All Your Base Are Belong To Us

(Primer on Lasso & WebAppSec)

Bil Corry
Lasso.Pro

Presented at:
Lasso Developer Conference
September 18-21, 2008
Chicago, Illinois, USA

INTRODUCTION

> *"How are you gentlemen !! All your base are belong to us. You are on the way to destruction. You have no chance to survive make your time. Ha Ha Ha Ha ...."*
>
> - CATS in Zero Wing

Web application security (aka WebAppSec) is one of many disciplines[1] a web developer must master in order to create secure, competent websites. This paper will discuss common vulnerabilities found in web applications and the methods you can use, programming in Lasso, to prevent them.

MONETIZING YOUR WEBSITE

Before we get to the "how" of WebAppSec, we're first going to cover the "why"; why is your website a target?

The short answer: money.

There is a belief among some that their little piece of the Internet is far too small and humble to be of any interest to cyber-criminals.[2] And logically, it may be confusing to understand how your local volunteer soccer league website could be of any value to cyber-criminals. Really, how can a site that has no personal information to steal, no financial information to steal, and simply functions to display the results of soccer games generate an income for cyber-criminals? Mostly likely not even the site owner is generating revenue from the site!

There are two forces that have converged that allow monetizing any website a reality: automated penetration tools and diversity of income streams. Automated penetration tools have made it so cyber-criminals do not have to individually and manually crack each website. Instead, they leverage automated tools that can crack tens of thousands of websites.[3] [4] And thanks to Google, they don't even have to crawl your web application, Google already has and can be used to target sites that match specific requirements and vulnerabilities.[5]

Once the website has been cracked, they can then use one or more strategies for monetizing the site. Here are some of the strategies they employ:

1. Link spam – you'll often see this in guest books and blog comments; link spam generates revenue by enticing your site users to visit the attacker's site (which sells products, has advertising, installs malware, etc). An interview with one such link spammer claimed he made over £100,000 per month![6] And the links also raise the relative value of the attacker's website in the eyes of the search engines, giving it a higher rank when doing searches.[7]
2. Hosting – free, anonymous hosting; cyber-criminals can use a cracked server to host their own content, including warez, malware, phishing sites, pirated content, etc.[8]

---

1 Top 10 Concepts That Every Software Engineer Should Know
http://www.readwriteweb.com/archives/top_10_concepts_that_every_software_engineer_should_know.php

2 Here's one example:
http://groups.google.com/group/google-safe-browsing-api/browse_thread/thread/950c15743e0a3c19/7d1f8600b53e1160

3 Hackers Hijack a Half-Million Sites
http://www.pcworld.com/businesscenter/article/145791/hackers_hijack_a_halfmillion_sites.html

4 Massive SQL Injection Attack 600.000++
http://www.0x000000.com/?i=556

5 Google Hacking Database (GHDB)
http://johnny.ihackstuff.com/ghdb.php

6 Interview with a link spammer
http://www.theregister.co.uk/2005/01/31/link_spamer_interview/

7 Spamdexing: Link Spam
http://en.wikipedia.org/wiki/Link_spam#Link_spam

8 Hacked bank server hosts phishing sites
http://www.computerworld.com/hardwaretopics/hardware/server/story/0,10801,109500,00.html

3. Infect users – your visitors represent an income stream by exploiting their computer and adding it to the attacker's botnet. Their computer can then be used to send spam, launch denial-of-service attacks, rented out to others, etc.[9] [10] [11] And your visitors' computer can also be searched for valuable data, and in combination with a keylogger, the attacker can steal personal information, drain bank accounts, and sell this information to others.[12] In fact, your visitor's computer is so valuable that botnet operators have been known to apply security patches and run defenses against malware to prevent other bot herders from taking over their bots.[13]

And should you decided to only allow certain IP addresses or registered users to interact with the site, know that if it sits on a shared server, it still may be vulnerable via a weakness in another webapp on the same server.[14]

GOLDEN RULE OF CLIENT INPUT

Now that you understand the "why" of WebAppSec, we're now going to delve into the "how" of specific web application vulnerabilities. As we do, keep in mind the golden rule of client input:

"All client input (headers and request) is hostile until proven otherwise or sanitized."

To understand how to prove client input isn't hostile, I'll first start with input validation, then follow it with discussions about four common web application vulnerabilities: SQL injection, cross-site scripting, cross-site request forgeries, and side-jacking. By the end of this paper, you should have a firm grasp of common web vulnerabilities.

9 Storm For Rent
http://www.forbes.com/technology/2008/01/09/storm-worm-cybercrime-tech-security-cx_ag_0109storm.html
10 Home PCs rented out in sabotage-for-hire racket
http://www.usatoday.com/tech/news/computersecurity/2004-07-07-zombie-pimps_x.htm
11 Know your Enemy: Tracking Botnets
http://www.honeynet.org/papers/bots/
12 Underground market for stolen IDs thrives
http://www.usatoday.com/money/industries/technology/2005-03-02-datathieves-usat_x.htm
13 Zombie PCs growing quickly online
http://news.bbc.co.uk/2/hi/technology/4685238.stm
14 MSN IP Search
http://ha.ckers.org/blog/20080803/msn-ip-search/

INPUT VALIDATION

Input validation and sanitation are important tools to protecting your web application. Any input coming from the user must be either validated (that is, make sure the data is in a verified, expected format) or sanitized (only accept whitelisted data or transform the data into a safe format). Failure to do either of these will expose your web application to a variety of attacks.

Let's first talk about whitelisting versus blacklisting. Whitelisting means you have a limited set of acceptable inputs; you either reject or remove anything non-valid. Blacklisting means you have a limited set of UNacceptable inputs; again, you either reject or remove anything non-valid. While they don't appear to be much different, whitelisting is much more secure because you're guaranteed to only get what you allow. Blacklisting has the disadvantage in that you must imagine all attack scenarios and try to block each one. A good example of this is trying to blacklist malicious HTML – imagine you wish to block the <script> tag by looking for "<script"; what you fail to realize is there are UTF-8 whitespace characters that can be inserted into the string that will thwart your blacklist, but still allow the <script> tag to be recognized by the browser (e.g. "<scr*ipt" where the asterisk is a Unicode whitespace character).[15] You should use blacklisting sparingly or as a secondary defense.

The types of validations you should perform depends on where and how the user input is used. In the latter sections on the vulnerabilities, I'll discuss the specific validations required for them. But for now, I'll go over generic validations.

Let's first identify the sources of user input[16]:

- Query parameters in the URL
- The path of the URL (may be echoed as part of "Document not found" error messages)
- Form fields (including hidden fields)
- Cookies
- Other parts of the HTTP request header (such as the referrer URL)
- Data that was inserted into a data store in an earlier transaction, possibly by a different user (e.g. messages in Google Groups, Orkut, GMail).
- Data obtained from a datafeed (e.g. merchant feeds in Google Product Search)
- Data crawled from the web (Google Search) or the local disk (Google Desktop)

As you can see, there are a variety of ways user input is received by Lasso. If you don't use the data (such as the referrer), then you don't have to validate it. You only validate the user input that gets used in your web application.

A general approach to validation is to only allow what you're expecting. So if you're expecting an integer, then either validate that it's an integer, or explicitly cast it as an integer. For example:

```
// validate integer
if(string_isDigit(action_param('quantity')));
      // it's an integer!
/if;

// cast to integer to ensure it's an integer
integer(action_param('quantity'));
```

Personally, I prefer to cast it to the type I want, then perform further validation on it (such as making sure it's non-zero for example). You'd do similar validation/casting on decimals and dates too.

---

15 More about whitespace characters used to thwart blacklisting:
http://www.gnucitizen.org/blog/snippets-of-defense-ptiv/

16 Verbatim from: Introduction to Cross-Site Scripting Vulnerabilities
http://code.google.com/p/doctype/wiki/ArticleIntroductionToXSS#Sources_of_Untrusted_Data

When using HTML forms, be sure checkboxes, selects or anything else with limited choices really only allows the limited choices. For example:

```
var('valid_choices') = (:'Red','Green','Blue');
var('choice') = 'Red'; // default
if($valid_choices->contains(action_param('choice')));
        $choice = action_param('choice');
/if;
```

When using hidden inputs on forms, you either should validate that they contain a value that's acceptable, or better, just encrypt them, then decrypt them on the response page. That will prevent a malicious user from changing them. You can use [lp_var_pack][17] and [lp_var_unpack][18] to perform the encryption/decryption.

Strings can be validated in a variety of ways, for example Lasso can validate email addresses and credit card numbers (validate that their format matches a known good format). You can also build custom validation for phone numbers, identifiers such as Social Security numbers (if in the States), etc.

For URLs, Google offers a "Safe Browsing API[19]" that allows you to verify if the URL is a known phishing/malware site. While a Lasso interface to the API doesn't currently exist, it's on my projects list, so expect one in the future.

Input validation is the corner-stone of web application security; remember the Golden Rule of Client Input:

> **"All client input (headers and request) is hostile until proven otherwise or sanitized."**

Now let's look at specific web vulnerabilities, starting with SQL injection.

17 [lp_var_pack]
http://tagswap.net/lp_var_pack
18 [lp_var_unpack]
http://tagswap.net/lp_var_unpack
19 Google Safe Browsing API
http://code.google.com/apis/safebrowsing/

SQL INJECTION

SQL injection attacks occur when unsanitized client input is allowed to be used in a SQL query. When unmitigated, an attacker can create their own SQL queries against your datasource (or more likely is an automated tool will compromise your web application[20]). This isn't an issue if you use classic inlines[21] as Lasso will sanitize the client input for you, but if you use -sql inlines, then you must understand how to protect against SQL injection.[22]

There are two ways in Lasso to protect against SQL injection when you want to write your own SQL query; one is to use Prepared Statements and the other is to carefully build a SQL query string using sanitize client input.

Prepared Statements provide the best protection because no sanitation is required on the client input yet you still get to use a SQL query string. Note that not all SQL queries are supported as Prepared Statements in MySQL, so be sure to read the MySQL documentation for more details.[23] How Prepared Statements in Lasso works is you simply specify the query you want to execute, provide the data one or more times and Lasso handles the rest.

An example of a Prepared Statement in Lasso looks like this: [24]

```
Inline(-Database='Contacts', -Table='People',
        -Prepare="INSERT INTO people (`first name`, `last name`) VALUES (?, ?)')";

        Inline(-Exec=Array('Bil','Corry'));
        /Inline;
        Inline(-Exec=Array($firstname,$lastname));
        /Inline;
        Inline(-Exec=Array(action_param('firstname'),action_param('lastname'));
        /Inline;
/Inline;
```

I won't delve into specifics about how to use Prepared Statements as it's well covered in the Lasso Language Guide[25] and a Tip of the Week[26] and is beyond the scope of this paper.

Building your own SQL query strings for -sql inlines is where developers get into trouble; all client input must be sanitized before it can be used in a -sql inline. How it gets sanitized depends on where in the SQL query string you're inserting the client input.

Let's walk through the four most common injection points in SQL query strings and how to secure them.[27]

---

20 Mass SQL Attack a Wake-Up Call for Developers
http://www.technewsworld.com/story/Mass-SQL-Attack-a-Wake-Up-Call-for-Developers-62783.html

21 By "classic inlines" I mean inlines where you specify an action other than -sql (e.g. -add, -delete). All Filemaker inlines are classic inlines.

22 Examples of real-world SQL injection is here:
http://www.evilsql.com/

23 MySQL Documentation - 12.7. SQL Syntax for Prepared Statements
http://dev.mysql.com/doc/refman/5.0/en/sql-syntax-prepared-statements.html

24 Adapted from example in Lasso Language Guide:
http://docs.lassosoft.com/Lasso%208.5/003%20Language%20Guide/002%20Database/007%20Database%20Interaction%20Fundamentals/index.lasso#PreparedStatements

25 Prepared Statements – Lasso Language Guide
http://docs.lassosoft.com/Lasso%208.5/003%20Language%20Guide/002%20Database/007%20Database%20Interaction%20Fundamentals/index.lasso#PreparedStatements

26 Lasso 8.5 MySQL Prepared Statements – Tip of the Week for September 8, 2006
http://www.lassosoft.com/Documentation/TotW/index.lasso?9185

27 Much of this is adapted from my 2004 article on SQL Injection:
http://tagswap.net/articles/SQL_Injection/

Strings

SQL injection into SQL strings happens when a user-provided string is merged into a sql inline and the quotes used to delimit the SQL string are also contained within the user-provided string. Here is a SQL query that plugs in the user-provided value for searching on a first name:

```
var('sql')="SELECT name_first, name_last FROM contacts WHERE
name_first='"+action_param('firstname')+"'";
```

Imagine firstname = "Bil" - the sql query becomes this:

```
SELECT name_first, name_last FROM contacts WHERE name_first='Bil'
```

So far, so good, right? Now imagine instead the user entered:

```
crackerboy'; delete from contacts; #
```

Now the query becomes:

```
SELECT name_first, name_last FROM contacts WHERE name_first='crackerboy';
delete from contacts; #'
```

Viola! Your entire contacts table is hosed. (The pound sign # tells MySQL that the rest of the line is a comment). So how do you prevent SQL injection into SQL strings? You must escape all the quotes in the data provided by the client. Fortunately, Lasso has a built-in tag to do it for you, [encode_sql][28].

Here's the original query rewritten to use it:

```
var('sql')="SELECT name_first, name_last FROM contacts WHERE
name_first='"+encode_sql(action_param('firstname'))+"'";
```

So properly encoded, now the query becomes:

```
SELECT name_first, name_last FROM contacts WHERE name_first='crackerboy\';
delete from contacts; #'
```

That slash \ in front of the embedded quote tells MySQL to ignore it, and thus, the entire string becomes the name_first search criteria.

So that's how you protect your strings in queries, but what about numeric values?

Numeric Values

Unfortunately, numeric values can not be protected using encode_sql; reason being you wouldn't (normally) have quotes around the value to begin with, so you wouldn't need to escape any quotes being sent.

So let's take a look at a query that uses a numeric value:

```
var('sql')="SELECT name_first, name_last FROM contacts WHERE
userid = "+action_param('userid');
```

28 [encode_sql] is for MySQL datasources. There's also [encode_sql92] for datasources that require a different type of escaping, such as SQLite.

Imagine userid = "123" - the sql query becomes:

```
SELECT name_first, name_last FROM contacts WHERE userid = 123
```

Already this has a problem, the user could feed sequential numbers to your query and pull out the entire database list. But that's another topic for another time.

Now imagine instead the user entered:

```
123; delete from contacts;
```

Now the query becomes:

```
SELECT name_first, name_last FROM contacts WHERE userid = 123; delete from contacts;
```

Viola! Your entire contacts table is hosed. (Notice the theme here?)

And even if we encode_sql the passed-in value, it wouldn't change anything. So how do you prevent an attacker from hosing your table this time? Make sure that you explicitly pass a numeric if that's what your query is expecting.

While you could validate the passed-in value to ensure it's numeric using something like [string_isdigit], I instead just cast it explicitly to a numeric using [integer] and [decimal] (depending on if the query needs an integer or decimal).

So let's rewrite that query to prevent sql injection for integer:

```
var('sql')="SELECT name_first, name_last FROM contacts WHERE
userid = "+integer(action_param('userid'));
```

So what happens, you ask, if the user provides the following?

```
123; delete from contacts;
```

Well, when Lasso converts a string to an integer or decimal, it takes every digit it can until it hits a non-numeric character. So casting to integer, the query would become:

```
SELECT name_first, name_last FROM contacts WHERE userid = 123
```

Decimals are handled in an identical fashion.

Date Values

Dates in MySQL can either be strings or numeric; MySQL accepts either. But to prevent SQL injection, the trick is to cast the passed-in value to a Lasso date type using [date] and formatting the output for MySQL:

```
var('lassoDate') = lp_date_stringToDate(action_param('date'),-error=date);
var('sql')="SELECT name_first, name_last FROM contacts WHERE
last_login <= "+$lassoDate->format('%Q');
```

Since a Lasso date type can not hold anything other than dates, you are assured that any bogus strings will be rejected when trying to cast it to a Lasso date.

LIKE Queries

LIKE queries are really just strings, but with an extra twist of pattern matching. What that means is LIKE queries will recognize special characters within the string as wildcard characters, namely, “%” and “_”. You can read more about what “%” and “_” do in a pattern-matching query in the MySQL documentation.[29]

To show how “%” and “_” can be abused, let's start off with an example query:

```
var('sql')="SELECT name_first, name_last FROM contacts WHERE
name_last LIKE '"+encode_sql(action_param('lastname'))+"%'";
```

This query will allow someone to enter the first letter(s) of a last name, and find all contacts that begin with them. So imagine the user entered "cor" (you plan to require and validate that they entered at least three characters), the query would look like:

```
var('sql')="SELECT name_first, name_last FROM contacts WHERE
name_last LIKE 'cor%'
```

But imagine if the user entered "%%%", that would validate as three characters and the query would become:

```
SELECT name_first, name_last FROM contacts WHERE name_last LIKE '%%%%'
```

That query would find all of your contacts! So how to handle LIKE? You have two options. Either remove all “%” and “_” found in the string or escape them by replacing “%” with “\%” and “_” with “_”. Be sure to still encode_sql the string. Note that you only escape “%” and “_” for LIKE queries (or any of the other pattern-matching queries that recognize “%” and “_”), don't go doing it on all your query strings.

29 MySQL Documentation - 3.3.4.7. Pattern Matching
http://dev.mysql.com/doc/refman/5.0/en/pattern-matching.html

CROSS-SITE SCRIPTING (XSS)

Cross-site scripting (XSS) attacks occur when unsanitized client input is used in content served to the client and/or others. When unmitigated, XSS can perform actions in the context of the site serving the page. This attack can steal cookies, modify the web application behavior, or modify the DOM/page content[30], often without the victim knowing the attack had occurred.[31]

To get a sense of how prevalent this attack is, it's interesting to note that financial sites such as Citibank, Barclays, Paypal and HSBC, security vendors such as McAfee, Verisign and Symantec, and large sites such as Google, eBay, Facebook and MySpace currently have, or have had XSS vulnerabilities within the last six months.[32]

When looking at solutions to protecting your web application, it should be noted that it is very difficult to blacklist all bad content – there are many ways to evade a blacklist filter[33]. A better approach is to only allow whitelisted content and/or encode the content to render it safe.

Let's look at a simple example. Imagine you have a page where the user enters a search term, which on the response page shows up as:

```
[var('search') = action_param('search_term')]
Your search for [$search] found [found_count] results.
```

Imagine the user entered:

```
test<script>alert('hello')</script>
```

Now the page, as served to the user, would look like:

```
Your search for test<script>alert('hello')</script> found 0 results.
```

The user would then see a dialog box with “hello” as the dialog alert text (assuming JavaScript was enabled in the user's browser). Now that isn't terribly interesting because most user's are not trying to exploit themselves. But imagine now that we take it one step further and create an URL that will take any user to the page and show them the exploit:

```
http://mysite.tld/response.lasso?search_term=test%3cscript%3ealert(%27hello%27)%3c%2fscript%3e
```

Now a malicious site can redirect a user to the above URL and execute a script of the attacker's choosing in the context of the vulnerable site. And while alerting 'hello' isn't terribly interesting, an attacker is much more likely to script a solution that steals cookies or performs some other maliciousness.

30 Google doctype – Introduction to Cross-Site Scripting Vulnerabilities
http://code.google.com/p/doctype/wiki/ArticleIntroductionToXSS

31 Wikipedia – Cross-site scripting
http://en.wikipedia.org/wiki/Cross-site_scripting

32 </xssed> xss attacks information
http://www.xssed.com/

33 XSS (Cross Site Scripting) Cheat Sheet
http://ha.ckers.org/xss.html

The above example used a request parameter to exploit the site, it is possible to use other client input vectors, such as user-agents[34], referrers[35], file downloads[36], cookies[37], or any other data received from the client that is used by the web application.

Now that we've discussed the issue, let's look at ways to protect against it. The first thing to understand is we're trying to prevent client input from becoming something malicious. How you protect against it depends entirely on what you're doing with the client input. For example, using our vulnerable code above, to protect our application against XSS, we can encode_html the user input:

```
Your search for [encode_html($search)] found [found_count] results.
```

Now the page, as served to the user, would look like:

```
Your search for test&lt;script&gt;alert(&#39;hello&#39;)&lt;/script&gt; found 0 results.
```

That fixes the problem by rending the <script> as plain text instead.

Let's look at protecting against XSS in the three most common situations:

HTML Body[38]

When client input is used within the <body> of your page, you should use [encode_html] to encode it:

```
Your search for [encode_html(action_param('search_term'))] found [found_count] results.
```

HTML Attribute[39]

When client input is used within a HTML tag attribute, again use [encode_html]:

```
<input type="text" name="test" value="[encode_html(action_param('test'))]">
```

In addition, be sure to surround the attribute with quotes to prevent attribute injection attacks, such as vulnerable code looking like:

```
<input type="text" name="test" value=[encode_html(action_param('test'))]>
```

Which when feed this:

```
test onmouseover=evil_script()
```

---

34 Vulnerable Vulnerability Databases
http://www.0x000000.com/?i=546

35 A Second-order of XSS
http://blogs.iss.net/archive/SecondOrderXSS.html

36 Google XSS
http://xs-sniper.com/blog/2008/04/14/google-xss/

37 XSS, Cookies, and Session ID Authentication – Three Ingredients for a Successful Hack
http://www.informit.com/articles/article.aspx?p=603037&rll=1

38 HOWTO filter user input in regular body text
http://code.google.com/p/doctype/wiki/ArticleXSSInBodyText

39 Google doctype – HOWTO filter user input in tag attributes
http://code.google.com/p/doctype/wiki/ArticleXSSInAttributes

Becomes:

```
<input type="text" name="test" value=test onmouseover=evil_script()>
```

That will cause a malicious script to execute when the mouse is moved over the <input>.

URL Attribute[40]

When client input is used within an URL attribute, use [encode_strictURL] to properly encode their input. For example, imagine your code looks like:

```
[var('test') = action_param('test')]
<a href="test.lasso?param=[$test]">Test</a>
```

If the user supplies the following for the "test" parameter:

```
test"<script>alert('ok')</script>
```

Then the page is served as:

```
<a href="test.lasso?param=test"<script>alert('hello')</script>">Test</a>
```

Which triggers the JavaScript. Adding [encode_strictURL] fixes the issue:

```
<a href="test.lasso?param=[encode_stricturl($test)]">Test</a>
```

The browser is served:

```
<a href="test.lasso?param=test%22%3cscript%3ealert(%27hello%27)%3c%2fscript%3e">Test</a>
```

Problem solved.

Other Attack Vectors

For further reading, Google's doctype project does a great job of providing details on protecting against XSS in a variety of scenarios, including JavaScript and Flash.[41] I highly recommend reading through it to better understand how to protect against XSS.

40 Google doctype – HOWTO filter user input in URL attributes
http://code.google.com/p/doctype/wiki/ArticleXSSInUrlAttributes

41 Google doctype – Web security
http://code.google.com/p/doctype/wiki/ArticlesXSS

CROSS-SITE REQUEST FORGERIES (CSRF)

Cross-site request forgery (CSRF aka XSRF) attacks occur when a victim's browser is secretly directed to a target site and as a result, an action is performed on behalf of the victim as if the victim him/herself had requested it.[42] Typically, there isn't any indication to the victim that any such action has even taken place.

A simple example to illustrate. Say you visit a random site and it has the following embedded in it:

```
<img src="http://www.google.com/setprefs?hl=xx-piglatin">
```

Your browser would try to retrieve an image from that URL, and in doing so, would send a request to Google using your credentials (assuming you've visited Google at least once). From Google's point of view, the request appears legitimate because it's coming from your browser with your cookies. If successful, the above attack would change your default language with Google to Pig Latin; fortunately Google has fixed that CSRF vulnerability (so don't try it at home).[43]

I should note here that with pure CSRF, the attacker is guessing you're a user that is authenticated with a particular website and that you're currently "signed in". And know that the attacker does not have access to the content returned by the server. The attacker can only trick your browser into sending a request to a server and that's the extent of the attack. But even with those limitations, there are a lot of interesting attacks that can be done.

Some real-world examples[44]: using CSRF to perform SQL injection against PHPMyAdmin[45], attacking a site that isn't vulnerable to XSS and using it's IMAP server against it in a reflected attack[46], cookie stuffing to defraud affiliate programs[47], and perhaps one of the more devious uses is Cross-Site File Upload attacks[48].

There are two preferred ways to defeat a CSRF attack: use a unique token or re-authenticate the user.

The way a unique token works is a sufficiently random token is added to each request, then the server verifies the unique token and allows the action. If the unique token is not present or incorrect, the server can reject the request and/or ask the user to re-authenticate. This prevents CSRF because the attacker won't know the token, and thus can't create a valid request for the victim.

Re-authenticating the user acts as a simple "Are you sure?" measure. Since the attacker won't know the username and password for the victim, it isn't possible to successfully execute a CSRF against the target.

Beyond those two methods, there are a few others to defeat a CSRF attack, but they have caveats that make them less desirable to implement. I don't recommend using them, but you can read more about using referrers, double-submit cookies, and POST-only requests at Wikipedia if interested.[49]

---

42 A nice CSRF overview is here:
http://www.0x000000.com/index.php?i=309&bin=100110101

43 Security Threat: Cross Site Request Forgery (CSRF)
http://itmanagement.earthweb.com/secu/print.php/3739621

44 CSRF Hacking Database
http://csrf.0x000000.com/csrfdb.php?do=browse

45 SQL Injecting PhpMyAdmin
http://www.0x000000.com/?i=587

46 The Extended HTML Form Attack Revisited
http://enablesecurity.com/2008/06/18/the-extended-html-form-attack-revisited/

47 Affiliate Programs Vulnerable to Cross-site Request Forgery Fraud
http://www.cgisecurity.org/2008/08/affiliate-progr.html

48 Cross-site File Upload Attacks
http://www.gnucitizen.org/blog/cross-site-file-upload-attacks/

49 Wikipedia – Cross-site request forgery
http://en.wikipedia.org/wiki/CSRF

To give a better idea of CSRF works and how to fix it, let's look at a typical Lasso implementation that's vulnerable to CSRF. I'm going to use a very benign example that wouldn't normally be of much concern in the real world, but it'll get the idea across.

Let's say we have a web application that has a simple “Sign Out” link on every page. The “Sign Out” link looks like this:

```
<a href=”/auth/signout.lasso”>Sign Out</a>
```

The page signout.lasso works by expiring the current Lasso session. So with that knowledge, a malicious site could target ours by adding this to one of its pages:

```
<img src=”http://oursite.tld/auth/signout.lasso”>
```

By visiting that malicious page, our session would be terminated on our site. If you remember, there are two ways to defend against this CSRF attack. One is to re-authenticate the user. So on the signout.lasso page, it would prompt the user to enter their username and password to authorize the signing out action. In the context of our malicious page, the request generated by the <img> HTML tag would no longer work since a username and password are now expected.

The other way to defend against this CSRF attack is to use tokens. Using this technique our sign out link would then become:

```
<a href=”/auth/signout.lasso?token=Yc9QOe1mLNeQ9SOSM05f”>Sign Out</a>
```

The page signout.lasso would look at action_param('token') and make sure it matched the value expected. You can store the value expected in the user's session, or in a database, or you could go another route and instead just encrypt_blowfish the current time and some user identifier (user id, IP address, etc) as the token, then on signout.lasso decrypt it making sure that the time is within five minutes of when it was created and the identifier still matches the current user.

SIDE-JACKING

Side-jacking attacks can occur when a user is (1) on a network that is sniffable, (2) the packets are traveling unencrypted, and (3) the user is using a web application that utilizes cookies to maintain user sessions. The attack is carried out by watching for traffic that contains cookies. By capturing the cookie and replicating it on the attacker's own system, the attacker can then pretend to be the victim to the web application.

As you can see from the description, side-jacking isn't so much a server issue as it is a client issue – as a user, you can protect yourself by never using open wi-fi networks (or if you must, use VPN to encrypt all of your traffic); always use HTTPS when possible; and you can mitigate some risk by logging out explicitly from web applications (which renders the session cookie useless).[50]

There are some measures a server can take to minimize the risk for its users. The best solution is to run all authenticated sessions (those in which a user has logged in) exclusively through HTTPS. When doing so, be sure to detect if a user tries to connect via HTTP and redirect to HTTPS if they do. Also, be sure to use the -secure option of [session_start] for the user session; this will prevent the session cookie from being sent on HTTP, which is an important defense to accidentally leaking the cookie data outside the HTTPS connection.

Another good practice is to provide a mechanism for users to log out of the application; as mentioned above, this helps mitigate the risk of side-jacking by invalidating the session. So even if the attacker captures the session cookie data, if the session is invalid, it can't be used. And of course, limiting the session life to something short (under a half-hour) is also good practice as many users never will log out themselves.

If you're not running HTTPS, then beyond the issue of having cookies stolen via side-jacking, you also have the problem of the user's username and password being sent in cleartext when logging into the web application. The best solution for that is to use Digest Authentication[51].

As for preventing side-jacking in a scenario where the packets are sniffable and unencrypted, there is no “bullet-proof” method. The best you can do is to create a “fingerprint” of the browser that successfully logged in, store the fingerprint in the session, then detect when a browser with a different fingerprint tries to use the session. The fingerprint can be anything unique to the request such as user-agent, ip address, language preferences, etc. Just know in the case of IP addresses, users behind a proxy may all share the same IP address (so you won't catch an attacker if he is on the same wi-fi network that is routed through a common proxy), and also one user behind a proxy may have multiple IP addresses, as is the case with AOL[52]. I'd recommend filtering on IP address unless the user requests that the IP check be turned off (make it optional).

Another solution is to rotate session IDs, issuing a new session ID on every page request.[53] The advantage to that is the session ID lives only for as long as the legitimate user stays on the current page, so it forces an attacker to move immediately against the victim. The disadvantage is it has some funky behavior when reloading the page or using the back button if you embed the session ID in the links rather than as a cookie, so experiment with it and see if it will fit your needs.

If you're interested in seeing side-jacking in action, there's a tool you can use called “Hamster[54]”, although I believe it's Windows-only currently.

---

50 'Sidejacking' Tool Unleashed
http://www.darkreading.com/document.asp?doc_id=130692

51 Wikipedia – Digest access authentication
http://en.wikipedia.org/wiki/Digest_access_authentication

52 AOL Proxy Info
http://webmaster.info.aol.com/proxyinfo.html

53 It's experimental in Lasso – Fletcher talks about it in the thread “Kill Session on Close Browser Window” from 2005-09-29:
http://www.listsearch.com/Lasso/Thread/index.lasso?12572#203922

54 SideJacking with Hamster
http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html

SUMMARY

It is clear that the days of security by obscurity is long dead. Given that the financial rewards for criminal activity has climbed into the billions[55] and the risk of being caught is slight[56] (and even if caught, it might work out favorably anyhow[57]), security must now be an integral part of any web application. It is no longer a question if the financially-motivated bot masters, with hundreds of thousands of computers at their disposal, will find you[58], it's just a matter of when. And when they do, what will they find?

The goal for this paper is to give you insight into the most common vulnerabilities found in web applications. With education and proper planning, security doesn't have to be something that is accounted for after an application has been deployed, but instead can be part of the development and testing cycle.

In the appendices that follow, you'll find a list of auditing tools, helpful Firefox plug-ins, additional topics to explore, and a couple of professional groups to consider joining. These lists are not exhaustive, but rather are a starting point from which you can expand your knowledge and education of WebAppSec.

The last appendix is a list of blogs that I read regularly which has greatly expanded my own personal knowledge of WebAppSec issues. I definitely suggest adding them to your favorite RSS reader; security is an on-going learning process and being exposed to new ideas is a great way to incrementally increase the depth and breadth of your WebAppSec knowledge.

ABOUT THE AUTHOR

Bil Corry is founder of Lasso.Pro, an international web application development and consulting firm specializing in scalable, secure web applications. Bil has been developing web applications using Lasso for more than 10 years. During this time, he also has been a major contributor to the Lasso community, including participating on LassoTalk (top 5 all-time contributor); code contributions for Email_Send, Auth Tags, PDF Tags, Memory Session Manager, File_Serve, LassoWiki and others; code contributions on TagSwap.net which include more than 100 custom tags; article and edit contributions on LassoTech; winner of a Lasso Programming Challenge; and has spoken at a previous Lasso Developer Conference. Bil holds a degree in Computer Science from California State University, Fullerton.

CONTACT INFORMATION

Bil Corry
Lasso.Pro (http://lasso.pro)
Phone: +1 240 337 2514
Email: (my first name)@lasso.pro

55 Online Threats Cost Consumers $8.5 Billion Over Last Two Years
http://news.yahoo.com/s/cmp/20080805/tc_cmp/209901659

56 Why Hackers Are A Step Ahead of the Law
http://news.cnet.com/2009-1017-912708.html?hhTest=1&tag=fd_lede

57 New Zealand Hacker Released As Police, Judge, Prosecutors All Praise His Mad Hacking Skillz
http://techdirt.com/articles/20080716/1236481702.shtml

58 Attacking Around the Globe Around the Clock
http://blog.imperva.com/2008/04/attacking-around-the-globe-aro.html

APPENDIX A – AUDITING TOOLS

Burp – integrated platform for attacking web applications:

http://portswigger.net/suite/

Google Ratproxy – passive web security assessment tool:

http://code.google.com/p/ratproxy/

Hamster/Ferret – tool for testing side-jacking:

http://erratasec.blogspot.com/2007/08/sidejacking-with-hamster_05.html

Metasploit – somewhat of a Swiss Army knife for attacking a target:

http://www.metasploit.org/

Pantera Web Assessment Studio – web application analysis engine:

http://www.owasp.org/index.php/Category:OWASP_Pantera_Web_Assessment_Studio_Project

Paros Proxy – web application security assessment:

http://www.parosproxy.org/

APPENDIX B – FIREFOX PLUG-INS

Firebug – Firefox plug-in that allows viewing XHR requests, script debugging, and much more:

https://addons.mozilla.org/en-US/firefox/addon/1843

Firecookie – Firefox plug-in that allows viewing and managing cookies:

https://addons.mozilla.org/en-US/firefox/addon/6683

Live HTTP Headers – Firefox plug-in for viewing the HTTP headers going between Firefox and the server:

https://addons.mozilla.org/en-US/firefox/addon/3829

NoScript – Firefox plug-in that allows turning JavaScript on/off; must have to protect your browser too:

https://addons.mozilla.org/en-US/firefox/addon/722

TamperData – Firefox plug-in to view and modify HTTP/HTTPS headers and post parameters:

https://addons.mozilla.org/en-US/firefox/addon/966

User Agent Switcher – Firefox plug-in that allows easy switching of the user agent:

https://addons.mozilla.org/en-US/firefox/addon/59

Web Developer – Firefox plug-in that provides a variety of tools for web development:

https://addons.mozilla.org/en-US/firefox/addon/60

XSS-Me & **SQL Inject-Me** – Firefox plug-ins for detecting XSS and SQL injection vulnerabilities:

http://securitycompass.com/exploitme.shtml

APPENDIX C – ADDITIONAL TOPICS

Additional topics to explore (in no particular order):

Timing Attacks – be sure to read Section 6, "Timing and its implications for Privacy" in the PDF:

> http://www.sensepost.com/research/squeeza/dc-15-meer_and_slaviero-WP.pdf
> http://ha.ckers.org/blog/20080828/more-timing-precision-enhancements/

Client Authentication – Do's and Don'ts:

> http://prisms.cs.umass.edu/~kevinfu/papers/webauth_tr.pdf

ZeroSum – create checksums of web directories to detect malicious changes to your web app files:[59]

> http://www.0x000000.com/?i=550

Logic flaws – some examples where logic flaws were exploited:

> Man Allegedly Bilks E-trade, Schwab of $50,000 by Collecting Lots of Free 'Micro-Deposits'
> http://blog.wired.com/27bstroke6/2008/05/man-allegedly-b.html
>
> Youtube's 18+ Filters Don't Work
> http://www.darkseoprogramming.com/2008/06/01/youtubes-18-filters-dont-work/
>
> Privacy flaw exposes Paris Hilton and Lindsay Lohan's private MySpace photos
> http://blogs.zdnet.com/security/?p=1244
>
> Yahoo SEM Logic Flaw
> http://ha.ckers.org/blog/20080616/yahoo-sem-logic-flaw/
>
> AT&T using User-Agent to give free wi-fi to iPhones at Starbucks:
> http://blogs.zdnet.com/security/?p=1067
> http://www.darkreading.com/blog.asp?blog_sectionid=447&doc_id=153200

Inside Jobs – some examples how people on the inside breach the system:

> Hidden Code Costs Poker Players Thousands
> http://catless.ncl.ac.uk/Risks/25.20.html#subj3
>
> S.F. officials locked out of computer network
> http://www.sfgate.com/cgi-bin/article.cgi?f=/c/a/2008/07/14/BAOS11P1M5.DTL
>
> One in three IT staff snoops on colleagues
> http://www.msnbc.msn.com/id/25263009/
>
> Day of Reckoning? Super Rich Tax Cheats Outed by Bank Clerk
> http://abcnews.go.com/Blotter/Story?id=5378080&page=1
>
> Data voyeurism is common
> http://redtape.msnbc.com/2008/03/surprised-by--1.html

59 On my to-do list is to port this to Lasso.

ModSecurity – an Apache module that acts as a web application firewall:

http://www.modsecurity.org/

Google DocType – articles on web security:

http://code.google.com/p/doctype/wiki/ArticlesXSS

Google Code University – web security courses:

http://code.google.com/edu/security/index.html

Web Application Hackers Handbook – attack checklist from the book

http://portswigger.net/wahh/tasks.html

Internet Jurisdiction – where are your websites physically located?

Do The Good People Of Florida Think Your Website Is Obscene? You Better Hope Not.
http://www.alleyinsider.com/2008/6/do_people_in_florida_think_your_website_is_obscene_

Host-Proof Data Encryption – data stored by the server is encrypted in such a way that only the user can view it:

http://ajaxpatterns.org/Host-Proof_Hosting#Solution

Rich Content Filters – allowing users to upload HTML can be risky:

Bullet-proof rich content filters:
http://www.gnucitizen.org/blog/bulletproof-rich-content-filters

HTML Purifier
http://htmlpurifier.org/

Infected Devices – brand-new devices pre-installed with malware:

HP USB Keys Shipped with Malware for your Proliant Server
http://isc.sans.org/diary.html?storyid=4247&rss

Back doors in embedded devices (printers, routers, etc)
http://blog.washingtonpost.com/securityfix/2008/04/get_paid_to_find_software_hard_1.html

Devices shipping from abroad with malware
http://www.darkreading.com/blog.asp?blog_sectionid=447&doc_id=148583

HTTP Response Splitting - http header injection:

http://www.aspectsecurity.com/documents/Aspect_File_Download_Injection.pdf
http://www.securityfocus.com/archive/1/425593

Open Redirects – spammers using open redirects to make their spam appear more legitimate:

http://blog.washingtonpost.com/securityfix/2008/07/study_site_redirects_abundant_1.html
http://ha.ckers.org/blog/20080716/redirection-report/

Internet Behavior – your online behavior can reveal much about yourself:

Google has patents that can detect your age, ethnicity, reading level, income, etc:
http://yro.slashdot.org/article.pl?sid=08/03/22/1314253

Using your browser URL history to estimate gender
http://www.mikeonads.com/2008/07/13/using-your-browser-url-history-estimate-gender/

Spyjax - :visited spy tool
http://www.merchantos.com/makebeta/tools/the-spy-is-dead/

Know which social sites the visitor uses
http://azarask.in/blog/post/socialhistoryjs/

Track users browsing via :visited link coloring
https://bugzilla.mozilla.org/show_bug.cgi?id=147777#c78

CSS Spying
http://jeremiahgrossman.blogspot.com/2006/08/i-know-where-youve-been.html

Logging – a case against logging:

http://www.0x000000.com/?i=612

CAPTCHA – it will stop amateurs, but not anyone motivated:

Inside Craigslist's Increasingly Complicated Battle Against Spammers
http://techdirt.com/articles/20080523/0327151211.shtml

How CAPTCHA got trashed
http://www.computerworld.com.au/index.php/id;489635775;fp;;fpid;;pf;1

Breaking The Google Audio Captcha.
http://www.0x000000.com/?i=560

Human CAPTCHA Breaking
http://ha.ckers.org/blog/20080311/human-captcha-breaking/

Inside India’s CAPTCHA solving economy
http://blogs.zdnet.com/security/?p=1835

Captcha's broken by mules
http://www.theregister.co.uk/2008/04/10/web_mail_throttled/

PWNtcha
http://libcaca.zoy.org/wiki/PWNtcha

File Uploads – allowing files to be uploaded is risky:

Evil GIFs: Partial Same Origin Bypass with Hybrid Files
http://radar.oreilly.com/2008/06/partial-same-origin-bypass-wit.html

GIFAR - A photo that can steal your Facebook account
http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9111298

Backdooring images
http://www.gnucitizen.org/blog/backdooring-images/

New Worm Transcodes MP3s to Try to Infect PCs
http://www.pcworld.com/businesscenter/article/148603/new_worm_transcodes_mp3s_to_try_to_infect_pcs.html

CPU Attacks – targeting errata in processors

Researcher to demonstrate attack code for Intel chips
http://www.infoworld.com/article/08/07/14/Researcher_to_demonstrate_attack_code_for_Intel_chips_1.html

Password Reset – security questions to reset a password are becoming a less secure way to authenticate:

Researcher mines blogs, social networks to access bank accounts
http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9113405

I forgot my password! (Now what?)
http://www.ravenwhite.com/iforgotmypassword.html

'Forgot your password?' may be weakest link
http://redtape.msnbc.com/2008/08/almost-everyone.html

Column truncation & max_packet_size vulnerabilities – interesting attacks:

http://www.suspekt.org/2008/08/18/mysql-and-sql-column-truncation-vulnerabilities/

OWASP TOP 10 – The top ten most critical WebAppSec vulnerabilities for 2007:

http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf

APPENDIX D – PROFESSIONAL GROUPS

There are two web security organizations I recommend you join, or at least become familiar with: OWASP and WASC. Both offer tools and informative articles.

Open Web Application Security Project (OWASP)

http://www.owasp.org/

Consider joining a local chapter and joining that chapter's email list:

http://www.owasp.org/index.php/Category:OWASP_Chapter

Web Application Security Consortium (WASC)

http://www.webappsec.org/

WASC has a great email list that discusses WebAppSec; I highly recommend it:

http://www.webappsec.org/lists/websecurity/

APPENDIX E – SECURITY BLOGS

A Day in the Life of an Information Security Investigator
http://it.toolbox.com/blogs/securitymonkey

Anachronic
http://www.anachronic.com/

Arbor Networks
http://asert.arbornetworks.com/

CERIAS Blog
http://www.cerias.purdue.edu/site/blog

cgisecurity
http://www.cgisecurity.com/

Chris Weber
http://lookout.net/

Dark Reading
http://www.darkreading.com/

Dark SEO Programming
http://www.darkseoprogramming.com/

Emerging Threats
http://www.emergingthreats.net/

Errata Security
http://erratasec.blogspot.com/

GNUCITIZEN
http://www.gnucitizen.org/blog/

Google Online Security Blog
http://googleonlinesecurity.blogspot.com/

ha.ckers
http://ha.ckers.org/blog/

HostExploit
http://hostexploit.blogspot.com/

HP Application Security Center Community
http://www.communities.hp.com/securitysoftware/blogs/

IBM Internet Security Systems
http://blogs.iss.net/

Jeremiah Grossman
http://jeremiahgrossman.blogspot.com/

Mantasano
http://www.matasano.com/log/

RISKS Digest
http://catless.ncl.ac.uk/Risks

Ronald van den Heetkamp
http://www.0x000000.com/

root labs rdist
http://rdist.root.org/

Rootsecure.net
http://www.rootsecure.net/

Schneier on Security
http://www.schneier.com/blog/

Spamhaus (uses Lasso!)
http://www.spamhaus.org/newsindex.lasso

Stefan Esser
http://www.suspekt.org/

StopBadware.org
http://blog.stopbadware.org/