Lecture 4: Intruding Address Space (How to Make Mods for Games)

Xin Liu

Florida State University
xliu15j@fsu.edu

CIS 5370 Computer Security
https://xinliulab.github.io/cis5370.html
February 6, 2025

Recap

Background:

- Linux builds the entire application program world from an initial process (state machine).
- Through `fork`, `execve`, and `exit`, we can create many child processes and execute them concurrently.

Question 3 in HW2

Objective: Understand how buffering works in the address space.

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    for (int i = 0; i < 2; i++) {
        fork();
        printf("Hello\n");
    }
}
```

Execution: Run the above program using the following commands:

1. `gcc hello.c`
2. `./a.out`
3. `./a.out | cat`

Task:

1. Explain why the outputs of the two commands differ.
2. Your explanation must include:
    - The exact outputs of both commands.
    - A detailed analysis of the buffering mechanism and how it affects output.
    - Screenshots of debugging using tools such as `objdump` or `gdb`.

Understanding `fork()`

Behavior Analysis:

- Running `./a.out` directly produces a different number of lines compared to `./a.out | wc -l`.
- Following the principle that *"the machine is always right"*, we analyze the cause:
  - Hypothesis: libc buffering effect.
  - Verification: Compare system call sequences using `strace`.

Buffering Control:

- Use `setbuf(3)` or `stdbuf(1)` to manage standard input/output buffering.

```
man setbuf
```

Understanding `Buffering`

When does the OS use line buffering?
- The operating system uses **line buffering** when writing to a terminal, meaning output is sent immediately when a newline character (`\n`) is encountered.
- If output is redirected (e.g., through a pipe), the standard output switches to **full buffering**, meaning data is only written when the buffer is full or when the program terminates.

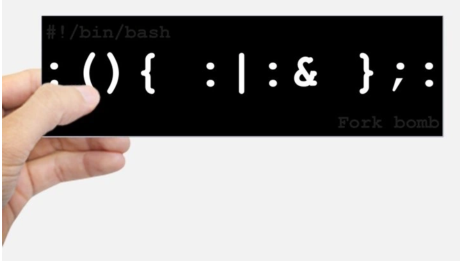
`fork()` creates an exact copy of the calling process, replicating every bit of its state, including the contents of buffers:

- The child process receives `0` as the return value.
- The parent process receives the child process ID.
- Other than the return value, the parent and child processes are identical and execute **in parallel** in the operating system.

Recap

Creating new state machines requires resources

- Continuously creating processes will eventually crash the system.
- **Don’t try it on linprog** (or try it in a container like Docker).
- Otherwise, I’ll have to go to the server room and reboot the system.



Code Analysis: Fork Bomb

```
:(){ :|:& };: # One-liner version

:() { # Formatted version
    : | : &
}; :

f() { # Bash: allows symbols as identifiers
    f | f &
}
f
```

Analogy to Nuclear Fission:

- A heavy atomic nucleus (U-235/Pu-239) is hit by a **neutron**, splitting into two lighter nuclei, releasing energy and more **neutrons**.
- This results in **self-replication**.

Recap

This Lecture:

- Based on our state machine model, a process's state consists of memory and registers.
- Registers are well-defined and can be examined using `gdb info registers`.
- What is inside the "flat" address space of a process (0 to $2^{64} - 1$)?
- Can we "invade" another process's address space?

State of State Machine

Register + Memory

Deterministic Execution:

- Given code and data, the initial state of a process is fully determined.
- Jumping to the entry point leads to a deterministic next state.
- Therefore, every state of the program should be deterministic.

Breaking Determinism:

- The only instruction that can break this determinism is a **system call**.

Invoking System Calls: `syscall`

- Delegates control completely to the operating system, allowing arbitrary modifications.
- An interesting question: What if a program never trusts the operating system?

Interacting with OS Objects:

- Read/write files (e.g., modify file contents via `mmap`).
- Modify process state (e.g., create processes, terminate itself).

Program = Computation + Syscall

Question: How do we construct the smallest possible "Hello, World"?

Question 1 in HW2

Constructing the Smallest Hello, World

```c
#include <stdio.h>

int main() {
  printf("Hello,␣World\n");
}
```

Why is the GCC output not the "smallest"?

- `gcc --verbose hello.c` shows all compilation options (there are many).
  - `printf` is transformed into `puts@plt`.
- `gcc --static hello.c` copies the entire `libc`.
  - Use `ls -l a.out` to check its size.
  - Use `objdump -d a.out` to check its code.

Core Idea

Hello, World is also a state machine. We only need to construct the state machine of several steps and finally invoke a syscall.

This is also the core idea behind the attack: **carefully crafting a sequence of states to eventually hijack execution and trigger the desired system call**.

Going Directly with Manual Compilation

Forcing Compilation + Linking: `gcc -c + ld`

- Directly using `ld` for linking fails:
  - `ld` does not know how to link library functions...
- An empty `main` function, however, works:
  - The linker produces strange warnings (can be avoided by defining `_start`).
  - But it results in a **Segmentation Fault**...

WHY?

- Naturally, we observe the execution of the **program (state machine)**.
- Beginners must overcome their fear: **STFW/RTFM** (Manual is extremely useful).
- `starti` helps us execute the program from the first instruction.
- `gdb` allows switching between two state machine perspectives (`layout`).
- `x/16x $rsp` allows us to check whether the return address or saved registers have been corrupted.

Handling Abnormal Program Exit

Can we make the state machine "stop"?

- Pure computation states: Not possible.
- Either an infinite loop or undefined behavior.

Solution: syscall

```
#include <sys/syscall.h>

int main() {
  syscall(SYS_exit, 5370);
}
```

Investigating Code: Where is `syscall` implemented?

- **Bad news**: It's inside `libc`, making direct linking inconvenient.
- **Good news**: The code is short, and it seems understandable.

Assembly Implementation of Hello, World

minimal.S

```
movq $SYS_exit, %rax # exit(
movq $1, %rdi # status = 1
syscall # );
```

Note: GCC supports preprocessing for assembly code (even defining __ASSEMBLER__ macros).

Where do I find these mysterious tech codes?

- `syscall (2)`, `syscalls (2)`
- **The Friendly Manual** is the richest source of information.

Recap: The state machine perspective on programs

- Program = Computation $\rightarrow$ `syscall` $\rightarrow$ Computation $\rightarrow$ ...

To observe system calls:

- Open two terminals and run the following commands separately:

```
$ gcc minimal_hello.s -c
$ ld minimal_hello.o
$ strace -f -o ./strace.log /bin/sh
$ ./a.out
```

```
$ tail -f ./strace.log
```

Why does Hello World have colors?

Easter Egg: ANSI Escape Code

Special encoded characters for terminal control:

- `telnet towel.blinkenlights.nl` (ASCII movie; `Ctrl-]` and `q` to exit)
- `dialog --msgbox 'Hello, OS World!' 8 32`
- `ssh sshtron.zachlatta.com` (online game)

Key takeaways:

- Programming doesn't have to be boring from the start.
- It may seem complex, but it's actually quite simple.

A Fundamental (but Difficult) Question

Registers are easy to understand (observable using `gdb` + `info registers`).

Process State Model:

- What is ”a process’s memory”?

Question 2 in HW2

Objective: Use debugging tools to understand how the address space works by analyzing program outputs and instruction locations.

```
#include <stdio.h>
int main()
{
   printf("%p\n", main);

   int x = *(int*)main;
   printf("%x\n", x);
}
```

Task:

1. Explain why the program produces two different outputs.
2. Specifically analyze how the first output relates to the second output.
3. Your answer should include:
   - Two outputs.
   - A detailed explanation with calculations of the relationship between the address, the instruction bytes, and the program's outputs.
   - Explanation supplemented by several screenshots of debugging using `objdump` or `gdb`.

What could the following program output?

```
#include <stdio.h>
int main()
{
   printf("%p\n", main);

   int x = *(int*)main;
   printf("%x\n", x);
}
```

What Memory Access is Valid in the Address Space?

What type of pointer access would NOT cause a segmentation fault?

```
char *p = random();
*p; // Load
*p = 1; // Store
```

How to View the Address Space of a Linux Process?

(Curious: How is `pmap` *implemented?)*

Process Address Space

RTFM: `man 5 proc`

- `/proc/[pid]/maps`
- `pmap`
- `gdb+info proc mappings`

E.g., use `gdb printmain` and `info proc mappings` to check the starting address of the ELF file for a better understanding of Question 2 in HW2.

- Each segment of the process address space:
  - Address range and permissions (rwxsp)
  - Corresponding file: offset, dev, inode, pathname
  - TFM provides detailed explanations
- Verified with the information from `readelf -l`

How about using `gdb minimal` and `info proc mappings`?

- vvar (Virtual Variable Page)
- vdso (Virtual Dynamic Shared Object)
- vsyscall

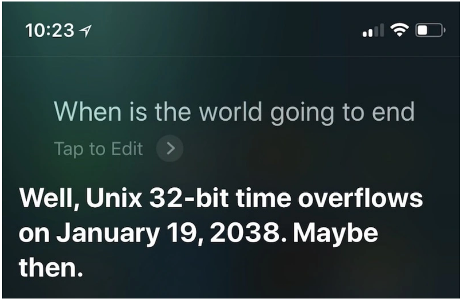
Our Treasure Found

System Calls Without Kernel Trap

- `vdso`: A shared library mapped into user space, containing functions (code) that user programs can call directly instead of invoking a system call.
- `vvar`: A read-only memory page in user space that stores kernel-exported data, such as timekeeping information, to support ‘vdso‘ functions.

VDSO

- These optimizations eliminate the need for a costly ‘int 0x80‘ or ‘syscall‘ instruction in specific cases, improving performance.
- Example: `time(2)` for seconds and `gettimeofday(2)` (a very clever implementation)
- `strace -e trace=gettimeofday ./vdso` Code



VDSO and VVAR: Communication Mechanism

- We do not need syscalls.
- What we need is a communication channel between user space and the kernel.
- **Shared Memory Page:**
  - In some extreme cases, a shared page can be read and written by user programs.
- **Periodic Updates:** The OS periodically updates the shared page.
- **Synchronization:** Spinlocks are used to protect the integrity of read and write operations on this page.

Further Questions

`execve` creates the initial state of a process, including registers and segments of memory.

Can we *control* the output of `pmap`?

- Modify the size of the `bss` segment in memory
- Allocate large arrays on the stack...

Managing Process Address Space

Perspective from the State Machine:

- Address space = memory segments with access permissions
  - Does not exist (inaccessible)
  - Exists but inaccessible (read/write/execute not allowed)
- Management: Add/Remove/Modify a segment of accessible memory

Question: What kind of system calls would you provide?

Memory Map System Calls

Adding/Removing/Modifying Accessible Memory in the State Machine:

- RTFM: `man 2 mmap`
- `MAP_ANONYMOUS`: Anonymous memory allocation
- `fd`: Map files into the process address space (e.g., loading libraries)
- Refer to the manual for more complex behaviors (complexity increases)

Code Injection Attack

Q: When you allocate memory using `malloc` and inject shellcode or assembly code into it, attempting to execute the code directly will typically result in a segmentation fault. This is because the memory allocated by `malloc` is marked as readable and writable but not executable, in accordance with modern operating systems' NX (Non-Executable) or W^X (Write XOR Execute) policies.

A: To execute code from such a memory region, you need to change its permissions to executable. This can be achieved by using `mmap` (with flags such as `PROT_READ | PROT_WRITE | PROT_EXEC`) or by using `mprotect` to modify the existing memory region's permissions.

Examples:

```
// Mapping
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);

// Modifying permissions
int mprotect(void *addr, size_t length, int prot);
```

Using mmap

Example 1: Allocating a Large Memory Space

- Instantaneous memory allocation
  - `mmap`/`munmap` provides the mechanism for `malloc`/`free`.
  - `libc`'s `malloc` directly invokes `mmap` for large allocations.
- Consider using `strace`/`gdb` to observe the behavior.

Example 2: Everything is a File

- Map a large file and access only part of it.

```python
with open('/dev/sda', 'rb') as fp:
    mm = mmap.mmap(fp.fileno(),
                   prot=mmap.PROT_READ, length=128 << 30)
    hexdump.hexdump(mm[:512])
```

Intruding Address Spaces

How to Make Mods for Games

Intruding Address Spaces

- A process (state machine) executes on a "dispassionate instruction machine."
  - The state machine is a self-contained world.
  - **But what if a process is allowed to access the address space of another process?**
    - It implies the ability to observe or modify another program's behavior.
    - Sounds pretty cool!

Examples of "invading" address spaces:

- Debugging (gdb)
  - *gdb* allows inspecting and modifying the state of a program.
- Profiling (perf)
  - Tools like *perf* help analyze the performance bottlenecks of a program.

Physical Intrusion into Address Spaces

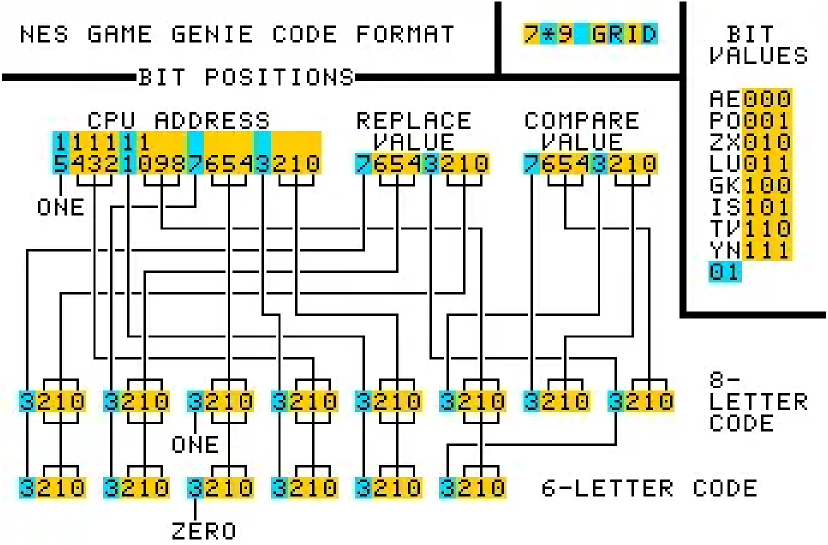
Golden Finger: Directly Manipulate Physical Memory

- Sounds distant, but it was achievable during the ”cartridge” era!

- Today, we have tools like Debug Registers and Intel Processor Trace.
- These tools assist systems in ”legally intruding” into address spaces.

Physical Intrusion into Address Spaces (cont'd)

Game Genie: A Look-up Table (LUT)



- Simple yet elegant: When the CPU reads address a and retrieves x,

Game Genie as a Firmware

Game Genie as a Boot Loader

- Configures the Look-Up Table (LUT) and loads the cartridge code.
- Functions like a simple ”Boot Loader.”

Expanding Game Exploration

Address Space: Where is the "Gold"?

- Includes dynamically allocated memory, with varying addresses every time.
- Insight: **Everything is a state machine.**
  - By observing the trace of state changes, you can identify the valuable addresses.

Search + Filter

- Enter the game: `exp = 4950`.
- Perform an action: `exp = 5100`.
- Match the memory locations where `4950 → 5100` occurs.
  - These memory locations are very few.
- Once found, you're satisfied!

Automation with Precision

Repeating Fixed Tasks at Scale (e.g., 2 seconds, 17 shots)

- Example shown demonstrates automating repetitive actions with precise timing.
- Such tools enable consistent execution of predefined tasks without manual intervention.

Implementing Precision Automation

Sending Keyboard/Mouse Events to Processes

- Developing Drivers (e.g., custom keyboard/mouse drivers)
- Leveraging System Window Manager APIs
  - **xdotool**: Useful for testing, including plugins for VSCode
  - **ydotool**
  - **evdev**: Commonly used for live streaming or scripting key sequences

Application in 2024: Implementing AI Copilot Agent

- Automating workflows: Text/Image Capture $\rightarrow$ AI Analysis $\rightarrow$ Execute Actions

Principle of Speed Modification: Theory

Programs as State Machines

- **Instructions** cannot perceive time.
- Situations like **spin count** may lead to issues such as:
  - Perception of the ”machine running too fast or the game being unplayable.”
- **Syscalls** are one way for programs to perceive time.

”Acceleration” and Time-Related Syscall/Library Functions

- Modify a program’s perception of time.
- Similar to adjusting a clock to make it appear faster or slower.

Custom Game Cheats

"Holding Code" is Essentially Debugger Behavior

- Games are programs and are also state machines.
- Cheats are like a **"dedicated gdb for the game"**.

Example: Locking Health Points

- Create a thread to spin and modify:

```
while (1) hp = 9999;
```

- However, conditions like `hp < 0` (e.g., one-hit kill) may still occur.
- Solution: Patch the code that checks `hp < 0` (soft dynamic updates).

Code Injection

Hooking Functions with Code

- Using a piece of code to **hook** the execution of a function.
- Allows tampering with the program's logic and gaining control.

Code Injection

Hooking Functions with Code

- Using a piece of code to **hook** the execution of a function.
- Allows tampering with the program's logic and gaining control.

Code Injection (cont'd)

Narrative Example:

> *"I heard the devil fruit is actually the incarnation of the devil in the sea. Eating it grants devil-like abilities but also condemns the user to the curse of the sea."*

On Cheats and Code Injection

Cheats Can Also Serve "Good" Purposes:

- **Live Kernel Patching:** Enable "hot" updates without stopping the system.
- Techniques, whether in computing systems, programming languages, or artificial intelligence, are meant to provide benefits to humans — for example, debugging tools and even cheats can help game developers or testers improve performance.

Ethics of Technology:

- Strong technology always has both "good" and "bad" applications.
- Any misuse of technology to harm others is a violation of integrity. Similarly, if cheats are used for malicious purposes in games, we should also consider the moral implications and use tools responsibly.