HOP - Hook Oriented Programming

Mouad Abid

DHBW Mannheim
Computer Science Department - Cybersecurity
TINF CS2

Abstract

*Software piracy is one of the concerns in the IT sector. Hackers leverage debuggers to reverse engineer the logic that verifies the license keys or bypass the entire verification process. Anti-debugging techniques are used to mitigate piracy using multiple techniques. However, anti-debugging methods can be defeated when the licensing protections are limited to what can be patched. This paper presents a custom anti-debugging solution, Hook Oriented Programming (HOP), tailored for Linux-based systems, relying on the fact that only one process can act as a debugger for another given process at a time. In other variations, debugger APIs exposed by the platform are hooked and monitored globally to detect debugging attempts[1]. Demo code is provided under this **github repo**, and all figures are added in full size at the end of the paper.*

I. Introduction

In software development and cybersecurity, reverse engineering and anti-reverse engineering represent two sides of the same coin. On the one hand, software engineers strive to protect their proprietary software from piracy through sophisticated copy-protection schemes, which could save companies millions in potential lost revenue. On the other hand, malware analysts face the challenge of dissecting malicious software, such as worms, to understand their behavior and develop countermeasures. The ability to reverse engineer such malware is critical to enhance antivirus products and secure networks.

These two scenarios underscore the importance of both reverse engineering for defense and anti-reverse engineering for protection. Anti-debugging techniques play a pivotal role in this ongoing battle, serving as a line of defense against those who seek to reverse engineer software for nefarious purposes. These techniques are designed to detect and disrupt the operation of debuggers, which are tools commonly used in the reverse engineering process[2].

Despite their significance, anti-debugging strategies have received relatively less attention in research compared to other anti-reverse engineering methods such as obfuscation, virtual machine detection, and tamper-proofing. This is surprising given the prevalence of anti-debugging in both malicious code and legitimate copy-protection mechanisms.

Anti-debugging involves the implementation of code that can identify and counteract debugging attempts on a target process. This can be done within the software itself or through system-level hooking in either user or kernel mode. By embedding these defenses into the final binary, software developers can make it significantly more challenging for unauthorized individuals to debug and reverse engineer their code[3][4].

In the domain of reverse engineering, debuggers are instrumental in analyzing the execution of a program by facilitating the modification of the process under inspection. This includes the insertion of breakpoints, which are achieved by overwriting instructions, modifying memory permissions, or altering exception handler chains. Consequently, anti-debugging strategies are designed to detect and counteract these modifications, often by restoring the original state of the program. This restoration process may necessitate the alteration of the program's own code, frequently employing self-modifying code techniques. The objective of such anti-debugging measures is not to provide an impenetrable barrier against reverse engineering but to increase the complexity and time required for successful analysis, thereby serving as a deterrent[5].

Anti-debugging techniques generally can be grouped into the following categories though not all techniques apply to all platforms[6]:

- Timing and Latency Analysis
- Process Detection
- Memory Analysis
- Break-point Detection
- Patching Detection
- Monitoring Debugger APIs
- Monitoring Exception Handlers
- Blocking Debuggers

This paper introduces a novel anti-debugging methodology termed Hook Oriented Programming (HOP), which leverages syscall and intermediate function hooking, capitalizing on the principle that a process can only be traced once. This approach ensures the Child[1] process is entirely reliant on the

[1] We mean by Child the Child process, spawning after fork()

Parent[2] for execution. Section II provides a review of related research, specifically focusing on Nanomite technology, a sophisticated anti-debugging and anti-dumping method. Section III lays the groundwork by elucidating key concepts essential to understanding HOP. Section IV delves into the system's design and the practicalities of its implementation. The paper concludes with Section V, summarizing the findings and implications of HOP[7][8].

II. Related Work

1. Nanomites Implementation

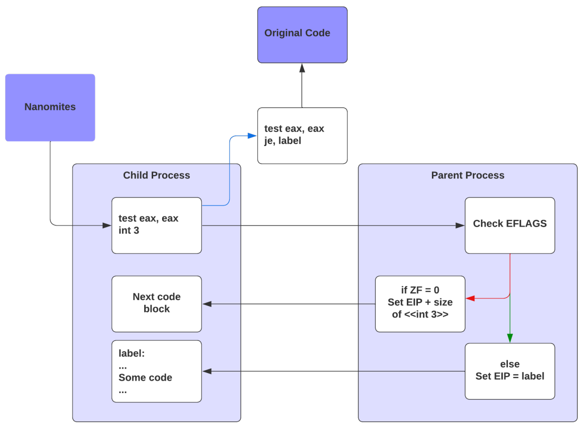

Figure 1: *Nanomite Scheme representing the communication between the Child and the Parent process*

The foundation of Nanomite technology relies on the Parent process protection paired with code segment extraction for packing, followed by obfuscation upon unpacking. The table of conditional and unconditional jumps, together with obfuscation, is used to remove marked code segments (Nanomites) from the source code and replace them with jumps in a certain way. In Windows protectors, Parent process protection (sometimes called debug blocker) launches the protected program as a Child process and attaches itself to it for debugging. Therefore, the application itself cannot be debugged by a third party; only the Parent process can.

"There are always some ways to fight any protection," Anton Kotik, Software Designer of Apriorit and Reverse Engineering Group member, says. "But with Nanomites, it's really hard and extremely time-consuming. You can detach the Parent process only after restoring all Nanomites – so while restoring, you have to work with the Parent process only. As there are no jumps in the software – they are all replaced – the application is a solid piece of code in disassemblers."

Figure 1 presents the operational framework of Nanomites, offering an overview of their functionality and how they are managed by the Parent process. It is a modern anti-debugging technique initially introduced by *Armadillo*. It operates by initially replacing branches with the **0xCC** (*INT_3*[3]) instruction, then creating a Child process and attaching to it. Throughout runtime, the Parent process governs the execution flow of its Child, such that whenever a debugging exception is triggered (as a result of the INT 3 instruction), the Parent resolves it by finding matches in the Nanomites table.

2. Fighting Nanomites

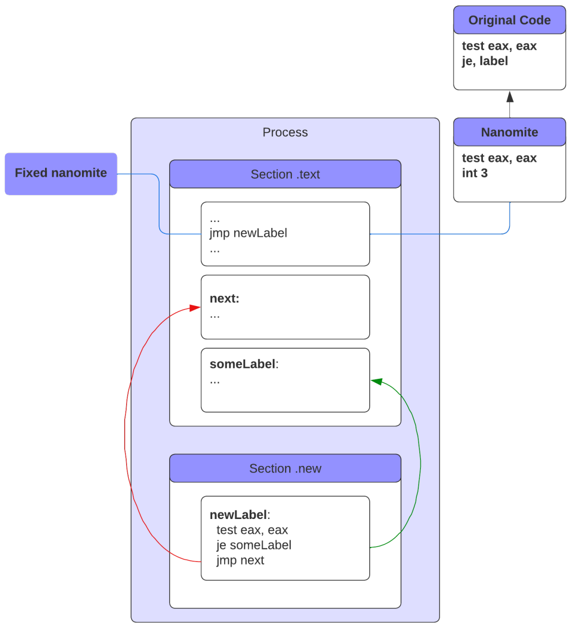

Figure 2: *Recovering Scheme representing a way to recover the execution flow from the Parent.*

Reversing applications protected with Nanomites is challenging due to several factors:

- The constant switching of context between the Parent and Child processes slows down the execution and complicates the debugging process.
- The presence of false entries in the Nanomite table can mislead reverse engineers, as these entries may never be used but are designed to confuse attempts to reconstruct the original code flow.
- The need to understand the communication and control flow between the Parent and Child processes, which requires a deep understanding of

[2] We mean by Parent the Parent process, the caller of fork()

[3] This debug exception is conventionally associated with the process of debugging

the application's structure and the Nanomite technology itself.

However, like any other mechanism, a novel approach exists. It could be summarized in adding an extra section to the executable file, extracting Nanomite jump instructions and addresses, and then adding the gadgets made by the *jmp* instruction and the addresses to the new section. This technology, while promising, has various technological hurdles that must be solved in order to properly circumvent Nanomite's defense. To begin with, the hacker must first identify the Nanomite-protected sections within the executable file. Once located, the analyzer must "execute" the code and navigate through all its branches, carefully identifying and restoring Nanomites according to the associated table. This process is complicated by the need to handle switch tables, table calls, and API functions with CALLBACK parameters, which require a deep understanding of low-level code execution and control flow. The newly created section serves as a repository for the extracted Nanomite jump instructions and addresses. So the last step involves creating the restored jump gadgets and their corresponding addresses obtained from the Nanomite table. However, this process is not without its challenges. The moved instructions may contain jump instructions with relative jumps, which will require corrections.

III. Background

1. Syscalls

In the Linux ecosystem, the management of virtual memory is crucial for the secure and efficient operation of the system. This virtual memory is divided into two distinct spaces: the kernel space, also known as **"Ring 0,"**, and the user space, often referred to as **"Userland."** The kernel space is a highly privileged area where the core of the operating system resides, while the user space is where unprivileged user applications and processes operate[1]. The boundary between these two spaces is carefully guarded, and any communication or interaction between them must be conducted in a controlled and secure manner. This is where system calls, or syscalls, come into play. A syscall serves as a bridge between the user space and the kernel space, enabling user applications to request services and functionality from the kernel[9]. When a user application needs to perform a task that requires kernel-level privileges or access to system resources, it initiates a syscall. This triggers a controlled transition of control from the user space to the kernel space, allowing the requested operation to be carried out by the kernel on behalf of the user application. Once the kernel has completed the requested task, control is returned to the user space[9]. Syscalls can be invoked in several ways, including via assembly code, by directly calling the syscall, or through wrapper functions in standard C libraries such as glibc[4].

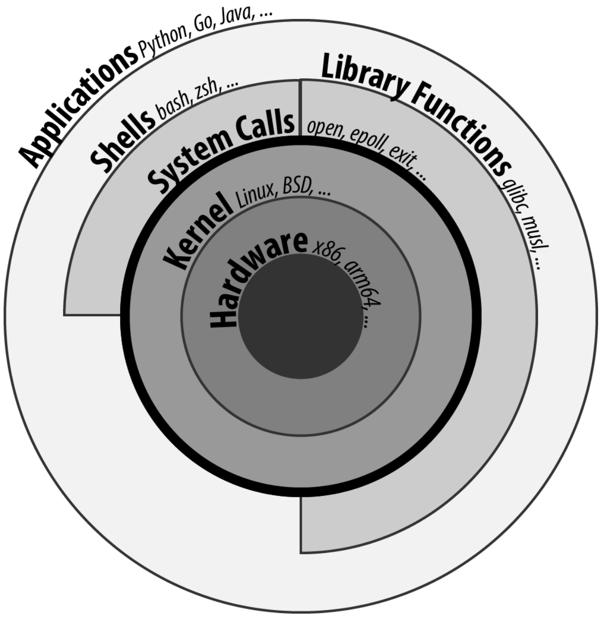

Figure 3: *Syscall chart representing, the intigrity of syscall in the linux world.*

2. Processes

In the Linux world, inter-process communication (IPC) is a fundamental concept that enables processes to exchange data and synchronize their actions. One of the key mechanisms for IPC involves the ptrace() system call, which allows a process (the "tracer") to observe and control the execution of another process (the "tracee"). When a process is being traced using ptrace(), the tracing process gains the ability to intercept and inspect various events in the traced process's execution, including system calls. This is achieved through the use of signals, with the kernel notifying the tracing process about specific events in the traced process's execution. For example, when a traced process (the "tracee" or Child process) makes a system call, the kernel interrupts the execution of that process and notifies the tracing process (the "tracer" or Parent process) through a signal, typically **SIGTRAP**. This interruption effectively "stops" the traced process, allowing the tracer to inspect and potentially modify the process's state, including the system call parameters, return values, and CPU registers. The ability to intercept system calls and inspect

[4] For our demo, we are going to use dietlibc.

the state of a traced process is a powerful feature that is commonly used for debugging and analysis purposes. Tracing processes can utilize this capability to gain insight into the behavior of the traced process, monitor its system call activity, and even modify its behavior under certain conditions.

IV. System Design

Building upon our foundational knowledge of inter-process communication (IPC) and system calls (syscalls), we can now apply this understanding to create a robust defense against anti-debugging techniques. This paper introduces Hook Oriented Programming (HOP), a method that involves hooking syscalls and libc calls to safeguard processes. Our demonstration will focus on employing the write syscall and dietlibc[5] as a practical example to illustrate the effectiveness of HOP. During the demonstration and as a proof of Concept (PoC), we will establish two hooks. The first hook will execute false randomized syscalls, with the Parent process poised to correct the syscall number immediately before the kernel executes it. The second hook will involve creating a write hook that generates a random number and records it in our newly established syscall_mapping. Figures 4, 5, 6, and 7 will serve as references throughout the demonstration, guiding us in implementing these hooks. By leveraging the concept of HOP, we aim to demonstrate a sophisticated approach to protecting critical code within processes from being debugged by unauthorized entities.

1. From write to syscall(1, ...)

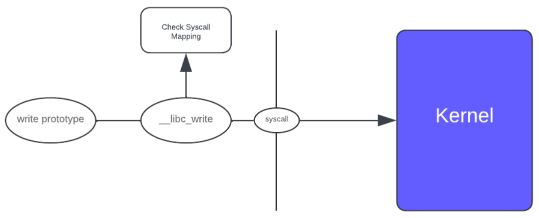

Figure 4: *The abstraction of write function in C.*

In C programming, when you call the write function to output data, you're using a high-level interface provided by the C standard library. This function eventually needs to make a system call to the operating system's kernel to perform the actual I/O operation. The transition from the high-level write function call to the low-level system call involves several steps, which are abstracted by the use of macros and assembly code. The syscall_weak macro is part of this abstraction, and in our example, **dietlibc** uses syscall_weak to make this transition. It defines two symbols, **wsym** and **sym**, as functions. **wsym** is marked as a weak symbol, which means it can be overridden by another symbol of the same name if it's defined elsewhere. While **sym** is declared as a global symbol, making it accessible from other parts of the program.

[5] A minimalistic version of libc, developed from scratch and thus only implements the most important and commonly used functions.

Listing 1: *The syscall_weak in dietlibc*

```
// syscall_weak(write, write, \
    __libc_write)
#define syscall_weak(name,wsym,sym) \
.text; \
.type wsym,@function; \
.weak wsym; \
wsym: ; \
.type sym,@function; \
.global sym; \
sym: \
.ifge __NR_##name-256 ; \
    mov $__NR_##name,%ax; \
    jmp __unified_syscall_16bit;  \
.else ; \
    mov $__NR_##name,%al; \
    jmp __unified_syscall; \
.endif
```

The numbers of the system call are hardcoded into the C library (libc). These numbers are used to identify which syscall should be executed by the kernel. They depend on multiple factors from architecture to OS, etc.

2. Hooking non-implemented syscalls

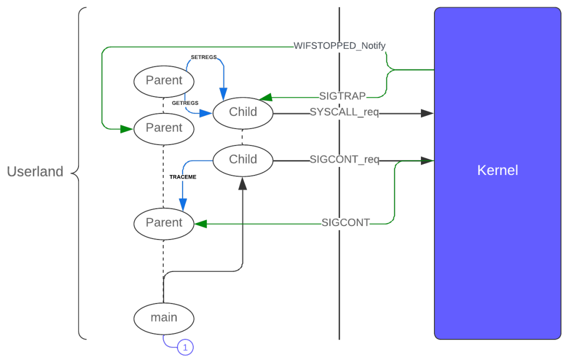

Figure 5: *IPC infra to achieve syscall hooking.*

To effectively hook a syscall, it's essential to understand that each process can be traced only once at a time. This limitation is crucial because implementing an anti-debugging mechanism involves establishing a dependent relationship between a Child and Parent process. In this context, the Parent process is responsible for hooking and correcting the syscall just before being executed by the kernel. This process is made possible by the PTRACE_TRACEME property, which enables the Parent process to receive notifications before every syscall while causing the Child process to enter a stopped state using **SIGTRAP**. Figure 5 provides a comprehensive overview of this relationship, beginning with (1). Alternatively, Figure 6 offers a different perspective on the same concept, providing additional insights into the intricacies of this process.

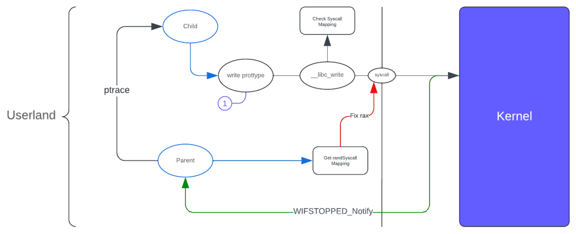

Figure 6: *PoC 1; hooking the syscalls through the Parent process.*

Following up, the next listing provides a more C oriented perspective, for further understanding of the concept:

Listing 2: *The tracee and The tracer*

```
// tracee.c
#include "main.h"
void
tracee(void)
{
    ptrace(PTRACE_TRACEME, 0, 0, 0);
    raise(SIGCONT);
    write(1, "Hello␣world\n", 12);
}

// tracer.c
#include "main.h"
void
tracer(pid_t Child_pid)
{
    // Some Code here ...
    ptrace(PTRACE_SETOPTIONS, \
        Child_pid, 0, PTRACE_KILL);
    while (WIFSTOPPED(status))
    {
        ptrace(PTRACE_SYSCALL, \
            Child_pid, 0, 0);
        waitpid(Child_pid, &status, \
            0);
        ptrace(12, Child_pid, 0, \
            &regs); // PTRACE_GETREGS
        if (regs.orig_rax == \
                CUSTOM_SYSCALL_WRITE)
        {
            regs.orig_rax = 1;
        ptrace(13, Child_pid, 0, \
            &regs); // PTRACE_SETREGS
        }
    }
}
```

3. Hooking the write function

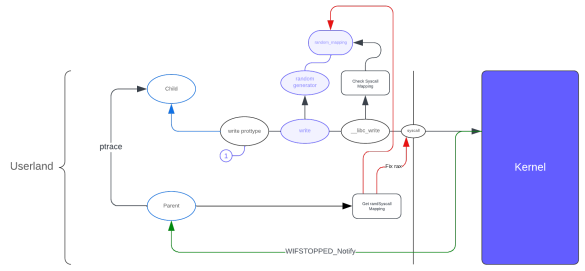

Figure 7: *PoC 2; hooking the write function through the Parent process.*

The previous PoC provides a relatively hard-to-debug interface, as it will require, carefully understanding the functionalities in case of dropping the Parent and attaching the debugger. However, the idea could be expanded to not only hook the syscall but also hook the write function hiding the pseudo-random generator, and save it in an extern object array, which should generate a different syscall on runtime.

The hooking was made by introducing a new node into our write abstraction, modifying the syscall_weak creating an alias of write, and hiding our random generator.

The initial proof of concept already presents a relatively challenging interface to debug, as it demands a thorough understanding of its functionalities before detaching the Parent and attaching the debugger. However, the concept could be further developed to not only hook the syscall, but also hook the write function and hide the pseudo-random generator within the write alias. This approach would result in the generation of a different syscall at runtime.

This process involves introducing a new node into our write abstraction and so modifying the syscall_weak function. Additionally, an alias of the write function was created to conceal our random generator.

Listing 3: *The write syscall and write hooked function*

```
// write.S
#include "syscalls.h"

// syscall_weak(write, write, \
    __libc_write)
.extern syscall_table_rand

.text
.type __libc_write,@function
.global __libc_write
__libc_write:
  lea syscall_table_rand(%rip), %rax
  add $4, %rax
  movl (%rax), %eax

  movzx %ax,%eax
  jmp   __unified_syscall_16bit

// write hook
ssize_t write(const void* buffer, \
    int fd, size_t len)
{
  int lower_bound = 501;
  int upper_bound = 0x1337;

  srand(time(NULL));
  syscall_table_rand[__NR_write] = \
    rand() % (upper_bound - \
    lower_bound + 1) + lower_bound;
  return __libc_write(fd, buffer, len);
}
```

Our mechanism can be combined with Nanomite technology to create a formidable anti-debugging strategy. Nanomites, as used by Armadillo, replace branch instructions with INT3 (a breakpoint instruction), and the original instructions are stored in an encrypted table. When the program runs, the debugger hits these INT3 instructions and must consult the table to resolve the correct execution path. By integrating HOP, which involves hooking syscalls and libc calls, we can ensure that the Parent process resolves not only syscalls but also these Nanomite-induced jumps. This creates a layered defense where the Parent process acts as a gatekeeper, resolving both syscalls and control flow changes.

To further complicate the debugging process, we can introduce Signal Oriented Programming, manipulating the program's flow using signals like segfaults and timers. By setting up handlers for these signals, we can transfer control between processes in response to events, rather than through traditional jumps. This method can catch debuggers off guard, as they typically expect control flow changes to occur through standard jumps or calls. When combined with HOP and Nanomites, this signal-based control flow manipulation can make it nearly impossible for debuggers like GDB to resolve the program's execution path correctly, as the Parent process orchestrates the resolution of syscalls and control flow behind the scenes, while signals dynamically alter the execution without typical control flow instructions.

V. Conclusion

In this paper, we discussed an anti-debugging mechanism named HOP, which includes hooking syscalls and intermediate libc function, within a Parent process, which creates a one-to-one relationship between the Child and the Parent.

However, it is crucial to acknowledge that without the integration of proper encryption and obfuscation techniques, the effectiveness of anti-reversing measures, including HOP, can be compromised. As highlighted in our review of related research and key concepts, the dynamic nature of code analysis and reverse engineering poses a persistent challenge to software protection efforts[10][11]. Theoretical and practical insights into software obfuscation reveal that while it is impossible to achieve provably secure obfuscation for all program functionalities, employing a combination of data obfuscation, static code rewriting, and dynamic code rewriting can significantly raise the bar against reverse engineering attempts[10]. Moreover, the application of hybrid obfuscation techniques further complicates the reverse engineering process, underscoring the importance of obfuscation as a complementary strategy to anti-debugging measures[12].

References

[1] *Linux system programming: talking directly to the kernel and C library*. https://www.semanticscholar.org/paper/Linux-System-Programming:-Talking-Directly-to-the-C-Love/2fa2d15a94c80b6504181d40e30f3bd0ea95555a. 2013.

[2] *Software Protection through Anti-Debugging*. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4218560. Mai 2019.

[3] *Anti Debugging Protection Techniques with Examples*. https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software. Mai 2019.

[4] Peter Ferrie. *The Ultimate Anti-Reversing Reference*. https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf. April 2011.

[5] *Research Summary of Anti-debugging Technology*. https://iopscience.iop.org/article/10.1088/1742-6596/1744/4/042186/pdf. 2020.

[6] *Thwarting Piracy: Anti-debugging Using GPU-assisted Self-healing Codes*. https://ieeexplore.ieee.org/document/10041271. Jan. 2023.

[7] *Nanomite and Debug Blocker for Linux Applications*. https://www.codeproject.com/Articles/621236/Nanomite-and-Debug-Blocker-for-Linux-Applications. Jul 2013.

[8] *Nanomite and Debug Blocker Technologies: Scheme, Pros, and Cons*. https://www.apriorit.com/white-papers/293-nanomite-technology. June 2015.

[9] *The devil is in the detail: Generating system call whitelist for Linux seccomp*. https://www.sciencedirect.com/science/article/abs/pii/S0167739X2200139X. 2022.

[10] *Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?* https://www.plai.ifi.lmu.de/publications/csur16-obfuscation.pdf. April 2016.

[11] *Purple Book of Sofwate Security*. https://www.thepurplebook.club/the-book. April, 2022.

[12] *Hybrid Obfuscation Technique for Reverse Engineering Problems*. https://www.researchgate.net/publication/368575192_Hybrid_Obfuscation_Technique_for_Reverse_Engineering_Problems. December 2020.

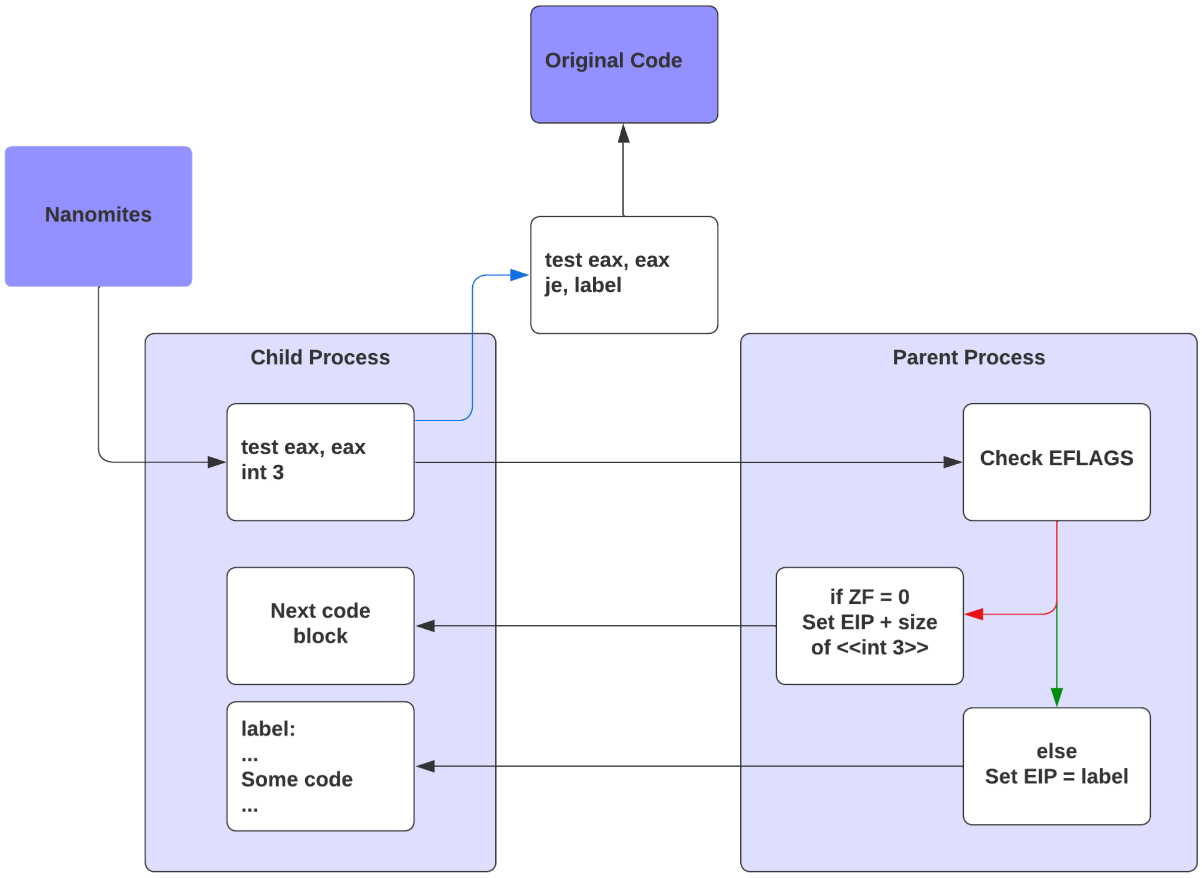

Figure 8: *Nanomite Scheme representing the communication between the Child and the Parent process*

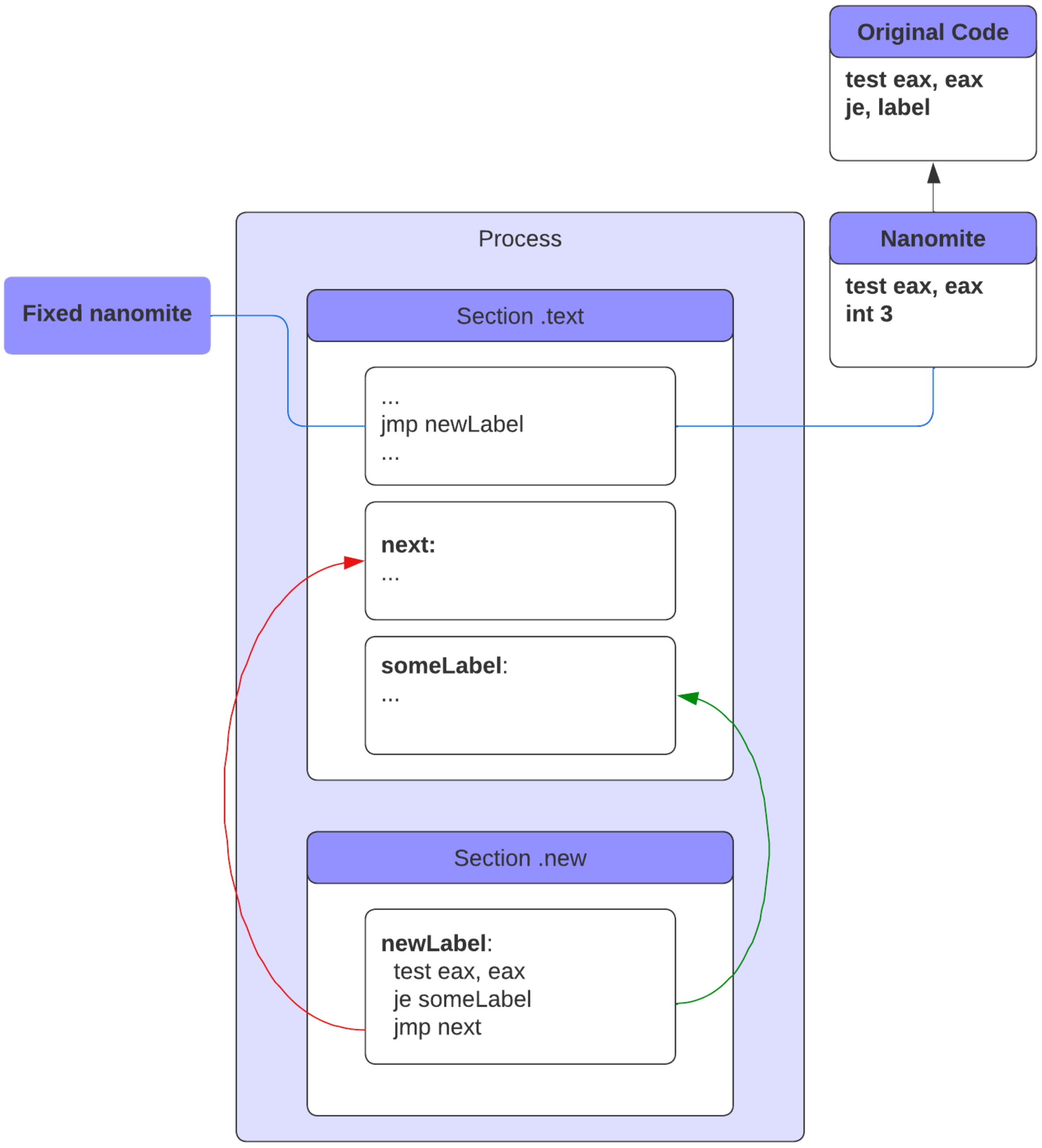

Figure 9: *Recovering Scheme representing a way to recover the execution flow from the Parent.*

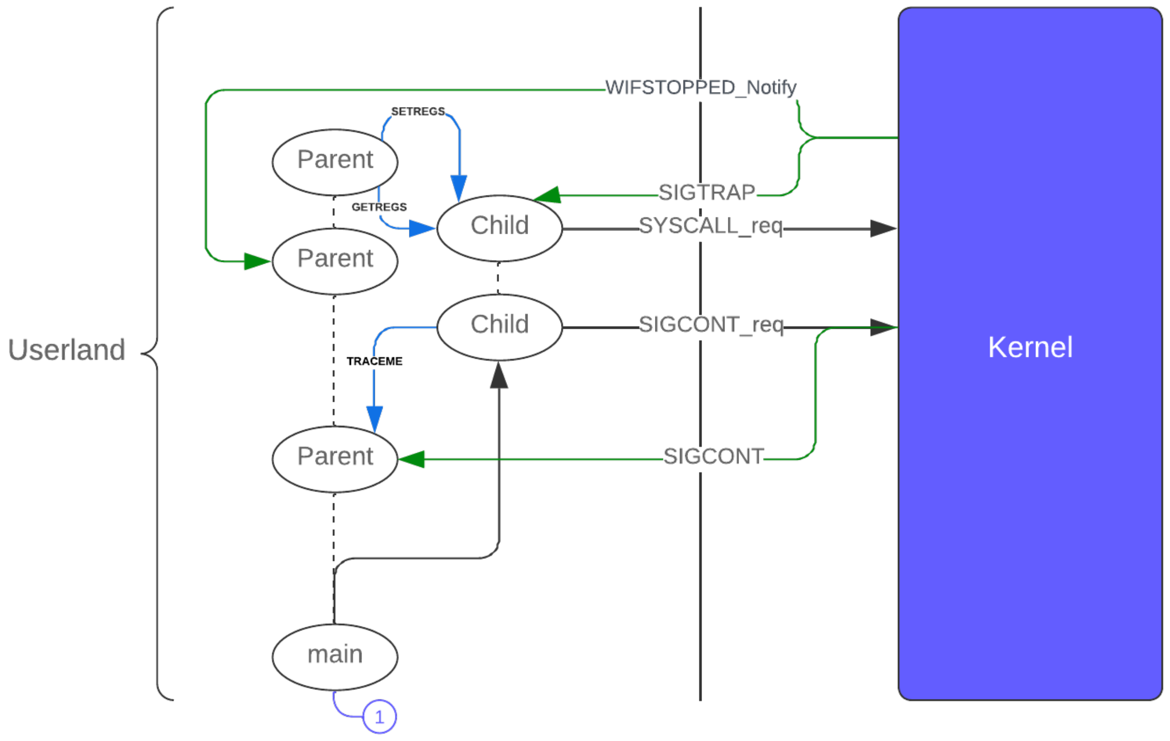

Figure 10: *IPC infra to achieve syscall hooking.*

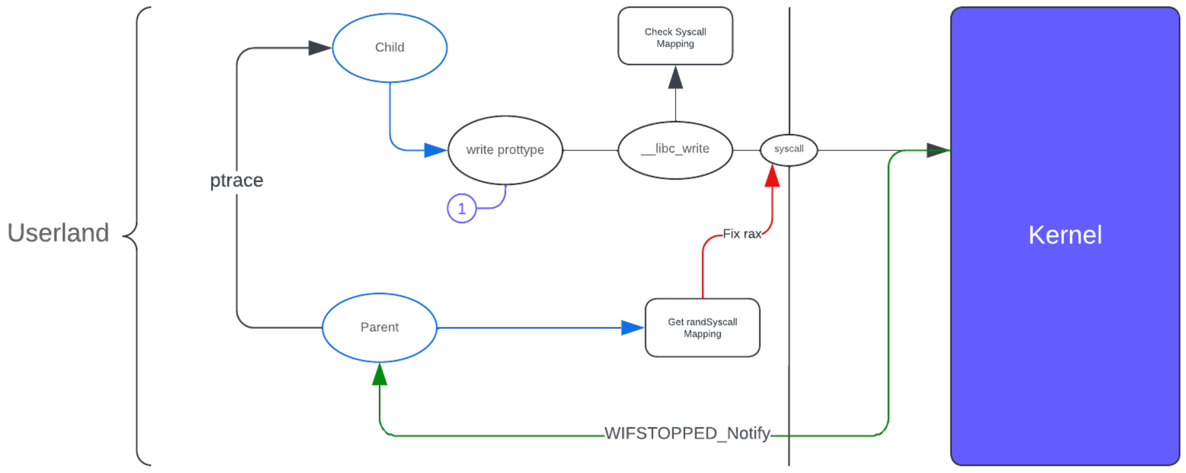

Figure 11: *PoC 1; hooking the syscalls through the Parent process.*

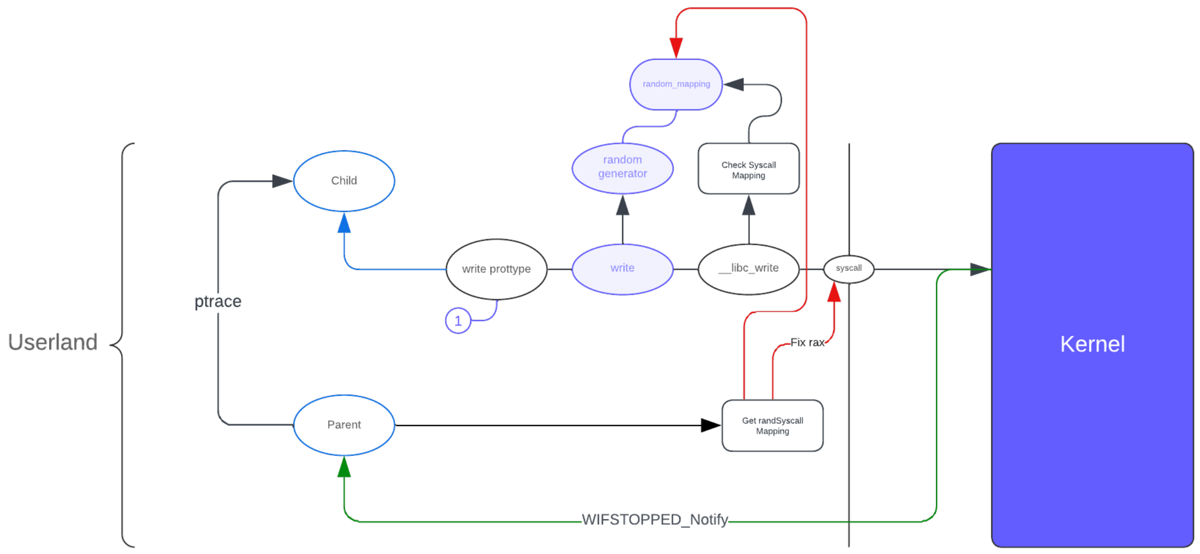

Figure 12: *PoC 2; hooking the write function through the Parent process.*