



Code Injection Input Validation

Defending against code injection
Examples
Input Validation

1



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

2



Code Injection cont.

- Defend against it by using input cleansing and validation; type casts may help if they are possible
- Need to keep track of which data has been cleansed, or keep track of all sources of inputs and cleanse as the input is received

3



Basic Example by Command Separation

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

4



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 user user confidential`
- Any statement after the ";" would also get executed, because ";" is a command separator.
- The data argument for "ls" has become code!

5



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"`

6



Answer

- Script1 is executed
 - first an "oups" directory is created
- The rest of the intended command, "grep oups afile", is executed

7



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 we run the following?
% ./a.out `./script`

8



Answer

- The program calls
 - `system("grep `./script` text");`
 - can be verified 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
 - Anyone running a.out would run arbitrary code as the owner of a.out
 - What if a.out runs with root privileges?

9



Shell Metacharacters

- ``` to execute something (command substitution)
- `;` is a command ("pipeline") separator
- `&` start process in the background
- `|` is a pipe (connecting standard output to standard input)
- `&&` , `||` logical operators AND and OR
- `<<` or `>>` prepend, append
- `#` to comment out something

Refer to the appropriate man page (`man csh`) for all characters

- How else can code be injected into a.out?

10



Defending Against Code Injection

- Input cleansing and validation
 - Model the expected input
 - Discard what does not 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

11



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...
 - HTML may contain JavaScript (Cross-site scripting vulnerabilities)

12



Intuitive Approach

Block or escape all metacharacters

- but what are they?

Problems:

- Character encodings
 - octal, hexadecimal, UTF-8, UTF-16, binary, Base-64, URL encoding, ...
- Obfuscation
 - Escaped characters that can get interpreted later
 - Engineered strings such that by blocking a character, something else is generated

13



Wrong Way to Cleanse Input (Sanitize)

```
int main(int argc, char *argv[], char **envp) {
    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 = '_';
    ...
}
```

- http://www.cert.org/tech_tips/cgi_metacharacters.html

14

Real Life example: 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

15

Real Life example: phf CGI cont. 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...

16



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

17



Defense: Input Sanitization

- Do not attempt to list all forbidden characters
 - It is easy to forget and one missed character leads to defeat
- Make a list of all allowed characters
 - Without metacharacters
- Convert to a variable of numerical type, if a number is expected
- Truncate input strings if the expected length is known

18



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

19



Order for Cleansing and Input Validation

- 1) Resolve all character encoding issues first
- 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)

20