Announcements

• Starting recording
• Please turn on video if you can
• Midterm 2 April 6 at 5pm PT
  – The exam will cover lectures from after midterm 1 (starting with hashing) until April 3rd
  – Review Saturday April 4 at 5pm Pacific
• Homework 3a, due Sunday, April 5, at 11:59pm PST
  – Mid-semester survey attached
Web security attacks
What can go bad if a web server is compromised?

- Steal sensitive data (e.g., data from many users)
- Change server data (e.g., affect users)
- Gateway to enabling attacks on clients
- Impersonation (of users to servers, or vice versa)
- Others
A set of common attacks

- SQL Injection
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query
- XSS – Cross-site scripting
  - Attacker inserts client-side script into pages viewed by other users, script runs in the users’ browsers
- CSRF – Cross-site request forgery
  - Bad web site sends request to good web site, using credentials of an innocent victim who “visits” site
Injection attacks
Historical perspective

• The first public discussions of SQL injection started appearing around 1998

In the Phrack magazine
First published in 1985

Hundreds of proposed fixes and solutions
# Top web vulnerabilities

<table>
<thead>
<tr>
<th>OWASP Top 10 - 2013</th>
<th>OWASP Top 10 - 2017</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A1</strong> – Injection</td>
<td>!!!</td>
</tr>
<tr>
<td><strong>A2</strong> – Broken Authentication and Session Management</td>
<td><strong>A2:2017-Broken Authentication</strong></td>
</tr>
<tr>
<td><strong>A3</strong> – Cross-Site Scripting (XSS)</td>
<td><strong>A3:2017-Sensitive Data Exposure</strong></td>
</tr>
<tr>
<td><strong>A4</strong> – Insecure Direct Object References [Merged+A7]</td>
<td><strong>A4:2017-XML External Entities (XXE) [NEW]</strong></td>
</tr>
<tr>
<td><strong>A5</strong> – Security Misconfiguration</td>
<td><strong>A5:2017-Broken Access Control [Merged]</strong></td>
</tr>
<tr>
<td><strong>A6</strong> – Sensitive Data Exposure</td>
<td><strong>A6:2017-Security Misconfiguration</strong></td>
</tr>
<tr>
<td><strong>A8</strong> – Cross-Site Request Forgery (CSRF)</td>
<td><strong>A8:2017-Insecure Deserialization [NEW, Community]</strong></td>
</tr>
<tr>
<td><strong>A9</strong> – Using Components with Known Vulnerabilities</td>
<td><strong>A9:2017-Using Components with Known Vulnerabilities</strong></td>
</tr>
<tr>
<td><strong>A10</strong> – Unvalidated Redirects and Forwards</td>
<td><strong>A10:2017-Insufficient Logging&amp;Monitoring [NEW,Comm.]</strong></td>
</tr>
</tbody>
</table>

Please don’t repeat common mistakes!!
General code injection attacks

- Attacker user provides bad input
- Web server does not check input format
- Enables attacker to execute arbitrary code on the server

- What attack does this remind you of?
Example:
code injection based on eval (PHP)

- $_GET[‘A’]: gets the input with value A from a GET HTTP request
- $_POST[‘B’]: gets the input with value B from a POST HTTP request

1. User visits calculator and writes 3+5 ENTER
3. Script at server receives http request and runs
   $_GET(“exp”) =“ 3+5”
Example: code injection based on eval (PHP)

- eval allows a web server to evaluate a string as code
  - e.g. eval('{$result = 3+5}’) produces 8

calculator: http://site.com/calc.php

http://site.com/calc.php?exp="3+5"

$exp = $_GET['exp'];
eval('{$result = ' . $exp . ';'});

Attack?:

http://site.com/calc.php?exp="3+5 ; system('rm *.*')"
Code injection using system()

- Example: PHP server-side code for sending email

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

Attacker can post

http://yourdomain.com/mail.php?
email=hacker@hackerhome.net &
subject="foo < /usr/passwd; ls"
SQL injection
Structure of Modern Web Services

URL / Form: command.php?arg1=x&arg2=y
Structure of Modern Web Services

Browser

URL / Form
command.php?arg1=x&arg2=y

Web server

Database query built from x and y

Database server
Structure of Modern Web Services

Browser

Web server

Custom data corresponding to x & y

Database server
Structure of Modern Web Services

Browser

Web page built using custom data

Web server

Database server
Databases

- **Structured** collection of data
  - Often storing tuples/rows of related values
  - Organized in tables

<table>
<thead>
<tr>
<th>AcctNum</th>
<th>Username</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1199</td>
<td>zuckerberg</td>
<td>35.7</td>
</tr>
<tr>
<td>0501</td>
<td>bgates</td>
<td>79.2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Databases

- Widely used by web services to store server and user information
- Database runs as separate process to which web server connects
  - Web server sends queries or commands derived from incoming HTTP request
  - Database server returns associated values or modifies/updates values
SQL

- Widely used database query language
  – (Pronounced “ess-cue-ell” or “sequel”)
- Fetch a set of rows:
  \[
  \text{SELECT column FROM table WHERE condition}
  \]
  returns the value(s) of the given column in the specified table, for all records where condition is true.
- e.g:
  \[
  \text{SELECT Balance FROM Customer WHERE Username='bgates'}
  \]
  will return the value 79.2
SQL (cont.)

• Can add data to the table (or modify):

\[
\text{INSERT INTO Customer VALUES (8477, 'oski', 10.00)};
\]

<table>
<thead>
<tr>
<th>AcctNum</th>
<th>Username</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1199</td>
<td>zuckerberg</td>
<td>35.7</td>
</tr>
<tr>
<td>0501</td>
<td>bgates</td>
<td>79.2</td>
</tr>
<tr>
<td>8477</td>
<td>oski</td>
<td>10.00</td>
</tr>
</tbody>
</table>

...
SQL (cont.)

• Can delete entire tables:
  
  DROP TABLE Customer

• Issue multiple commands, separated by semicolon:
  
  INSERT INTO Customer VALUES (4433, 'vladimir', 70.0); SELECT AcctNum FROM Customer WHERE Username='vladimir'

  returns 4433.
SQL Injection Scenario

- Suppose web server runs the following code:

  ```php
  $recipient = $_POST['recipient'];
  $sql = "SELECT AcctNum  FROM Customer
          WHERE Username='$recipient' ";
  $rs = $db->executeQuery($sql);
  ```

- Server stores URL parameter “recipient” in variable $recipient and then builds up a SQL query
- Query returns recipient’s account number
- Server will send value of $sql variable to database server to get account #s from database
SQL Injection Scenario

• Suppose web server runs the following code:

```php
$recipient = @$_POST['recipient'];
$sql = "SELECT AcctNum FROM Customer WHERE Username='$_recipient' ";
$rs = $db->executeQuery($sql);
```

• So for “?recipient=Bob” the SQL query is:

```
"SELECT AcctNum FROM Customer WHERE Username='Bob' 
```
Basic picture: SQL Injection

1. **post malicious form**
   - $recipient specified by attacker

2. **Victim Web Server**
   - unintended SQL query

3. **SQL DB**
   - receive valuable data

How can $recipient cause trouble here?
Problem

Untrusted user input ‘recipient’ is embedded directly into SQL command

Attack:

$recipient = alice'; SELECT * FROM Customer;'  

Returns the entire contents of the Customer!
CardSystems Attack

• CardSystems
  – credit card payment processing company
  – SQL injection attack in June 2005
  – put out of business

• The Attack
  – 263,000 credit card #s stolen from database
  – credit card #s stored unencrypted
  – 43 million credit card #s exposed
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 front-end 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
Another example: buggy login page (ASP)

```asp
set ok = execute(

"SELECT * FROM Users
    WHERE user='" & form("user") & "'
    AND pwd='" & form("pwd") & "'");

if not ok.EOF
    login success
else
    fail;
```
Normal Query

Web Browser (Client) → Web Server → DB

Enter Username & Password

SELECT *
FROM Users
WHERE user='me'
AND pwd='1234'

(1 row)
Another example: buggy login page (ASP)

```asp
set ok = execute( "SELECT * FROM Users
    WHERE user=' " & form("user") & 
    " '
    AND pwd=' " & form("pwd") & 
    "' ");

code
if not ok.EOF
    login success
else  fail;
```

Is this exploitable?
Bad input

• Suppose user = “'or 1=1 -- ” (URL encoded)

• Then scripts does:
  ok = execute( SELECT ...
               WHERE user=' ' or 1=1 -- 
               ...
  )
  – The “--” causes rest of line to be ignored.
  – Now ok.EOF is always false and login succeeds.

• The bad news: easy login to many sites this way.

Besides logging in, what else can attacker do?
Even worse: delete all data!

• Suppose user = 
  "' ; DROP TABLE Users -- "

• Then script does:

  \[
  \text{ok = execute( SELECT ...} \\
  \text{    WHERE user= ' ' ; DROP TABLE Users ... })
  \]
What else can an attacker do?

• Add query to create another account with password, or reset a password

Suppose user =

```
    "'"; INSERT INTO TABLE Users ('attacker', 'attacker secret');
```

And pretty much everything that can be done by running a query on the DB!
How to prevent SQL injection?

- Ideas?
SQL Injection Prevention

• Sanitize user input: check or enforce that value/string does not have commands of any sort
  ◆ Disallow special characters, or
  ◆ Escape input string

SELECT PersonID  FROM People WHERE Username='alice\'; SELECT * FROM People;’
How to escape input

You “escape” the SQL parser

Web Server → query → Parser → commands → DB
How to escape input

• The input string should be interpreted as a string and not as a special character
• To escape the SQL parser, use backslash in front of special characters, such as quotes or backslashes
The SQL Parser does...

- If it sees ' it considers a string is starting or ending
- If it sees \ it considers it just as a character part of a string and converts it to '

For

```
SELECT PersonID FROM People WHERE Username='alice\'; SELECT * FROM People;
```

The username will be matched against 'alice'; SELECT * FROM People;' and no match found

- Different parsers have different escape sequences or API for escaping
Examples

• What is the string username compared to (after SQL parsing), and when does it flag a syntax error? (syntax error appears at least when quotes are not closed)

```sql
[..] WHERE Username='alice';  alice
[..] WHERE Username='alice\';  Syntax error, quote not closed
[..] WHERE Username='alice"';  alice'
[..] WHERE Username='alice\\';  alice\  
    because \\ gets converted to \ by the parser
```
SQL Injection Prevention

• Avoid building a SQL command based on raw user input, use existing tools or frameworks

• E.g. (1): the Django web framework has built in sanitization and protection for other common vulnerabilities
  – Django defines a query abstraction layer which sits atop SQL and allows applications to avoid writing raw SQL
  – The execute function takes a sql query and replaces inputs with escaped values

• E.g. (2): Or use parameterized/prepared SQL
Parameterized/prepared SQL

- Builds SQL queries by properly escaping args: ' → \'

- Example: Parameterized SQL: (ASP.NET 1.1)
  - Ensures SQL arguments are properly escaped.

```csharp
SqlCommand cmd = new SqlCommand("SELECT * FROM UserTable WHERE username = @User AND password = @Pwd", dbConnection);

cmd.Parameters.Add("@User", Request["user"]);

cmd.Parameters.Add("@Pwd", Request["pwd"]);

cmd.ExecuteReader();
```
How to prevent general injections

Similarly to SQL injections:

• Sanitize input from the user!
• Use frameworks/tools that already check user input
HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR — DID HE BREAK SOMETHING?
IN A WAY—
Summary

• Injection attacks were and are the most common web vulnerability

• It is typically due to malicious input supplied by an attacker that is passed without checking into a command; the input contains commands or alters the command

• Can be prevented by sanitizing user input