Building and Stopping Next Generation XSS Worms

Arshan Dabirsiaghi

3rd International OWASP Symposium on Web Application Security
Refereed Papers Track at OWASP AppSec Conference Europe 2008

Abstract. There has been much analysis of the recent MySpace and Yahoo! cross-site scripting worms. While the web development world slowly comes to recognize this method of attack, attackers are in the wild, presumably improving on the work of their predecessors.

In this paper we will analyze the design choices made by past worm authors and hopefully illuminate how future attackers will improve on the current paradigm when building the next generation of cross-site scripting worms. Also, the paper will highlight some new defense mechanisms in both preventing current and next generation cross-site scripting worms, and include some original recommendations on how to respond to such attacks.

Keywords: security, web application security, webappsec, xss, worms, viruses, owasp.

1. Introduction

There is an inverse relationship between the prevalence of cross-site scripting (XSS) vulnerabilities in web applications today and an overall awareness of XSS among web application developers. These circumstances have led researchers to discover empirically that 84.57% [21] percent of web sites suffer at least some form of XSS vulnerability.

XSS protections (input validation, canonicalization, and output encoding) generally cannot be applied globally across all application data without breaking much of the UI layer. This means inbound data and outbound data must be secured on a case-by-case basis, which is a condition that generally leads to failure as there are many reasons a developer may “miss” validating an input field or “forget” to output encode a field. This speaks nothing to the fact that most developers are not aware of most security issues and most managers do not budget security as part of their project expenditure.

The situation is compounded by the fact that attackers are always finding financial reasons to trick users into executing JavaScript inside their browser, so attacker motivation for discovering and exploiting these vulnerabilities is fairly high.

Web applications will always be exposing XSS vulnerabilities in their code and attackers will always want to exploit those vulnerabilities. The advent of Web 2.0 makes things worse as one of the fundamental characteristics of the paradigm is movement of code off of the server and onto the client or “into the cloud”. Hastily arranged trust boundaries, “improved” client side technology and traditional attacker cleverness can combine to create a situation where the shortcomings of current XSS worms are going to become quickly antiquated.

The paper will begin by briefly establishing what is presently going on in the world of XSS worms (also called “Web worms”), including the strategies used to infect and re-populate, and also the strategies used to “remove” a XSS worm from an infected application.

The paper then makes predictions about the future of XSS worms, based on experiments demonstrating that all the individual components of a “next-generation” worm can be built today.

Finally, the paper will conclude with a summary of the predicted traits and experiments to show that all it will take to make the next infamous event in web application security is for an attacker with the right motivation to find an opportune vulnerability.

2. The Present

There have been a handful of large-scale XSS worms worthy of analysis in depth, and we will briefly analyze those worms here in deference to the excellent analysis of those worms in [11] and [12].

2.1 MySpace: Samy Worm

Samy Kamkar, a then-19 year old prankster from Los Angeles, USA created the first known XSS worm after exploiting MySpace's blacklist-based validation mechanism. Samy's worm was fairly important for two reasons. The first reason is that it demonstrated the capability for a XSS vulnerability to become self-propagating, and the second is that it illuminated the inability of negative validation mechanisms, even good ones, to work with high assurance when permitting rich user data.

Samy infected his own user profile with a stored XSS attack that caused viewers of his profile to execute several XmlHttpRequest calls to bypass MySpace's CSRF protection, add Samy to their friends list, add his name to their list of heroes, and copy the infection code into their profile. Within 18 hours, Samy had attained 1 million friends and MySpace responded by bringing the site down and cleaning the infection.

2.2 Yahoo! Yamanner Worm

The Yahoo! Yamanner worm was an XSS worm that infected Yahoo!'s webmail application. The attack bypassed the XSS filtering system Yahoo! mail utilized to prevent XSS attacks from being stored in email messages. The author used the XSS vulnerability to propagate the XSS message by sending the exploit message to the contact addresses of a victim.

The worm author attempted to use the worm to profit, which was a first at the time. Aside from propagation, the exploit code attempted to pop up a window to the author's click-through site and forwarded the harvested email addresses of the victims to a remote site, where the author would presumably sell the list to spammers.

2.3 Orkut Worm

Google's social networking platform, Orkut, was hit by another extremely fast-spreading XSS worm in mid-December 2007. The worm propagated by viewing a malicious message on the victim's "scrapbook" which added the propagation code and joined the user to an "infected users" group.

The worm quickly grew, infecting between 300,000 and 600,000 victims before being removed.

2.4 Other Worms

There have been a handful of other notable worms, including the MySpace Zango/QuickTime and Nduja worms, and each is worthy of its own lengthy analysis. Although these worms were researched for the paper, an analysis of them would not help clarify any issues regarding XSS worms.

2.5 Traits of Current Worms

An analysis of past worms show that there are a number of traits common to the current wave of XSS worms.

2.5.1 Static Payloads

Contemporary Internet worms at all layers use static payloads. Static payloads are payloads that are constant, literal values. Most traditional exploits contain static payloads. This is a major shortcoming as payloads can be easily signatured for IDS, IPS, and removal purposes which are discussed more in depth later.

Polymorphic payloads have been used to bypass character restrictions in shellcode development, but not for the purposes of having shifting payloads. In effect, these "polymorphic" payloads only morph once after during payload execution, and are still essentially static.

2.5.2 Uncontrolled Growth

Contemporary worms have uncontrolled growth. Closely related to the fact that all XSS worm payloads are static is the fact that all contemporary worms have no method of growth control. This is a fatal flaw for worms in general as it ultimately guarantees their destruction. As the worm grows and grows exponentially, it will quickly and inevitably reach an infection rate that can't help but be detected and dealt with in an emergency-type fashion.

This is similar to Slammer and other traditional Internet viruses that congest the bandwidth of infected networks so much that it disrupts normal network use and demands response [3].

2.5.3 Passive Infection Models

All contemporary XSS worms have a passive infection model. The victim of the worm attack must perform some action to be infected. In the case of Samy, a new victim must choose to view the profile of a previous victim to become infected. In the case of the Yamanner worm, a new victim must open an email delivered by a previous victim. It is not likely that this will change because of the nature of web applications. Traditional viruses such as [26] can infect users without any requisite action on the part of the victim. As long as their computer is plugged into the Internet, they can be infected and start to pass on the virus because the injected cargo code (shellcode/machine code) has access to perform any system calls and thus perform any operation on the computer.

JavaScript, on the other hand, when executing in a non-local zone, is heavily sandboxed by the browser and will not allow the user to access local files, start programs or execute arbitrary system code. The language itself is not meant to heavily interact with the operating system, so much of the desired attacker capability is just not there to use maliciously.

2.5.4 No Cross-application Infections

All contemporary XSS worms stay on the same web application on which they were pre-programmed to exploit. Although an exploit for one web application is not necessarily very different from an exploit for another web application (i.e., "userid=' or 1=1—" vs. "acctNo=' or 1=1—"), the steps required to get to the stage where the exploit can be delivered are normally quite different and complex. For a worm to jump to another application, the worm would require an understanding or learning capability of how to register, login, find the vulnerability and perform the necessary exploit. This is no small task, especially for a JavaScript scanner which is hampered by the browser’s same origin policy.

Although there is the opportunity for unauthenticated XSS in many applications (blog comments, message boards, etc.), the sites with opportunities for large propagation will mostly require authentication.

The Njuda worm did cross domains, but it was pre-programmed to attack the sites it did attack [23].

2.6 Removing Current XSS Worms

What led to the lines of research in this paper was the initial question: How can one effectively remove a XSS worm from a fully-infested application? The options are quite limited, but they could be effective against the current wave of XSS worms.

2.6.1 Manual Purging

The first option is to shut down the application and manually purge the tainted records. Obviously this would result in downtime losses and does not scale. Also, it is unlikely that a non-security person would be able to recognize a properly obfuscated XSS payload with a high rate of accuracy.

2.6.2 Database Snapshot Restore

The second option is to restore the database to previous state which was snapped before the infection began. While effective in removing all the worm code from the tainted columns, it also causes all other application data changes since the infection to be lost.

2.6.3 Search and Destroy

The last option is to halt column updates momentarily to perform a search-and-destroy operation on the payload an attacker uses. This option seems the only viable solution now, but its effectiveness against a smart payload is dissected later in the paper.

3 The Future

What follows is an analysis and set of predictions on how future attackers (worm authors) can improve the functionality, stealth, infection rate and operational stability of their worms in the world of Web 2.0. Also, an analysis and set of predictions regarding next generation defense mechanisms for developers and browser vendors will be discussed.

3.1 Traits of Next-Generation Attacks

In this section we will discuss a number of ways in which future XSS worms will increase in capability, stealth and growth.

3.1.1 Payloads will use command and control channels

The most obvious mechanism that contemporary XSS worms don’t utilize is command-and-control channels (C&C). A C&C channel is either a one-way or two-way information line that an infected client uses to send messages to and/or receive information from a malicious server. C&C is a vital piece of malware because it allows the worm operator to give infected hosts arbitrary data, e.g. new instructions or targets for DoS attacks, etc.

The same origin policy does not prevent a hijacked browser from communicating with a host that is in collusion with the client for many reasons. Remote scripting, remote stylesheets, image dimensions, and other half-blind techniques can be used as covert channels between an infected browser and an evil web server.

A simple and effective use of worms would be to limit virus density. As discussed previously, worms would want to control density to increase the stealth and predictability of the infected network.

Having a static C&C channel introduces a single point of failure into the worm’s design. If a worm uses a C&C remote script at http://www.evil.com/payload.js, in one sense it is advantageous to the attacker because the web application can’t enforce a policy change on the server to prevent the user’s browser from referencing the malicious site. However, the developers/defenders (hereto referred to simply as defenders) may be able to remove the channel (malicious site) either by contacting the ISP or registrar for evil.com as discussed in [8].

Therefore, it’s optimal to have a worm that uses a dynamic or distributed command-and-control channel, which is difficult to implement when considering Kerchoff’s principle [16]. The principle would state that the defenders have access to the client side worm code so nothing in the worm code can be considered secret. The defenders would then know what the C&C host is or how the worm would deduce it. Given that, how could the worm author keep the dynamic C&C hosts anonymous?

Another challenge to C&C is poisoning. If a worm is taking its commands from a malicious host, it is usually possible to poison or disrupt the infected client’s channel to the malicious host. This technique has been used against the Storm worm and other viruses in the past [25].

3.1.2 Future worms will contain more subtle payloads

The payloads seen in previous worms were extremely noisy. The Samy worm altered the profile and “Heroes” list of every user it infected. Obviously, leaving visible cues to users that their profile has been altered is going to dramatically reduce the time-to-discovery of malicious code, and evading detection was not one of Samy’s concerns.

A more subtle payload would have been instructions that added the infection code to the user’s profile page and left everything else alone. A more subtle payload in the

Yamanner worm would have created an invisible `<iframe>` that pointed to the click-through adware. This would have allowed the worm to operate without obvious signs of abnormal behavior.

3.1.3 Polymorphic payloads will dramatically increase the longevity and stealth of a worm

There has been a lot of research into polymorphic JavaScript code. The idea of polymorphic attack code in general is very old. For our purposes, it would be useful to have JavaScript that is so different from one victim to the next that it would be difficult to create a pattern-based signature to identify the instances of the worm. This functionality implies the use of a non-deterministic JavaScript obfuscator, which most closely maps to the research in Gareth Heyes' Hackvertor [10].

Although Heyes' work on Hackvertor clearly demonstrates the ability to highly obfuscate JavaScript, it suffers from two main weaknesses. The first weakness is that keywords still appear in a Hackvertor payload. A keyword scanner looking for `document.cookie` in user-supplied code could still "detect an attack" that's been obfuscated.

Also, the Hackvertor function is deterministic. What is needed for a polymorphic payload to be truly difficult to signature is an algorithm that is not possible to de-obfuscate. By combining random encoding, expression types, string fragmenting and ordering, a payload can be made invisible by most signaturing techniques. These techniques are discussed in more detail in [2].

It is obviously possible that a polymorphic algorithm could perform these techniques. At this point the issue is an exercise in tough engineering to implement them, and naturally the best place to start would be with the already strong Hackvertor engine.

However, assuming that a human would always be the best candidate to create a small set of similarly-acting-but-differently-looking JavaScript infection codes, it would be effective to manually create a dozen or so "seeding" payloads which serve as the "roots" of the worm. That way, even if the application does a 100% successful search-and-destroy operation on the payload and its morphed copies, they will still not find the hand-crafted dozen "seeding" payloads which can continue infecting (if just to distribute malware once the XSS vulnerability is closed).

3.1.4 Victim browsers become distributed vulnerability scanners

One of the major limitations of XSS worms today is the fact that they are limited to the site on which they are born. Because of the same origin policy and the complexity involved in delivering complex attacks to other sites that involve reading responses (nonces, CAPTCHAs, etc.), a domain crossing worm has not yet been seen. Same-domain worms have a hard maximum number of infections: the number of users of the target application.

This is obviously not ideal for attackers who would like a continuously, outwardly spreading worm that is not trying to attain coverage over a certain site's user base, but is trying instead to penetrate and infect multiple user bases (sites).

Web application vulnerability scanners have been written in JavaScript by Billy Hoffman and others independently [13]. Although the effectiveness of these tools is hampered by the limitations of the browser, they both demonstrate a basic capability of scanning remote web applications using intermediary sources.

In order for a cross-domain worm to propagate, the scanning code must either execute on the client or the server. Forcing the server to execute the code is not possible since XSS is an attack that affects the client, although PHP remote file includes or similar exploits may allow this kind of server-side propagation to happen.

The fundamental problem in Web 1.0 with trying to make a JavaScript scanner execute in the clients browser: running time. The JavaScript will only execute as long as the user is on the page and allowing the script to run. If a script is running for too long the browser will prompt the user to stop the script or allow execution to continue, which will immediately trigger the user into navigating away from the page. Even if timers are implemented to avoid the script message from the browser, the JavaScript will run until the user closes their browser or navigates to a new page. One could lure the user into staying while reading content or playing a Flash game, but user response to the "bait" will be varied and is generally unpredictable.

GoogleGears WorkerPool allows JavaScript to spawn "worker threads" that execute asynchronously in the "background" of a web page. These threads live past when the browser navigates to the next page and only die when the browser closes. Therefore the time that a compromised browser can scan is the lifetime of the browser instance rather than the length of time that the browser remains on the page that's been injected.

Unfortunately, this type of client-side scanning tool is mostly academic because malware installed on the operating system could perform the same vulnerability scanning without any browser limitations. Therefore, vulnerability scanning malware in combination with XSS-based delivery is the key to cross-domain propagation. However, it should be noted that no browser vulnerability is required to use GoogleGears WorkerPool or any other RIA functionality.

With all the new RIA frameworks coming to market, each new browser capability will probably introduce some security implication that the framework authors did not intend. Because of this, a close eye should be kept on the direction of rich functionality that the frameworks produce as each one will have security implications.

3.1.5 Future worms will remain dormant during propagation

The widespread propagation of an XSS worm by itself is not what's likely to lead to its discovery. The payload and the resulting effects on the user experience are what usually prompt attack discovery.

Therefore, it would make sense for a slow moving worm to quietly propagate until its desired virus density or infection total reach a desired goal before activating its payload. Cross-domain worms, as mentioned before, are likely to be slow moving due to the difficulty in finding stored XSS through which to propagate, and a dormancy period only makes sense. Dormancy has been seen in previous Internet worms [19], but not around XSS.

It should be noted, however, that propagation in XSS worms is extremely quick when compared to traditional Internet worms. Therefore, the worm author of the future may choose to write a worm that utilizes no dormancy, opting to spread as deliver attacks as quickly as possible in order to decrease the window of time when the payload can be noticed casually.

3.1.6 Targets will change

The most sought after targets of layer 7 attackers in Web 2.0 will be content distribution channels. Content distribution channels were available and valuable in Web 1.0, but amount and type of these channels has increased dramatically in Web 2.0, and the capability of those channels has also been greatly enhanced.

Most websites that are compromised are compromised through network layer attacks, such as attacks against their *ftp*, *mail*, or *ssh* servers. These are traditionally C-based exploits that give the attacker shell access on the system, which the attacker then uses to either deface the website, attack the website’s intranet, or retain as a zombie host for spamming or denial of service attacks. However, the profit that could be gained from compromising a website is becoming known among attackers as it has been known in the spamming community for a long time- as has been demonstrated by the recent hack of Al Gore’s “Inconvenient Truth” website.

The Yamanner worm opened up a new window on the client’s browser to a static click-through site and harvested the victim’s email addresses to a remote site.

Search engine optimization is extremely important to spammers, as spam-filtering tools look at the relevance of links in emails as a way of detecting spam. By having a legitimate site, i.e., Al Gore’s documentary website, have links to the spammer’s site, it lends legitimacy to the link the spammers send out in their emails according to the spam filters because the spam filters rely on search engines to decide the legitimacy of a link [6]**.**

Advertising channels have long been target of attackers, and have also themselves been attackers to client browsers with malware and adware installations. The value of the advertising channels will remain fairly high because of the level of control over browsers they can exert, even without attacking them with traditional browser exploits.

RSS and aggregator feeds in general are excellent content distribution channels. RSS channels are usually insecure (not over SSL) and are thus vulnerable to tampering, but no such attacks have been seen in the wild. If an attacker near the RSS source were able to hijack the communication channel, an attacker could deliver their malicious payload to a massive number of subscribing users.

Mashup data feeds are also traditionally not over SSL. The feature SSL is reserved for the most is confidentiality, and mashup data is usually not sensitive, so no use of SSL is generally warranted. Also, speed is a key factor in delivering a successful mashup, and SSL does introduce speed issues. However, an attacker could as easily hijack an RSS delivery as they could hijack a mashup data feed. Alternatively, the attackers could attempt to infiltrate the mashup data feed’s application and attempt to alter the channel’s messages from inside using CSRF, XSS or some other vulnerability as was tried in the attack against Robert Hansen’s blog [20].

3.2 The Next Great XSS Worm: A Case Study

What follows is a proposed dynamic C&C channel design which provides poison-resistance, stealth, anonymity using Google, remote scripting and code signing.

Let assume that there are a number of users on a social networking site, www.myfacenovel.com. An attacker has found a persistent XSS vulnerability on the profile submission which would force other users to execute the attacker's arbitrary JavaScript when viewing the attacker's page. The JavaScript would then perform the following steps:

1. Import the Google Ajax JSON API
2. Generate a token based on the current date and some pre-defined string (i.e., var msg = Base64Encode("arshan_worm@<Today's Date>"))
3. Invoke the Google Ajax JSON API with remote scripting to find pages on the Internet that have that token on them
4. Iterate through results until one is found that finds a correctly tokenized message
5. If none is found, then simply copy the infection code to your own profile (like the Samy worm)
6. If one is found, tokenize the message into 3 parts: payload code, infection code, and digital signature
7. If the digital signature is valid according to the public key stored in the client's JavaScript, then execute the payload and store the new infection code in the user's profile

This technique is resistant to poisoning for a few reasons. Defenders can't prevent access to the sites which host the worm operator's message because the sites are distributed over the Internet, and the sites the worm goes after will change every day. Working with a real person to remove access to a malicious page does not scale when the attacker can store thousands or even millions of copies of the message on the web.

Also, defenders cannot forge messages from the author because they do not have the private key of the worm author so the worm will not execute their message when it fails the digital signature.

If Google was willing to take part in defending a high profile worm, it could signature the queries the infected hosts send in and not return results, but the attacker could get the same information from any search engine that had remote scripting libraries.

Also, the lag time between indexes both helps and hurts attackers. It hinders attackers because it forces them to deploy the distributed payloads well before the actual deployment of the worm to allow Google time to index the sites where they're hosted.

The lag time also helps attackers, though. Defenders attempting to interfere with the results of Google queries would have to wait until their sites were indexed as well, although it is likely that they would have the SEO ranking to speed up indexing.

Regardless, it seems possible that an attacker could cloak their malicious lookup queries inside "normal looking" queries using some language tricks and steganography.

In summary, the C&C method described previously exhibits the following qualities:

- The payload can't be stopped on the client side without NoScript [17] or an equivalent protection because the server can't enforce any client side policies in respect to remote scripting. This is compounded by the fact that the host they are on is likely to be whitelisted
- The C&C messages are distributed throughout the Internet, so removing access to one copy will not prevent the infection code from retrieving it
- The C&C channel can't be poisoned because the infected hosts only execute code that has been signed by the worm operator

3.3 Next Generation Defense Mechanisms

The good news is that there is a lot of room for improvement as far as defensive techniques in Web 2.0. This means better defensive techniques in our application security on the server side, smarter browser security and better response techniques when dealing with an already infected application.

3.3.1 Web 2.0 Isolationism

A major pitfall in the understanding of Web 2.0 security is the fact that it is the browser's support of Web 2.0 capabilities that makes the situation worse – isolationism from Web 2.0 is not a security solution. Simply not implementing any server side Web 2.0 features does not protect an application against a Web 1.0 exploit that leverages the victim's browser's Web 2.0 functionality. If I can execute XSS against a victim, I can use that victim's browser to execute a Web 2.0 payload. In this sense, Web 2.0 browser improvements make security "worse".

3.3.2 Effective Rich Input Validation

A major piece of the Web 2.0 defensive puzzle is rich input validation. The rising prevalence of mobile rich input leads to the need for stronger rich input validation. The OWASP AntiSamy project attempts to solve the problem of validating rich input by using transformation, policy-based validation and reliable serialization. This API allows users to safely upload rich content without exposing the application to phishing or XSS attacks.

3.3.3 Content Restrictions

The browser vendors can help developers prevent XSS and phishing attacks by implementing content restrictions, an idea proposed by Robert Hansen [9]. Content restrictions are essentially a sandbox that allows developers to specify policies that prevent any user-supplied markup from executing any unauthorized code. Exactly how content restrictions could be implemented has been debated.

It been proposed that content restrictions be implemented as an HTTP header delivered by the server to the client browser, informing it of what capabilities the client browser will need to use the resulting page in the way the application intended.

This approach is not prone to success because it most web pages, especially in Web 2.0, use all the same capabilities an attacker would use in a realistic exploit. The policy dictating what functionality is allowed to be invoked for the whole page would end up being far too permissive. If a page uses `XmlHttpRequest` and the use of the `innerHTML` property, the policy referenced by the header would have to allow the usage of those capabilities. This analysis leads us to conclude that any content restriction implementation that forces the developer to apply policy across a whole page is prone to failure, regardless of how that policy is communicated to the user (in a header or in the page somewhere).

In a more simple form, content restrictions could be implemented as a tag-based jail that prevents any JavaScript from being fired. The simplest content restriction would be something found in Figure 1.

```
  <jail>

<!-- dangerous JavaScript here -->

  </jail>
```

Fig. 1. A very naïve approach to tag-based content restrictions.

Unfortunately, the user input could contain the end tag of the jail itself, telling the browser that the jailed section is finished. This effectively terminates the jail and allows the user to supply unhindered code after the end tag. An example exploit could be seen in Figure 2.

```
</jail>
<script>alert(‘hi’)</script><jail>
```

Fig. 2. An example attack that would prematurely close the jail tag an insert a malicious script.

To prevent this, the jail needs some “secret” to prevent the attacker from being able to finish the jail tag prematurely. A suggestion previously made was to contain the secret in both the opening and closing tags as seen in Figure 3.

```
  <jail secret=”ZXcOQW#iht*”>

  <!-- dangerous script here -->

  </jail secret=”ZXcOQW#iht”>
```

Fig. 3. A content-restricting jail with the secret in the open tag and close tag.

This solution would work, but is not optimal because it violates SGML standards, although Internet Explorer has honored some attributes in end tags in the past. In this paper I am proposing an alternative to the jail tag idea that requires much less work than the jail just discussed. An example of the jail can be found in Figure 4.

```
  <start-jail
secret=”zxQ#W0rwef8H”/>

<!-- dangerous JavaScript here -->

  <end-jail secret=”zxQ#W0rwef8H”/>
```

Fig. 4. A content-restricting jail (with secret) consisting of differing open start and end elements.

This approach, which I’m calling “jail pairs”, has two advantages over previous jail ideas because it does not require browsers to honor non-standard code, and it is valid markup according to most parsers. Unfortunately, its usage is counter-intuitive to normal SGML markup, but what it lacks in this concern it makes up for in ease of implementation as it does not require the browsers or parsers to honor attributes in end tags.

It’s worth noting that none of the content restriction implementations right now address presentation layer attacks. Presentation layer attacks are code injection attacks that involve altering the UI without invoking JavaScript, at least directly. Given the XSS pandemic seen in the wild right now, the prevention of XSS (through the prevention of unauthorized JavaScript) is a good start. For more information on presentation layer attacks, see [12] [1].

Preventing presentation layer attackers is a challenging issue for browsers because it would involve teaching the content restricting code some visual context. The validation code would have to recognize that the user-supplied code is attempting to clobber a div that is located outside the user’s jail, or that the dimensions and `z-index` on the user-introduced `<div>` would end up overlaying other sections of the page. Given these challenges, it is extremely unlikely that the first implementation of content restrictions will even attempt to solve this problem.

3.3.4 Technology Switch

An expensive alternative in avoiding most of the Web 2.0 issues is migrating to a client side technology that does not haphazardly mix up code and data. Although it’s not a realistic solution for everyone because of compatibility and user expectation reasons, it should be noted for comprehensiveness that a subset of the web-related vulnerabilities made worse in Web 2.0 can be sidestepped entirely by using an alternative client side technology. For example, Java applets are delivered as bytecode, which is not injectable and is therefore less susceptible to cross-user attacks.

Although all the traditional client-to-server attacks would still exist, most peer-to-peer attacks involve tricking peers into executing/viewing rich content. However, there are generic, peer-to-peer application security threats such as cookie sniffing that would still exist.

3.3.5 Utilizing cross-domain workflows

Web applications can host user content in off-domain pages to prevent cross-domain pollution. Web applications can host un-trusted user content on their application's UI in an `<iframe>` that is hosted on a separate sub-domain than the rest of the application (e.g., utnrusted.socialnet.com). If a user were able to store attacks in their profile, the attacks would be hampered by the same origin policy which would limit the user from interacting with the only domain that application data is accessible from (www.socialnet.com). This technique prevents loss of application data, but it will still be vulnerable to XSS, which can lead to remote browser takeover or malware installation, phishing, and most other attacks.

However, there are many negative consequences to running an off-domain `<iframe>` in a page. Search engines punish pages that use `<iframe>` because they typically represent less reputable web sites. Also `<iframe>` usage in general is slow and tends to promote accessibility issues. Robert Hansen and has spent a lot of time researching this particular issue and found that the business cost of this solution in loss of accessibility, functionality and search engine optimization is too great to implement [18].

Hansen's has described a solution that is a slight departure from this technique, but affords the application protection from negative search engines policies, and it can be found in [7].

3.3.6 Character frequency analysis

Character frequency analysis could easily be used to detect most payloads. Although typical user data may content rich content and thus have a different distribution of characters on average than plain language, some "normal" baseline of characters could be used to detect anomalous data among users' rich input, infected or otherwise. However, a worm that has been designed to morph while still retaining the character frequencies of normal data could be developed in response.

3.4 Techniques for worm removal

Thinking about how to remove a worm discussed in the paper leads to interesting avenues of research in preventing malicious code already in an application from functioning according to the author's intent.

3.4.1 Signaturing Polymorphic JavaScript

Coming up with reliable signatures for polymorphic JavaScript payload come will be important for performing search-and-destroy operations on a worm that has fully infected an application.

Without effective search-and-destroy, a worm that can't propagate may still be able to deliver malware, perform phishing attacks or execute other nefarious operations.

3.4.2 C&C channel poisoning

C&C channel poisoning is an effective way of dealing with distributed malicious agents. Although this paper has shown that a piece of JavaScript can communicate through an intermediary in a way that is resistant to C&C channel poisoning, no XSS worms have implemented this type of functionality yet.

3.4.3 Exploit Egress Filter

An infected site can quickly stop the propagation of an XSS worm without fixing the XSS vulnerability that is used to deliver the payload by implementing an exploit egress filter. If an XSS worm with a static payload has infected a site, the site can setup an egress filter (outbound response filter) that prevents the malicious payload from being delivered to users.

The good news is that this can be done in a relatively transparent way to the user. Because of JavaScript's asynchronous nature, there are many places where execution flow is transferred based on some event taking place. Consider an example of some payload code in Figure 5.

```
xhr.onreadystatechange=handleIt;
```

Fig. 5. A code snippet from an XHR call inside an attack payload.

In this snippet, the attacker is telling the JavaScript VM that when the state of the `XmlHttpRequest` communication changes, it should invoke the `handleIt()` function that was part of their payload. If for some reason the `handleIt()` method never gets called, the rest of the payload will simply never execute.

In this case, an effective egress filter would simply prepend the string "//" before any place the filter finds the exploit, as shown in Figure 6.

```
//xhr.onreadystatechange=handleIt;
```

Fig. 6. The attack code snippet modified by an HTTP egress filter.

This type of mechanism requires no source code update and can be hot deployed in an HTTP filter or web application firewall.

3.4.4 JIT Anomaly-Based Profiling

One option for browser protection in the future could be JIT JavaScript profiling to prevent anomalous behavior. More simply, if the browser detects that AspectSecurity.com doing something that AspectSecurity.com doesn't normally do, it could simply refuse to perform the operation.

Such anomaly-based execution prevention has been deployed in host-based IPS systems in the past [4]. Unfortunately, such a solution may not be realistic as most sites use the functionality that attackers would leverage when using XSS attacks, making profiling a legitimate application very hard to do effectively.

4. Conclusions

It has been shown that the individual pieces of a truly subtle, highly evasive and far-reaching XSS worm can be created with the increased client side capability available to attackers in Web 2.0 capable browsers. The effects of a worm that implements the functionality discussed will undoubtedly serve as a wake-up call for web developers and managers who still do not consider security seriously. Unfortunately, billions of dollars in damages will result as companies will ineffectually pressure anti-virus vendors to quickly learn layer-7 security to deal with the issue. Unfortunately, this is not the domain of expertise for anti-virus vendors, so response will be slower than traditional Internet worms.

In next generation worms, the attacker can now use the Internet as an API thanks to the accessibility and ease-of-publishing of distribution channels in Web 2.0.

Tangentially, a trend that is already developing in attacks is the push towards compromise of content distribution channels for search engine optimization, malware distribution or XSS delivery. This trend should continue as attackers realize the profit to be made from their skill is easily made in infecting end-users en masse.

However, it has been shown that using agile defense techniques, a XSS worm that has already been deployed may be able to be stopped on a single infected application with an egress filter that is deployed quickly, or by poisoning the C&C channel by which the worm is controlled.

Web application developers can help secure their own application by utilizing user input workflows that cross domains and use nonces to prevent XSS propagation. Also, peer-reviewed rich input validation mechanisms like OWASP's AntiSamy project can be used to filter HTML/CSS according to some business policy in order to prevent any XSS vulnerability in the first place.

Browsers can also improve the state of security by implementing content restrictions that are easy to use by developers, optional to implement in an application, and does not require major changes to the HTTP response.

References

1. Dabirsiaghi, A. January 5, 2008. HTML/CSS Injections – Primitive Malicious Code. *omg.wtf.bbq*. Retrieved February 25, 2008 from http://i8jesus.com/?p=10.
2. Dabirsiaghi, A. February 25, 2008. Improving Hackvertor: Polymorphic JavaScript Payloads. *omg.wtf.bbq*. Retrieved February 26, 2008 from http://i8jesus.com/?p=15.
3. F-Secure Corporation, December, 2003. *F-Secure Corporation's Data Security Summary for 2003*. Retrieved February 25, 2008 from http://www.f-secure.com/2003/.
4. Gong, F. March, 2003. Deciphering Detection Techniques. *Anomaly-Based Intrusion Detection*. Retrieved February 25, 2008 from http://www.mcafee.com/us/local_content/white_papers/wp_ddt_anomaly.pdf.
5. Grossman, J. April, 2006. Cross-Site Scripting Worms and Viruses. *WhiteHat*. Retrieved February 25, 2008, from http://www.net-security.org/dl/articles/WHXSSThreats.pdf.
6. Grossman, J. November 27, 2007. Inconvenient Truth blog, SE0wN3d!!1. Retrieved February 27, 2008 from http://jeremiahgrossman.blogspot.com/2007/11/inconvenient-truth-blog-se0wn3d1.html.
7. Hansen, R. XSS Worm Analysis and Defense. *ha.ckers.org*. Retrieved February 25, 2008, from http://ha.ckers.org/xss-worms/.
8. Hansen, R. et. al. Creating and Combating the Ultimate XSS Worm. *sla.ckers.org*. Retrieved February 25, 2008 from http://sla.ckers.org/forum/read.php?2,19143.
9. Hansen, R. June 1, 2006. Content restrictions and XSS. *ha.ckers.org*. Retrieved February 29, 2008 from http://ha.ckers.org/blog/20060601/content-restrictions-and-xss/.
10. Heyes, G. January 21, 2008. Code Morphing. *The Spanner*. Retrieved February 25, 2008 from http://www.businessinfo.co.uk/labs/morph/morph.php.
11. Higgins, K. December 19, 2007. Google's Orkut Social Network Hacked. *Dark Reading*. Retrieved February 25, 2008 from http://www.darkreading.com/document.asp?doc_id=141761&WT.svl=news1_2.
12. Hoffman, B. and Sullivan, B. *Ajax Security*. Addison-Wesley, 2007.
13. Hoffman, B. April 2, 2007. Jikto in the wild. *The HP Security Laboratory*. Retrieved February 27 from http://portal.spidynamics.com/blogs/spilabs/archive/2007/04/02/Jikto-in-the-wild.aspx.
14. Jackson, C., Barth, A., Bortz, A., Shao, W., Boneh, D. Protecting Browsers from DNS Rebinding Attacks. Retrieved February 25, 2008 from http://crypto.stanford.edu/dns/dns-rebinding.pdf.
15. Kaplan, D. December 5, 2007. Duke University Law School Infiltrated by Hackers. *SC Magazine*. Retrieved February 25, 2008 from http://www.scmagazineus.com/Duke-University-Law-School-website-infiltrated-by-hackers/article/99613/.
16. Kerckhoffs, A. 1883. La Cryptographie Militaire. *Journal Des Sciences Militaires, IX, 5-83, 161-191*.

17. Maone, G. *NoScript – JavaScript/Java/Flash blocker for a safer Firefox experience!* Retrieved February 25, 2008 from http://noscript.net/.
18. Markham, G. February 24, 2005. Auto-Sizing IFRAMEs? *Hacking for Christ*. Retrieved February 25, 2008 from http://weblogs.mozillazine.org/gerv/archives/007610.html.
19. Rhodes, K. August 29, 2001. Code Red, Code Red II, and SirCam Attacks Highlight Need for Proactive Measures. *United States General Accounting Office*. Retrieved February 25, 2008 from http://www.gao.gov/new.items/d011073t.pdf.
20. Sirdarkcat. November 8, 2007. Inside History of hacking rsnake for fun and pagerank. *SIRDARKCAT: Security and Programming Blog*. Retrieved February 25, 2008 from http://sirdarckcat.blogspot.com/2007/11/inside-history-of-hacking-rsnake-for.html.
21. Sutton, Michael. December 31, 2006. Web Application Security Statistics. *Web Application Security Consortium*. Retrieved February 25, 2008 from http://www.webappsec.org/projects/statistics.
22. Unknown. *Spam Mimic* .Retrieved February 25, 2008, from http://www.spammimic.com.
23. Valotta, R. *Nduja Connection*. Retrieved February 25, 2008 from http://rosario.valotta.googlepages.com/home.
24. Veness, C. SHA-1 Cryptographic Hash Algorithm. *Movable Type Scripts*. Retrieved February 25, 2008 from http://www.movable-type.co.uk/scripts/sha1.html.
25. Zhou, Y., Cui X., Wu, B. Worm Poisoning Technology and Application. *CNCERT/CC*. Retrieved February 27, 2008 from http://www.first.org/conference/2006/papers/xiang-cui-papers.pdf.
26. Spafford, E. The Internet Worm Program: An Analysis. Purdue Technical Report CSD-TR-823. Department of Computer Sciences, Purdue University. Retrieved February 27, 2008 from http://homes.cerias.purdue.edu/~spaf/tech-reps/823.pdf.