worms, we see the increasing abuse of legitimate JavaScript obfuscators by malware authors. We have seen an online XSS worm tutorial suggesting people obfuscate their code using a legitimate JavaScript packer [8]. We believe unsophisticated malware authors normally would not take the effort to write their own JavaScript obfuscators. When we first examined the obfuscated source code of the Orkut worm, we noticed that it used an unusual decoding function which has six parameters: “ (p, a, c, k, e, d) ”. After some research, it turned out that the obfuscated code was generated by a legitimate and publicly available JavaScript packer written by Dean Edwards [8]. The decoding function acts as a signature function of his JavaScript packer.

The purpose of using JavaScript obfuscators is to turn JavaScript source code into functionally equivalent JavaScript code that is more difficult to study, analyze and modify. Eventually any obfuscated JavaScript code must be read and correctly interpreted by a JavaScript engine. Common JavaScript obfuscation techniques include the following:

- Code shuffling and code nesting.
- String manipulations such as string reversion, split and concatenation.
- Character encoding.
- Insertions or deletions of arbitrary comments, spaces, tabs, and newline characters.
- Variable renaming and randomized function names.
- The use of encryption and decryption.

With the combination of our automatic decoder and trigram algorithm, our approach is robust in dealing with the first four obfuscation techniques. For example, the code shuffling technique has no impact on our string-based similarity detection process. We tested our approach on widely used JavaScript obfuscators that we have collected. We used each obfuscator to obfuscate five real-world XSS worms, namely the Samy, Orkut, Yamanner, Hi5, and Justin.tv worms. We computed the similarity between original source code and obfuscated code, and calculated the average similarity for each JavaScript obfuscator. We show the results in Figure 3.

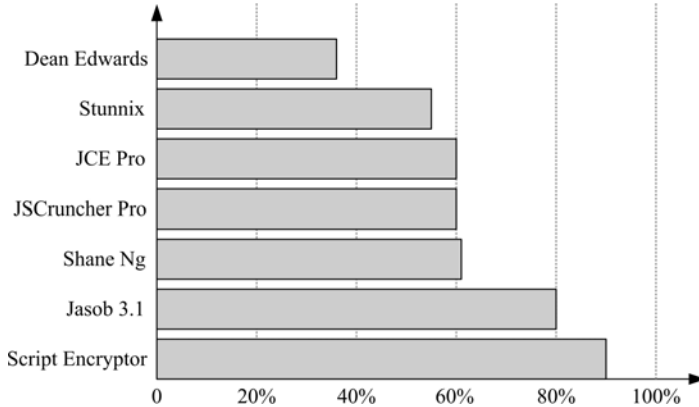


Fig. 3. Similarity results for common JavaScript obfuscators

Based on the real-world XSS worms and obfuscated JavaScript code that we have collected, we set the string similarity threshold for the trigram algorithm to 20%. As can be seen in Figure 3, the similarity results for these obfuscators all exceed the conservative 20% threshold. Although the similarity result for Dean Edwards’s JavaScript packer is relatively low, code obfuscated by this tool can be easily discerned by its signature function. An alternative is to simply replace the `eval` function with a `print` equivalent function for the complete exposure of the original worm payload. We observed that Stunnix, a commercial JavaScript obfuscator, applied multiple encoding routines using different character encoding schemes. Thanks to our automatic decoder, our extension was still able to detect a high similarity between the original and obfuscated code generated by Stunnix.

4.3 Overhead Measurements

To estimate the performance overhead imposed by our extension, we visited the top 100 most popular United States Web sites ranked by Alexa [3]. We measured the page load time for the top 100 Web sites with Firebug [9], an open source Firefox Extension for Web development. To eliminate the impact of cache on measured time, we disabled browser cache by setting both `browser.cache.disk.enable` and `browser.cache.memory.enable` to `false` in our browser configuration. We first visited each Web page five times with our extension disabled, and then visited the same page another five times with our extension enabled. Due to space limitation, we show in Table 2 the performance overhead imposed by our extension for the top 20 Web sites. Column 1 and 5 show the names of the Web sites; column 2 and 6 show the average page load time in seconds without our extension; column 3 and 7 show the average page load time in seconds with our extension enabled; column 4 and 8 show the performance overhead for each Web site tested.

For all the 100 Web sites we visited, the number of requests a page load generates ranged from 3 to 390. All generated HTTP requests were monitored by our Firefox extension. The average number of requests a Web page generates on a page load event

Table 2. Performance overhead for top 20 Web sites

Web site	w/o avg(s)	w/ avg(s)	overhead average	Web site	w/o avg(s)	w/ avg(s)	overhead average
Google	0.240	0.243	1.17%	AOL	3.796	3.878	2.16%
Yahoo!	0.724	0.738	1.96%	Blogger	0.745	0.762	2.34%
Facebook	0.662	0.683	3.14%	Amazon	2.616	2.696	3.06%
YouTube	1.738	1.784	2.65%	Go	3.574	3.614	1.12%
MySpace	0.914	0.948	3.72%	CNN	6.450	6.746	4.59%
MSN	1.192	1.214	1.85%	Microsoft	1.756	1.782	1.48%
Windows Live	0.361	0.371	2.66%	Flickr	0.618	0.641	3.76%
Wikipedia	1.308	1.336	2.14%	ESPN	2.694	2.786	3.41%
Craigslist.org	0.283	0.285	0.78%	Photobucket	1.274	1.308	2.67%
eBay	0.712	0.722	1.41%	Twitter	1.098	1.118	1.82%

was 89; the average content length of corresponding HTTP responses was 666KB. Our extension increased page load time by 2.62% on average.

The primary overhead of running our extension is due to the latency introduced by the decoding and similarity detection processes. Web sites that sent more outgoing HTTP requests with a large number of parameters incurred higher performance overhead than other Web sites. Among the listed top 20 Web sites, CNN sent out the largest number of requests, 249 requests in total, for a page load event; our extension incurred the highest overhead on CNN. Most Web sites ran smoothly with our extension enabled. Of all the Web pages tested, `Craigslist.org` generated the fewest number of requests; we observed only minor performance overhead for `Craigslist.org`. Overall, the overhead incurred by our extension is reasonably low for everyday use.

4.4 Parameter Settings

Minimum URI link length. To avoid unnecessary computation and increase the efficiency of our algorithm, we conservatively set the minimum URI link length to 12. Protocol declarations, such as `http://` which has seven characters, are also counted in URI links. If the length of a parameter value is of length less than 12, we skip that parameter value all together without searching for URI links in it.

Minimum length of XSS worm payloads. We conservatively set the minimum length of an XSS worm payload p to 155 characters according to the result of an XSS worm replication contest [1]. This contest aimed to find the smallest amount of code necessary for XSS worm propagation. Having to deal with application-specific requirements, real-world XSS worms should have larger code base than the winners of this contest. In order for an XSS worm to propagate itself, it needs to at least obtain, reconstruct and embed its payload, and send an HTTP request. The smallest real-world XSS worm we have seen so far, the MyYearBook worm, is composed of 769 characters after a normalization process. If any element in set \mathcal{P} or set \mathcal{D} is of length less than p , we remove it from the set.

4.5 Discussion

Of all the Web sites that we have tested, most of them exhibit normal behavior. Only a few Web sites propagate advertisement URI links with long query strings similarly to the way that XSS worms propagate their payloads. However, files pointed to by these URI links are not dynamically loaded into the DOM tree and thus do not pose any threats. To limit the impact of these URI links on the similarity detection process, we used a URI query string filter to remove such query strings. We tested our extension on the top 100 Web sites and enabled our extension in our everyday browsing for two weeks. With the current settings, we have not observed any false positives.

Some Web sites include identical style sheets or JavaScript library modules in multiple Web pages. However, such Web sites do not raise false alarms because these remote files are only loaded in the Web pages, but not propagated through outgoing requests.

As mentioned in Section 4.1, our approach does not deal with server-side self-replicating worms. This is because worms that propagate on the server side store their payloads directly on the Web application server rather than in parameter values or external files. In such cases, triggering the worm payload requires additional user actions rather than simply viewing Web pages. For this reason, server-side XSS worms are relatively rare in the wild. To detect such XSS worms, server-side coordination is necessary.

We have shown that our solution is effective with regard to popular JavaScript obfuscators. However, determined attackers might be willing to take the effort to create their own obfuscation techniques and write their own encryption and decryption routines to create highly obfuscated worms. For advanced obfuscation techniques, we expect behavior-based approaches to be more effective.

5 Related Work

We survey closely related work in this section.

5.1 Worm Detection

Researchers have proposed signature-based approaches to detect polymorphic worms. The key idea for signature generation is to find invariant substrings [15] or structural similarities in all variants of a payload. Such techniques can also be applied to highly obfuscated XSS worms.

Techniques for the detection of traditional Internet worms include content filtering [6], network packet analysis [7], honeypots, and worm propagation behavior analysis. Wang *et al.* [27, 28] proposed a similarity-based approach using an n-gram based detection algorithm.

Spectator [17] also detects JavaScript worms by identifying the propagation behavior of JavaScript worms. In particular, it tags the traffic between browsers and Web applications, and sets a threshold to detect worm-like long propagation chains. Although it is effective in detecting JavaScript worms, their approach can only detect JavaScript worms that have propagated far enough. Different from their approach, we can detect XSS worms in a timely manner.

5.2 Client-Side Protection

Several client-side approaches have been proposed to address XSS vulnerabilities. Most of them are not purely client-side solutions and require server-side cooperation.

Client-side policy enforcement mechanisms aim to enforce security policies provided by Web application developers to make sure that browsers interpret Web pages in expected ways. BEEP [12] provides two kinds of policies: a whitelist policy for trusted scripts and a blacklist policy for DOM sandboxing. Similarly, Noncespaces [10] use whitelist and ancestry-based sandbox policies along with the Instruction Set Randomization technique to constrain the capabilities of untrusted content.

To prevent injection attacks, several approaches [24, 18, 20] in the literature rely on the preservation of intended parsing behavior. BLUEPRINT [18] seeks to minimize the trust placed on browsers for interpreting untrusted content by enabling a web application to effectively take control of parsing decisions. DSI [20] ensures the structural integrity of HTML documents.

Noxes [14] is the first purely client-side solution to mitigate XSS attacks. It works as a personal Web firewall that helps mitigate XSS attacks with both manually and automatically generated rules to protect against information leakage. However, as most of the above approaches, it aims to detect XSS attacks but may not work in the face of XSS worms, which exploit the trust between users and cause damage without accessing sensitive user information.

5.3 Server-Side Analysis

Previous server-side techniques mostly address Web application vulnerabilities using information flow analysis. Static analysis [29, 30] aims to detect cross-site scripting vulnerabilities before the deployment of Web applications, while dynamic analysis [21, 5, 16] aims to provide detailed information of vulnerabilities and exploits at run time. Saner [4] uses both static analysis and dynamic analysis to analyze custom sanitization processes. Sekar [23] recently proposed a black-box taint-inference technique that works by observing inputs and outputs of a Web application.

The challenge of applying server-side analysis lies in the difficulty of finding all XSS vulnerabilities in Web applications. Our approach, in comparison, seeks to detect XSS worm payloads rather than to find all XSS vulnerabilities within a Web application.

6 Conclusions

This paper presents the first purely client-side solution to effectively detect XSS worms by observing the propagation of worm payloads. The main idea of our approach is to identify similar strings between the set of parameter values in outgoing HTTP requests and retrieved external files, and the set of DOM scripts and loaded external files. We implemented our approach as a cross-platform Mozilla Firefox extension. We evaluated its effectiveness on some real-world XSS worms and its resilience against some common JavaScript obfuscators. Finally we measured the performance overhead incurred by our Firefox extension on the top 100 U.S. Web sites. Our empirical results show that our extension is effective in detecting self-replicating XSS worms on the client side with

reasonably low performance overhead. Because our approach is general and effective, it can be applied to other browsers besides Firefox to detect self-replicating XSS worms on the client side.

References

- [1] Diminutive XSS worm replication contest (2008)
<http://sla.ckers.org/forum/read.php?2,18790,page=19>
- [2] Ahmed, T.: The trigram algorithm,
<http://search.cpan.org/dist/String-Trigram/Trigram.pm>
- [3] Alexa. Top sites in United States, <http://www.alexa.com/topsites>
- [4] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 387–401. IEEE Computer Society Press, Los Alamitos (2008)
- [5] Chang, W., Streiff, B., Lin, C.: Efficient and extensible security enforcement using dynamic data flow analysis. In: Proceedings of the 15th ACM conference on Computer and communications security, pp. 39–50. ACM Press, New York (2008)
- [6] Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: End-to-End Containment of Internet Worms. In: Proceedings of the Symposium on Systems and Operating Systems Principles, pp. 133–147 (2005)
- [7] Crandall, J.R., Su, Z., Wu, S.F., Chong, F.T.: On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: Proceedings of the 12th ACM conference on Computer and communications security, pp. 235–248. ACM Press, New York (2005)
- [8] Edwards, D.: Dean Edwards Javascript packer,
<http://dean.edwards.name/packer/>
- [9] Firebug, <http://getfirebug.com/>
- [10] Gundy, M.V., Chen, H.: Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (2009)
- [11] Hansen, R.: XSS cheat sheet, <http://ha.ckers.org/xss.html>
- [12] Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with Browser-Enforced Embedded Policies. In: WWW, pp. 601–610 (2007)
- [13] Kamkar, S.: The Samy worm (2005), <http://namb.la/popular/tech.html>
- [14] Kirda, E., Kruegel, C., Vigna, G., Jovanovic, N.: Noxes: A client-side solution for mitigating cross-site scripting attacks. In: SAC, pp. 330–337 (2006)
- [15] Li, Z., Sanghi, M., Chen, Y., Kao, M.-y., Chavez, B.: Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 32–47. IEEE Computer Society Press, Los Alamitos (2006)
- [16] Liang, Z., Sekar, R.: Fast and automated generation of attack signatures: A basis for building self-protecting servers. In: Proceedings of the 12th ACM conference on Computer and communications security (2005)
- [17] Livshits, B., Cui, W.: Spectator: detection and containment of JavaScript worms. In: USENIX 2008 Annual Technical Conference on Annual Technical Conference, pp. 335–348. USENIX Association (2008)
- [18] Louw, M.T., Venkatakrishnan, V.N.: Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In: Proceedings of the 30th IEEE Symposium on Security and Privacy (2009)

- [19] Mozilla Corporation. Same origin policy for JavaScript, https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript
- [20] Nadj, Y., Saxena, P., Song, D.: Document structure integrity: A robust basis for cross-site scripting defense. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (2009)
- [21] Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the 12th Annual Network and Distributed System Security Symposium (2005)
- [22] OWASP, <http://www.owasp.org>
- [23] Sekar, R.: An efficient black-box technique for defeating Web application attacks. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (2009)
- [24] Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: Proceedings of the 33rd Annual Symposium on Principles of Programming Languages, pp. 372–382. ACM Press, New York (2006)
- [25] Symantec Corporation. Symantec Global Internet Security Threat Report, vol. XIII (2008)
- [26] W3C, <http://www.w3.org/>
- [27] Wang, K., Cretu, G., Stolfo, S.J.: Anomalous payload-based worm detection and signature generation. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, pp. 227–246 (2005)
- [28] Wang, K., Parekh, J.J., Stolfo, S.J.: Anagram: A content anomaly detector resistant to mimicry attack. In: Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection, pp. 226–248 (2006)
- [29] Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: Proceedings of the 30th International Conference on Software Engineering, pp. 171–180. ACM Press, New York (2008)
- [30] Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: Proceedings of the 15th conference on USENIX Security Symposium, USENIX Association (2006)