Processes + Signals

Nimish Mishra
@manwe sulimo

Agenda

Creating processes

C programs...

Exit codes

And why you do exit(0) to remain sane...

Signals

The messengers of OSes

Appendix

Some extra things:

-Function pointers
-Virtual memory introduction

Questions

-Technical?
-Non-technical?

Creating processes

- Recap: process image
- Family of syscalls to create processes
- Time sharing demonstration

Family of syscalls to create processes

- **system**(): A part of stdlib, system() can execute any bash program from within C/C++ through creating a new process image
- **fork()**: Duplicates a process image and allows separate code sections to run in parent and child processes
  - **Verify**: Through use of **getpid()**, which code section is running where
  - **Verify**: from the process image explored in previous session, explore memory regions and verify if the process image is an actual duplication
  - **Verify:** What if similar memory regions have different value in parent and in child
- Family of **exec*()** syscalls
  - Unlike **fork()** syscalls, they replace the process image with a new image (means new mapping, new environment etc)
  - **Verify:** Pass in a custom environment and verify it in **cat /proc/<CHILD PID>/environ**

Time sharing demonstration

- Normal sane systems have **schedulers**, and (not so surprisingly!), they don't allow one process to capture the system forever
- In the era of multicore computing, true parallelism is possible. However in the times of unicore computing, time sharing allowed for a *perception* of multitasking
- Use **sched_setaffinity()** to bind processes to a specific core and verify irregular scheduling. For this to make sense though, the process must run way beyond the **time quantum** Linux's CFS scheduler (or your OS) gives for each process to run
    - **Verify:** That this actually happens

- What are exit codes
- Capturing exit codes in C
- Why do zombie processes exist

Exit codes

What are exit codes

- Exit codes are 8 bit unsigned integers that are used to communicate status of exited process to the parent process
- To capture exit code of last run command in **bash**, do a **echo $?**
- Generally, a value of **0** means **SUCCESS** while other positive values indicate several sorts of problems that may occur to a particular process. The parent process or the OS can use these exit codes to adapt accordingly
- Examples:
  - **126:** Execution permission denied (**/etc/shadow**)
  - **127:** Command to be executed not found in PATH (**literally anything that is not in PATH)**

Capturing exit codes in C

- **waitpid()** waits upon a process to return an exit code which it can then capture

- Through the return status, the parent process employs macros **WIFEXITED()** and **WEXITSTATUS()** to glean out the exit code

Why do zombie processes exist?

- Every process has an exit code. And it is the responsibility of the parent to collect and process it.
- But what if the parent forgets to **waitpid()** on a child process and exits without it?
- Such a child process is called a **zombie process** and the following happens:
  - The child process is *adopted* in as child by a *grandparent*
  - The grandparent's job is to collect **zombie processes** under its hood and to wait upon them at regular intervals
  - **Verify:** A zombie process in the process table (use **ps -al** and **pstree** to verify grandparent)
- What if no *grandparent* waits upon the orphaned child?
  - That doesn't happen!
  - **But what if it did happen?** From our previous session, we know that a process has a process image and a PCB which has entry in the so called **process table** (**ps ax** outputs a section of process table itself). If no one waits upon a child, that process remains forever alive unless you shut down the system eating up resources.

Signals

- What be signals?

What be signals?

- *Signals* are events generated as software interrupts which the software or operating system may choose to handle as they see fit
- Signals are usually raised by error conditions (like segmentation faults) but they can also be generated from userspace
- To handle signals: **void (*signal (int signal, void (*handler)(int))) (int);**
  - Treated simply, **signal** is a function pointer to a function that takes an input another **function pointer func** and a **signal integer,** and returns void
  - **handler()** is basically another function pointer to the function to be called to handle signal
- To generate signals: **kill(PID, signal)**
  - Send signal **signal** to the process **PID**
  - **Verify:** Are there signals which *can't* be sent using the current permissions.

- Function pointers
- VA introduction

Appendix

Function pointers

- Very much like normal pointers, except that they *point* to functions

- Syntax: **return_type (*name)(arg_list);**
  - int (*func)(int): A function pointer to a function that accepts an *int* and returns an *int*
  - void (*func)(void): A function pointer to a function that accepts nothing and returns nothing
  - And so on...

- Examples

VA introduction

- Revisit **va_address.c** and note a peculiar thing

```
manwe@manwe-Lenovo-IdeaPad-S540-15IML-D:~/Desktop/exploit-dev/notes/sessions/processes_details$ make va_address
[PID 4732] Integer value: 10, integer address: 0x7ffc03a81b54
[PID 4733] Integer value: 20, integer address: 0x7ffc03a81b54
```

- How can same memory address have two different values?
  - Because it is not **physical memory** address (i.e. it is not the address where the values are stored in actual memory)
  - It is rather **logical memory** address, implies each process has a **base address** and this **physical address = base address + virtual address**
  - This value of **base address** is different for both parent and child
- How is this magic happening?
  - Next session probably...

Link to everything:
https://github.com/NimishMishra/exploit-dev/tree/master/notes/sessions/process_details

Questions?