kGuard++: Improving the Performance of kGuard with Low-latency Code Inflation

Jordan P. Hendricks
Brown University

Abstract

In this paper, we introduce low-latency code inflation for `kGuard`, a GCC plugin that provides runtime defense against return-to-user (*ret2usr*) attacks. Our changes decrease performance overhead of kGuard's current code inflation mechanism by reducing the overall number of instructions needed to execute a check, while maintaining the benefit of a randomized address space. This solution improves the feasibility of using kGuard as a production solution to prevent ret2usr attacks, offering better performance without sacrificing functionality.

1 Introduction

Return-to-user (*ret2usr*) is a kernel exploitation technique that leverages *weak memory separation* of kernel and userspace memory to exploit memory vulnerabilities in the kernel. If such a vulnerability exists, an attacker may cause kernel code to jump to malicious code in userspace memory, allowing an exploit to run with full privileges of the kernel.

kGuard defends against ret2usr by inserting runtime checks at compile-time that ensure addresses of indirect branch targets are within kernel space. To increase difficulty in bypassing these checks, kGuard also uses *code inflation* to randomize the location of these checks within the kernel text segment.

The current implementation of code inflation introduces a minor, but unnecessary performance overhead: kGuard inserts a random number of no-op instructions (`NOP`s) to provide address space randomization. These NOPs are always executed before kGuard's runtime checks run.

In this paper, we remove this overhead, modifying kGuard to use a low-latency code inflation mechanism. Our implementation relies upon NOPs only as a means of randomizing the address space, without introducing unnecessary overhead of actually executing them. Instead, we have added an additional instruction to jump from the beginning of the NOP sled to the check. To do so, we rely on GCC's API for Register Transfer Language (RTL) to extend the original kGuard plugin.

2 Background

2.1 Memory Separation in Modern OSes

Modern commodity operating systems employ a *weak memory separation* between userspace and the kernel. This is a consequence of two features an OS typically provides: virtual memory, and a system call interface for user programs.

Virtual Memory In short, virtual memory is an abstraction of physical memory that, among other things, provides processes access to a private virtual address space. From a user program's perspective, the entire address space may be used.

In practice, it is likely that the physical memory available is far less than the amount of memory that can be addressed, much less enough memory for many processes to have that much memory on the system. Virtual memory implementations thus provide a *mapping* of virtual addresses for a process to a physical addresses in memory, such that memory pages can be stored in primary memory, or swapped to persistent storage if primary memory is low.

At a high level, virtual memory is implemented through the following mechanism. The processor keeps a cache of virtual addresses to physical addresses that it uses to translate addresses from machine code. When an address is used in an instruction, the processor will attempt to translate it, and generate a hardware *page fault* if it does not already have the translation cached. If the virtual address is valid, the kernel will provide the virtual-to-physical mapping, and the program may continue executing as normal.

System Calls In general, user programs communicate

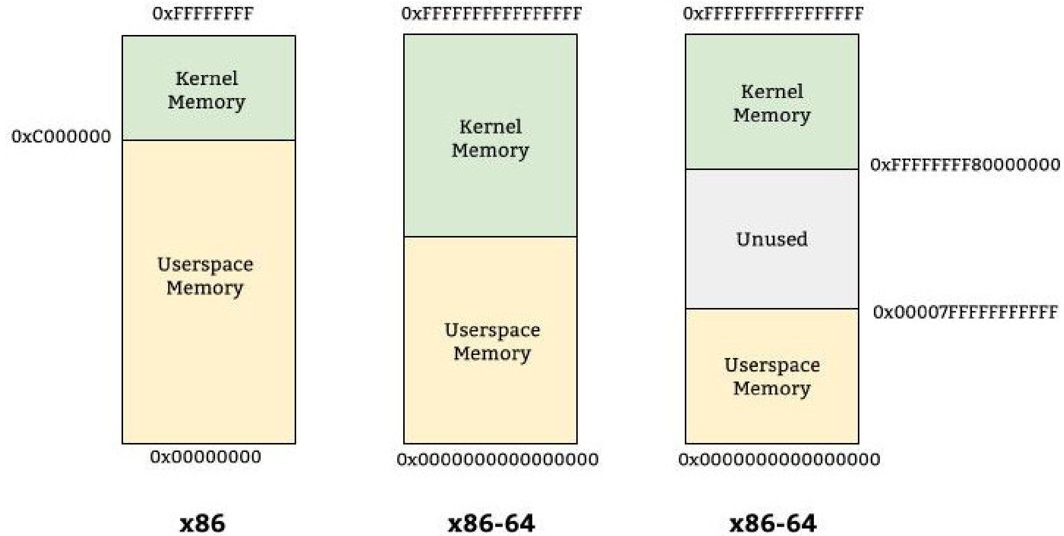

FIGURE 1: The virtual memory layout for the x86 and x86-64 architectures. Note that in x86 the kernel is allocated 1 GB of a process's virtual address space, beginning at the memory location 0xC0000000, while user space is allocated 3 GB. In x86-64, the address space is officially split in half between user and kernel memory. Kernel addresses have an uppermost bit of 1, while userspace addresses begin with 0. In practice, much of the address space is unused, as Linux only uses the lower 48 bits for addressing on AMD64. On AMD64, the next 15 bits are the same as the first — thus, all kernel addresses begin with a bit-string of 15 ones, while user addresses begin with 15 zeroes. The kernel-user memory boundary for x86-64 is thus 0xFFFFFFFF80000000: the first possible kernel address.

with the operating system kernel through system calls, perhaps to request data from various devices or to reach other parts of the system — for example, communicating with another process through pipes. In order to execute a system call, the kernel may need to access userspace memory. Consider *read(2)*, which conventionally uses a userspace buffer as an argument that the kernel fills with the requested data. In order to do so, the kernel thus needs a mechanism to write to a userspace address.

As a result, modern operating systems allow the kernel to share part of the virtual address space with each user process, as laid out in Figure 1. If a user program makes a system call, the system call code executes in privileged mode on the processor, and reads and writes from the process's address space as needed.

2.2 ret2usr

ret2usr is an exploitation technique that leverages weak memory separation to exploit the kernel. Because the entire virtual address space of the process is accessible when kernel code is running in a user process, if a vulnerability in the kernel exists, an attacker may be able to devise an exploit that causes kernel code to jump to userspace and execute code of the attacker's choice, with the privileges of the kernel.

A motivating and relatively straightforward example of how *ret2usr* might be done is through a NULL function pointer vulnerability in the kernel. Suppose that it was possible for a user program to cause a code execution path in kernel code, through normal use of system calls, such that a function pointer for some data structure was never properly initialized — and thus is NULL. Additionally, suppose the attacker could invoke a code path that executes the NULL function pointer. This is a reasonable assumption given that the data structure containing the function pointer was initialized by normal system calls as well.

Typically, a program trying to dereference a NULL pointer would crash. Virtual memory enforcements from the kernel provide this: page 0, where the "NULL" address lies, is not normally mapped into the virtual memory space. Thus, when the hardware attempts to translate this address, it causes a page fault, which the kernel deals with by killing the offending process using `SIGSEGV`: also known as a "segmentation fault."

It may seem as if this is a major roadblock to exploiting such a vulnerability. But it is trivially bypassed by using the system call *mmap*, which allows for explicitly mapping pages of the virtual address space. Prior to invoking the vulnerable code path, the attacker can *mmap* page 0, and place their shellcode there.

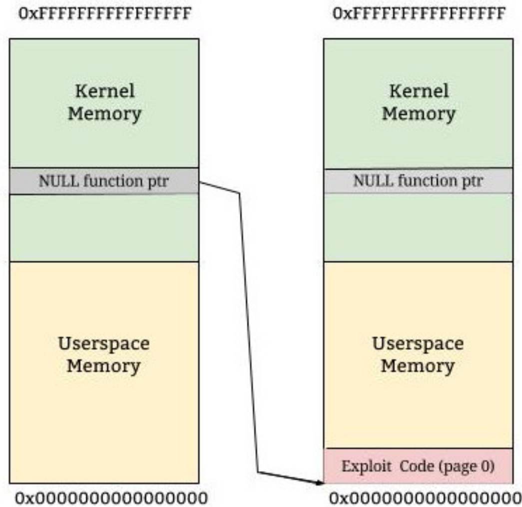

FIGURE 2: An illustration of a ret2usr attack. Here we see a NULL pointer dereferenced in kernel code, which will try to access page 0, a user page. If this page has been mapped into the address space by the attacker, then the NULL dereference will run the attacker code with full kernel privileges.

2.3 kGuard Overview

`kGuard` [3] is a compiler plugin that provides a lightweight defense against *ret2usr* attacks [2]. It relies on GCC[1], which is free and widely available, and used to compile many operating systems. In order to use kGuard, one must simply compile their kernel with it, and reboot.

kGuard defends against *ret2usr* by inserting runtime checks directly into machine code that prevent kernel code from jumping to userspace. In particular, these checks determine if an address of an indirect branch — instructions that load an address from a memory or register operand — is indeed a kernel virtual address.

At first glance, such a check may seem difficult, or at the least, potentially wreak havoc on performance, if computing whether an address is a kernel address is difficult.

The virtual memory space in the x86 and x86-64 architectures actually make this straightforward. In both architectures, user and kernel addresses are separate memory regions. To compute if an address is in kernel space, we simply check if it falls below or above the boundary between the regions. In x86, kernel addresses are greater than 0xC000000. In x86-64, they are less than 0xFFFFFFFF800000. See Figure 1 for an illustration.

[1] `https://gcc.gnu.org`

FIGURE 3: The layout of a kGuard runtime address check in memory. Note the NOP sled precedes the check instructions.

3 Low-latency Code Inflation

We implemented a performance improvement to kGuard: low-latency code inflation. In particular, we added one instruction that will prevent execution of many instructions added by kGuard to provide randomization to the address layout, but are not needed for functionality of the runtime checks.

3.1 Motivation

One mechanism of bypassing kGuard's defenses would be to detect where the checks are, and devise a mechanism to avoid them, perhaps using other kernel exploitation techniques such as Return Oriented Programming (ROP) [4]. kGuard makes bypasses more difficult using code inflation. A random-length "NOP sled" is inserted before the runtime checks — a series of no-op instructions that vary the binary's text segment layout.

A NOP sled is useful for memory space randomization, but not ideal for performance, particularly in the kernel, where performance is vital. NOP instructions are no-ops, in that they do not execute anything. Still, like any other instruction, they must be fetched from memory, decoded and executed by the processor, incurring a small amount of overhead.

Our goal for this project was to eliminate this unneeded overhead completely, while retaining the address space layout randomization code inflation provides.

3.2 Design

Because the NOP sled does not perform any functional purpose for the runtime checks, it would be ideal if we could avoid executing its instructions all together. One simple way to do this is to add another instruction before the NOP sled: a direct `jump` from the beginning of the sled to the check to the end, thus skipping unneeded instructions.

3.3 Implementation

We implemented this using the GNU Compilers Collection (GCC) Register Transfer Language (RTL) API for GCC plugins, building upon the existing kGuard implementation's use of RTL.

Register Transfer Language GCC, like other compilers, does multiple *passes* in its job of converting source code to machine code. See Figure 4 for an illustration of the GCC architecture.

The last of these passes are done in RTL, GCC's generic representation of machine code. RTL instructions roughly share a one-to-one mapping with typical machine code instructions, such as ADD or XOR [1].

To modify the instructions kGuard inserts, we used functions from the RTL API. In particular, we made use of the `emit_jump_insn_before` class of functions to cause a direct jump to be inserted and the `gen_jump` API call, which creates a jump instruction as represented in the RTL implementation. Unfortunately, the `gen_jump` API function is only available in GCC 4.7, while kGuard also works for 4.6.

3.4 kGuard v. kGuard++

To demonstrate how programs compiled with `kGuard++` differ from `kGuard`, we will show some code snippets from a sample program compiled with both.

In this example, the program was compiled for x86-64. The handler for runtime check failures located at address `0xFFFFFFFFFF40054e` and instructions are from a function, `my_foo`, located at `0xFFFFFFFFFF400527`. Note that these checks are for the x86-64 architecture, and would differ if compiled for x86.

Function Returns First, we consider a function return — indicated by the presence of a `ret` (or `retq`) instruction. The x86 calling convention dictates that when a function is to return, the address of the next instruction is located on the top of the current stack. To continue execution, the program should pop a *return address* (4 bytes or 8 bytes, for x86 and x86-64, respectively) from the stack, then jump to that address and continue executing. Return addresses are a target kGuard protects.

For the sample program compiled with kGuard, we see a return instruction as follows — note that there is a 4-NOP sled that is executed before the check begins.

```
400551: nop
400552: nop
400553: nop
400554: nop
400555: cmpq   $0x0,(%rsp)
40055a: js     400564 <my_foo+0x3d>
40055c: movq   $0x4003f0,(%rsp)
400563: retq
```

As shown in the snippet, the check for the x86-64 return does the following. First, it compares the address on the top of the stack with 0. If the address starts with a 1 — that is, it is a kernel address — the sign flag on the register will be set; otherwise, the flag will be unset.

If the sign flag is not set — as checked by the `js` instruction — then it is assumed that the address has been compromised in a ret2usr attack. The next instruction overwrites the return address on the stack with the address of the check failure handler — the default handler is `panic`. Otherwise, `js` skips the `retq` instruction, and the function returns normally.

Compiled with kGuard++ with the same-length NOP sled, we see the following:

```
400552: jmp 0x400588
400554: nop
400555: nop
400556: nop
400557: nop
400558: cmpq   $0x0,(%rsp)
40055c: js     400564 <my_foo+0x3d>
40055e: movq   $0x4003f0,(%rsp)
400563: retq
```

Using kGuard++, a direct jump is added from the beginning to the end of the sled, bypassing the NOPs, and reducing the check by 3 instructions.

The default maximum length sled for kGuard is a random value up to 16 NOPs. This means kGuard++ saves up to 15 instructions for the typical function call at a minimum.

NOP sled length is configurable by the user. A kernel compiled with a larger max length could see a greater performance improvement than 15 instructions per function call.

Function Calls

Compiled with kGuard, a function call in the example program is guarded with a check like this:

```
40053f: nop
400540: nop
400541: nop
400542: nop
400543: nop
400544: test   %rdx,%rdx
400547: js     40054e <my_foo+0x27>
400549: mov    $0x4003f0,%edx
40054e: callq  *%rdx
```

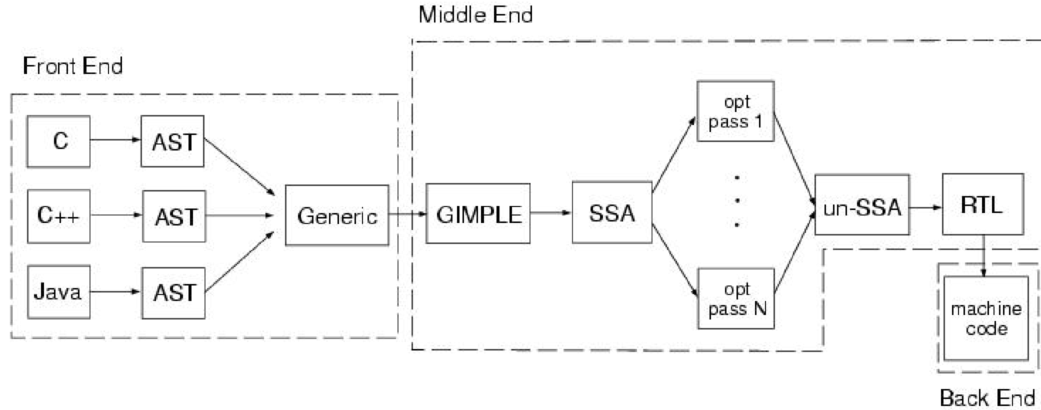

FIGURE 4: The high level architecture of GCC. RTL is the last set of passes between source code and machine code in the compilation process. [5]

Function call checks differ from function return checks. In this case, the program is loading a function pointer from a register, which could be corrupted by an attacker. The `test` instruction performs a bitwise AND on the function pointer with itself, and will set the sign flag if the result is signed — if it is a kernel address.

Similar to the function return case, if the address is valid, the instruction loading the check failure handler is skipped; otherwise, the handler overwrites the function pointer to be called. The handler will be executed instead of the corrupt function pointer, stopping the attack.

Using kGuard++, a function call runtime check is as follows:

```
40053d: jmp 0x400544
40053f: nop
400540: nop
400541: nop
400542: nop
400543: nop
400544: test   %rdx,%rdx
400547: js     40054e <my_foo+0x27>
400549: mov    $0x4003f0,%edx
40054e: callq  *%rdx
```

Again, there is direct jump to the runtime check, skipping the NOP sled.

Testing To test kGuard++, we compiled userspace programs with an alternate runtime check failure handler and inspected the binary output to ensure they properly inserted the jump. We then compiled a Linux kernel to ensure the kernel would compile and run on a hardware virtual machine.

4 Future Work

There are many opportunities to extend work on kGuard, both for improving existing code and expanding it to support new architectures.

In particular, supporting ARM would allow for the use of kGuard on Android phones, another interesting attack surface for hackers. Extending to support new architectures would involve writing new runtime checks, as well as porting the architecture-specific GCC plugin code to work with ARM as well.

Another potential performance improvement leverages the x86 and x86-64 variable-length NOPs. Using larger NOPs provides the same amount of memory space randomization as their one-byte counterparts, but without the overhead of executing as many of them.

5 Conclusion

We have proposed a low-latency code inflation solution for kGuard, a GCC plugin that provides a lightweight defense against ret2usr attacks, by inserting runtime checks for indirect branch targets at kernel compile time.

Our solution leverages one of kGuard's existing code inflation solutions, which inserts a random number of `NOP` instructions prior to the runtime checks to vary the address space. We continue to use the NOP sled to provide address randomization, but remove the overhead of executing them by inserting a direct `jmp` instruction from the beginning of the sled to the runtime check.

This proposal improves the feasibility of using kGuard as a production solution to ret2usr attacks, by improving performance without sacrificing functionality.

Acknowledgments

I would like to thank my project adviser, Professor Vasileios Kemerlis, for his patient mentorship and guidance on this project throughout the year.

References

[1] GCC DEVELOPER COMMUNITY. RTL Representation. In *GNU Compiler Collection Internals*. pp. 227–280.

[2] KEMERLIS, V. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. PhD thesis, Columbia University, 2015.

[3] KEMERLIS, V. P., PORTOKALIDIS, G., AND KEROMYTIS, A. D. kGuard: Lightweight Kernel Protection Against Return-to-user Attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 39–39.

[4] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 552–561.

[5] SMIRNOV, A. GCC front end, middle end, and back end with source file representations. `https://en.wikibooks.org/wiki/file:gcc.jpg`, Jan 2007.