Common Vulnerabilities

Junghoo Cho

cho@cs.ucla.edu

Common Vulnerabilities - Junghoo Cho - cho@cs.ucla.edu

Dangerous Software Errors

- OWASP Top Ten
- CWE Top 25
  1. Improper Restriction of Operations within the Bounds of a Memory
  2. Improper Neutralization of Input During Web Page Generation (Cros Scripting)
  3. Improper Input Validation
  4. Information Exposure
  5. Out-of-bounds Read
  6. Improper Neutralization of Special Elements used in an SQL Comma Injection) …

What We Will Discuss

- Buffer overflow
- SQL/command injection
- Client state manipulation
- Cross-site scripting (XSS)
- Cross-site request forgery (XSRF)

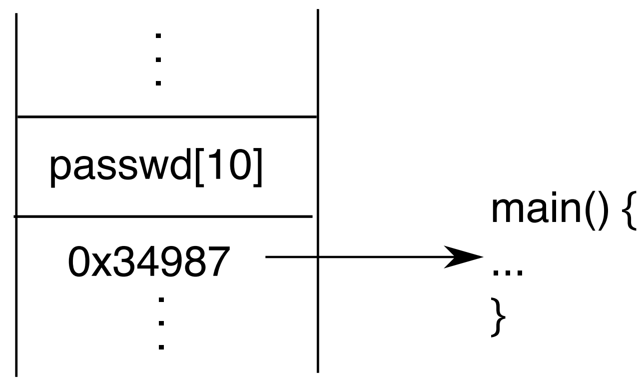
Buffer Overflow

```
int main() {
    if (login()) start_session();
    return 0;
}

int login() {
    char passwd[10];
    gets(passwd);
    return (strcmp(passwd, "mypasswd") == 0);
}

int start_session() {
    ...
}
```

- Q: Anything wrong?

Stack and Local Variable

- Q: `main()` → `login()`. How does the system know where to after a function call?
- Structure of stack after call `main()` → `login()`



- Q: What will happen if the user input is longer than 10 chara

Buffer Overflow Attack

- By making a local variable “overflow”, a malicious user may ju
part of the program
- *Attack string*: carefully constructed user input for attack

Standard C Is Dangerous

- ***NEVER*** use C str functions, such as `gets`, `strcpy`, `strcat`, `spr`
- Modern languages like Java, C#, JavaScript, …, are safer
  - They explicitly check for incorrect address and array bounds
  - This doesn't mean that their implementation isn't buggy and vulner
- Most of all, ***NEVER trust user input!!!***

Stackguard

- General protection mechanism for legacy C code without co rewriting
  - Only recompilation is needed
- Inserts random *canary* before return address and checks cor before return.
  - Not a complete protection, but covers most common attack
  - `-fstack-protector-all` for `gcc`

SQL/Command Injection Attack

```
"SELECT price FROM Product WHERE prod_id = " + user_input + ";"
```

- Q: Any problem?
- Q: What if `user_input` is `"1002 OR TRUE"`? Q: What if `user_i`
  `"0; SELECT * from CreditCard"`?
- CardSystems lost 263,000 card numbers through SQL injecti
  and went out of business

```
system("cp file1.dat $user_input");
```

- Q: Any problem?

SQL Injection: Protection

- Basic idea: ***NEVER trust user input***
  - Reject unless it is absolutely safe
- Use prepared statements and bind variables

```
String sql = "SELECT * from Product WHERE id = ?";
PreparedStatement s = db.prepareStatement(sql);
s.setInt(1, Integer.parseInt(user_input));
ResultSet rs = s.executeQuery();
```

  - Only integers can make it into SQL
  - *Input validation + white listing*

Command Injection: Protection

- JavaScript `eval()` is dangerous. Never use it
  - Including `exec()` in C/C++/php/…
- Java `Runtime.exec(command_string)` is safer
  - Executes only the first word as a command and the rest as paramete
- *Taint propagation*
  - User supplied strings are marked "tainted"
  - If tainted string is used inside sensitive commands (SQL, shell,…) sy generates error
  - Tainted string must be explicitly "untainted" by programmer
  - Supported in Perl, Ruby, …

Mitigating Damage

- Contain damage even after a successful attack
  - Give *only necessary privileges* to your application
  - Encrypt sensitive data even for local storage
    - Never store user passwords in plain text!

Client State Manipulation

```
<form>
    <input type="hidden" name="price" value="5.50">
    ...
</form>
```

- Q: Any problem?
- Similar problems with cookies
  - Whenever “state” is provided by a client
- Q: How can we avoid the problem?

Client State Manipulation: Protection

Basic idea: ***NEVER trust user input***

1. Authoritative state stays at the server
    - Idea: store values only at the server and send a session ID only
    - *Session ID*: random number generated by the server
    - To avoid stolen session ID attack
        - Pick a random session ID from a large pool
        - Make session ID short lived
2. Send signed-states to client
    - Detect tempering by checking the signature
    - Make the state short-lived
        - e.g., price fluctuation over time

Cross Site Scripting (XSS)

```
<body>
Welcome to {{user_name}}'s Profile!
</body>
```

- Q: Any problem?
- Q: What will happen if `user_name` is `John Cho<script src="http://x.com/hack.js"></script>`?
- If a page includes user-provided string, a hacker may execut JavaScript code in people's browser!

XSS: Protection

- Q: Do not allow any HTML tag?
- At the minimum, escape `<`, `>`, `&`, `"`, `'`
- Q: What if HTML tags must be allowed (like HTML email)?
- Note
  - See example XSS attack vectors
  - Complete protection against all XSS attacks in general is VERY diffic
  - Important to use *white list* as opposed to *black list*
  - Use both input validation and output sanitization

`Content-Security-Policy` Header

- Generalization of same-origin policy
- Explicitly control the list of allowed content "origins"
- Can be used to mitigate XSS vulnerability
- Example:
  - `Content-Security-Policy: default-src 'self' *.trusted.com`
    - Include resources only from the same host or from `*.trusted.com`
    - Separate policy for `img-src`, `script-src`, …, may be specified

Cross-Site Request Forgery (XSRF)

- HTTP cookie
  - Arbitrary name/value pair set by the server and stored by client
  - Session cookie: track an authenticated user's login session
  - Same-origin policy
    - A script can access only cookies from the same site
    - Cookies are sent back only to the same site
    - Minimal data protection from malicious web sites
- Q: Can a malicious page "see" cookie from another site in the

XSRF: Example

1. A user visits http://victim.com and does not logged out
2. The user visits the following page at http://evilsite.com

```
<form action="http://victim.com/transfer" onload="submit()">
    <input type="hidden" name="amount" value="$1M">
    <input type="hidden" name="to" value="hacker">
</form>
```

- Q: What will happen? Will http://victim.com reject the reque

XSRF: Problem

- Attacker cannot “see” a cookie but they can still “use” it!
  - Same-origin policy prevents an attacker from "see"ing it
  - But they can send it with their own request
- Q: How can we prevent it?
- Idea: Ask user for a password for every request?

XSRF: Protection

- Basic idea: Make sure that all valid requests include a “secret
  malicious page cannot include
- *Action token*
  1. Generate an action token: secret-key signed signature of session ID
     - Assume session ID is random, unique per session, short lived, and hard to gue:
  2. Embed the action token as a hidden field in a form
  3. When request is received, validate received action token
- Q: Can a malicious page include a valid action token?

What We Learned

- Buffer overflow
- SQL/Command injection
- Client state manipulation
- Cross-site scripting
- Cross-site request forgery
- Input validation and output sanitization
- White list, not black list
- ***NEVER TRUST USER INPUT***

Thank You

- Thank you for your hard work
- I hope you learned something useful
- Please provide feedback!
- Read Developer Roadmap to learn where to go from here