Secure Programming

Input Validation

Ahmet Burak Can

Learning objectives

- Understand the definition of code injection
- Know how code injection happens
- Learn how to perform input validation and cleansing

Code Injection: Outline

- Definition
- Defending against code injection
- Examples
- Input Validation
- Dynamic code generation with SQL

Code Injection

- Goal: trick program into executing an attacker's code by clever input construction that mixes code and data
- Mixed code and data channels have special characters that trigger a context change between data and code interpretation
  - The attacker wants to inject these meta-characters through some clever encoding or manipulation, so supplied data is interpreted as code

Basic Example by Command Separation

- cat > example
  - `#!/bin/sh`
  - `A=$1`
  - `eval "ls $A"`
- Permissions for file "confidential" before exploit:
  - `% ls -l confidential`
    `-rwxr-x---  1 pmeunier  pmeunier confidential`
- Allow execution of "example":
  - `% chmod a+rx example`
- Exploit (what happens?)
  - `%./example ".;chmod o+r *"`

Results

- Inside the program, the eval statement becomes equivalent to:
- `eval "ls .;chmod o+r *"`
- Permissions for file "confidential" after exploit:
  - `% ls -l confidential`
    `-rwxr-xr--  1 pmeunier  pmeunier confidential`
- Any statement after the ";" would also get executed, because ";" is a command separator.
- The data argument for "ls" has become code!

Other Code Injection by Command Substitution

- Backtick: execution in a command line by command substitution
- `command` gets executed before the rest of the command line
- Imagine a malicious script called "script1":
  - `mkdir oups`
  - `echo oups`
  - `etc...`
- Imagine a program that calls a shell to run grep.
- What happens when this is run?
  - `eval "grep `./script1` afile"`

Answer

- Script1 is executed
  - an “oups” directory is created
- The rest of the intended command, “grep oups afile”, gets executed

A Vulnerable Program

```
int main(int argc, char *argv[], char **envp)
{
   char buf [100];
   buf[0] = '\0';
   snprintf(buf, sizeof(buf), "grep %s text",argv[1]);
   system(buf);
   exit(0);
}
```

What happens when this is run?

```
% ./a.out  \`./script\`
```

Answer

- The program calls
  - `system ("grep `./script` text");`
  - You can verify by adding "`printf( "%s", buf)`" to the program
- So we could make a.out execute any program we want
  - Imagine that we provide the argument remotely
  - What if a.out runs with root privileges?

Shell Metacharacters

- '`' to execute something (command substitution)
- ';' is a command ("pipeline") separator
- '|' is a pipe
- '&&'
- '||'
- '<<' or '>>'
- # to comment out something
- Refer to the appropriate man page for all characters
- How else can code be injected into a.out?

Exercise and Discussion

- Take the vulnerable code on the previous slides
- Can you get it to do something else using command separators or meta-characters?
  - Which meta-characters work best, and how?

Defending Against Code Injection

- Input cleansing and validation
  - Model the expected input
    - Discard what doesn't fit (e.g., metacharacters)
  - Keep track of which data has been cleansed
    - e.g., Perl's taint mode
  - Keep track of all sources of inputs
    - Or cleanse as the input is received
- Type and range verification, type casts
- Separating code from data
  - Transmit, receive and manipulate data using different channels than for code

Input Cleansing

- Key to preventing code injection attacks
- Common problem where code is generated dynamically from some data
  - SQL (database Simple Query Language)
  - System calls and equivalents in PHP, Windows CreateProcess, etc...
  - Environment pollution (already covered in another set of slides)
  - HTML may contain JavaScript (Cross-site scripting vulnerabilities)

Intuitive Approach

- Block or escape all metacharacters
  - but what are they?
- Problems:
  - Character encodings
    - octal, hexadecimal, UTF-8, UTF-16...
  - Obfuscation
  - Escaped characters that can get interpreted later
- Engineered strings such that by blocking a character, something else is generated

Wrong Way to Cleanse and Sanitize

```
static char bad_chars[] = "/ ;[]<>&\t";
char * user_data;
char * cp;

/* Get the data */
user_data = getenv("QUERY_STRING");

/* Remove bad characters. WRONG! */
for (cp= user_data; *(cp += strcspn(cp,bad_chars)); /**/)
   *cp = '_';
```

phf CGI

- CVE-1999-0067

```
strcpy(commandstr, "/usr/local/bin/ph -m");
escape_shell_cmd(serverstr);
strcat(commandstr, serverstr);
(...)
phfp = popen(commandstr,"r");
```

- What could be the problem?
  - besides the potential buffer overflows

Black List of Characters

```
void escape_shell_cmd(char *cmd) {
    (...)
    if(ind("&;`'\"|*?~<>^()[]{}$\\",cmd[x]) != -1){
    (...)
}
```

- Author forgot to list newlines in "if" statement...
- Exploit: input “newline” and the commands you want executed...

More Robust Cleansing

```
{...}
static char ok_chars[] =
       "1234567890!@%-_=+:,./\
        abcdefghijklmnopqrstuvwxyz\
        ABCDEFGHIJKLMNOPQRSTUVWXYZ";
                {...}
for (cp = user_data; *(cp += strspn(cp,
ok_chars)); )
                *cp = '_';
```

- http://www.cert.org/tech_tips/cgi_metacharacters.html
- a.k.a. White List vs Black List design principle

Other Input Validation Issues

- Range of types
  - Short vs long integers
  - Unsigned vs signed
- Integer overflows
  - Validate range (e.g., array indexes)
    - Attacks can make something negative to reach forbidden data
    - Attacks can reset a counter to zero
      - Data structure reference count vs garbage collection
- Strings in numerical inputs
  - e.g., PHP will accept both string and numerical values for a variable, which may allow unexpected attacks
  - Use typecasts

Order for Cleansing and Input Validation

1) Resolve all character encoding issues
2) Cleanse
   - If combinations of characters can produce metacharacters, you may need to do several passes. Example:
     - "a" and "b" are legal if separated from each other, but "ab" is considered a metacharacter. The character "d" is not allowed. After you filter out "d" from "adb", you may be allowing "ab" through the filter!
3) Validate type, range, and format
4) Validate semantics (i.e., meaning of input)

SQL

- SQL uses single and double quotes to switch between data and code.
- Semi-colons separate SQL statements
- Example query:
  - ```
    "UPDATE users
    SET prefcolor='red'
    WHERE uid='joe';"
    ```
- This command could be sent from a web front-end to a database engine.
- The database engine then interprets the command
    ```

Dynamic SQL Generation

- Web applications dynamically generate the necessary database commands by manipulating strings
- Example query generation:
  - ```
    $q = "UPDATE users
    SET prefcolor='$INPUT[color]'
    WHERE uid='$auth_user'";
    ```
- Where the value of "`$INPUT[color]`" would be originating from the client web browser, through the web server.
- And where the value for "`$auth_user`" would have been stored on the server and verified through some authentication scheme
    ```

Client Web Browser

- Forms in client browsers return values to the web server through either the POST or GET methods
  - "GET" results in a url with a "?" before the values of the form variables are specified:
    - http://www.example.com/script?color=red
    - The value of "`$INPUT[color]`" is set to "red" in the script
- "GET" urls are convenient to hack, but there isn't any significant difference in the security of either "GET" or "POST" methods because the data comes from the client web browser regardless and is under the control of the remote attacker

The SQL Table

- Tables are used to store information in fields (columns) in relation to a key (e.g., "uid")
- What other fields could be of interest?
- ```
  CREATE TABLE users (
    prefcolor varchar(20),
    uid VARCHAR(20) NOT NULL,
    privilege ENUM('normal', 'administrator'),
    PRIMARY KEY (uid)
  );
  ```
  ```

A Malicious SQL Query

- What if we could make the web server generate a query like:
  - ```
    "UPDATE users
    SET prefcolor='red', privilege='administrator'
    WHERE uid='joe';"
    ```
- Can we engineer the value of "color" given to the web server so it generates this query?
  - Note how code and data are mixed in the same channel
    - Better database interfaces provide separate channels
      - Java prepared statements
      - Stored procedures
    ```

Malicious HTTP Request

- `http://www.example.com/script?color=red',privilege='administrator`
- The "color" input is then substituted to generate SQL:
  - ```
    $q = "UPDATE users
    SET prefcolor='$INPUT[color]'
    WHERE uid='$auth_user'";
    ```
- It gives the query we wanted!
  - Joe now has administrator privileges.
    ```

Adding Another SQL Query

- Let's say Joe wants to run a completely different query:
  - "DELETE FROM users"
    - This will delete all entries in the table!
- How can the value of "color" be engineered?

Malicious HTTP Request

- `http://www.example.com/script?color=red'%3Bdelete+from+users%3B`
  - %3B is the url encoding for ";"
- What happens when the "color" input is used to generate SQL?
  - ```
    $q = "UPDATE users
    SET prefcolor='$INPUT[color]'
    WHERE uid='$auth_user'";
    ```
    ```

Result

```
UPDATE users
SET prefcolor='red';
delete from users;
WHERE uid='$auth_user'";
```

- The last line generates an error, but it's already too late; all entries have been deleted.
- The middle query could have been anything

FAQs

- Couldn't the database have a separate account for "Joe" with only the privileges he needs (e.g., no delete privilege)?
  - In theory yes, but in practice the management of such accounts and privileges, and connecting to the database with the correct IDs, adds significant complexity
    - Most often a database account is created for the entire web application, with appropriate limitations (e.g., without privileges to create and drop tables)
    - A good compromise is to create database accounts for each class of user or class of operation, so:
      - if Joe is a regular user he wouldn't have delete privileges for the user table
      - Changing user preferences, as an operation type, doesn't require delete privileges

FAQs

- Doesn't SSL protect against this sort of attack?
  - No
- But what if you authenticate users with a username/password over SSL? Then, if the user does SQL injection, the server admins will know who perpetrated the crime, right?
  - Not necessarily; only if you have sufficient audit logging.

Other SQL Injection Methods

- Let's say you've blocked single quotes, double quotes and semi-colons.
- What else can go wrong?
  - How about "\"?
  - If attacker can inject backslashes, then escaped quotes could get ignored by the database

PHP-Nuke SQL injection CAN-2002-1242

- iDefense advisory dated Oct. 31, 2002
- Malicious url:
  - modules.php?name=Your_Account&op=saveuser&uid=2 &bio=%5c&EditedMessage= no&pass=xxxxx&vpass=xxxxx &newsletter=,+pass=md5(1)/*
- %5c is the encoding for '\'

Let's Look at the SQL

```
UPDATE nuke_users
   SET name          = '', email         = '',
       femail        = '', url           = 'http://',
       pass          = 'xxxxx', bio          = '\',
       user_avatar   = '',
       user_icq      = '',
       user_msnm     = '',
       newsletter    = ',
       pass=md5(1)/*' WHERE uid='2'
```

- Notice how bio would be set according to the text in red?
  - '' (two single quotes) make the database insert a single quote in the field, effectively the same as \'
- Notice how the comment field, '/*', is used to comment out the "WHERE" clause for the uid? This means that the query applies to **all** users!

What Happened?

- All passwords were changed to the value returned by the function "md5(1)"
  - Constant: "c4ca4238a0b923820dcc509a6f75849b"
- Attacker can now login as anyone