Global Information Assurance Certification Paper

Vulnerability Testing Techniques for Web Application Programmers

Alex Kuhn
December 6, 2004

GSEC Version 1.4c, Option 1

Abstract
Prior to releasing any new web application on the Internet or for use within a company, it is a best practice to test the application for potential vulnerabilities. A wide variety of free and commercial tools are available to aid in this task. However, the developers who have to make changes to those applications do not necessarily have the tools or the appropriate techniques to correct these vulnerabilities.

This paper provides an introduction to some techniques a developer can use to test their own applications while under development or when remediating particular vulnerabilities.

Introduction

Securing a web application today is like putting together a puzzle: all the pieces have to be put together in the right way or a hole will remain. SANS' Defense In-Depth concept teaches the security professional about the many components that need to be considered in order to secure an environment. (Cole et al. 2:14)

In the case of an Internet web application, there are many components that play a role in securing the application. Some examples are firewalls with restrictive rules, having current patches applied to host operating systems and web servers, having proper permissions on the files that make up the application, and a web server configuration that prevents inadvertent access to files. All of these need consideration prior to making an application available on the Internet.

But all of these components are not generally part of a web developer's responsibility. In many enterprises, one organization may administer the firewalls, another group the servers, and a third the web server application. Even if all of these components are properly secured, the application itself may have vulnerabilities due to the techniques used within the web application code. Many organizations employ commercial tools, such as Appscan and WebInspect, to assess web applications for vulnerabilities.

Imagine you are a web developer whose new application has just been subjected to a vulnerability assessment by the corporate security department. The report describes vulnerabilities such as cross-site scripting, SQL injection and form parameter tampering. You have never had any formal training in security, and now are expected to resolve these problems before your application is released. What can you do? Where do you even start?

This paper will discuss some common vulnerabilities that are frequently found during the vulnerability assessment process, demonstrate some techniques to remediate these vulnerabilities, provide links to other reference materials for remediation information, and provide an overview of one very useful tool developers can use as they test their fixes.

Common Attacks

I have run vulnerability assessment tools against many web applications, and several categories of vulnerabilities repeatedly appear: those based on cross-site scripting, SQL injection, and form field manipulation. Several other categories frequently appear which are not usually resolved by changing source code: directory traversal, debugging settings, and improper access to files. Upgrading or patching the operating system and/or web server can usually resolve these latter categories of vulnerabilities, or by changing settings within the web server itself. Because these vulnerabilities are frequently outside of the control of the web developer, they will not be discussed further here.

The three vulnerabilities that will be discussed have one thing in common: validating data input can be protection against all of them. In *Writing Secure Code*, 2nd edition, Howard and LeBlanc sum up the problem with this statement: "*all input is evil until proven otherwise*." (341)

Any place where an attacker can input data, whether in a visible or hidden form field, as a parameter in a URL, or through a cookie should be validated upon receipt by the web server. Howard and LeBlanc also sum up the solution well: "*look for that which is provably valid, and disallow everything else*." (385)

Cross-site Scripting

In their paper "Penetration Testing for Web Applications (Part Two), Melbourne and Jorm describe Cross Site Scripting (XSS) as occurring "wherever a developer incorrectly allows a user to manipulate HTML output from the application" ("Part two"). XSS depends on a server not filtering input from a user and then presenting it back to the user's web browser, where the browser may interpret the response as executable code.

Specific XSS attacks may only occur with specific pairings of the web server and web browser; not every variation may work with every combination of browser and server. Therefore, many possible flavors of XSS exist, and a good vulnerability assessment tool will check an application for as many of these as possible. Melbourne and Jorm give a simple example of an XSS vulnerability where this string is input as a parameter in a URL string ("Part Two")

```
<script>alert(document.cookie);</script>
```

If the server responds with a message including that string, then it shows that the web server is not filtering out the <script> tags from the input, and could be susceptible to a XSS attack.

It would seem that checking for certain kinds of strings, such as <script>, should be easy. Unfortunately, the grammar that the web application would have to examine is very complex. Howard and LeBlanc in Writing Secure Code, 2nd edition include examples of many of the complex constructions that an application would need to filter to minimize the risk of executing undesirable code. Here are a few examples (the malicious code would be placed within the [code] block): (428-429)

```
<img src=javascript:alert([code])>
<body background="javascript:alert([code])"
<!-- -- --><script>[code]</script><!-- -- -->
<\xC0][\xBC]script>[code][\xC0][\xBC]/script>
```

The first two examples would display a dialog box that would run arbitrary code. The third hides code within a comment block, and the fourth hides the opening angle bracket by encoding it in a different character representation.

There are many other ways to generate an XSS attack, and the references listed at the end of this paper contain descriptions of many more versions along with more detail than can be included herein.

Cross-site scripting remediation

XSS attacks could be eliminated if users did not have scripting languages permitted in their browsers, but doing so would cause many web sites to cease to work properly. Email clients that display dynamic content are also common programs that permit XSS attacks. Web browsers also need regular updating to patch new holes but this is a constant catch-up game as new vulnerabilities are constantly found. In the meantime, web developers need to filter as much as possible and limit input to that which is known good. Within the code that runs on the web server, whether it is ASP, CGI or another language, there should be routine logic that parses user input and strips out undesired data, leaving only that which can be proven safe. Some examples of code that can do this are shown below.

This regular expression from Writing Secure Code allows some html formatting tags and safe input (430):

```
/^(?:[\s\w\?\!\,\.\'\"]*|(?:\<\/?(?:i|b|p|br|em|pre)\>))*$/i
```

In Q article 252985, Microsoft lists this javascript function that filters special characters that allow scripts to execute (Q252985). In this example, these characters will be filtered out: < > " ' % ;) (& + -

```
function RemoveBad(strTemp) {
    strTemp = strTemp.replace(/\<|\>|\"|\'|\%|\;|\(|\)|\&|\+|\-/g,"");
    return strTemp;
}
```

If you implement code like this in a web browser function, do not make it your only defense! It should be only one part of a defense-in-depth strategy: the server should validate all data as well.

If you are expecting only alphabetic or numeric data, write a filter that will only allow those values through. In the article "How To Remove Meta-characters From User-Supplied Data in CGI Scripts", cert.org provides this Perl script that will only allow letters, numbers and a few special characters through: ("Cert")

```
#!/usr/local/bin/perl
        $_ = $user_data = $ENV{'QUERY_STRING'};       # Get the data
        print "$user_data\n";
        $OK_CHARS='-a-zA-Z0-9_.@';     # A restrictive list, which
                                       # should be modified to match
                                       # an appropriate RFC, for
                                       # example.
        s/[^$OK_CHARS]/_/go;
        $user_data = $_;
        print "$user_data\n";
        exit(0);
```

SQL Injection

SQL injection is another kind of input validation vulnerability that can be executed using similar techniques as Cross-Site Scripting. In their paper "Penetration Testing for Web Applications (Part Two)", Melbourne and Jorm describe SQL Injection as those where developers "do not properly strip user input of potentially 'nasty' characters before using that input directly in SQL queries". ("Part Two") SQL Injection attacks do not require an attacker to send an email or post a link on a web site and convince a user to click a link; these attacks can be initiated by an attacker directly against a web site.

In a SQL injection attack, merging the SQL statement on the web server with malicious data from an input field creates unexpected output that an attacker desires. A simple example is an application that has two fields: user ID and password. In Hacking Exposed Web Applications, Scambray and Shema demonstrate this SQL statement that might be used to validate a logon: (157)

SELECT * from AUTHENTICATIONTABLE WHERE Username = 'username input' AND Password = 'password input'

If in the username field an attacker inputs this text:

Username' --

Then the SQL string sent to the database would be this:

SELECT * from AUTHENTICATIONTABLE WHERE Username = Username' -- 'username input' AND Password = 'password input'

The -- means the rest of the SQL statement is a comment and should be ignored, so in effect, the password check would be eliminated. However, a valid username would still be required. Howard and LeBlanc in Writing Secure Code list these database servers that use the -- operator: Microsoft SQL Server, IBM DB2, Oracle, PostgreSQL and MySql. (399) This makes the **--** operator a very powerful enemy to your SQL statements.

What if the password input was this string, also from Hacking Exposed Web Applications (157):

DUMMYPASSWORD' OR 1 = 1 --

The SQL string sent to the database would become this:

SELECT * from AUTHENTICATIONTABLE WHERE Username = 'username input' AND Password = 'DUMMYPASSWORD' OR 1 = 1 --

In this case, the logic of the statement would check whether the valid password associated with the input username was DUMMYPASSWORD or whether the value 1 is equal to the value 1! Since the clause "1=1" is always true, then the password check would not be effective. An attacker would need to know only a valid username and could bypass the password.

A more sophisticated attack involves the use of the UNION clause to display information from the database. The UNION statement appends the result of one query onto the end of another query. In this example, Thompson and Whittaker demonstrate that a Microsoft SQL Server could be instructed to display all the tables in the database by querying the built-in sysobjects table (62)

' union all select 0,name,xtype,0,0,0 from sysobjects --

In order for this kind of attack to be effective, the attacker would do reconnaissance on the application, testing various combinations of strings to see what errors would result. In some cases, the errors give away the brand and version of the database and some quick checks on the Internet can result in lists of vulnerabilities for that database.

In this further example from Thompson and Whittaker, simply by putting a single quotation mark (') in a user field, the syntax for the Select statement can be made erroneous, resulting in an ODBC error (62)

Error Type:
Microsoft OLE DB Provider for ODBC Drivers (0x80040E14)
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the character string ' and Pin = ", /process.asp, line 40

Now the attacker knows to focus on SQL Server attacks, and knows that the name of at least one field name: Pin.

SPIDynamics' website has several good free whitepapers that have more examples of tests for SQL injection, as do Chris Anley's papers available at www.ngssoftware.com. See the references at the end of this paper for specific links.

Access to built-in stored procedures can be dangerous as well. Many databases provide stored procedures that can run essentially any command on the database server. Some of these stored procedures need administrator-level rights, but not all do. Here Thompson and Whittaker provide another example of a statement that could spawn a shell and save a list of all files on the root of the C drive and save it to a file that could then be accessed via a web browser: (62)

'; EXEC master.dbo.xp_cmdshell 'cmd.exe /c dir c:\ > c:\inetpub\wwwroot\dir_c.txt' --

Vulnerability assessment tools will run many tests for strings such as these and examine any return error messages for information that could aid an attacker to learn more about the system, and to see whether field input is being properly sanitized.

SQL Injection Remediation

No single technique can protect against SQL injection attacks – there need to be several layers of protection to be effective. Several that are very effective and should be considered include input data validation, controlling access within the database and the use of stored procedures.

In Chapter 12 of Writing Secure Code, Howard and LeBlanc lay out several best practices to preventing SQL injection attacks. (403-406)

1) Never Ever Connect as sysadmin

Always use an account that has the minimum rights needed to access the database. Sysadmin can access system-level stored procedures

2) Building SQL Statements Securely

In a SQL server environment, you can use VBScript within the database to create parameterized statements that can validate data type. This is an example of a parameterized query statement: (404)

SELECT count(*) FROM client WHERE name=? AND pwd=?

Within the VB script function, name and pwd would be defined with specific type and size characteristics and can be validated against certain string patterns, such as an email address. After that check the values would be passed to the database through the select statement.

In his MSDN article "Stop SQL Injection Attacks Before They Stop You", Paul Litwin demonstrated several good techniques for validating usernames and passwords using this regular expression:

[\d_a-zA-Z] {4,12}

This limits the input to entries between 4 and 12 characters of digits, alphabetic characters and the underscore.

Building the SQL statement within a stored procedure is another good way to prevent SQL injection. Chris Anley's paper "Advanced SQL Injection in SQL Server Applications" lists several approaches to validating data: (22-23)

1) Massage data so that it becomes valid. This is problematic because some characters that we may want to eliminate are possibly valid. Take the -- operator. If we eliminate hyphens in a user name field, then that will cause problems for people with hyphenated names.

2) Reject input that is known to be bad. This is one good technique to use as a part of an input validation routine. In his paper he has a code example that will filter out the strings “select”, “insert”, “update”, “delete”, “drop”, the double-hyphen and the single quote. If your application does not need these characters, filter them out.

3) Allow only good input. After removing known bad input, such as the SQL reserved words above, then validate the input for known good characters, such as the characters in Litwin’s regular expression shown above.

The process I recommend to protect against SQL injection is to work with database administrators to ensure a low-privilege account is used by the web application, and ensure that the only accessible stored procedures are the ones the DBA or web developer wrote for that application. Then ensure the web developers validate all input with routines such as those in #2 and #3 above. That is, remove any known bad strings, such as the SQL reserved words, and then check the remainder against the known good characters. By doing this, other characters that should not be present, such as quotes, periods, double-hyphens and semicolons would be filtered out, greatly reducing the risk of SQL injection attacks.

Hidden Form Elements and Cookies

A third kind of vulnerability regularly found in web applications is not validating hidden form elements or cookies. Melbourne and Jorm write in “Penetration Testing for Web Applications (Part One)” that unvalidated item prices in web shopping carts “is still common on many sites, though to a lesser degree”. I still see this regularly in custom-written applications. Web developers have a more difficult time testing their code for these vulnerabilities because the data is transferred behind the scenes, not using user-accessible fields in the web browser. The same is true of cookies, which are either stored on disk or in memory. To discuss vulnerabilities of hidden fields and cookies, I will introduce the use of the Achilles web proxy tool as a way to test how an application reacts to unexpected changes in these values.

A good vulnerability scanning tool will test all the input fields in a form, both visible and hidden. Hidden fields and cookies are frequently used to maintain state or logon information within applications. If these fields are modified and not validated, it is possible for a user to access another user’s account, or change prices for items being purchased.

A user can view these hidden fields if they do a “view source” within their browser, and they can view their persistent cookies if they look in the directory where their browser stores them. But it is much harder to see session cookies, which are stored in memory and not saved to disk. Achilles lets the user see all

the data traveling from the web browser to the web server, including cookies, html and http headers, even if SSL-encrypted.

An example of manipulating a hidden field that stores the price of an item in a shopping cart is shown in this form code, from "Penetration Testing for Web Applications (Part One)" by Melbourne and Jorm:

```
<FORM METHOD="LINK" ACTION="/shop/checkout.htm">
<INPUT TYPE="HIDDEN" name="quoteprice" value="4.25">Quantity:
<INPUT TYPE="text"
NAME="totalnum"> <INPUT TYPE="submit" VALUE="Checkout">
</FORM>
```

If the web server did not validate or ignore the value coming back from the browser, the user could purchase something at a lesser price.

Cookies can be manipulated as well. In "Penetration Testing for Web Applications (Part Three)" Melbourne and Jorm show how cookies are set and retrieved. As part of the response to a request for a web resource, the web server sends a statement like this in the http header:

```
Set-Cookie: SESSION=123456; path=/; expires=Sunday, 24-Oct-04
23:23:23 GMT; domain=somedomain.com
```

For each subsequent request for a resource in that path "/", the client sends that cookie along with the request. In this case, the cookie is a server-generated session identifier.

An attacker could modify the session identifier cookie during a subsequent request and see if he could access someone else's account. If so, they could potentially perform actions as that user. In most commercial application servers, the session ID is not sequentially assigned but in some cases can be reverse-engineered, even if encrypted or hashed. An encrypted cookie could look like this:

```
Set-Cookie: SESSION=A$0(k38v; path=/; expires=Sunday, 24-Oct-04
23:23:23 GMT; domain=somedomain.com
```

Depending on the encryption algorithm being used, and the amount of data an attacker can gather on the application, this value could still be spoofed, allowing an attacker access to someone else's account.

In Chapter 7 of Hacking Exposed Web Applications, Scambray and Shema go into some detail on how an attacker could derive other valid session values for various application servers. This analysis is beyond the scope of this paper, but several of their countermeasures are germane to this discussion (199)

Strong hashes or encrypted contents

Place dynamic data, such as a timestamp, as part of the string.

Enforce concurrent login limits.

If a second connection comes in from the same ID, possibly close out both sessions.

Validate contents of state information

Use similar validation checks as described earlier for SQL injection for the fields that carry state

Use checksums or message authentication techniques

Use multiple related values so that if one changes but not the other, it invalidates the session. An interesting paper available from Advosys Consultants, Inc. includes techniques for hashing certain hidden fields to detect whether they have been modified.

Use SSL

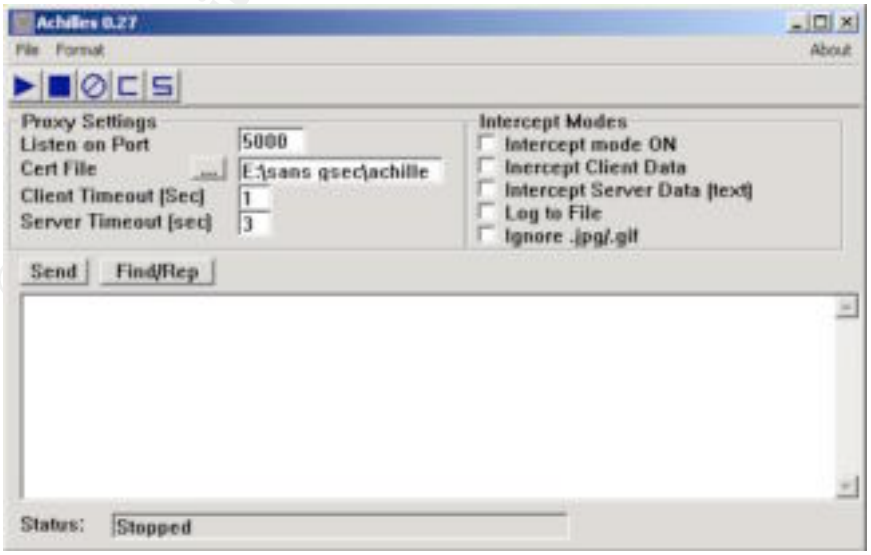
It is not a solution to every problem but it does limit sniffing attacks

If a web developer is writing code to protect against these attacks, how can Achilles be used to manipulate these values to see if the defensive code works properly? This will be shown below, by using Achilles to modify a hidden form field.

The Achilles Web Proxy as a testing tool

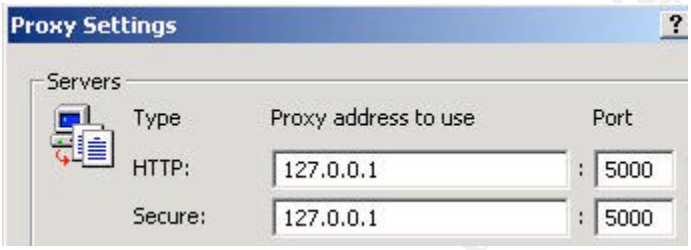
Achilles is a free tool, written by Robert Cardona and David Rhoades and is available at www.mavensecurity.com/achilles. Achilles works as a web proxy; it acts as a pass-through for http requests, and allows the user to control the traffic flows between the browser and server. This makes it easy to change data in mid-stream for testing. Achilles has the additional great advantage of being able to intercept SSL-encrypted HTTPS traffic. This means the developer can see all the communication within an encrypted session.

After installing and running Achilles the main configuration screen appears:



By default, Achilles listens on port 5000 although this is easily changed by modifying the “Listen on Port” field.

You will need to reconfigure your web browser proxy settings to point to port 5000 to have Achilles proxy your requests. Internet Explorer may need to be closed and restarted in order for those changes to take effect. This change can be made in the Internet Explorer Tools -> Internet Options -> Connections tab. Click on LAN settings, then click Advanced. Set both HTTP and Secure to your local loopback address (127.0.0.1) and set the port to the Achilles default port of 5000, unless you changed it on the screen shown above.



Internet Explorer Proxy Settings Page

To start using Achilles to proxy your web browser traffic, check all the boxes under Intercept Modes. “Log to file” is optional and will create a record of the data transferred. If “Ignore .jpg/.gif” is checked you will not see the GET requests for those files, only the html pages. If your site has lots of images, seeing all the GETs for those can be distracting. Click the blue arrow in the upper left corner to begin capturing data.

With Intercept Mode on, Achilles will not forward any request from the browser to the web server on the Internet until the user clicks the “Send” button inside Achilles. Because it is intercepting the traffic, the user can modify the information after the web browser created the request, but before the web server sees it. In effect, Achilles is facilitating a “man-in-the-middle” attack, but at the request and under the control of the user. If you have checked the Intercept Client data and Intercept Server Data, you will need to click the Send button after every GET and web server response.

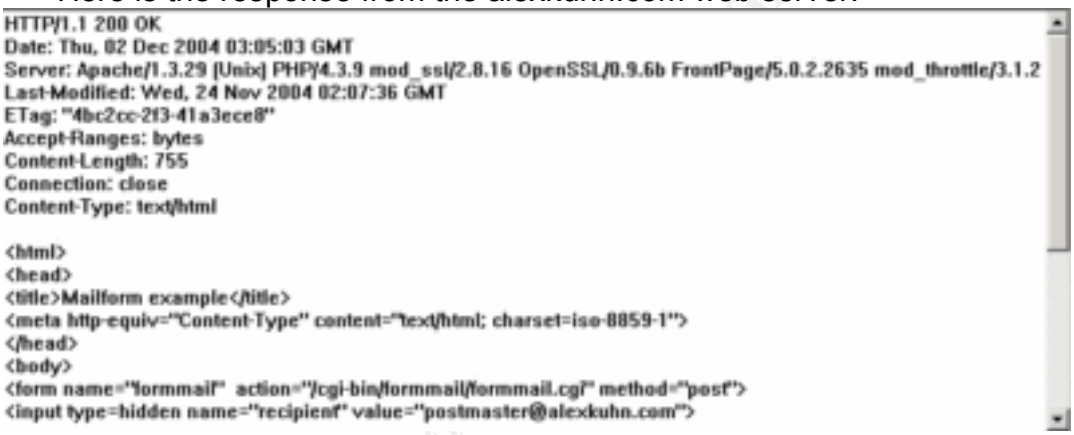
In the below screen shots, we will load an example support web page that uses the Formmail script, written by Matthew M. Wright (Formmail). This form allows a user to submit a support question to a webmaster, and the form has the webmaster email address hardcoded as a hidden form field. Without changing the code on the web server, we will change the email address to which the script will send the support request. This will demonstrate how Achilles allows you to change data after it is submitted from the web browser, but before it is sent to the server. For this test we will use an example web site I constructed for this demonstration which is called alexkuhn.com.

Each screen that is displayed as Achilles intercepts the data stream is shown. Note that because the “Ignore .jpg/.gif” box is checked we will not see any of the requests for those image files, but this particular form does not use images.

Here is the initial GET for the Formmail page (/cp/scripts/formmail-doc/example.html) from the web browser:

```
GET /cp/scripts/formmail-doc/example.html HTTP/1.0
Accept: */*
Referer: http://alexkuhn.com/cp/scripts/formmail.html
Accept-Language: en-us
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: alexkuhn.com
```

Here is the response from the alexkuhn.com web server:



```
HTTP/1.1 200 OK
Date: Thu, 02 Dec 2004 03:05:03 GMT
Server: Apache/1.3.29 (Unix) PHP/4.3.9 mod_ssl/2.8.16 OpenSSL/0.9.6b FrontPage/5.0.2.2635 mod_throttle/3.1.2
Last-Modified: Wed, 24 Nov 2004 02:07:36 GMT
ETag: "4bc2cc-2f3-41a3ece8"
Accept-Ranges: bytes
Content-Length: 755
Connection: close
Content-Type: text/html

<html>
<head>
<title>Mailform example</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>
<form name="formmail"  action="/cgi-bin/formmail/formmail.cgi" method="post">
<input type=hidden name="recipient" value="postmaster@alexkuhn.com">
```

You can see that this form uses a hidden field, called “recipient” and sets it to the value “postmaster@alexkuhn.com”.

Now let’s do this again but manipulate the request, using a test that will show whether data is being changed in Achilles as expected. This time, we will request the same page in the web browser, but will change that within Achilles to a page that is not present. If we get back a 404 “Not Found” response, we know Achilles has intercepted the data, forwarded it on to the web server and received a response based on our modification.

After the GET is issued from the web browser, the data is displayed in Achilles as in the first screen shown above. This time, put your cursor after the GET / and type in some random page, as shown below:

```
GET /cp/scripts/formmail-doc/badpage.html HTTP/1.0
Accept: */*
Referer: http://alexkuhn.com/cp/scripts/formmail.html
Accept-Language: en-us
Proxy-Connection: Keep-Alive
If-Modified-Since: Wed, 24 Nov 2004 02:07:36 GMT
If-None-Match: "4bc2cc-2f3-41a3ece8"
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: alexkuhn.com
Pragma: no-cache
```

Then click the Send button.

The response you get from the web server shows the 404 Not Found error and a custom Not Found page.

```
HTTP/1.1 404 Not Found
Date: Thu, 02 Dec 2004 03:06:43 GMT
Server: Apache/1.3.29 (Unix) PHP/4.3.9 mod_ssl/2.8.16 OpenSSL/0.9.6b FrontPage/5.0.2.2635 mod_throttle/3.1.2
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>404 Not Found</TITLE>
</HEAD><BODY>
<H1>Not Found</H1>
The requested URL /cp/scripts/formmail-doc/badpage.html was not found on this server.<P>
<HR>
<ADDRESS>Apache/1.3.29 Server at alexkuhn.com Port 80</ADDRESS>
</BODY></HTML>
```

The page as displayed in the web browser:

Not Found

The requested URL /cp/scripts/formmail-doc/badpage.html was not found on this server.

This demonstrates that the request was intercepted and changed within Achilles between the browser request and reception by the server. It should be apparent that if you can change the page you requested in-between the browser and the web server, that other data could be changed as well: cookies, http headers, session identifiers, and hidden form parameters. Let's try this with a hidden field.

The web browser View Source function can display the form code. If doing this on a large html page, use the Find function (Ctrl-F) to bring up the Find window and search for the string "form", which is the tag in html that creates an input form. In this example the page is very small so the form is easy to find. The text from this form is shown below. Note the hidden field which is boldfaced.

```
<form name="formmail"  action="/cgi-bin/formmail/formmail.cgi" method="post">
<input type=hidden name="recipient" value="postmaster@alexkuhn.com">
<p><b>Subject:</b><br>
<input type=text name="subject" size="45" value=""></p><br>
<p><b>Your Name:</b><br>
<input type=text name="realname" size="45" value=""></p><br>
<p><b>Your Email Address:</b><br>
<input type=text name="email" size="45" value=""></p><br>
<p><b>Comments:</b><br>
<textarea name="body" rows="10" cols="60" wrap="virtual"></textarea></p><br>
<p><input type="submit" value="Send Email"> <input type="reset" value="Reset
Form"></p>
</form>
```

This hidden field is the destination for the email that the script will generate. Since it is hidden it is not displayed in the web browser so the user cannot change it. However, if we change the destination in Achilles and the response from the web server shows our alternate email address, not postmaster@alexkuhn.com, then we will have shown not only that Achilles allowed us to perform a validation test on the web server's cgi script, but also that that script did not restrict the email address to only the expected value.

When an HTML form is submitted, all the input fields are appended onto a string composed of the field names, an equals sign, and the value for the field. An ampersand (&) separates the fields.

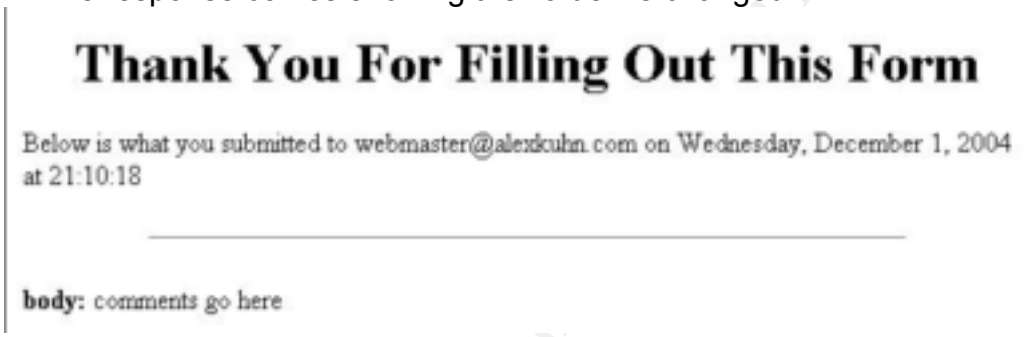
In this example, we will change the hidden field value and see if it persists in the response. I used simple values in the other fields to demonstrate how the data from multiple form fields is concatenated:

```
POST /cgi-bin/formmail/formmail.cgi HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-powerpoint,
application/vnd.ms-excel, application/msword, application/x-shockwave-flash, */*
Referer: http://alexkuhn.com/cp/scripts/formmail-doc/example.html
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: alexkuhn.com
Content-Length: 105
Pragma: no-cache

recipient=postmaster@alexkuhn.com&subject=test&realname=Alex&email=foo@alexkuhn.com&body=comments
+go+here
```

We will change the recipient value from postmaster@alexkuhn.com to another value, webmaster@alexkuhn.com:

```
POST /cgi-bin/formmail/formmail.cgi HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-powerpoint,
application/vnd.ms-excel, application/msword, application/x-shockwave-flash, */*
Referer: http://alexkuhn.com/cp/scripts/formmail-doc/example.html
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: alexkuhn.com
Content-Length: 105
Pragma: no-cache

recipient=webmaster@alexkuhn.com&subject=test&realname=Alex&email=foo@alexkuhn.com&body=comments
+go+here
```

The response comes showing the value we changed:



Thank You For Filling Out This Form

Below is what you submitted to webmaster@alexkuhn.com on Wednesday, December 1, 2004 at 21:10:18

---

body: comments go here

We changed the destination email from *postmaster* to *webmaster*. This same technique of testing different values can be used on **any value** in the web application, including cookies and http headers, not just the hidden field and URL GET demonstrated here. If you use this technique of manipulating hidden fields or cookies, you should understand which fields hold the most important information, such as session or user identification credentials, and test these fields for manipulations such as I have demonstrated here. Some of the works in the List of References to this paper describe ways to defend against these kinds of manipulations.

Summary

The web developer's challenge of securing potential vulnerabilities is a daunting one. New vulnerabilities are discovered every day, and it is difficult for even full-time security professionals to stay abreast of every new risk. But many newly discovered vulnerabilities are similar to other vulnerabilities, and can be mitigated through the same kinds of techniques.

This paper has shown some of the more common, significant vulnerabilities that continue to surface in applications, and ways for a web developer to test for these vulnerabilities, and to test the fixes to determine if they are effective. By implementing a defense-in-depth strategy, where not only the network infrastructure, operating systems, and web servers are secured, but also the applications themselves, you can be more confident in your ability to repel potential attackers against both known and future attacks.

List of References

Anley, Chris. "Advanced SQL Injection in SQL Server Applications" 24 Oct. 2004 <http://www.nextgenss.com/papers/advanced_sql_injection.pdf>

Anley, Chris. "(more) Advanced SQL Injection" 24 Oct. 2004 <http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf>

Cardona, Robert and Rhoades, David. Achilles 24 Oct. 2004 <http://www.mavensecurity.com/achilles>

Clover, Andrew. "GOBBLES SECURITY ADVISORY #33." Online posting, May 11, 2002 Bugtraq. 24 Oct. 2004. <http://www.securityfocus.com/archive/1/272037>

Cole, Eric et al. SANS Security Essentials, version 2.2, volume 1.2: Defense In-Depth. 2004.
--- SANS Security Essentials, version 2.2, volume 1.3: Internet Security Technologies. 2004.

Dhanjani, Nitesh. Web App Security Testing with a Custom Proxy Server. 22 Jan 2004 <http://www.onlamp.com/pub/a/php/2004/01/22/php_proxy.html>

"FAQ". Online Web Application Security Project. 24 Oct. 2004 <http://www.owasp.org/documentation/appsec_faq.html>

Howard, Michael and LeBlanc, David. Writing Secure Code, 2nd edition. Redmond, Washington: Microsoft Press. 2003.

"How To Remove Meta-characters From User-Supplied Data in CGI Scripts." Cert.org. 24 Oct. 2004 <http://www.cert.org/tech_tips/cgi_metacharacters.html>

Internet Security Systems Internet Scanner. <http://www.iss.net/find_products/vulnerability_assessment.php>

Litwin, Paul. "Stop SQL Injection Attacks Before they Stop You." Microsoft Developer Network. 24 Oct. 2004 <http://msdn.microsoft.com/msdnmag/issues/04/09/SQLInjection/default.aspx>

Melbourne, Jody and Jorm, David. "Penetration Testing for Web Applications" (in three parts). 24 Oct. 2004
<http://www.securityfocus.com/printable/infocus/1704>
<http://www.securityfocus.com/printable/infocus/1709>
<http://www.securityfocus.com/printable/infocus/1722>

"Microsoft HOWTO: Prevent Cross-Site Scripting Security Issues." Microsoft. 24 Oct. 2004 <http://support.microsoft.com/default.aspx?scid=kb;EN-US;q252985>

"Paros" web vulnerability assessment tool. 24 Oct. 2004 <http://www.proofsecure.com>

"Preventing HTML form tampering" 7 Aug. 2001 Advosys Consulting, Inc. <http://advosys.ca/tips/form-tampering.html>

Rhoades, David. "Hacking Web Apps" 24 Oct. 2004 <http://www.mavensecurity.com/presentations>

Scambray, Joel and Shema, Mike. "Hacking Exposed – Tools" 24 Oct. 2004 <http://www.webhackingexposed.com/tools.html>

Scambray, Joel and Shema, Mike. Hacking Exposed Web Applications. Berkeley: McGraw-Hill/Osborne, 2002.

Sima, Caleb. Security at the next level: Are your web applications vulnerable? <http://www.spidynamics.com/support/whitepapers/webappwhitepaper.pdf>

"Sleuth: web proxy." 24 Oct. 2004 <http://www.sandsprite.com/Sleuth>

Spett, Kevin. Blind SQL Injection: Are your web applications vulnerable? SPIDynamics. 24 Oct. 2004 <http://www.spidynamics.com/support/whitepapers/Blind_SQLInjection.pdf>

Spett, Kevin. Cross-site scripting: Are your web applications vulnerable? SPIDynamics. 24 Oct. 2004 <http://www.spidynamics.com/support/whitepapers/SPIcross-sitescripting.pdf>

Spett, Kevin. SQL Injection: Are your web applications vulnerable? SPIDynamics. 24 Oct. 2004 <http://www.spidynamics.com/support/whitepapers/WhitepaperSQLInjection.pdf>

Thompson, Herbert & Whittaker, James "String-Based Attacks Demystified." Dr. Dobb's Journal, June 2004, 61-63

Wright, Matthew M. "Formmail" 6 Dec. 2004. <http://www.scriptarchive.com/formmail.html>