

# Lecture 7-9: 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>

## 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.

```
#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.

## 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**.

## 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"?

## Constructing the Smallest Hello, World

```
#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.

**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 → syscall → Computation → ...

# 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
```

## 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`

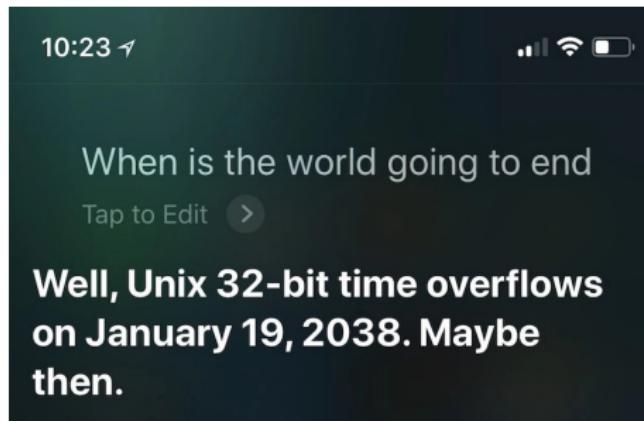
How about using `gdb minimal` and `info proc mappings`?

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

## 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.

- 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](#)



- 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...

## 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?**

## **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<sup>X</sup> (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);
```

## 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.

```
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

# Game Cheat 1: 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)
  - !ps or !pmap in gdb a.out
  - *gdb* allows inspecting and modifying the state of a program.
- Profiling (perf)
  - Tools like *perf* help analyze the performance bottlenecks of a program.

- **How gdb Uses ELF Files**

- ELF contains function symbols, variable locations, and debugging metadata.
- *gdb* reads the ELF file to get debugging symbols.

- **Accessing Another Process's Address Space**

- *gdb* can attach to a running process.
- It allows inspecting and modifying memory and registers.
- Achieved through system calls (e.g., `ptrace` in Linux).

## Key Concept: The OS as an API and Object

- The OS provides APIs that allow a process to debug another.
- Can these APIs ensure security and prevent unauthorized access?

# Physical Intrusion into Address Spaces

## Golden Finger: Directly Manipulate Physical Memory

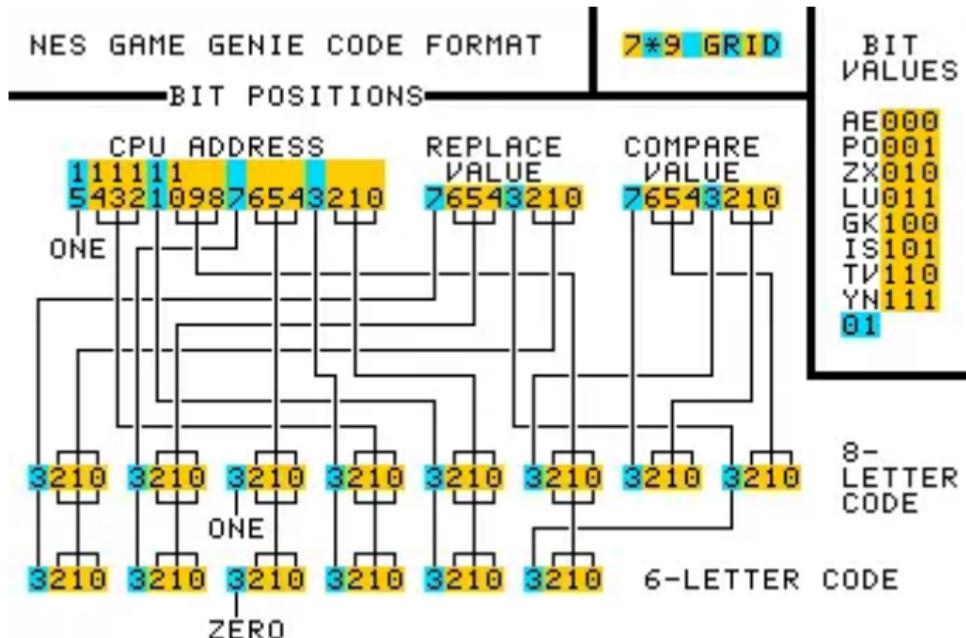
- Sounds distant, but it was achievable during the "cartridge" era!



- 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  $x$ , replace it with  $y$ .
- [Technical Notes \(Patents, How did it work?\)](#)

# 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."



# The Blurring Boundaries Between I/O Dev and Comp

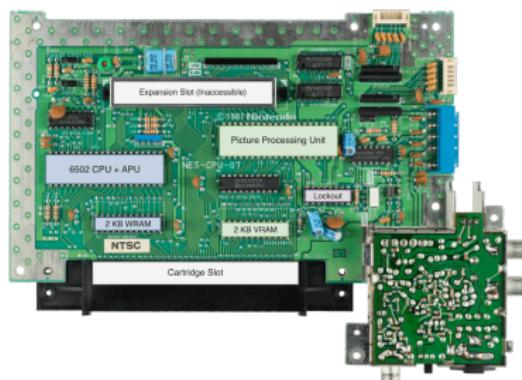
- How can we have CPUs for various tasks?

## Example: Displaying Patterns

```
#include <stdio.h>

int main() {
    int H = 10;
    int W = 10;

    for (int i = 1; i <= H; i++) {
        for (int j = 1; j <= W; j++) {
            putchar(j <= i ? '*' : '_');
        }
        putchar('\n');
    }
}
```

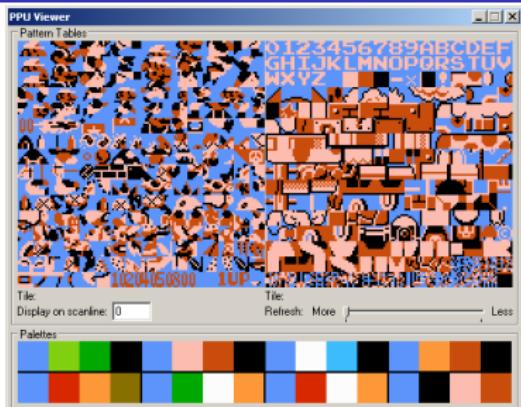
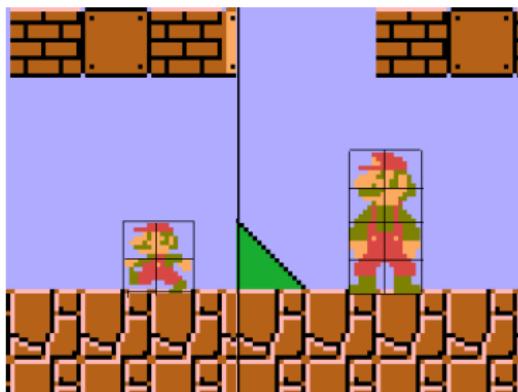


Nintendo Entertainment System (NES)  
Motherboard

**The Challenge of Performance:** NES: 6502 @ 1.79MHz; IPC = 0.43

- Screen resolution:  $256 \times 240 = 61K$  pixels (256 colors)
- 60FPS  $\Rightarrow$  Each frame must complete within 10K instructions
  - How to achieve 60Hz with limited CPU computing power?

# NES Picture Processing Unit (PPU)



The **CPU** only **describes** the arrangement of 8x8 tiles

- The background is part of a larger image
  - No more than 8 foreground tiles per line
- The PPU completes the rendering
  - A simpler type of "CPU"
- Enjoy!

7	6	5	4	3	2	1	0	
-	-	-	-	-	-	+	+	Palette
-	-	+	-	-	-	-	-	Unimplemented
-	+	-	-	-	-	-	-	Priority
+	-	-	-	-	-	-	-	Flip horizontally
-	-	-	-	-	-	-	-	Flip vertically

# Providing Rich Graphics with Limited Capability

Why do the characters in KONAMI's Contra adopt a prone position with their legs raised?

- Video



## What if we have more powerful processors?

- The NES PPU is essentially a "tile-based" system aligned with the coordinate axes.
  - It only requires addition and bitwise operations to work.
- Greater computational power = More complex graphics rendering.

## 2D Graphics Accelerator: Image "Clipping" + "Pasting"

- Supports rotation, material mapping (scaling), post-processing, etc.

## Achieving 3D

- Polygons in 3D space are also polygons in the visual plane.
  - Thm. Any polygon with  $n$  sides can be divided into  $n - 2$  triangles.

# Simulated 3D with Clipping and Pasting

## GameBoy Advance

- 4 background layers; 128 clipping objects; 32 affine objects
  - CPU provides the description; GPU performs the rendering (acting as a "program-executing" CPU)



V-Rally; Game Boy Advance, 2002

# But We Still Need True 3D

## Triangles in 3D space require correct rendering

- Modeling at this stage includes:
  - Geometry, materials, textures, lighting, etc.
- Most operations in the rendering pipeline are massively parallel



*"Perspective correct" texture mapping (Wikipedia)*

# Solution: Full PS (Post-Processing)

## Example: GLSL (Shading Language)

- Enables "shader programs" to execute on the GPU
  - Can be applied at various rendering stages: vertex, fragment, pixel shaders
  - Functions as a "PS" program to calculate lighting changes for each part
    - Global illumination, reflections, shadows, ambient occlusion, etc.



# Modern GPU: A General-Purpose Computing Device

A complete multi-core processing system

- Focuses on massively parallel similar tasks
  - Programs are written in languages like OpenGL, CUDA, OpenCL, etc.
- Programs are stored in memory (video memory)
  - nvcc (LLVM) compiles in two parts
    - Main: Compiles/links to a locally executable ELF
    - Kernel: Compiles to GPU instructions (sent to drivers)
- Data is also stored in memory (video memory)
  - Can output to video interfaces (DP, HDMI, ...)
  - Can also use DMA to transfer to system memory

# Example: PyTorch and Deep Learning

What is a "Deep Neural Network"?

How do we "train"?

- Requires computationally intensive tasks

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512), nn.ReLU(),
            nn.Linear(512, 512), nn.ReLU(),
            nn.Linear(512, 10), nn.ReLU(),
        )
    ...
model = NeuralNetwork().to('cuda')
```

Many components can perform the "same task"

- The key is to choose the component with the most suitable power/performance/time trade-off!

## Examples of Components:

- CPU, GPU, NPU, DSP, DSM/RDMA

# Game Cheat 2: 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 = 4610.`
- Perform an action: `exp = 5370.`
- Match the memory locations where  $4610 \rightarrow 5370$  occurs.
  - These memory locations are very few.
- Once found, you're satisfied!

## Repeating Fixed Tasks at Scale (e.g., 1 second, 5370 shots)

Enjoy!

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

## 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 → AI Analysis → Execute Actions

# Game Cheat 4: Adjusting Logic Update Speed

## Adjusting the Game's Logic Update Speed

- For example, a certain mysterious company's game is so slow that both map traversal and combat feel unbearable.
- The gaming industry today has become so competitive that if a new player's progression path isn't smooth, the game will be heavily criticized.



# Principle of Speed Modification: Theory

## Program = State Machine

- "Compute instructions" are inherently unaware of time.
- Using count for timing can lead to issues where the game becomes unplayable on faster machines.
- **Syscalls** are the only way for a program to perceive time.

## "Hijacking" Time-Related Syscall/Library Functions

- `gettimeofday`, `sleep`, `alarm`
- Replacing the system call's code with our own code allows us to alter the program's perception of time.
- Similar to adjusting a clock to make it appear faster or slower.

# 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.



# Hooking in Game Cheats

## How Hooking is Used in Game Cheats

- Hooking intercepts and modifies game functions to manipulate game behavior.
- Commonly used in ESP (Extra Sensory Perception) cheats, Aimbots, and Wallhacks.

## Methods of Hooking:

- DirectX/OpenGL Hooking: Modifies rendering functions like D3D11Present to draw ESP overlays.
  - System Call Hooking: Alters time-related functions (e.g., `gettimeofday`) to manipulate game physics.
  - Memory Hooking: Modifies in-game variables (e.g., `hp = 9999`) in real-time.

## Example: ESP Wallhack

- Hooks rendering APIs to bypass depth checks.
- Modifies enemy rendering to make them visible through walls.

## The Essence of "Hijacking Code" is Debugger Behavior

- A game is also a program, and a state machine.
- A cheat tool is essentially a `gdb` designed specifically 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., instant death) may still occur.
- Solution: Patch the code that checks `hp < 0` (soft dynamic updates).

# Code Injection (cont'd)

*"I heard that Devil Fruits are the incarnations of sea demons. Eating one grants devil-like abilities, but in return, the sea will reject the user."*

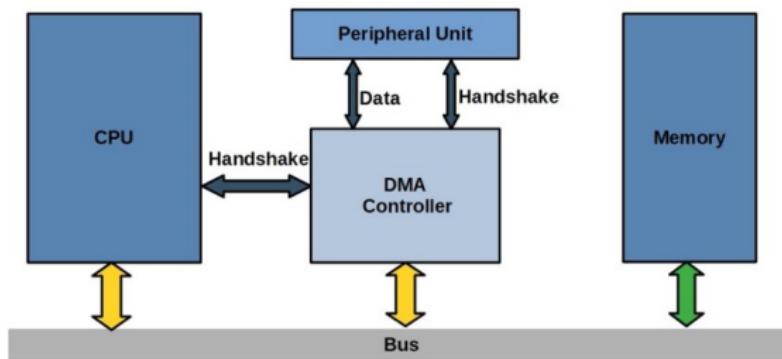


Enjoy!

# Game Cheat 5: DMA

**DMA (Direct Memory Access): A dedicated CPU for executing "memcpy" operations**

- Adding a general-purpose processor is too costly
- A simple controller is a better solution
- Supported types of memcpy:
  - memory → memory
  - memory → device (register)
  - device (register) → memory
    - Practical implementation: Directly connect the DMA controller to the bus and memory
    - Intel 8237A



- CPU is not involved in copying data
- A process cannot access in-transit data
- PCI bus supports DMA
  - Handles a large number of complex tasks

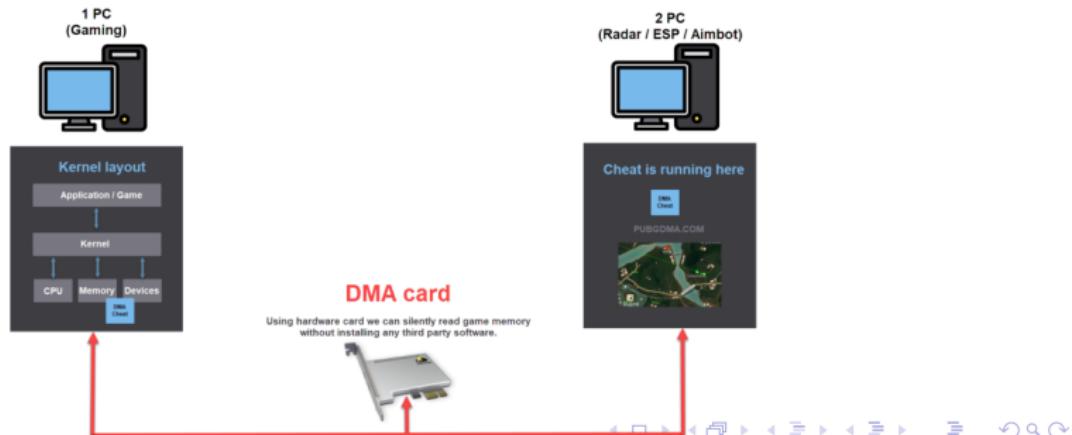
# Why Does DMA Cheating Exist?

- Modern anti-cheat methods rely on detecting memory modifications.
- Kernel-level anti-cheat software (e.g., Vanguard, BattleEye) prevents direct process memory access.
- Reading memory via software (e.g., external cheats) is highly detectable.
- **DMA bypasses all software-based detection** because it directly accesses memory **without CPU intervention**.

# How DMA Cheats Work

- ① A second computer with a **DMA capture card** is used.
- ② The card is installed in the main gaming PC via **PCIe**.
- ③ The DMA card **reads game memory** and extracts relevant data (e.g., player positions).
- ④ The extracted data is sent to the second PC for processing.
- ⑤ The second PC renders an **ESP (extra-sensory perception) overlay**, giving the player an unfair advantage.
- ⑥ Since the main PC runs no cheat software, anti-cheat solutions fail to detect it.

## How does DMA works



# Why Is It Hard to Detect?

- **No modification of game memory** (only reading).
- **No injected code**, unlike traditional hacks.
- **Appears as a legitimate PCIe device**, making it difficult to blacklist.

## Current Anti-Cheat vs. DMA

Anti-Cheat Method	Effectiveness Against DMA
Signature Scanning	Ineffective (DMA is external)
Kernel-Level Hooks	Ineffective (DMA doesn't use system calls)
Code Integrity Checks	Ineffective (No code modification)
Behavior Analysis	Partially Effective (Detecting unnatural movements)

# Future of Anti-DMA Methods

- **Hardware-based solutions:** Restricting PCIe device access via BIOS/firmware.
- **AI-based detection:** Tracking suspicious player behavior.
- **Encrypted memory:** Preventing DMA from extracting useful data.
- Currently, **no effective universal countermeasure exists.**

## 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.