Common Web Vulnerabilities

Common Web application vulnerabilities to discuss

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

Buffer Overflow

- Example

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

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

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

  - Q: main() -> login() -> start_session(). How does the system remember where to return after a function call?
    - Structure of stack after call `main()` -> `login()`

    - Stack typically grows bottom up
  - Q: What will happen if the user-input is longer than 10 characters?
    - By making a local variable "overflow", a malicious user may jump to any part of the program
    - *Attack string*: carefully constructed user input for attack
- Modern languages like Java, C#, JavaScript, etc., are mostly safe from buffer overflow attack
  - Java runtime engine actively checks for incorrect address, buffer overflow, array-bound checking, ...
  - C++ STL string class actively checks for overflow
  - But of course, every code is bound to have a bug that may be exploited
  - ***NEVER*** use C str functions: `gets`, `strcpy`, `strcat`, `sprintf`, ...
- **Q: Any general solution?**
  - *Stackguard*: inserts random "canary" before return addr and checks corruption before return.
    - Not a complete protection against buffer overflow, but covers most common attack
    - `-fstack-protector-all` for `gcc`
  - Most of all, ***NEVER trust user input!!!***

SQL/command injection attack

- **Q: Is there any problem with the following code?**

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

  - Q: What if user_input is "1002 OR TRUE"'?
  - Q: What if user_input is "0; SELECT * from CreditCard"?
  - CardSystems lost 263,000 card numbers through SQL injection vulnerability and was acquired by another company
- **Q: Any problem?**

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

- Protection
  - ***Never trust user input!*** Reject unless absolutely safe
  - For SQL: prepared statements and bind variables
    - Example

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

    - Invalid input cannot make it into SQL. It is filtered out during parsing
  - Java `Runtime.exec(command_string)` executes the first word in the string as the command and the rest as the parameters.
    - Not as vulnerable as C/C++/php/...
  - JavaScript `eval()` is dangerous. Do NOT use it
- *Taint propagation* in Perl/Ruby
  - User supplied strings are marked "tainted"
  - If tainted string is used inside sensitive commands (SQL, shell,...) system generates error
  - Tainted string must be explicitly "untainted" by programmer
- To contain damage even after a successful attack
  - Give *only necessary privileges* to your application
  - Encrypt sensitive data in DBMS
    - Never store user passwords in plain text!

Client state manipulation

- **Q: Any problem?**

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

  - Similar problems with cookies
  - ***NEVER trust user's input!!!***

- **Q: How can we avoid the problem?**

  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)

- **Q: Any problem?**

```
Welcome to $user_name$'s page!
```

  - Q: What will happend if `$user_name` is `<script>hack()</script>`?
  - Note: If a page includes user input string, users may execute *any* JavaScript code!

- **Q: How to protect?**

  - Q: Do not allow any HTML tag?

    - At the minimum, escape &, <, >, “, ’

  - Q: What if HTML tags must be allowed (like HTML email)?
  - Q: What about `<img src="$user_url">`? $user_url can be "javascript:attack-code;"!

- Note
  - Complete protection against all XSS attack is VERY difficult
  - Important to use *white list* as opposed to *black list*
  - Use both input validation and output sanitization

Cross site request forgery (XSRF)

- HTTP cookie
  - Arbitrary name/value pair set by the server and stored by client
  - Server -> client: `Set-Cookie: foo=bar; path=/; domain=cs144.edu;`
  - Client -> server: `Cookie: foo=bar`
  - Frequently used to track a user's login session
    - Session cookies are "valid" only during a browser session
  - Q: Can a malicious page "see" cookie from another site?
  - Same-origin policy
    - A script can access only the documents and cookies that are from the same site
    - Cookies are sent back only to the same site
    - Same-origin policy gives minimal data protection from malicious web sites
- 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">
</form>
```

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

- Note
  - Due to same-origin policy, an attacker cannot "see" a session cookie from another site
  - But XSRF still allows the attacker to "use" the cookie to send a request!
- Q: How to prevent it?

  - S1: Check Referrer header?
    - Note: Referrer header may be missing for legitimate reasons
  - S2: Ask user password for every request?
- *Action token*
  - Basic idea: make sure that a valid request from our page includes a "secret" that a malicious page cannot get
  - Embed action token
    1. Generate an *action token*:
       - Action token: secret-key signed signature of session ID
       - We assume session ID is random, unique per session, short lived, and hard to guess
    2. Embed the action token as a hidden field in a form
  - Verify action token
    1. Compute the action token of the request
    2. Take action only if it matches with the session ID
- Q: Can a malicious page obtain the action token from our page?