CITS3007 Secure Coding
Inter-Process Communication

Unit coordinator: Arran Stewart

Highlights

- memory isolation
- sandboxing
- IPC

Memory isolation

We know that one principle of secure design is to limit shared resources.

Memory isolation helps with this

- Goal: Prevent one program from interfering with other running programs, the operating system, and maybe itself.

Memory isolation

- Each running process sees *virtual addresses*
- The process is isolated from, and cannot access, other processes' virtual memory.
- Whenever code in that process accesses memory, this gets *translated* into the correct accesses to physical memory
  - In modern architectures, address translation is usually done by hardware, by the Memory Managment Unit (MMU) or Memory Protection Unit (MPU), using information provided by the OS.

Sandboxing

Sandboxing

- If we need *more* communication between processes, we can use the filesystem, or one of the IPC techniques we look at later
  - e.g. pipes, FIFOs, sockets, shared memory
- What if instead we want *less* communication?

Sandboxing

Sandboxing: a security technique for isolating untrusted code from the host platform

Usual reason is to limit the potential damage code can do, and what access it has to information about the host OS and other processes.

Types of damage

A process could deliberately or accidentally . . .

- corrupt application data
- corrupt memory
- corrupt local files/filesystem
- corrupt other processes
- spread via the network to other computers

Types of separation

Some types of separation are:

- Whole-system virtualization
  - Same architecture as host
  - e.g. VirtualBox, VMWare
- Whole-system emulation
  - Different architecture to host, emulated entirely in software
  - e.g. QEMU
- Containerization / partial system resources emulation
  - e.g., Docker, Podman
- In-process application sandboxes
  - e.g. using techniques like capabilities and seccomp

Containerization

A containerized process is given the illusion that it is the only application running on the platform (other than, perhaps, its own dependencies).

Docker and Podman are examples of containerization technologies.

How containerization is done

- By default, processes on Unix systems can see a fair amount of information about each other, via the `/proc` filesystem (or tools like `ps`)
  - And this information is normally true and accurate
- Who or what decides what information a process gets? The OS.
- But they don't have to be.
- e.g. An OS designer might decide that some user IDs can be categorized as "guest" users, perhaps, with less privilege than a normal user.

How containerization is done

On Linux, done using a number of kernel features:

- namespaces - limits what a process can *see*
- cgroups - limits what resources a process can *use*
- seccomp - limits what system calls a process can make

Docker combines these, but it's also possible to use particular techniques individually if desired.

cgroups

- **Control groups** ("cgroups") – kernel feature for limit, doing accounting of, and isolating resource usage of a collection of processes
  - e.g. "Allow this process(es) to use at most 1024MB of memory", or "... 20% of 1 CPU".
- Allows management of
  - memory
  - CPU
  - block I/O
  - network
  - device drivers

namespaces

cgroups limits what a process can use; ns limits what a process can see.

- pid: what other processes can be seen?
- net: networking interfaces can be seen?
- mnt: what files/filesystems can be seen?
- uts: what hostname is seen?
- ipc: allows namespace-specific IPC
  - e.g. semaphores, message queues, shared memory
- user: what user IDs does the process see? what user ID does it think it runs with?
  - allows mapping of user IDs
  - within namespace, process could "think" it's root
- time: allow slower/faster/different clock times

namespaces

Can be demonstrated from shell.

```
$ sudo unshare --uts
```

- Create new uts (hostname) namespace
- Set a new hostname:

  ```
  $ hostname cits3007
  ```

- In another shell can run `hostname` and see that it's unaltered for other processes.

seccomp

- Prevents execution of certain system calls by a process through a customizable filter.

Steps to use BPF filter:

1. Construct filter using BPF rules
2. Install filter using `seccomp()` or `prctl()`
3. `exec()` new program or use a function in dynamically loaded shared libraries

IPC

Inter-process communication (IPC)

IPC any mechanism provided by an operating system for processes to share data.

Technically, using the filesystem counts as a type of IPC – but we normally mean more direct methods.

IPC technologies

There are many . . .

- pipes
- named pipes (aka FIFOs)
- Full-duplex pipes (STREAMS pipes)
- System V message queues
- System V semaphores
- shared memory
- sockets
  - Unix domain sockets
  - network sockets
- streams
- signals
- . . .

Not really a single, coherent design, rather many distinct technologies.

Someone would invent a new way for processes to communicate on Unix-like systems, it would get adopted by others.

IPC technologies

Stream-based IPC

- Things with file-like operations (read and write) on them
- Examples: pipes, FIFOs, sockets, streams

Segment-based IPC

- Techniques based on sharing memory segments
- Examples: shared memory, memory-mapped files

Message-based IPC

- Things that allow individual **messages** or signals of some sort to be sent
- Examples: signals, message queues

Choosing an IPC technology

Normally, we want to keep IPC as private as possible.

e.g. If communicating with child processes – can rely on *inheritance* of open files when forking, and use pipes.

Absent kernel exploits, should be few ways to intercept the communication.

If paranoid, could always encrypt information as well.

Signals in Unix

- On Unix systems, **signals** are a fundamental IPC mechanism
- Signals are **software interrupts** delivered to a process
- Used to notify a process that a specific event has occurred.

Commonly used Signals

Some commonly used signals on Linux:

- **SIGTERM (15)**: Terminate a process gracefully. Allows cleanup before exit.
- **SIGKILL (9)**: Forcefully terminate a process. No cleanup is performed.
- **SIGHUP (1)**: Hang up. Typically used to restart a process or reload its configuration.
- **SIGINT (2)**: Interrupt from the keyboard (e.g., Ctrl+C).
- **SIGQUIT (3)**: Quit signal. Similar to SIGINT but also generates a core dump.
- **SIGSTOP (19)**: Stop the process. Can be resumed with SIGCONT.

Signalling processes

Processes can signal other processes, or themselves, in various ways:

kill command The `kill` command sends signals to processes. E.g., `kill -TERM <pid>` sends a termination signal.

Keyboard shortcuts In the terminal, Ctrl+C sends a SIGINT signal to the foreground process.

Other shortcuts (e.g. Ctrl+Z) can potentially send others.

Programmatic Use the `kill()` function (more violent than it sounds)

Signal handling in programs

- Processes can define custom behavior on receiving signals.
- They do so by setting **signal handlers** – functions executed in response to signals.
  - Set using `signal()` or `sigaction()`
- Can be used for e.g., reloading configurations
  - Webservers often do this: instead of having to re-start server, you send it a signal

Practical use cases

- Graceful process termination and cleanup.
- Reloading configurations without restarting.
- Managing daemon processes.
- Debugging and profiling processes using signals like SIGUSR1/SIGUSR2.

Default behavior

When a process receives a signal, its behavior is determined by the signal type. Unless customised, the default behavior is:

SIGTERM (15): Termination signal Terminate the process gracefully. The process can perform cleanup operations before exiting.

SIGKILL (9): Forced termination signal Terminate the process immediately without any cleanup. Data loss may occur.

SIGHUP (1): Hang-up signal Terminate the process. Historically used to restart or reconfigure daemons.

Default behavior

SIGINT (2): Interrupt signal Terminate the process. Often triggered by pressing Ctrl+C in the terminal.

SIGQUIT (3): Quit signal Terminate the process and generate a core dump for debugging. Process can send *itself* SIGQUIT

SIGSTOP (19): Stop signal Stop the process. Can be resumed with SIGCONT.

Security implications of signals

Race conditions:

- Using handlers can be tricky
- Timing issues can lead to race conditions when handling signals.
- What happens if a signal is received when already handling a signal?
  - Need to consider this
  - Behaviour depends on OS settings, particular signal – some are marked as "reentrant"
- Attackers could try to exploit for unauthorized access or privilege escalation.
- (And, of course, the usual memory-based vulnerabilities are relevant)

Security implications of signals

Information Leakage:

- Signals could expose process information to attackers, potentially leaking sensitive data.

Best practices

Signal whitelisting

- Allow only trusted signals and sources.
- Limit which signals a process can receive.

Pipes (aka “unnamed pipes”)

Aka “anonymous pipe”

You’ve already used these . . .

```
$ ls | grep somefile
```

- Unidirectional data channel using standard file-writing and reading mechanisms

Pipes

- Typically used for communication between a parent process and its child process.
- Each process normally gets one "end" of the pipe (read or write)
- Data written to the write-end of the pipe is buffered by OS
- Until read from the read-end of the pipe.
- For two-way communication – can use two pipes in opposite "directions".
- Pipes have limited capacity; writing to a full pipe may block the writer.
- Reading from an empty pipe may block the reader until data is available.

Pipe documentation

- See `man 2 pipe`
- `man 7 pipe` - overview of pipes and FIFOs

Creating a pipe

- To create a pipe, use the `pipe()` system call:

```
int pipe(int filedes[2]);
```

- Creates a pipe with two file descriptors:
  - `filedes[0]` (read end)
  - `filedes[1]` (write end)
- Data written to the write end (`filedes[1]`) can be read from the read end (`filedes[0]`).

Writing to the pipe

- Use `write()` to write data to the pipe:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- Example:

```
write(filedes[1], "Hello, Pipe!", 12);
```

Reading from the pipe

- Use `read()` to read data from the pipe:

```
ssize_t read(int fd, void *buf, size_t count);
```

- Example:

```
char buffer[20];
read(filedes[0], buffer, 20);
```

Closing

Close unused pipe ends to prevent resource leaks.

```
close(filedes[0]);
  // Close the read end in the writing process
close(filedes[1]);
  // Close the write end in the reading process
```

FIFOs

Aka “named pipes”

- Very similar to pipes, but:
  - FIFOs exist as files in the file system.
- Can be used for both related and unrelated processes.

Creating a FIFO

- Create a FIFO using the `mkfifo()` system call:

```
int mkfifo(const char *pathname, mode_t mode);
```

- Specify the path to the FIFO and its permissions in the `mode` argument.
- Creating a FIFO:

```
mkfifo("/tmp/myfifo", 0666);
```

FIFO example

- Process 1 writing to the FIFO:

```
int fd = open("/tmp/myfifo", O_WRONLY);
// ... write some data to the fd
close(fd);
```

- Process 2 reading from the FIFO:

```
int fd = open("/tmp/myfifo", O_RDONLY);
// Read data from the FIFO
close(fd);
```

Security considerations

- Like any file – FIFOs must be secured with appropriate permissions.
- Unauthorized access can lead to data exposure or tampering.