CAULDRON PANDA/UNC3886 -Use of Reptile Linux rootkit

https://www.trendmicro.com/en_us/research/25/g/revisiting-unc3886-tactics-to-defend-against-present-risk.html

Change permissions for ptrace to work:
cat /proc/sys/kernel/yama/ptrace_scope
sudo echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope

To compile each exercise:
gcc exercise1.c -o exercise1
./exercise1

repeat for the rest

Exercise 1: Using `getpid()` and `fork()`

What's happening:

- `getpid()` → asks Linux "What number (PID) did you give me?" Every process gets its own number so the OS can track it.
- `fork()` → makes a copy of the current **(parent)** process. After `fork()`, there are **two programs running the same code**: the *parent* and the *child*.
- The parent gets the child's PID, the child gets `0` returned. That's how you tell them apart.

lua Copy Edit

```lua
Parent PID=1001
    |
    +-- Child PID=1002
```

- The child is **always under its parent**, unless the parent dies.
- PPID = the PID of the current parent at that exact moment.
- PID = unique process identifier for each process.

Beginner takeaway:

- Linux thinks of programs as *processes*, each with its own number (PID).
- `fork()` is like cloning yourself — now there are two of you running!

Question:

- *"So after `fork()`, who runs first, parent or child?"* → It depends! The scheduler decides. Sometimes parent runs first, sometimes the child.

Exercise 2: Running Another Program with `execl`

What's happening:

- `execl()` replaces the current process with a new program.
- Example: the child calls `execl("/bin/ls", "ls", NULL)` → suddenly that child is no longer the original code, it's now running the `ls` program!
- The PID stays the same, but the code and memory are swapped out.

Beginner takeaway:

- `fork()` duplicates, `execl()` transforms.
- Shells work this way: type `ls` → the shell forks → child does `execl("ls")`.

Question:

- *"Does the parent also change into `ls`?"* → No. Only the child calls `execl()`, the parent continues as before.

Exercise 3: Tracing with `ptrace`

What's happening:

- `ptrace()` is a special system call that lets one process (the parent) *watch* or *control* another (the child).
- Debuggers (like `gdb`) use this to pause, inspect, or step through programs.
- In the exercise, the parent tells the kernel "let me trace this child," then it can see when the child runs system calls or stops. CTRL-C to stop program.

Beginner takeaway:

- `ptrace()` = process surveillance. One process can spy on another with the OS's help.

Question:

- *"Can any program spy on another?"* → No, normally you must be the parent, or have special permissions.

Exercise 4: Structs and System Calls

What's happening:

- A `struct` is just a way to organize related pieces of data (like a box with labeled compartments).

- This exercise shows how system calls often take structs as input/output. For example, you might call `stat()` and it fills in a struct with file information.
- The program demonstrates passing structured data between user code and the kernel.

1. The **child process** creates a buffer containing `"Hello, World!"` and prints its address so the parent knows where to look.
2. The child then **stops itself** with `raise(SIGSTOP)` so the parent can safely attach and examine it.
3. The **parent process** attaches to the child using `PTRACE_ATTACH` and waits until the child stops.
4. The parent reads memory from the child's buffer using `PTRACE_PEEKDATA`, demonstrating how a process can inspect another process's memory.
5. After reading, the parent **detaches** using `PTRACE_DETACH` (with the "continue" flag), allowing the child to resume execution and exit normally.
6. The exercise shows the **relationship between parent and child processes**, how `ptrace` can be used for memory inspection, and the importance of stopping, attaching, reading, and detaching safely. Enter CTRL-C to stop the program.

Beginner takeaway:

- Think of structs like forms you hand to the OS. You give Linux a struct, it fills in the blanks with answers (file size, process info, etc).
- `ptrace` allows **one process to observe and control another**.
- The child must **stop** before the parent can safely trace it.
- Reading memory directly is done with **`PTRACE_PEEKDATA`**, which returns raw bytes.
- Proper detaching ensures the child continues and the program doesn't hang.

Question:

- *"Why not just return everything directly?"* → Because system calls often return lots of info, so putting it in a struct is cleaner.

Exercise 5: Multiple System Calls Together

What's happening:

In this exercise, the parent process uses `ptrace` to modify a variable inside the child process. The child first stops itself using `SIGSTOP` so the parent can attach safely. The parent then writes a new string, `"Modified"`, directly into the child's memory using `PTRACE_POKEDATA`. After writing, the parent reads back the child's memory with `PTRACE_PEEKDATA` to verify the change. Finally, the parent detaches, which automatically resumes the child, and both the parent and child print the updated buffer. This demonstrates how `ptrace` allows a process to inspect and alter another process's memory in real time.

- This program chains several system calls (like `open`, `read`, `write`, `close`, etc).
- It shows how Linux programs talk to the kernel step by step — open a file, read it, print it out, then close it.
- Each system call is like knocking on the OS's door asking for help.

Beginner takeaway:

- Programs don't access hardware directly. Instead, they use system calls as requests to the OS.

Question:

- *"Why not just read the disk directly?"* → The OS protects hardware; you must ask politely through system calls.

Exercise 6: More System Calls (Combined Example)

What's happening:

In this exercise, the parent process demonstrates manual memory injection into a child process using the `ptrace` system call. The child process allocates a buffer and prints its PID and memory address, then stops itself (`SIGSTOP`) so the parent can attach. The parent attaches to the child, writes a new string into the child's buffer in small memory chunks using `PTRACE_POKEDATA`, and then detaches, allowing the child to continue. Finally, the child prints its buffer to show that the data has been successfully injected.

- This exercise expands on Exercise 5 with more system calls working together (maybe process, file, or memory management).
- It shows how small building blocks (`fork`, `exec`, `wait`, `open`, etc) combine to create real software.
- It's less about one new call and more about how they interact.

Beginner takeaway:

- Real programs are just lots of small system calls glued together.
- The OS is the "middleman" between your program and the hardware.

- How `ptrace` allows one process to **observe and modify another process's memory**.
- Using `raise(SIGSTOP)` to pause a process for inspection.
- Writing data to a child's memory in fixed-size chunks.
- Synchronizing parent and child processes during memory manipulation.

Question:

- *"So is everything a system call?"* → Pretty much any time you need the OS (files, processes, networking). Regular calculations happen in user space, but OS services require system calls.