CITS3007 Secure Coding
Injection and validation

Unit coordinator: Arran Stewart

Highlights

- ▶ Terminology
  - ▶ Injection
  - ▶ Neutralization, escaping, filtering, validating, parsing, canonicalization
- ▶ Vectors for injection
  - ▶ command string
 - ▶ environment

Terminology and techniques

Injection

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

The CWE describes CWE-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."

Injection

- ▶ "Upstream" and "downstream" refer to the *flow of data* between components
- ▶ "Components" could be functions, methods, objects, programs, or entire systems – it depends on what you're looking at
  - ▶ Example: flow of data through a web-based forum system
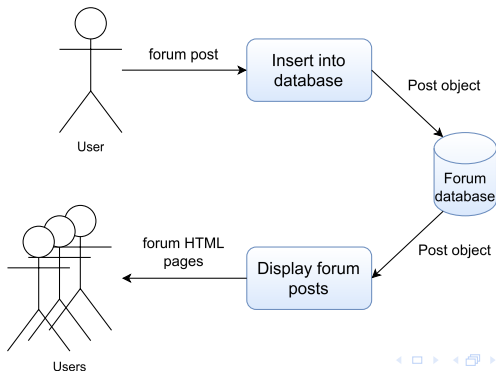
Input-validation vulnerabilities

Injection vulnerabilities arise because we've failed to validate and/or sanitize input properly.

In all of them, an attacker is able to use specially crafted inputs to subvert our security goals.

Some vulnerabilities of this sort are common enough to have their own names.

OS command injection (CWE-78)

> External input is used when invoking some external command, and allows an attacker to run commands we don't intend

SQL injection (CWE-89)

> External input is used when constructing an SQL database query, allowing an attacker to run queries we don't expect (altering the database or returning confidential data)

Input-validation vulnerabilities

### Cross-site scripting ("XSS", CWE-79)

> External input is used when constructing content to be displayed by a web site, and allows an attacker to run malicious code.

Path traversal (CWE-22)

> External input is used to construct a path which *should* be restricted to a particular directory – but due to improper validation, an attacker can gain access to other directories

Neutralization

- ▶ Injection problems arise because particular characters can have **special meanings** for downstream components.

- ▶ For instance, web servers send HTML to browsers.
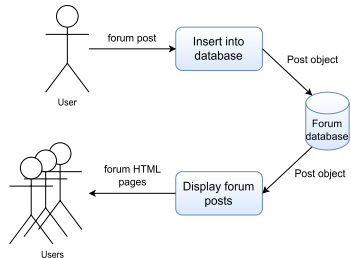
- ▶ If posters on a forum are allowed to create any HTML they want, they can emit the sequence "`<script> ... </script>`"

- ▶ The poster can include arbitrary JavaScript between the "`script`" tags, and when the post is read by some user, that JavaScript will be executed by the user's browser

- ▶ (Browsers contain safety features which attempt to limit the harm that can be done by a web page – but allowing a poster to execute arbitrary JavaScript is still a very bad idea.)

Solution

▶ Limit what "special" HTML sequences users are allowed to include in their posts

▶ *Neutralize* (render harmless) or remove (filter) everything else

▶ This needs to be part of a coherent approach to managing **untrusted inputs** and ensuring they can cause no harm

Solution

We consider this further when we look at secure software development processes.

As part of a secure design, we need to

- ▶ identify sources of input
- ▶ identify flows of data
- ▶ identify languages and formats used, and their special elements
- ▶ neutralize special elements before they are either relied upon by some component, or included in output

Causes of most input vulnerabilities[1]

Unintended parsing

> Data treated as special when it shouldn't be

Overlooked input channels

> Failing to notice ways untrusted data can be inserted

Overlooked data flows

> Failing to notice some circuitous route untrusted data can take

Unexpected expressivity

> A language or format being used is more expressive than intended

---

[1]Poll, *Secure Input Handling* (version 1.0, Nov 2023)

Terminology – neutralization

No universally accepted terminology, but usually:

Escaping Replacing some sequence of characters with an "escaped" or "quoted" equivalent, so that it loses its special meaning.

Example: HTML

`<b>` ... `</b>` means "make the enclosed text bold". So what if we want to actually show the reader the *literal* characters "`<b>`"?
We escape the `'<'` and `'>'` so they lose their special meaning – we write

 ...

We're *often* talking about text, so I'll use the word "characters"; but we could be talking about something more general (bytes, sequences of numbers, data structures) – hence the CWE uses "elements".

Escaping

Example: C

In C, we can represent a character literal by putting it in single quotes.

To represent the single quote character itself, we use the backslash ("\") to escape it.

```c
char c = '\'';
```

Neutralization – filtering

Filtering Stripping out some sequence of characters entirely. In some contexts might be called "whitelisting" or "blacklisting", depending how implemented.

Example: HTML

We could *filter* HTML "special characters" from some source by stripping them out entirely.

('<' and '>' are two examples, but there are more.)

Neutralization – validating

Validating Comparing a sequence of characters (or other input) against a pattern or rule which determines what input is allowable.

Example: Year

Given a 4-character sequence, we can write a function to determine whether it represents a valid *year* in the second or third millenium.

`Pseudocode:`

— `The first character must be the digit '1' or the digit '2'`
— `The remaining characters must be digits`

Often, regexes are used to check whether some sequence of characters is valid.

Parse, don't validate[2]

- ▶ Booleans give you a "yes/no" to the question "Is input X valid?"
- ▶ Better is to **parse** the input into some struct or object so that it can't be confused with other strings, ints, etc

---

[2]For further reading: see Alexis King, "Parse, don't validate" (2019)

Example

Suppose we have some string that should represent a URL.

Validation Write a boolean validation function in your preferred language (e.g. `bool isValidURL(const char *)` in C) that checks whether the string is really a URL.

Parsing *Parse* the input string, and if it represents a URL, return a new type with the invariant "Represents a valid URL".

e.g. parsing in Java:
`URL parseURL(String s) throws InvalidURLException`

In C, we'd likely need to create our own `struct URL` type. (Query: what would the signature for our function be? How would we indicate failure?)

Advantages

Validating leaves the input as a string; but a string is (almost) bare, unstructured data – tells us nothing about what the string *represents*.

Parsing

▶ Helps ensure we only parse *once* – we can't confuse a parsed URL for a string

▶ Ensures we don't pass un-parsed data to functions – our URL-handling functions will only take a URL, not a string

Disadvantages

Main disadvantage of parsing is extra code.

Some languages make it easier than others to create new types.

If your code passes around data as strings – especially when better types are available (often the case in Python, Java) – it is often said to be "stringly typed", a code smell.

Neutralization – sanitizing

Sanitizing Sometimes used to mean "filtering".

Sometimes used to mean some combination of escaping, filtering, and validation that ensures some input does not trigger undesired behaviour. Equivalent to the general term **neutralization**.

Canonicalization

▶ In addition to neutralization – another relevant technique.
Also called **normalization**.

▶ Example: paths.

`/etc/passwd` and `/etc/../etc/passwd` represent the same path.

Canonicalization – why?

If you have data (like a path) where there can be multiple representation of the same "thing", it becomes impossible to know whether two things are actually the same thing.

Solution: canonicalize them.

On Unix-like systems, canonicalizing *paths* typically means making them absolute, instead of relative, and removing any "/.." and "/." sequences.

Before taking any security decision based on some input, it should be put into canonical form (if relevant).

Canonicalization – why?

Canonicalization can help deal with *unintended expressivity*.

File paths *look* like they're "just" strings – but they are their own "language", with its own metacharacters with special meanings

Unexpected expressivity

Some language or format being used is more expressive or complex than a designer or developer realized, and can be used to express things they didn't intend.

Another example – homograph attacks (similar to "typosquatting", but not the same)

Identifying sources of input

Suppose we're writing a command-line program in C.

Question: What are possible sources of *input* to the program?

Managing input

Check your assumptions about what sources are "trusted".

Example: "Information taken from the database is assumed to be trustworthy".

Managing input

Check your assumptions about what sources are "trusted".

Example: "Information taken from the database is assumed to be trustworthy".

That's only the case if you checked it for trustworthiness on the way *into* the database.

Overlooked data flows

A designer or developer fails to notice a route by which malicious input can end up affecting or being processed by an application. Example: "second-order injection" attacks. Untrusted data is inserted into a database, but no harm is caused until its retrieved from the database, incorrectly treated as trustworthy, and used by a downstream component.

Dealing with input

Question: "Do we need it?"

▶ If not: filter or remove what you don't need

▶ e.g. If your program doesn't make use of environment variables, you may as well strip out all the ones you don't need

Dealing with input

Assuming you do need the input:

- ▶ Be wary of strings
    - ▶ especially when passed to a downstream component
    - ▶ especially if that downstream component implements or makes use of some *language* (i.e. there are characters/elements with special meaning)

Examples of downstream components which use or implement a language:

- ▶ the `system()` function (interprets shell scripts and commands)
- ▶ SQL database systems (interprets SQL queries)

Example: the `system` library function

$$\texttt{int system(const char *command)}$$

▶ Invokes the host environment's *command processor* with the parameter `command`
    ▶ On Unix-like systems, `bin/sh` is typically invoked
    ▶ On Windows, `cmd.exe` is typically invoked
▶ The command processor interprets `command` according to the rules for its language
▶ That language will have particular special characters – e.g. ";" to separate commands.

system – injection example

From 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]).

system – 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].

What can go wrong here?

Unintended parsing

This is an example of *unintended parsing* – the shell interprets what input is given to it, and we let that input contain sequences with special meanings

Unintended parsing

Something is treated as special (e.g. as HTML, as SQL, as part of a Bash command sequence) when it shouldn't have been

Unintended parsing

The German art collective !Mediengruppe Bitnik published a book entitled "`<script>alert("!Mediengruppe Bitnik");</script>`"

Many online bookshops discovered they weren't properly sanitizing book *titles* before publishing details on the web.

An example of both *unintended parsing* (titles should contain no, or only limited, special characters) and *overlooking input channels* (book details not noticed as a source of input).

Overlooked input channels

A designer or developer overlooks a way in which malicious input can end up affecting or being processed by an application (Example: environment variables, local files)

Dealing with input

- Keep track of what *source* input came from, and to what degree it is trustworthy
- Try to use more appropriate types than strings
- Parse from string to type
- Or if not parsing, at least validate

Dealing with input

- ▶ Keep track of what languages are used by downstream components.

- ▶ If possible, use appropriate types to represent them, so you don't get confused.

- ▶ *Before* passing data to a component, ensure any portions that came from untrusted input are escaped or quoted.

- ▶ Details of how to correctly escape or quote are often complex
  - ▶ If possible - rather than writing your own escaping/quoting routines, use a well-tested library.

Downstream component

A "downstream component" could be

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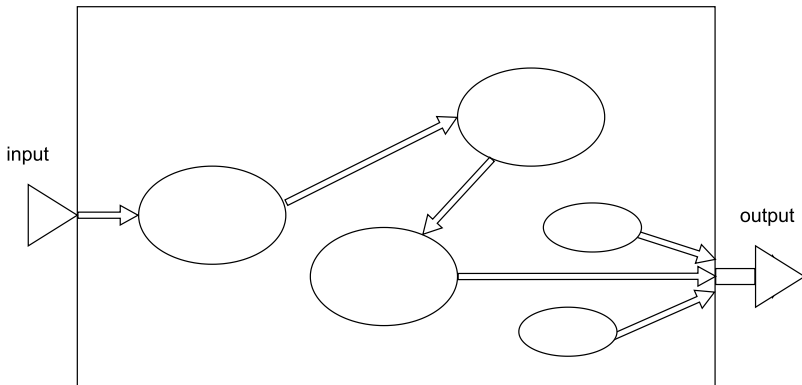 - ▶ play an animation
  - ▶ execute an OS command
- ▶ a message sent to another service.
  For example, to
  - ▶ send a web request to some server
  - ▶ query a database

Injection

We can imagine the situation looking something like this:

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"

Invoking downstream components

When invoking/passing data to downstream components – select the safest alternative for doing so.

C `system`, `exec` family vs safe wrapper libraries

Python `os.system` vs `subprocess` module

SQL Hand-constructed queries vs prepared statements

Command injection

Why programmers use `system`

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?

Preferred use libraries of safe versions (e.g. the "O'Reilly Secure Programming Cookbook" functions, discussed later)

Less preferred 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

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:

```python
# ... 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

Metadata

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)

Meta-characters

Meta-**characters** are single characters with a special meaning.

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 often indicate some special encoding/meaning to be used when intepreting other characters.

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/)

Sources of data – environment variables

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 envp[])
```

▶ `execve()` doesn't copy variables 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.

Further reading

▶ Alexis King, "Parse, don't validate" (2019)
▶ Erik Poll, *Secure Input Handling* (version 1.0, Nov 2023)