POSTER: A Path-Cutting Approach to Blocking XSS Worms in Social Web Networks

Yinzhi Cao
Northwestern University
Evanston, IL
yinzhi.cao@eecs.northwestern.edu

Vinod Yegneswaran
SRI International
Menlo Park, CA
vinod@csl.sri.com

Phillip Porras
SRI International
Menlo Park, CA
phillip.porras@sri.com

Yan Chen
Northwestern University
Evanston, IL
ychen@northwestern.edu

ABSTRACT

Worms exploiting JavaScript XSS vulnerabilities rampantly infect millions of webpages, while drawing the ire of helpless users. To date, users across all of the popular social networks, including FaceBook, MySpace, Orkut and Twitters, 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* of the current forms of XSS worms, including those that exploit traditional XSS, DOM-based XSS, and content sniffing XSS vulnerabilities. We demonstrate PathCutter using WordPress and perform a preliminary evaluation on a proof-of-concept JavaScript Worm.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software (e.g., viruses, worms, Trojan horses); Information flow controls

General Terms

Security

Keywords

Cross site scripting (XSS), JavaScript Worms, Security, Social Network

1. INTRODUCTION

The high degree of connectivity and dynamism observed in modern social networks enables worms to spread quickly by making unsolicited transformations to millions of pages. In particular, JavaScript-based Cross Site Scripting (XSS) worms pose a severe security concern to operators of modern social networks. In October 2005, the MySpace Samy worm [2] infected over one million users on the Internet within a span of 20 hours. More recently, outbreaks of similar worms have infected other major social networks (such as Renren [4] and Facebook [1]), while garnering significant public attention.

Copyright is held by the author/owner(s).
CCS'11, October 17–21, 2011, Chicago, Illinois, USA.
ACM 978-1-4503-0948-6/11/10.

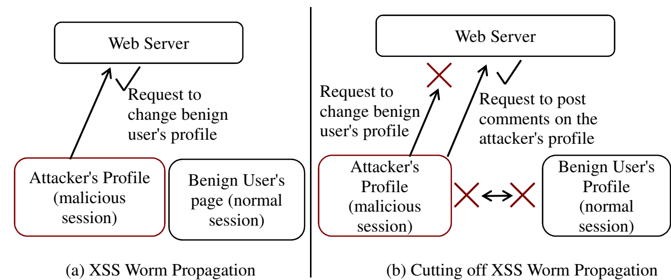

Figure 1: PathCutting an XSS Worm Propagation

We propose PathCutter as a new approach to XSS worm prevention to protect modern social networks. The twin goals of PathCutter are to (i) block the propagation of an XSS worm early, and (ii) 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 these 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, using an abstraction called *pseudodomains*. 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 introduces *per-url session tokens* and *referrer-based view validation* to ensure that requests are only made by views with the corresponding capability.

Scenario: PathCutting an XSS Worm Propagation. To illustrate how PathCutter blocks the propagation of a JavaScript-based XSS worm, we begin by describing the steps involved in a typical XSS worm exploit. Although XSS worms exploit different types of XSS attacks, they all share a common need to issue an unauthenticated cross-view request (shown in Step 3).

- *Step 1 – Enticement and Exploitation:* The victim is tricked into visiting (or stumbles upon) a malicious social network page with embedded worm logic. 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 rights and privileges of the victim user, with the goal of propagating its malicious logic to the victim's social network page. For example, if the

user logged into the social network account, the worm may modify the victim's profile and send messages to his friends.

- *Step 3 – Replication:* The worm copies itself onto the victim's page. As shown in Figure 1(a), the malicious JavaScript worm uses the victim's stolen privilege to send the server a request to change the victim's profile page.
- *Step 4 – Propagation:* When other benign users subsequently visit the infected victim's page, steps 2 and 3 are repeated, continuing the propagation of the worm

Related Work. The growing threat of XSS worms has been recognized by the academic community, notably in the following three papers. The Spectator [5] system proposed one of the first methods to defend against JavaScript worms. Their proxy system tracks the propagation graphs of activity on a web site 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 contrast, Sun et al. [6] propose a purely client-side solution, implemented as a Firefox plugin, 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. Meanwhile, Xu et al. [7] propose social graph monitoring as a means to detect worms that spread through social networks. However, as stated in their paper, their approach cannot detect XSS worms, such as the MySpace Samy worm, which do not modify the social graph and generate activities that are "passively noticeable by friends". PathCutter's goals and approach are complementary, addressing the limitations identified in the above systems.

2. DESIGN

PathCutter first isolates different pages from the server at the client side, then authenticates the communication between different pages and the server. Doing so, effectively halts the XSS worm's propagation path by essentially preventing the worm's unauthorized request on its infected page from altering the victim's own social network page, as illustrated in Figure 1(b).

- **Malicious HTTP request to the server from the infected page.** This is the most common exploit method employed by XSS worms, i.e., a content modification request of the victim's social network page is sent from the worm's infected page through the browser. Because the request is from the victim's browser, the social network's server will honor this request. However, using PathCutter, the origin page of each request is cross-checked against the target page, enabling the server to distinguish the request as an unauthorized modification attempt.
- **Malicious DOM access to the victim's page from an infected page at client-side.** An XSS worm can also modify the victim's page at the client side to send a request on the behalf of that page. To prevent this attack, PathCutter introduces pseudodomain isolation, which enables the browser to prevent cross-domain page modifications from the client-side.

2.1 Concepts Definition

Views. A view refers to a portion of a web application. From the 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 represent 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 client-side action may be a request from blog X (view X) to post a comment on X's blog post.
Access Control List (ACL) or Capability. An Access control list

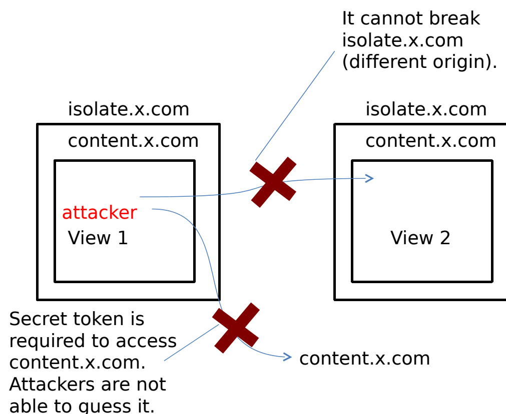

Figure 2: Isolating Views Using Pseudodomain Encapsulation

(ACL) defines all the actions that a view can perform. As we described request authentication, PathCutter prevents a request from site X to post on Y's blog. A Capability is a secret key that a view owns, which enables it to perform a specific action. PathCutter can enforce request authentication using either ACLs (in the form referrer-based validation) or capabilities (in the form of per-url session tokens) for access control.

2.2 Web Application Modification

We explore different implementation strategies to securing an application using PathCutter.

2.2.1 Dividing Web Applications into Views

By Semantics. A web application can be divided into views by its semantics. For example, blog sites can be divided based on blog names. Forums can be divide based on threads and subject matter.
By URLs. An alternate way to divide web applications is by URLs. For example, when clients visit `blog.com/options` and `blog.com/update`, we can consider those two to be from different views.
Isolating vulnerable actions or user injected content into separate views. In some web applications, user injected content, such as comments, might be in the same web page as vulnerable actions such as posting comments. In such cases, we need to either isolate those user comments or the vulnerable actions.

2.2.2 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. PathCutter **encapsulates views within a pseudodomain** to achieve isolation. As shown in Figure 2, for each view from contents.x.com, we embed it within an iframe from pseudodomain name *isolate.x.com*. Therefore, even if a worm obtains control of *contents.x.com* in one view, it 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, which can be used to further strengthen the isolation between different views.

2.2.3 Checking 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 may be implemented to authenticate actions.

- **Secret Tokens.** PathCutter may explicitly embed a secret token (e.g., a capability-based strategy) with each action or request, especially those that tend to modify content on the server side. A simple request could be implemented as follows:

```
http://www.foo.com/submit.php?sid=****&...
```

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

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

Figure 3: Session Isolation

```
document.onloand = 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 4: Inserting secret tokens into actions.

The server will check the sid (secret token) of each request to see if it has the right to modify the contents. As the client-site XXS worm cannot guess the sid value, a request from the attacker's view will not have right to modify contents on the target victim's page.

- **Referrer-based View Validation.** The referrer header in the HTTP request can be used to distinguish the view from which an action request originates. Then server can then deny any action for which the ACL does not specifically authorize.

2.3 Severing Worm Propagation

For a JavaScript worm which seizes control of a certain view of an application by exploiting an XSS vulnerability, there are two possible avenues from which to propagate, as shown in Section 2. Blocking the worm propagation can be considered in terms of blocking the following two forms of malicious behavior.

Enforcing View Separation. The worm may attempt a direct modification on a page whose origin is associated with a different view. Because PatchCutter checks every action originating from each view, the worms cross-view propagation attempt will be prevented.

Protecting View Boundaries at the Client-side. The worm can open another view on the client, and then infect that view by modifying its content. However, PathCutter isolates different views at the client side. Thus, the worm cannot break boundaries of different views belonging to a web application at the client side.

3. IMPLEMENTATION

To illustrate the feasibility of implementing the server-side modifications required by PathCutter, we use WordPress [3], an open source blog platform. We find that just forty three lines of additional code were required to add support for secret token authentication and view isolation. It took the authors five days to understand WordPress source code and insert modifications.

Dividing and Isolating Views. We enable the multi-site 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 supported. As a proof of concept, we implemented separation by different blogs. As shown in Figure 3, a view will be isolated at the client-side using iframes.

```
<script>
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();
   }
} ;
str = payload;
xmlhttp.setRequestHeader("Content-type",
             "application/x-www-form-urlencoded");
xmlhttp.setRequestHeader("Content-length", str.length);
xmlhttp.send(str);
</script>
```

Figure 5: A Proof-of-concept XSS Worm

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 output to a user through *comment-template.php* and handled in *wp-comments-post.php*. Similarly, the *blog posting/updating* functionality is 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 4, using JavaScript, such that the client's post request to the server always includes a capability.

The ideal location for implementing authentication is 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.

4. PRELIMINARY EVALUATION

Experiment Setup. We deployed WordPress with and without our modification on a Linux machine with Apache-PHP-mySQL installed. To simulate XSS vulnerabilities, we simply disabled XSS filering in WordPress. XSS filter is at wp-includes/formatting.php. esc_js, esc_html and esc_sql are used to filter the corresponding languages. We block all of them.

A Proof of Concept Worm. Next, we developed a simple worm to propagate on the network as shown in Figure 5. The functionality of the worm is to post itself on the benign user's blog comments by AJAX when the benign user visits an infected page.

Results. Finally, we validated that before adopting our PathCutter approach, the worm is able to easily post comments on visiting user pages and propagate. After adopting our approach, the worm is unable to post itself as a comment on visitor's blogs, because the posting request and the benign user's blog belong to different views that are isolated. The worm propagation is thus prevented.

5. CONCLUSION

In this paper, we propose a new 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 based on WordPress, and evaluate it using a simple proof-of-concept worm. Our preliminary evaluation demonstrates that the PathCutter approach requires minimal modifications to the server application and is effective against most XSS worms.

6. REFERENCES

[1] Boonana java worm. `http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx%?Name=Worm%3AJava%2FBoonana`.

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

[3] Wordpress. `http://wordpress.org/`.

[4] XSS worm on renren social network. `http://issmall.isgreat.org/blog/archives/2`.

[5] Livshits, B., and Cui, W. Spectator: Detection and containment of JavaScript worms. In *Proceedings of the Usenix Annual Technical Conference* (July 2008).

[6] 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.

[7] 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.