PathCutter: Severing the Self-Propagation Path of XSS JavaScript Worms in Social Web Networks

Yinzhi Cao[§], Vinod Yegneswaran[†], Phillip Porras[†], and Yan Chen[§]
yinzhi.cao@eecs.northwestern.edu, {vinod, porras}@csl.sri.com, ychen@northwestern.edu
[§]Northwestern University, Evanston, IL [†]SRI International, Menlo Park, CA

Abstract

Worms exploiting JavaScript XSS vulnerabilities rampantly infect millions of web pages, while drawing the ire of helpless users. To date, users across all the popular social networks, including Facebook, MySpace, Orkut and Twitter, have been vulnerable to XSS worms. We propose PathCutter as a new approach to severing the self-propagation path of JavaScript worms. PathCutter works by blocking two critical steps in the propagation path of an XSS worm: (i) DOM access to different views at the client side and (ii) unauthorized HTTP request to the server. As a result, although an XSS vulnerability is successfully exercised at the client, the XSS worm is prevented from successfully propagating to the would-be victim's own social network page. PathCutter is effective against all the current forms of XSS worms, including those that exploit traditional XSS, DOM-based XSS, and content sniffing XSS vulnerabilities.

We present and evaluate both a server-side and proxy-side deployment of PathCutter. We implement PathCutter on WordPress and Elgg and demonstrate its resilience against two proof-of-concept attacks. We also evaluate the PathCutter implementation on five real-world worms: Boonana, MySpace Samy, Renren, SpaceFlash, and the Yamanner worm. We show that although the worms themselves exploit different vulnerabilities, at either the client side or server side, they are successfully thwarted by PathCutter as it is vulnerability agnostic and blocks the propagation path of the infection. Our performance evaluation shows that rendering overhead of PathCutter is less than 4%, and memory overhead for one additional view is less than 1%.

1 Introduction

JavaScript is a cornerstone of the modern Internet that enables enhanced user interactivity and dynamicism. It is universally adopted by all modern e-commerce sites, web portals, blogs, and social networks. However, JavaScript code also has a demonstrated penchant for attracting vulnerabilities. JavaScript-based Cross Site Scripting (XSS) worms pose a severe security concern to operators of modern social networks. For example, within just 20 hours in October 4, 2005, the MySpace Samy worm [10] infected more than one million users on the Internet. More recently, similar worms [16–18] have affected major social networks, such as Renren and Facebook, drawing significant attention from the public and media.

JavaScript worms typically exploit XSS vulnerabilities in the form of a traditional XSS, document object model (DOM)-based XSS, or content sniffing XSS vulnerabilities. Incorporated into web applications, Javascript worms can spread themselves across social networks. Although they are referred to as worms, these JavaScript malware activities are more akin to viruses, in that they rely on interactions by users on the social network to replicate themselves. Once a vulnerable user is infected, malicious logic residing on the user's page coerces browsers of other victim visitors to replicate the malicious logic onto their respective pages.

The high degree of connectivity and dynamicism observed in modern social networks enables worms to spread quickly by making unsolicited transformations to millions of pages. While the impact of prior XSS worms has been quite benign, it is conceivable that future worms would have more serious implications as underground economies operated by cybercriminals have become increasingly organized, sophisticated, and lucrative. In [29] Billy Hoffman describes a hypothetical 1929 Worm that uses a self-propagating XSS attack on a brokerage site to wreak havoc on financial markets.

The growing threat of XSS worms has been recognized by the academic community, notably in the following two papers. The Spectator [34] system proposed one of the first methods to defend against JavaScript worms. Its proxy system tracks the propagation graphs of activity on a website and fires an outbreak alarm when propagation chains exceed a certain length. A fundamental limitation of the Spectator approach is that it does not prevent the attack propagation until the worm has infected a large number of users. In con-

trast, Sun et al. [41] propose a purely client-side solution, implemented as a Firefox plug-in, to detect the propagation of the payload of a JavaScript worm. They use a string comparison approach to detect instances where downloaded scripts closely resemble outgoing HTTP requests. However, this approach is vulnerable to simple polymorphic attacks.

In this paper, we propose PathCutter as a complementary approach to XSS worm detection that addresses some of the limitations of existing systems. In particular, PathCutter aims to block the propagation of an XSS worm early and seeks to do so in an exploit agnostic manner. To achieve its objectives, PathCutter proposes two integral mechanisms: *view separation* and *request authentication*. PathCutter works by dividing a web application into different views, and then isolating the different views at the client side. PathCutter separates a page into views if it identifies the page as containing an HTTP request that modifies server content, e.g., a comment or blog post. If the request is from a view that has no right to perform a specific action, the request is denied. To enforce DOM isolation across views within the client, PathCutter encapsulates content inside each view within pseudodomains as shown in Section 4. However, isolation by itself does not provide sufficient protection against all XSS attacks. To further prevent Same Origin Cross Site Request Forgery (SO CSRF) attacks, where one view forges an HTTP request from another view from the same site, PathCutter implements techniques such as per-url session tokens and referrer-based view validation to ensure that requests can be made only by views with the corresponding capability.

The design of PathCutter is flexible enough to be implemented either as a server-side modification or as a proxy application. To evaluate the feasibility of a server-side deployment, we implement PathCutter on two popular social web applications: Elgg and WordPress. We find that only 43 lines of code are required to inject PathCutter protection logic into WordPress and just 25 lines of additional code are required to secure Elgg[1]. We also evaluate a proxy-side implementation of PathCutter. The proxy seamlessly modifies content from popular social networks like Facebook on the fly to provide protection from XSS injection attacks.

Based on published source code and press reports, we analytically investigate PathCutter's efficacy against five real-world JavaScript worms: Boonana [17], Samy [10], Renren [16], SpaceFlash [12], and the Yamanner worm [8]. Together, these worms span diverse social networks and exploit various types of XSS vulnerabilities, including Flash XSS, Java XSS, and traditional XSS. However, they converge in their requirement to send an unauthorized request to the server in order to spread themselves. PathCutter exploits this need to successfully thwart the propagation of all these worms. Finally, we conduct performance evaluations to measure the overhead introduced by our PathCutter implementation at the client side. Our results show the rendering overhead (latency) introduced by PathCutter to be less than 4% and the memory overhead introduced by one additional view to be less than 1%. For highly complex pages, with as many as 45 views, the additional memory overhead introduced by PathCutter is around 30%.

[1]Elgg has built-in support for request authentication but not view separation.

Contributions: Our paper makes the following contributions in defending against XSS JavaScript worms:

- We identify two key design principles (view separation by pseudodomain encapsulation and request authentication) for fortifying web pages from XSS worms.
- We develop prototype implementations of the server-side and proxy-side designs.
- We validate the implementation against five real-world XSS social network worms and experimental worms on WordPress and Elgg.
- We demonstrate that the rendering and memory overhead introduced by PathCutter is acceptable.

The remainder of this paper is organized as follows. In Section 2, we provide the problem definition. In Section 3, we provide a survey of related work. We introduce the PathCutter design and our prototype implementation in Section 4 and Section 5 respectively. An evaluation of the PathCutter methodology and implementation is provided in Section 6. We discuss related and open issues in Section 7. Finally, in Section 8, we summarize our findings and discuss future work.

2 Problem Definition

A cross-site scripting (XSS) attack refers to the exploitation of a web application vulnerability that enables an attacker to inject client-side scripts into web pages owned by other users [15]. To illustrate how PathCutter blocks the propagation of a JavaScript-based XSS worm, we begin by describing the steps involved in the life cycle of a typical XSS worm exploit. Although XSS worms exploit different types of XSS attacks, they all share a need to acquire the victim's privilege (in Step 2) and thus issue an unauthenticated cross-view request (in Step 3), which PathCutter seeks to block.

Step 1 – Enticement and Exploitation: A benign user is tricked into visiting (or stumbles upon) a malicious social network page with embedded worm logic that has been posted by an attacker. The worm is in the form of potentially obfuscated, self-propagating JavaScript, which is injected via an XSS vulnerability.

Step 2 – Privilege Escalation: The malicious JavaScript exploits the XSS vulnerability to gain all the victim's rights and privileges to the website that is currently connected to from within the victim's compromised browser. For example, if the victim is logged into his social network account, the worm has the ability to modify the victim's home page and can send messages to the victim's friends.

Step 3 – Replication: The worm now replicates itself. As shown in Figure 2(a), the JavaScript worm uses its victim's own privileges to send the social network web server a request to change the victim's home page. The victim's home page is now altered to include a copy of the Javascript worm.

Step 4 – Propagation: When other benign users subsequently visit the infected victim's page, Steps 2 and 3 are repeated. Such a strategy has been demonstrated in the wild to support worm epidemics that can grow beyond a million infections.

3 Related Work

3.1 XSS and JavaScript Worm Defense

Researchers have proposed numerous defenses against XSS attacks and JavaScript worms that directly relate to our work. A comparison of our work with closely related work is shown in Table 1. Each individual defense mechanism targets different stages of an XSS worm propagation and it can be deployed at either the client or server. We explore different XSS attack strategies and defenses in more detail below.

Cross-Site Scripting Attacks and Defenses: Cross-site scripting attacks can be broadly classified into two categories, traditional *server-side* XSS attacks (stored and reflected [15]), and *client-side* XSS attacks (DOM-based XSS [4], plug-in-based XSS [6], and content sniffing XSS attacks [21]), as shown in Figure 1.

In a *traditional XSS attack*, clients receive injected scripts from the server. Many techniques have been proposed that operate at the server side to defeat traditional XSS attacks, including [19, 24, 25, 30, 32, 35, 37, 46]. While these systems are quite successful at identifying XSS vulnerabilites, their tracking of information flow is restricted to the server side and is blind to the client-side behavior of browsers and vulnerabilities in browser plug-ins. BEEP [31] and Noxes [33] are the first client-side systems to defend against traditional server-side XSS attacks. Later, recent papers on systems such as Blueprint [44] and DSI [38] discuss browser quirks [14] and propose client-side solutions to traditional XSS attacks. Bates et al. [23] criticize client-side filtering and propose their own solutions. Content Security Policy (CSP) [2], proposed by Mozilla, injects a very fine grained policy that is specified at the server side into HTTP headers. This policy is then adopted by client browsers and enforced during every HTTP request.

In a *DOM-based XSS attack*, clients inject scripts through an unsanitized parameter of dangerous DOM operation, such as $document.write$ and $eval$. A simple example is that of the client-side JavaScript of web application calls $document.write(str)$ where str is part of $window.location$. Therefore, the attacker can inject scripts into parameters of URLs. A few defense mechanisms [40] are proposed for DOM-based XSS. Futhermore, CSP can be used to prohibit the use of dangerous functions such as *eval* and *document.write*, but such policies also limit website functionality. Recently, Barth et al. [21] proposed a new class of XSS attack called *Content Sniffing XSS attacks* where an image or a pdf file may also be interpreted as a JavaScript file by the client browser. Moreover, malicious JavaScript could also be injected by plug-ins. This has led to the proliferation of plug-in-based XSS vectors such as *Flash-based XSS attacks*, as a means to inject scripts into web pages. For example, the Renren worm [16] exploited a Flash vulnerability to enable access to the infected web page vulnerability, and inject malicious JavaScript. To fix the attack, users had to update their Adobe Flash plug-in to prevent such malicious accesses. These defenses all target **Step 1** in the propagation of an XSS worm.

JavaScript Worm Defense Techniques: Sun et al. [41] propose a Firefox plug-in that detects JavaScript worms using payload signatures. Their mitigation mechanism targets **Step 3** (Replication) in the life cycle of an XSS worm. Their approach is limited in that it protects only the specific client and not the entire web application. Furthermore, it is vulnerable to polymorphic worms where the payload dynamically changes during the worm's propagation. As shown by Dabirsiaghi et al. [27], the next generation of JavaScript XSS worms could integrate advanced polymorphic payloads, which may prevent direct payload fingerprinting of worm instances and thereby prolong and widen the epidemic.

Spectator [34] adopts a distributed tainting and tagging approach that detects the spreading behavior of JavaScript worms. The mitigation mechanism, which can be implemented as a proxy, targets Step 4 (Propagation) in Section 2. A deficiency of their approach is that it only detects the worm once a critical mass of users have been infected. Xu et al. [47] propose building a surveillance network that uses decoy nodes to passively monitor the social graph for suspicious activities. Their approach is complementary to ours in that it can detect worms like Koobface, that spread through malicious executables and are delivered through re-

	Spectator [34]	Sun et al. [41]	Xu et al. [47]	BluePrint [44]	Plug-in Patches	Barth et al. [21]	Saxena et al. [40]	PathCutter
Blocking Step	4	3	4	1	1	1	1	2
Polymorphic Worm Prevention	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Early-Stage Prevention	No	Yes	No	Yes	Yes	Yes	Yes	Yes
Types of XSS JavaScript Worms that Can Be Defended	All	All	Passively Observable Worms	Traditional Server-Side XSS Worms	Plug-in XSS Worms	Content Sniffing XSS Worms	DOM-based XSS Worms	All
Deployment	Server or Proxy	Client	Server	Server	Client	Client	Client	Server or Proxy
Passive/Active Monitoring	Active	Passive	Passive	Active	Active	Active	Active	Active

Table 1. Design Space of XSS Worm Defense Techniques

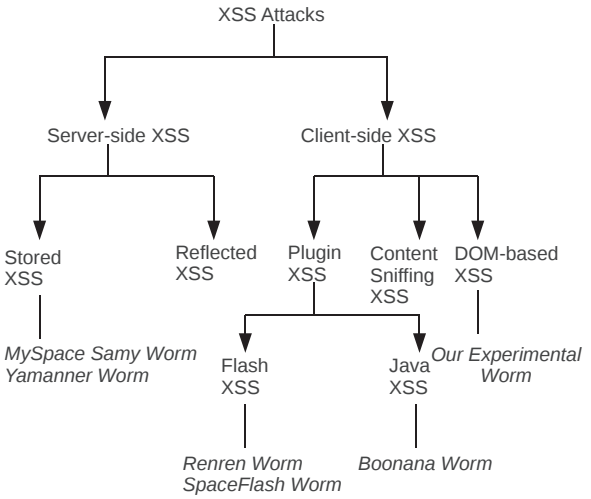

Figure 1. Taxonomy of XSS JavaScript Attacks and Worms

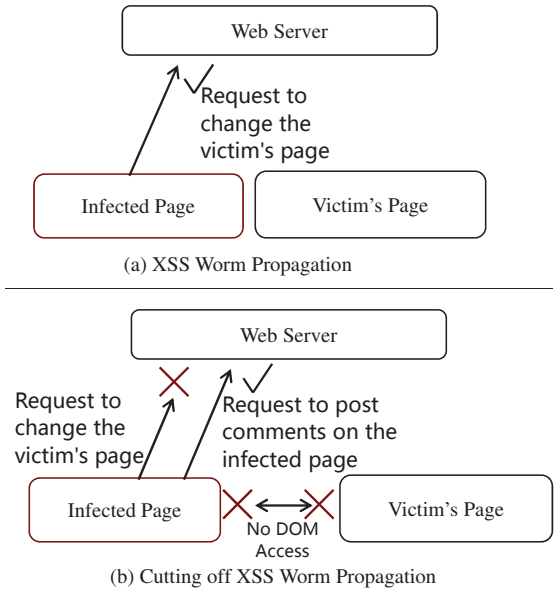

Figure 2. XSS Worm Propagation

mote browser exploits, which PathCutter cannot. In contrast, they acknowledge that their approach cannot detect worms like MySpace Samy because it "does not generate passively noticeable worm activities". Another limitation of their approach is that Xu's decoy nodes, like Spectator, require a minimal threshold of users to be infected before detection. Both of these graph-monitoring systems target **Step 4** in the propagation of an XSS worm.

3.2 Request Authentication and View Separation Techniques

Two main techniques used in PathCutter include request/action authentication and view separation. Here, we discuss related work that informed the development of these two techniques.

Request/Action Authentication Techniques: Barth et al. [22] propose the use of an *origin* HTTP header to validate the origin (<scheme, host, port>) of an HTTP request. Previous attempts have also used secret tokens and the *referer* header to validate requests from a different origin. Defending against CSRF attacks is similar to cutting off the propagation path of a JavaScript worm in the sense that both of them need to validate the request. Therefore, *referer* and secret tokens can be used in both cases. However, there is also the following fundamental difference. A CSRF attack is across different same-origin policy (SOP) origins but a JavaScript XSS worm propagation usually happens within the same SOP origin. For example, `malicious.com` may try to modify contents on `bank.com`. Those two websites are properly isolated at the client side. Compared to a typical CSRF attack, a JavaScript worm spreading is much harder to defend and detect, because the forged request is actually from the same website—the MySpace worm spreads within MySpace, so SOP is not violated). And in theory, the worm can modify the user's contents from the client side because they are in the same origin. Hence, although we leverage CSRF defense methods within PathCutter, they cannot by them-

selves prevent XSS worm propagation as the *origin* header proposed by Barth et al. [7] contains only origin information such as `http://www.foo.com` and cannot distinguish requests from the same origin. To better illustrate this point, secret tokens are indeed adopted by MySpace. However, because the token is accessible from the same origin, the MySpace Samy worm [10] successfully steals that token. Similarly, attackers can also easily create and control an iframe with the same origin URL to make a request with correct *referer* header.

As a complementary strategy, social networks could also incorporate techniques such as CAPTCHAs [1] to authenticate actions/requests and defend against XSS attacks, at the cost of some user inconvenience. Similar defense techniques are used by websites such as Amazon that always prompt users to input their username and password before performing a potentially dangerous action.

View Separation Techniques: MiMoSA [20] proposes a view/module extraction technique to detect multi-step attacks at server side. Their concept of a view is different from ours, and they can detect only traditional server-side XSS vulnerabilities. Many blogs such as WordPress adopt different subdomain names like `name.blog.com` to augment users' self-satisfaction of owning a subdomain. Since the purpose is not actually to prevent XSS worms, they do not really combine view separation with request authentication. View separation is also very coarse, such that vulnerable actions cannot always be isolated. For example, in many social web networks, such as Facebook, updates from your friends will also be shown on your own page, thus launching unauthorized requests. In PathCutter, as shown later by the Elgg example in Section 5, contents in the same page can be separated into different views. Finally, there has been a recent research thrust [26, 28, 39, 42, 43, 45] on building better sandboxing mechanisms for browsers. We argue that these approaches are complementary. While sandboxing provides a strong containment system, it is entirely up to the programmer to decide which contents to put into the container. In PathCutter, we can adopt any of these approaches to make the isolation of different views stronger.

4 Design

Here, we first provide an overview of the approach taken by PathCutter. Next we define the concept of views and describe strategies used by PathCutter for implementing view isolation and action authentication. Finally, we describe how view isolation and action authentication can prevent the propagation of an XSS worm.

4.1 Design Overview

PathCutter first isolates different pages from the server at the client side, and then authenticates the communication between different pages and the server. By doing this, the worm propagation path, in the form of an unauthorized request from a different page will be successfully blocked. The two main self-propagation routes of an XSS worm are cut off as shown in Figure 2(b).

- **Malicious HTTP request to the server from the infected page.** This is the most common exploit method employed by XSS worms, i.e., they send a request to modify the benign user's profile/contents at the server from the attacker's (or infected user's) page. Because the request is from the victim's client browser, the server will honor that request. In our system, because the originating page of each request will be verified, the server can easily deny such a request.
- **Malicious DOM access to the victim's page from infected page at client side.** An XSS worm can modify the victim's page at the client side to send a request on behalf of that page. Because pages are well isolated at client side, this self-propagation path is cut off.

Key Concepts. Key concepts used in PathCutter are defined as follows.

- **Views.** A view is defined as a portion of a web application. At client side, a view is in the form of a web page or part of a web page. As a simple example, one implementation at a blogging site might consider different blogs from different owners to be different views. It might also consider comment post forms to be a separate view from the rest of the page.
- **Actions.** An action is defined as an operation belonging to a view. For example, a simple action might be a request from blog X (view X) at client side to post a comment on X's blog post.
- **Access Control List (ACL) or Capability.** Access control list records all the actions that a view can perform. In the previous example, a request from X cannot post on blog Y's blog post, because X does not have the right to do this action. Capability is a secret key that a view owns that enables it to perform a specific action. Our system supports the use of either ACLs (in the form of referrer-based validation) or capabilities (in the form of per-url session tokens) for access control.

4.2 Web Application Modification (View Separation, Isolation and Authentication)

We explore and evaluate different strategies for implementing view separation, view isolation and view authentication, and securing an application using PathCutter.

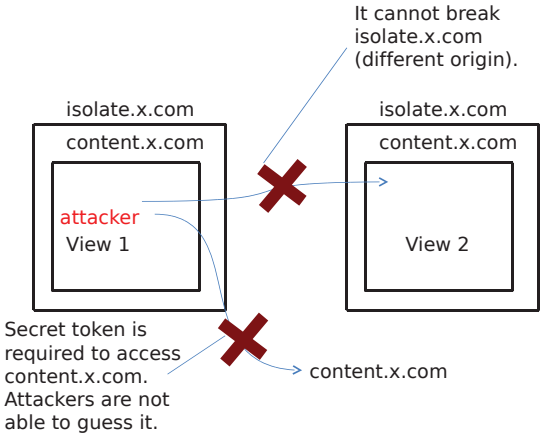

Figure 3. Isolating Views Using Pseudodomain Encapsulation

Dividing Web Applications into Views. We imagine that there are at least three potential strategies for separating web application content into views. First, a web application can be divided into views based on semantics. For example, a blog website can be divided using blog names. Forums can be divided based on threads and subject matter. A second way to divide web applications is based on URLs. For example, when clients visit `blog.com/options` and `blog.com/update`, we can consider those two to be from different views. Finally, in some web applications, user-injected contents like comments might be on the same web page as vulnerable actions such as posting comments. In such cases, we need to isolate either those user comments or the vulnerable actions.

Isolating Views at the Client Side. According to the same-origin policy (SOP), DOM access for different sessions from the same server is allowed by default. Theoretically, we can isolate each view inside a new domain. But numerous domain names are required. PathCutter **encapsulates views within a pseudodomain** to achieve isolation by just two domain names. As shown in Figure 3, for each view from contents.x.com, we embed an iframe with pseudodomain name *isolate.x.com* inside the main page. Therefore, an attacker who obtains control of *contents.x.com* in one view, cannot break *isolate.x.com* to access contents inside another view that also belongs to *contents.x.com* due to the same-origin policy. HTML5 also provides a *sandbox* feature for preventing the origin access that can be used to further strengthen isolation between different views.

Authenticating Actions. PathCutter checks the originating view for each action (e.g., posting a comment) to ensure that the view has the right to perform the specific action. Either of the following two strategies might be implemented

http://www.foo.com/blog1/index.php:

```
<iframe src="contents.foo.com/blog1/index.php?token=**"
 sandbox="allow-forms, allow-scripts">
</iframe>
```

Figure 4. Implementing Session Isolation in WordPress

to authenticate actions.

- **Secret Tokens.** We could explicitly embed a secret token with each action or request, especially those tending to modify contents on the server side, as a *capability*. A simple request might look like the following:
  `http://www.foo.com/submit.php?sid=****&...`
  The server will check the sid of each request to see if it has the right to modify the contents. As an attacker will not be able to guess the value of the secret token (sid), a request from the attacker's view will not have the right to modify contents on another user's page.
- **Referer-based View Validation.** The referer header in the HTTP request can be used for recognizing views from which an action originated. Then servers can check if the action is permitted in the *access control list*. If not, the action is denied.

4.3 Severing Worm Propagation

For a JavaScript worm that seizes control of a certain view of an application by exploiting an XSS vulnerability, there are two possible avenues to propagate as shown in Section 4.1. Blocking the worm propagation can be considered in terms of blocking the following two forms of malicious behavior. First, the worm can initiate an illegal action to the server in order to exploit other views. Because PathCutter checks every action originating from each view, illegal actions will be prevented. Second, the worm can open another view at client side, and then infect that view by modifying its contents. PathCutter's view isolation logic ensures that the worm cannot break boundaries of different views belonging to a web application at the client side.

5 Implementation

5.1 Case Study 1: Server-side Implementation - WordPress

We use WordPress [13], an open source blog platform, as an example to illustrate the feasibility of implementing PathCutter by modifications at the server side. We find that just 43 lines of additional code were required to add support for secret token authentication and view isolation. It took the authors less than five days to understand WordPress

```
document.onload = function() {
  forms = document.getElementsByTagName("form");
  for (i=0; i<forms.length; i++) {
     forms[i].innerHTML="<input type=\"hidden\" value=\""
               +window.mySID+"\"/>" +forms[i].innerHTML;
  }
}
```

Figure 5. JavaScript Function Added to WordPress for Inserting Secret Tokens into Actions

source code and insert those modifications.

Dividing and Isolating Views. We enable the multisite functionality of WordPress, and our implementation classifies different blogs in WordPress as belonging to different views. For example, `www.foo.com/blog1` and `www.foo.com/blog2` will be divided into different views. A finer-grained separation of views, such as different URLs, can also be adopted. As a proof of concept, separation by different blogs is implemented. As shown in Figure 4, a view will be isolated at client side by iframes. Every time a client browser visits another user's blog, the real contents will be embedded inside the outer frame to achieve isolation. Borders, paddings, and margins will be set to zero in order to avoid any visual differences.

Identifying Actions. Vulnerable actions in WordPress are normally handled by a post operation in a $form$ tag. For example, the *comments posting* functionality is the output to a user through *comment-template.php* and handled in *wp-comments-post.php*. Similarly, the *blog posting/updating* functionality is the output to a user through *edit-form-advanced.php* and handled in *post.php*.

Authenticating Actions. We use capability-based authentication (using a secret token) to validate user actions. Every action belonging to comment or blog posting categories must be accompanied by a capability, or else the action will be rejected. We implement this by injecting a hidden input into the form tag, as shown in Figure 5 by JavaScript, such that the client's post request to the server always includes a capability.

The ideal locations for implementing authentication are at points where client-side actions affect the server database. WordPress has a class for all such database operations and because every database operation will go through that *narrow interface*, we can quickly ensure that our checks are comprehensive.

5.2 Case Study 2: Server-side Implementation - Elgg

Elgg [5] is an open social network engine with many available plug-ins. We use Elgg 1.7.10 with several basic embedded plug-ins such as friends and blogs. Just two additional lines of code were required to add support for view isolation into the Elgg source code base. An additional file was also required to support the modification which had 23 lines. It took the authors less than three days to understand the Elgg source code and insert the corresponding modifications.

Dividing and Isolating Views. As discussed below, Elgg has built-in mechanisms to protect the *post* action. However, a JavaScript worm can still steal the secret token just as in the case of the MySpace Samy Worm. For example, the worm could send an XMLHttpRequest to the server to get the posting page and then steal the token. Therefore, we need to isolate specific views at the client side to protect the secret token as shown in Figure 6. Instead of using a div to submit a comment, we adopt methods mentioned in Section 4.2 to isolate the view for posting comments.

Identifying Actions. The action we wish to protect is the comment posting action in blog functionality of Elgg. It is handled in mod/blog/actions/add.php.

Authenticating Actions. The blog plug-in functionality in Elgg has already implemented action authentication. A secret token, named $\_\_elgg\_token$, is embedded in each post action that is checked by add.php upon each post request. If the token is incorrect or missing, an error message is returned to the user. We extend this logic to also check the *referer* header of each post action.

Fixing Cascaded Style Sheet (CSS) Issues. After isolating the *post* action into a separate view, we still need to fix several outstanding CSS issues, including the following. First, we make the iframe body transparent to leave the original background unaltered. Second, we include all original CSS files in order to retain the original style and layout. Finally, we make the iframe size automatic and use the seamless attribute in HTML5 to ensure that the iframe is well integrated with its parent.

5.3 Case Study 3: A Proxy Implementation

Although a server-side implementation is most desirable, a proxy-based deployment approach is attractive in certain scenarios because it provides greater flexibility. For example, this enables the service provider to deploy PathCutter without changing the application, or alternatively, PathCutter could be deployed at the client's enterprise network.

Original code:

```
echo elgg_view('input/form', array('body' => $form_body, 'action' => "{$vars['url']}action/comments/add"))
```

After PathCutter modification:

```
echo "<iframe style = 'background:inherit;border:0;margin:0;padding:0'
        sandbox='allow-forms' scrolling='no' height='400pt' width='100%'
        src='http://other.com/echo.php?content="
     .urlencode(elgg_view('input/form', array('body' => $form_body,
     action' => "{$vars['url']}action/comments/add"))."'/>";
```

Figure 6. Isolating Views in Elgg by Modifications to edit.php in views/default/comments/forms/

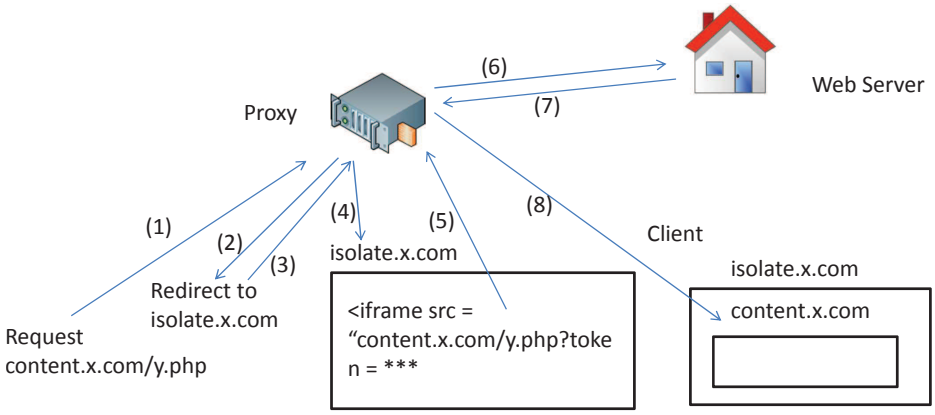

Figure 7. View Isolation in PathCutter Proxy Implementation

Here, we describe a prototype implementation of PathCutter over Privoxy [11] that secures Facebook.

Isolating Existing Views. As shown in Figure 7, when the client browser requests a URL, such as content.x.com, which requires isolation, the proxy redirects the URL to isolate.x.com with an embedded iframe containing a secret token in the src property. When the client browser requests the redirected URL, the proxy forwards the request to the real web server and displays the returned content in the iframe.

Dividing Nonexisting Views. As an example, we consider an individual's Facebook page that typically includes comments and updates from other users in the individual's friend circle. Hence, we need to isolate multiple views within each page. When we look at the HTML source code of Facebook, we find that each comment and update is embedded inside a `span` tag. So, the PathCutter proxy identifies span tags and simply uses a regular expression to replace them with an iframe. For example, as shown in Figure 8, we replace the `span` tag with an iframe and echo back the other users' comments. Even if a malicious script is injected by an attacker, our transformation ensures that the malicious script operates in a separate view, isolated from the view with the capability to modify the victim user's content.

Original code:

```
<span data-jsid="text"> user comments </span>
```

After PathCutter modification:

```
<span data-jsid="text">
<iframe scrolling='no' height='100%' sandbox style='..'
 src='http://foo.com/echo.php?content=user%20comments'/>
 </span>
```

Figure 8. Implementing View Separation for Facebook

Our proposed approach to dividing views at the proxy is vulnerable to *injection attacks*, i.e., the attacker can inject the same pattern that we are looking for into the comments. For example, as shown in Figure 8, the attacker can inject `<span data-jsid="text">` or `</span>` to confuse the proxy. The proxy needs to *modify the signature* in order to deal with such injection attacks. For example, if the attacker tries to inject `</span>`, the proxy needs to find the last matching `</span>` instead of the first match.

Authenticating Actions. The proxy checks the *referer*

header of each request. If the request is from *foo.com* (our echoing server), this indicates that it is potentially a forged request originating from a malicious comment. Hence, such requests are rejected.

6 Evaluation

We analyze the effectiveness of the PathCutter approach against five real-world worms. We further evaluate the server-side implementation against two proof-of-concept worms.

6.1 Evaluation against Real-world Worms

We evaluate PathCutter against two server-side XSS (MySpace Samy, Yamanner) worms and three client-side XSS (Renren, SpaceFlash, Boonana) worms by analyzing the source code and online descriptions of these worms.

1. Boonana Worm. Boonana [17] is a Java applet worm that was released in October 2010. The propagation of this worm can be divided into the following steps:

1. *A benign user visits an infected profile with a malicious Java applet posted by the attacker.*
2. *The malicious Java applet exploits a Java vulnerability to inject malicious JavaScripts on the client side, thus escalating its privilege to the victim user.*
3. *The worm posts itself on the visiting user's wall using the stolen cookie.*
4. *Boonana proliferates over the social network when more people visit the malicious applet.*

PathCutter Defense: PathCutter blocks the Boonana worm propagation at Step 2, by ensuring that the worm gains only the privilege of a view containing the page of the malicious Java applet, and not the privilege of the user's Facebook profile page. Therefore, the web server declines the request to post on the user's wall.

2. Renren Worm. The Renren worm [16] was a Flash-based worm that spread on the Renren social network (one of the largest Chinese social networks). The worm was released in 2009 and affected millions of users. The propagation of this worm can be divided into the following steps:

1. *A victim user visits an infected profile with a malicious Flash movie posted by the attacker.*
2. *The malicious Flash movie exploits a Flash vulnerability, which injects malicious JavaScripts at the client side, thus escalating its privilege to that of the victim* (as shown in Figure 9).
3. *The injected script replicates itself on the victim's wall.*
4. *When other users visit the infected user's profile, the worm repeats the infection and replication process, and thus spreads.*

PathCutter Defense: A user who wants to share something on the Renren social network, needs to get a page `http://share.renren.com/share/buttonshare.do?link=..`, and then send the real share request. PathCutter isolates the real sharing request in a view A that is different from view B where updates from friends are displayed. Therefore, at Step 2, the worm obtains only the privilege of that specific view B, and so is unable to replicate itself on behalf of the victim user.

3. MySpace Samy Worm. The MySpace Samy worm [10] was one of the first cross-site scripting worms that spread in the MySpace social network, affecting over a million users in 2005. The attack steps of Samy worm are as follows:

1. *The victim visits an infected profile page, which carries a malicious script (due to a script filtering problem in MySpace). The infected user's profile has the following code to embed a malicious* $< div\ style = background : url('java\backslash nscript : eval(...)')$
2. *The worm first steals the secret token, required by MySpace, using a GET HTTP request to escalate its privilege to the view that can send a POST HTTP request.*
3. *The worm posts itself to* `/index.cfm?fuseaction=profile.previewInterests&Mytoken=**` *on the victim's profile via XMLHttpRequest.*
4. *The Samy worm proliferates over the social network as more victims visit the growing list of infected profiles.*

PathCutter Defense: Two propagation paths are severed. First, when the worm tries to steal the secret token required by MySpace, that access is denied because different profiles are isolated into different views. The XMLHttpRequest is sent to a different domain and the response is not accessible by the worm. Second, when the worm sends out the POST request, that request is actually from the infected user's profile and not from the victim's profile. PathCutter correctly checks the capabilities of the originating view and denies such modifications.

4. SpaceFlash Worm. The SpaceFlash worm [12] was released in 2006 as another JavaScript worm spreading on the MySpace network by exploiting a Flash vulnerability. The steps of a SpaceFlash infection are as follows:

1. *A victim user visits the attacker's "About Me" page with a malicious Flash applet.*
2. *The malicious Flash applet is executed, exploits a Flash vulnerability to access the MySpace page, and retrieves the victim's profile by visiting* `http://editprofile.myspace.com/index.cfm?fuseaction=user.HomeComments`, *thus escalating its privilege to match the victim.*

Allowing DOM access from Flash:

```
XN.template.flash=function(o){
return  <embed src=\"+o.filename+\ type=\application/x-shockwave-flash\
+width=\"+(o.width||320)+\ height=\"
+(o.height||240)+\ allowFullScreen=\true\ wmode=\"
+(o.wmode||transparent)+\ allowScriptAccess=\always\></embed>;
};
```

Modifying DOM to add and invoke the malicious script:

```
var fun = var x=document.createElement(SCRIPT);
x.src=http://n.99081.com/xnxss1/evil.js;
x.defer=true;document.getElementsByTagName(HEAD)[0].appendChild(x);;
flash.external.ExternalInterface.call(eval,fun);
```

Figure 9. Flash Vulnerability Exploitation in the Renren Worm

3. *The worm sends out an AJAX request to the server to post itself on the victim's "About Me" page.*
4. *SpaceFlash proliferates over the social network as more victims visit the growing list of infected "About Me" pages.*

PathCutter Defense: In step 2, the worm cannot escalate its privilege and therefore the unauthorized post request in Step 3 is rejected, as it is does not originate from the victim's "About Me" page.

5. Yamanner Worm. The Yamanner worm [8] was released in 2006 and infected tens of millions of users. It was a JavaScript worm spreading in Yahoo! mail. The steps of infection and propagation were as follows:

1. *A victim user receives malicious email from the attacker.*
2. *The victim user clicks on the email and the malicious scripts inside the email are executed due to a bug in Yahoo's script filter. Using these scripts the worm acquires the victim's privilege.*
3. *The worm opens the victim's address book and sends out malicious email containing itself to those who are listed in the book.*
4. *Yamanner proliferates across the email social networks of those victims who open the email.*

PathCutter Defense: Even though the worm logic gets executed at the client, it does not have the privilege of sending email to others. In the PathCutter approach, a secret token is required to perform the action and the worm cannot steal the token because it is isolated inside a different domain.

Summary. Although the five worms described above all use different vulnerabilities and techniques to achieve JavaScript execution privileges on behalf of a victim at a specific social network web site, they are all blocked by PathCutter. PathCutter exploits the fact that aforementioned worms all have to send a request to the server from an untrusted view in order to post themselves on the victim's profile. The common propagation path is severed in each case.

```
<form<?php echo $enctype; ?> id="upload-file" method="post" action="<?php
echo get_option('siteurl')
. "/wp-admin/upload.php?style=$style&amp;tab=upload&amp;post_id=$post_id";
?>">
```

Figure 10. CVE-2007-4139 (Untainted Input in wp-admin/upload.php)

6.2 Evaluation against Experimental Worms

We evaluate our implementation using experimental worms that operate on WordPress and Elgg. We implemented two XSS attacks based on a published vulnerability in WordPress (CVE-2007-4139 stored XSS [3]) and a custom DOM-based attack on Elgg that is based on a proof-of-concept DOM-based attack [4]. We adopt the same code as published in [4] to let Elgg read language settings in URL parameters and write it on the web page without sanitizing. Figure 10 shows the WordPress XSS vulnerability, which does not properly cleanse one input from the user.

To convert the proof-of-concept attacks into worms, we incorporated two propagation modules: a custom propagation module that we developed and a published worm template [9]. The custom propagation module implements a simple worm to propagate on the network as shown in Figure 11. The functionality of the worm is to post itself (for DOM-based XSS attack, it will be the URL with code injected into language settings) on the victim user's blog comments by AJAX when the victim visits an infected page. The published worm template is the universal JavaScript XSS worm template [9] as shown in Figure 12. We updated the worm template with the XSS vulnerability we created,

```
check_infected();
// check if the user is infected or not
xmlhttp = new XMLHttpRequest;
xmlhttp.open("POST", post_url,true);
xmlhttp.onreadystatechange=function() {
  if (xmlhttp.readyState==4) {
    set_infected();
  }
}
xmlhttp.setRequestHeader("Content-type"
    , "application/x-www-form-urlencoded");
xmlhttp.setRequestHeader("Content-length"
    , payload.length);
xmlhttp.send(payload);
```

Figure 11. Propagation Logic of Custom Proof of Concept Worm

```
  //Worm's HTTP request object
function xhr() { ... }
  Object.prototype.post = function(uri,arg) {
   /*** usage: xhr().post('foo.php'); ***/
   this.open('POST', uri, true);
   this.setRequestHeader('Content-type'
    ,'application/x-www-form-urlencoded');
   ...
   this.send(arg);
  };
  /*** source morphing component ***/
  Object.prototype.morph = function(s) {
   ...
   switch(morphtype) {
     case "unicode": ...
     case "charcodes": ...
   }
 }
```

Figure 12. Propagation Logic of a Published Worm Template

and the worm replicates itself as blog comments.

For evaluation, we deployed WordPress and Elgg with and without our modifications on a Linux machine with Apache-PHP-mySQL installed in our network. Before integrating PathCutter, we found that both worms propagate easily in both networks by replicating themselves in the form of blog comments. After adopting PathCutter, worm propagation is effectively stifled, as view separation ensures that comments must be posted from the same view as that of the victim's blog.

6.3 Performance Evaluation

We summarize memory and rendering time overhead introduced by PathCutter. No measurable CPU overhead was introduced by the system at the client.

Memory Overhead. The memory overhead introduced by PathCutter depends on the complexity of pages on the social network and strategy used by PathCutter to separate views. In the Elgg and WordPress examples of Section 5, we chose to isolate html elements with *vulnerable actions* into separate views. Hence, only two frames were required per blog page, and the memory overhead we introduced was negligible.

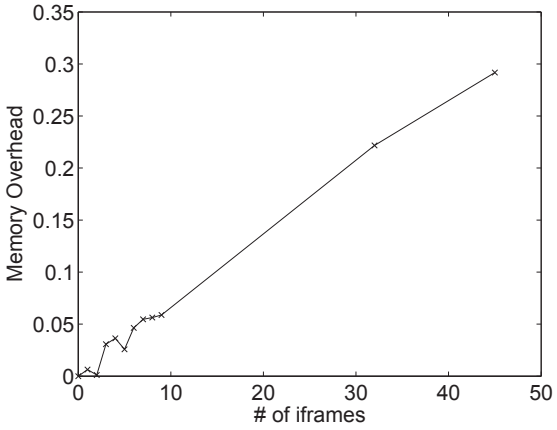

Figure 13. Memory Overhead for Isolating Comments from Friends

Instead, if we choose to isolate views based on *content provenance*, like comments in the Facebook example in Section 5, the memory overhead we introduce depends on the number of comments on the web page. We conducted an experiment where we visited Facebook, using our proxy implementation, on a Linux client running Firefox 3.6.18 with 2 GHz dual core Xeon processors and 2 GB of memory. The results are shown in Figure 13. We find that when the number of comments is less than 10, PathCutter's view isolation iframes introduce less than 10% overhead. If we have 45 comments, the iframes introduce nearly 30% overhead. On Facebook threads are folded by default when the number of comments in a thread exceeds three. Therefore, we do not expect the overhead to be significant. Finally, if Facebook adopts PathCutter's approach and isolates potentially dangerous actions (e.g., comment posts) into a different view, we do not need to isolate comments at the proxy and introduce those overheads.

Rendering Time Overhead. We perform a rendering time comparison between our modified Elgg and the original Elgg implementation. The experiment is performed on a Linux machine running Firefox 3.6.18 with 2 GHz dual core Xeon and 2 GB memory. We use Firebug to monitor the onload event. Experiments are performed ten times for the blog posting page of each version. The average rendering time for modified Elgg is 1.18 seconds, and for original Elgg is 1.14 seconds. The additional rendering time overhead is about 3.5%.

7 Discussion

While we argue that PathCutter provides an effective security solution for defending against XSS worms in social web networks, it is by no means a complete security solution. Here, we discuss certain known limitations of PathCutter.

1. Vulnerability to Cookie and Content Stealing Attacks. Networks running PathCutter are still vulnerable to cooking stealing and other content exfiltration XSS attacks, as we do not block the exploitation of the XSS vulnerability. The sole objective of our system is to cut off the self-propagation path of a JavaScript worm. Hence, although portions of a victim user's information may be revealed, the worm cannot infect the user and spread any further. Websites need to adopt auxiliary approaches to prevent such attacks, e.g., using HTTP-only cookies to prevent cookie-stealing attacks.

2. Vulnerability to Phishing and ClickJacking Attacks. Networks running PathCutter also remain vulnerable to phishing and clickjacking attacks. First, the worm can use phishing or clickjacking to steal a user's password or some other confidential information. This is out of scope for this paper. Second, one might argue that because the worm is running under the privilege of the hosting web site, it is much easier for the worm to launch such attacks. This is true. However, both phishing and clickjacking require user interaction. The user is not likely to input the worm directly because the source code of the worm is unlikely to be in the form of a normal comment, and the worm cannot induce the user to input one because of the isolation mechanism in Section 4.2. Therefore, phishing and clickjacking will not help the propagation of a JavaScript worm even if launching those attacks is easier in this case.

3. Vulnerability to Drive-by Download Worms. Finally, networks running PathCutter remain vulnerable to worms such as Koobface that upload binaries to victim hosts using drive-by exploits. Such attacks are also out of scope for this paper. However, systems such as Blade [36] may be used to prevent such attacks.

Summary. Since PathCutter allows the exploitation of an XSS vulnerability at the client-side, there are certain attacks that remain possible. Specifically, damage to the first view in each attack is not within the scope of this paper. However, PathCutter strives to minimize the harm that an XSS worm can cause, by containing it to a specific view of a website and blocking its propagation to other views and other users.

8 Conclusion

We propose a new architectural approach to blocking the two main propagation paths of JavaScript worms – DOM access to a different view and unauthorized HTTP requests to the server. We implement a prototype upon WordPress and Elgg. We evaluate our system using five real-world worms and two proof-of-concept worms. Our preliminary evaluation demonstrates that the PathCutter approach requires minimal modifications to the server application and is effective against most XSS worms.

9 Acknowlegements

This material is based upon work supported in part by the National Science Foundation under grant no. CNS-0831300, CNS-0716612 and the Army Research Office under Cyber-TA Grant no. W911NF-06-1-0316. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily refect the views of the National Science Foundation or the Army Research Office.

References

[1] CAPTCHA. http://en.wikipedia.org/wiki/CAPTCHA.

[2] Content security policy. http://people.mozilla.com/~bsterne/content-security-policy/index.html.

[3] CVE-2007-4139. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-4139.

[4] DOM-based XSS attack. https://www.owasp.org/index.php/DOM_Based_XSS.

[5] Elgg. http://www.elgg.org/.

[6] A firefox pdf plug-in XSS vulnerability. http://lwn.net/Articles/216223/.

[7] IETF draft about HTTP origin. http://tools.ietf.org/html/draft-abarth-origin-00#section-6.

[8] Javascript Yamanner worm. http://www.theregister.co.uk/2006/06/12/javscript_worm_targets_yahoo/.

[9] A modular universal XSS worm. http://groups.google.com/group/ph4nt0m/msg/d75435c75fc6b81b?pli=1.

[10] Myspace samy worm. http://namb.la/popular/tech.html.

[11] Privoxy. www.privoxy.org.

[12] Spaceflash worm on MySpace. http://news.cnet.com/Worm-lurks-behind-MySpace-profiles/2100-7349_3-609%5533.html.

[13] Wordpress. http://wordpress.org/.

[14] XSS Cheat Sheet. http://ha.ckers.org/{XSS}.html.

[15] XSS definition and classification in Wikipedia. http://en.wikipedia.org/wiki/Cross-site_scripting.

[16] XSS worm on Renren social network, 2009. http://issmall.isgreat.org/blog/archives/2.

[17] Boonana Java worm, 2010. http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx%?Name=Worm%3AJava%2FBoonana.

[18] Facebook hit by XSS worm, 2011. http://news.softpedia.com/news/Facebook-Hit-by-XSS-Worm-192045.shtml.

[19] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 387–401.

[20] BALZAROTTI, D., COVA, M., FELMETSGER, V. V., AND VIGNA, G. Multi-module vulnerability analysis of web-based applications. In *CCS: Conference on Computer and Communication Security* (2007).

[21] BARTH, A., CABALLERO, J., AND SONG, D. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *SP:IEEE Symposium on Security and Privacy* (2009).

[22] BARTH, A., JACKSON, C., AND MITCHELL, J. Robust defenses for cross-site request forgery. In *CCS: Conference on Computer and Communication Security* (2008).

[23] BATES, D., BARTH, A., AND JACKSON, C. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference on World Wide Web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 91–100.

[24] BISHT, P., AND VENKATAKRISHNAN, V. N. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2008), DIMVA '08, Springer-Verlag, pp. 23–43.

[25] CHONG, S., VIKRAM, K., AND MYERS, A. C. SIF: Enforcing confidentiality and integrity in web applications. In *USENIX Security Symposium* (2007).

[26] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for web applications. In *SP: IEEE Symposium on Security and Privacy* (2006).

[27] DABIRSIAGHI, A. Building and stopping next generation XSS worms. In *3rd International OWASP Symposium on Web Application Security* (2008).

[28] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *SP: IEEE Symposium on Security and Privacy* (2008).

[29] HOFFMAN, B. Analysis of web application worms and viruses. In *Blackhat 06*. http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Hoffman/B%H-Fed-06-Hoffman-up.pdf.

[30] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *WWW: Conference on World Wide Web* (2004).

[31] JIM, T., SWAMY, N., AND HICKS, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, pp. 601–610.

[32] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP: IEEE Symposium on Security and Privacy* (2006).

[33] KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC: ACM Symposium on Applied Computing* (2006).

[34] LIVSHITS, B., AND CUI, W. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the Usenix Annual Technical Conference* (July 2008).

[35] LIVSHITS, V. B., AND LAM, M. S. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), USENIX Association, pp. 18–18.

[36] LU, L., YEGNESWARAN, V., PORRAS, P. A., AND LEE, W. BLADE: An attack-agnostic approach for preventing drive-by malware infections. In *ACM Conference on Computer and Communications Security'10* (2010), pp. 440–450.

[37] MARTIN, M., AND LAM, M. S. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 31–43.

[38] NADJI, Y., SAXENA, P., AND SONG, D. Document structure integrity: A robust basis for cross-site scripting defense. In *Network and Distributed System Security Symposium* (2009).

[39] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *EuroSys: European conference on Computer systems* (2009).

[40] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 513–528.

[41] SUN, F., XU, L., AND SU, Z. Client-side detection of XSS worms by monitoring payload propagation. In *ESORICS* (2009), M. Backes and P. Ning, Eds., vol. 5789 of *Lecture Notes in Computer Science*, Springer, pp. 539–554.

[42] TANG, S., GRIER, C., ACIICMEZ, O., AND KING, S. T. Alhambra: A system for creating, enforcing, and testing browser security policies. In *WWW: Conference on World Wide Web* (2010).

[43] TANG, S., MAI, H., AND KING, S. T. Trust and protection in the illinois browser operating system. In *OSDI'10: Proceedings of the 9th USENIX Symposium on Operating Systems Design & Implementation* (Berkeley, CA, USA, 2010), USENIX Association.

[44] TER LOUW, M., AND VENKATAKRISHNAN, V. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *30th IEEE Symposium on Security and Privacy* (2009).

[45] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal OS construction of the Gazelle web browser. In *18th Usenix Security Symposium* (2009).

[46] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium* (2006).

[47] XU, W., ZHANG, F., AND ZHU, S. Toward worm detection in online social networks. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 11–20.