Introduction to Web Application Security Flaws

Jake Edge
LWN.net
jake@lwn.net

Abstract

You hear the names of the most common web security problems frequently: cross-site scripting, SQL injection, cross-site request forgery—but what do those terms mean? This paper will provide an introduction to those vulnerabilities along with examples and ways to avoid them. This introduction is language-independent, as the problems can occur in any language used to develop web applications.

Developers of web applications sometimes get caught up in the excitement of developing the application and forget to consider the security implications. This paper will help them get a handle on what to avoid so that the excitement doesn't get squashed by an attacker. Others who are curious about the kinds of attacks made against web applications will also find much of interest.

1 Introduction

Web application vulnerabilities make up a fairly large slice of security vulnerabilities reported on Bugtraq and other security mailing lists. In addition, they are probably the type of vulnerability that Linux users are most likely to come across.

The consequences of a web vulnerability vary greatly, from full compromise of a vulnerable server application—potentially the server machine itself as well—to stealing authentication information so that a user's account, often on an unrelated site, can be accessed by an attacker. This highlights the broad reach of web application vulnerabilities as they can affect particular sites *or* the users who visit them.

2 Hypertext Transfer Protocol (HTTP)

Hypertext Transfer Protocol (HTTP) is the language spoken by web applications. It is a fairly simple, text-oriented protocol that is easy to read and understand. A web browser sends an HTTP request and awaits a response from the server. That response is generally text in Hypertext Markup Language (HTML), but can also be other types of data: images, audio, video, etc. The browser then displays the response from the server and awaits another user action (e.g. clicking a link, submitting a FORM, using a bookmark to go elsewhere, etc.).

The most common HTTP requests are of the following three types:

- **HEAD** – this retrieves the headers and dates associated with a page so the browser can determine whether it can use its cached version of the object (HTML, image, etc.).
- **GET** – This is the workhorse of HTTP. Retrieve content based on a URL, with parameters passed as part of the URL (e.g. `http://foo.com/bar?baz=42`).
- **POST** – This is used by FORMs. The parameters are encoded into the request and POSTed to a specific URL, which is specified in the FORM tag.

Figure 1 shows a short example of using `telnet` to talk to a web server. The "GET" and "Host:" lines are typed by the user (followed by two carriage returns) with the response from the server following. The headers are sent first followed by a blank line and then the content, in this case the HTML of the document (abbreviated in the figure).

There is an important distinction between GET and POST that web application programmers should be aware of. GET requests should be *idempotent*, that is they should not change the state of the application. Multiple GET requests should return the same content, unless the underlying state of the application has been changed via a POST request. It is common for web

HTTP EXAMPLE

```
$ telnet lwn.net 80
Connected to lwn.net.

GET /talks/ols2008/ HTTP/1.1
Host: lwn.net

HTTP/1.1 200 OK
Date: Mon, 28 Apr 2008 03:54:40 GMT
Server: Apache
Last-Modified: Mon, 28 Apr 2008 03:49:48 GMT
ETag: "428105-30c-4815495c"
Accept-Ranges: bytes
Content-Length: 780
Content-Type: text/html

<html>
<head>
<title>OLS 2008</title>
</head>
...
```

Figure 1: Example of HTTP using telnet

applications to have state-changing links (which correspond to GETs), though there are two good reasons not to do that.

One classic example is a web page with links to delete content that looked something like: `http://somehost.com/delete?id=4`. A web spider then came along to index the site and found all of these links to follow—promptly deleting all the content on the site. The other reason to avoid state-changing GET requests is to prevent trivial cross-site request forgery as will be described below.

Another important thing to note about HTTP is that it is a stateless protocol. There is no inherent state information kept by the server and client. Each request is an entity unto itself. Various mechanisms have been used to achieve stateful web applications, the most common is the idea of a *session*. Sessions are typically set up by the server, given some kind of identification number (i.e., session ID), and then set as a *cookie* in the user's browser in the response from the server. Cookies are persistent values, associated with a particular website, that are sent by a browser whenever a request is made of that website.

3 User input cannot be trusted

Many web application vulnerabilities share a fairly simple characteristic: insufficient or incorrect filtering of user-controlled input. This input can come as part of the URL, from FORM data, or from cookie values. These inputs make up most, if not all, of the attack surface of an application and *must* be appropriately filtered before use. The filtering should use a whitelisting, rather than blacklisting, approach—only allowing known-good inputs is far safer than trying to construct a list of all "bad" inputs.

One common mistake that web application programmers make is to assume that all traffic generated to their program will come from a web browser. They assume that certain things are "impossible" because a web browser does not do it that way. This is a grievous error, as it is trivial to generate HTTP traffic from any programming language—many provide full-featured libraries to do just that. An attacker can use a browser and easily manipulate parameters passed as part of the URL in a GET request, but using FORMs is no protection. Generating a POST request with the appropriate parameters is a simple task that is often done in Javascript as part of an exploit. Cookie values can also be created or stolen from another user.

Perhaps the most common manifestation of this mistake is in using Javascript validation. Web application programmers will often write some Javascript to run on the browser to validate values typed into a form. For example, a form might have a place to type in an IP address, with Javascript that ensured the values were legitimate—integers in the right range—popping up an alert box if the values were not valid. This may help some users and is a reasonable thing for a web application to do. The mistake is in not doing the same validation on the server. *Any* validation done by Javascript needs to be repeated on the server side. There is no guarantee that the user has Javascript enabled—even if the application tries to force it—or even that it is a browser at the other end. A program can easily be written to submit any value of any kind for that parameter. Browser-based limitations on length or type of a field in a form are *not* enforced if the browser is not present.

4 Cross-site scripting (XSS)

One of the more common vulnerabilites seen for web applications is *cross-site scripting* (XSS[1]). XSS results from taking user input and echoing it back to the browser without properly filtering it for HTML elements. Many web applications allow users to store some content, a comment on a story for example, in a database on the server. This content is then sent back to the browser for that user or others as appropriate for the application. Consider the following "content" `<script>alert("XSS")</script>`. If that is sent to the browser unchanged, it will cause Javascript to pop up an alert.

Any user input that gets sent back to the browser is a potential XSS vector. One common mistake is for an error message to contain the unrecognized input, which is helpful for a legitimate user who made a mistake, but can also be used as part of an XSS attack. Typically, XSS vulnerabilities are described with a proof-of-concept that just uses a Javascript alert which tends to make some underestimate the power of XSS. It is important to note that XSS attacks are capable of anything Javascript can do, which is a *lot*.[2] One of the more common uses of XSS is to steal cookie information from the browser which can then be used for session hijacking or other attacks.

There are two major flavors of XSS, the non-persistent XSS, where the attack comes as part of the link—like the error message example above—and persistent XSS, where the attacker stores something persistently on the server that can attack each time that content is accessed. Both flavors can have serious effects, but the amount of malicious code that can be contained in a link is somewhat limited, whereas the database will happily store a great deal more. Vulnerable applications may be storing page contents for MySpace or Facebook-like uses, comments on a blog posting, or some other lengthy content, any of which may be effectively unlimited in size.

The only defense against XSS is to filter all user input before echoing it back to the browser. Table 1 shows the recommended filtering rules. Mapping each listed character to its HTML entity equivalent will prevent XSS.

Input Character	Output HTML Entity
<	&lt;
>	&gt;
(	&#40;
)	&#41;
&	&amp;
#	&#35;

Table 1: Character mapping for HTML entities

Depending on the implementation language, there may be functions (like `htmlentities()` or `cgi.escape()`) that do some or all of the filtering job. Note that some implementations may not do all of the recommended transformations, which could possibly lead to an XSS hole.

5 Cross-site request forgery (XSRF or CSRF)

Another type of web vulnerability—in some ways related to XSS—is *cross-site request forgery* (CSRF or XSRF[3]). XSRF abuses the trust that a web site has in the user, typically in the form of cookies, to cause an action on that site from an unrelated site.

To see how this works, consider a state-changing GET on a particular web site, perhaps one for a broadband router. If a particular URL on the site, `http://router/config?setDNS=1.2.3.4` for example, will change the router's DNS setting, a completely unrelated attack site could use that URL in an image tag: `<img src="http://router/config?setDNS=1.2.3.4">`. This would cause a request to be made of the router *with* any cookie the browser has stored for the router. If the cookie was used for authentication and had not yet expired, the action would be performed.

It is not just GETs that can be attacked via XSRF, POSTs are vulnerable as well. Using Javascript—and often hidden away in an IFRAME—FORMs can be constructed and submitted with values under the control of an attacker. A user could be lured to the attack site via a link in an email or other web page. Once they arrive, the attack site could generate a FORM submission to a

[1] For web abbreviations, CSS was already taken by Cascading Style Sheets.

[2] For some examples, including network scanning behind firewalls, stealing web browser history, and more, see `http://jeremiahgrossman.blogspot.com/2006/07/my-black-hat-usa-2006-presentation.html`.

[3] For consistency with XSS, this paper will use XSRF.

Username: `x`
Password: `' OR 1=1; --`
Query: `SELECT id FROM users WHERE name='$name' AND password='$pword'`
Results in: `SELECT id FROM users WHERE name='x' AND password=''` **`OR 1=1; --'`**

Figure 2: SQL injection example

```
$stmt = prepare("SELECT id FROM users WHERE name=? AND password=?")
execute($stmt, $name, $pword)
```

Figure 3: SQL prepared statement and placeholder example

completely unrelated site—a banking, stock trading, or shopping site for example—which would be transmitted along with any cookies the user has for the site. If the user has recently logged in, the form action will be taken, just as if the user visited the site and filled in the form that way.

For high-profile sites, the URL is easy to know, but even for local devices like the broadband router mentioned above, the URL can often be guessed. It is very common for a particular model of router to be handed out en masse to subscribers of a particular service—often by default their IP address is fixed. So an attacker that wanted to do a phishing scam might choose a vulnerable router type and try to do XSRF attacks to 192.168.1.1, for example.

Getting rid of XSRF holes is somewhat complicated. First, as described above, state-changing GETs must be eliminated from the site. These are clearly the easist XSRF hole to exploit.

For FORMs, there needs to be something that cannot be reliably predicted—or brute forced—in the FORM data. The best way to do that is with a `TYPE=HIDDEN` FORM element that has unpredictable NAME and VALUE attributes. Each should be generated separately, be long enough to resist brute force, and be tracked on the server side associated with the session ID. Whenever a FORM is submitted, the NAME and VALUE of the element should be validated, with any action dependent on that validation.

It should be noted that XSS can provide a means for Javascript to read the FORM and gather the required information, so a site must be free of XSS issues or the avoidance mechanism above can be circumvented.

It should also be noted that for very sensitive operations, re-authenticating the user can provide absolute protection against XSRF—assuming a good password has been chosen. It is common for web applications to require the current password before changing to a new password, which is an example of this technique.

6 SQL injection

SQL injection is, after XSS, the most commonly reported web site application flaw. Because many web sites are backed by some kind of SQL database, there are a large number of applications that are potentially vulnerable. SQL injections can lead to data loss, data disclosure, authentication bypass, or other unpleasant effects.

SQL injection abuses the queries that the web site does as part of its normal operation by injecting additional SQL code—under the control of the attacker—into the query. It is usually done through parameters to GET or POST requests by taking advantage of naïve—or nonexistent—attempts to protect the query from the parameter values.

In Figure 2, there is an example of how a SQL injection can happen. The SQL query is generated by interpolating the FORM variables for username and password into the statement. Under normal circumstances, when a user is trying to log in, the SQL statement works fine to select an ID if the username/password matches someone in the database. If an attacker types in the **`' OR 1=1; --`**[4] string for the password, he modifies the query as shown. This has the effect of returning every row in the *users* table. Typically application code is written to just take the

[4] The "--" tells the SQL engine to ignore the rest of the query, similar to a comment.

first returned result in that case, which should be a valid user—and may in fact be the first user added, which is often the administrative user.

Depending on how the web application is structured and what database system is used, other abuses are possible. Some databases allow multiple statements separated by ";" so a password of **'; DROP TABLE users; --** would end up removing all users from the database. There are ways to use SQL injection to discover all of the tables in the database and their contents, again depending on the database system.[5]

When trying to work out how to create a SQL injection for a site, an attacker may need to try multiple different techniques. The error messages returned by the web application often make it easier to determine what needs to be added to the injection to make it work because they disclose what the problem is (i.e., "Missing parentheses," "Unterminated string," and the like). Even unhelpful error messages can give clues to an attacker if the application responds differently to well-formed SQL that uses correct table and column names versus illegal SQL. That difference can be exploited by a technique known as *blind SQL injection*.

Thankfully, there are straightforward ways to avoid all SQL injection attacks. Converting an existing codebase may be somewhat tedious and time-consuming, but the method is easy. Essentially all of the techniques boil down to having the database treat the user input as a single entity that is used in the proper place in the query—as opposed to textually substituting that text into a query string.

The overall best technique is to use *prepared statements* with *placeholders* for the values that are to be used in the query. Different database systems use different placeholder syntax—and various languages' database libraries obscure it more—but a common choice is the "?" character.

Queries are then created using the placeholder and passed to the database `prepare()` function. Figure 3 gives an example in a kind of pseudocode. Instead of textually substituting the `$name` and `$pword` variables into the query, the database system uses them internally to match. Doing it that way, the only way the query will return any results is if there is a user named `x` with the password `' OR 1=1; --`.

If the database (or language library) does not have placeholder support, strong consideration should be given to changing to one that does. If that is impossible, any database library should have some kind of support for a database-specific `quote()` function. This will take the user input and do whatever necessary to escape special characters in the input so that they can be used directly in the query string.

Stored procedures offer similar protections to prepared statements, but are set up in the database itself ahead of time. It is somewhat less flexible than just tossing a query in where needed, but will also handle parameters in a safe method.

7 Authentication bypass

Authentication bypass comes in various flavors, but at the core it is a way to circumvent logging in while still being able to perform privileged actions. Unlike cracking a password—which still uses the standard authentication mechanism—authentication bypass, as the name implies, circumvents the authentication method completely. It abuses some aspect of the application to "reach around" the requirement to be logged in.

While not truly an authentication bypass, default passwords that remain unchanged have essentially the same effect. If default passwords are for some reason required—and finding a way to not require them would be a better choice—it should be difficult to get very far with the application without changing the default.

Applications should be structured in such a way that it is impossible to view or submit a page that is privileged without also supplying the proper credentials. One common mistake is for an application to check the URL against a list of privileged URLs, requiring authenticated users for any that are on the list. This kind of testing can fall prey to *aliasing*.

Most web servers will allow multiple URLs to reach the same page, but those URLs can look quite different. A trivial example is `http://vulnsite/foo//bar` which is equivalent—in web server terms—to `http://vulnsite/foo/bar`, but is very different when

[5] Microsoft's SQL Server is said to be particularly susceptible to this.

matching the URL. By adding the extra slash, an attacker gets around the authentication requirement. Similar things can be done using HTML URL encoding (using `%2F` instead of `/`, for example).

Another common mistake is to assume that any links or forms that are only presented to the user after they have logged in are somehow protected. While in the normal course of events, the user has no access to those elements *through the application*, there is nothing stopping an attacker from using them. Some applications will use a separate program to process FORM submissions—without checking authentication—believing that because those FORM URLs are only presented post-login, they cannot be accessed otherwise. Both of these are a kind of "security through obscurity" that provides no protection at all.

It is imperative that the code for each page that requires authentication check for it before displaying or taking any action. If a separate program is used to process FORMs, it must also check authentication. No matter what kind of aliasing might be happening, the page code must be invoked, so that is the proper place for checking credentials.

8 Session hijacking

As described above, *sessions* are a standard way for adding state to HTTP. A session is assigned a particular ID that is stored in a cookie. Each HTTP request from the client is accompanied by the session ID, allowing the application to track a related series of requests. For applications that require authentication, the session stores the status of that authentication. This means that a valid session ID can be presented to the application by an attacker to hijack it.

This hijacking works only as long as the session is valid. Short-lived sessions—on the order of minutes—reduce the window of vulnerability, but can be annoying to users because they have to reauthenticate whenever the session times out. For sensitive web applications, it is worth the user annoyance.

An attacker can gain access to the values of a victim's cookies in a number of ways. If the application runs on an unencrypted connection, cookie values can be sniffed on the wire. XSS provides another means to get access to cookie values. Some web applications do not even use cookies—instead sending the session ID in the URL or a hidden FORM element—making it even easier to access them.

Some applications store IP address information in the session which is verified on each subsequent request. This technique is not particularly useful, as there are often lots of computers sharing a single IP address (e.g. NAT); also, some ISPs effectively assign a new IP on each request which would require the user to re-authenticate for each page accessed.

Re-authentication is an important safeguard for extremely sensitive operations. It can be annoying, but does protect against leaked session IDs.

9 Conclusion

The vulnerabilities presented are the most common flaws that are found in web applications. There are others, of course, but these should be at the top of any web application designer's list. Properly handling user input while ensuring that authentication is correctly implemented will go a long way towards securing these applications.

Modern web frameworks—like Ruby on Rails, Django, and others—often provide mechanisms to eliminate or seriously lessen these vulnerabilities. Quite a bit of thought has gone into the security model of these frameworks and there is typically an active security group maintaining them. For new web applications, it is worth looking into them so as to benefit from those protections.

Proceedings of the
Linux Symposium

Volume One

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,*
Thin Lines Mountaineering
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
Matthew Wilson, *rPath*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
Eugene Teo, *Red Hat, Inc.*
Kyle McMartin, *Red Hat, Inc.*
Jake Edge, *LWN.net*
Robyn Bergeron
Dave Boutcher, *IBM*
Mats Wichmann, *Intel*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.