December 2019

Return-Oriented Programming Detection and Prevention Utilizing a Hardware and Software Adaptation

Alex Levin

Return-Oriented Programming Detection and Prevention Utilizing a Hardware and Software Adaptation

Abstract:

This publication describes techniques aimed at detecting and preventing return-oriented programming (ROP) attacks. The techniques consist of a software adaptation which enables supplemental hardware, specifically a system on a chip (SoC), to chronologically log return (ret) addresses of pushed stack frames and compare those logged ret addresses to ret commands executed by a central processing unit (CPU) of a computing system. When the SoC determines that ret commands executed by the CPU have deviated from the logged ret addresses, then the SoC can take action to thwart a ROP attack.

Keywords:

Return-oriented programming (ROP), computing system, system on a chip (SoC), call stack, stack frame, hacking, computer security exploit, return (ret) address, attack, gadget, control flow, stack overflow

Background:

Computing systems (*e.g.*, smartphones, computers, tablets) store and process data according to executable machine instructions provided by software (*e.g.*, programs, applications operating systems (OS)), referred to as program control flow. When software on a computing system invokes a callable unit (*e.g.*, a function, a procedure, a subroutine), a processor (*e.g.*, Central Processing Unit (CPU)) in the system allocates space in its memory (*e.g.*, the random-

access memory (RAM), read-only memory (ROM)) for that callable unit to operate. This allocated space is often referred to as a stack frame.

Each stack frame corresponds to an invocation of a callable unit which has not yet terminated with a return. In other words, a stack frame is produced when a function, for example, is invoked, but has not yet executed. For instance, take the factorial of the number five as a function. In order to generate the factorial of five, the factorial of four must be computed, and so on unto the number one. For this process to occur, the factorial of five function was put on hold and the factorial of four function was invoked; this was continued unto the first executable function, specifically the factorial of the number one. The number one would be multiplied by itself, resulting in one. The preceding function, specifically the factorial of the number two, would then multiply the result by two. This method of returning to the previous function would continue unto the factorial of five function and produce a result of 120. In summary, the factorial of five function instigated the generation of many functions to produce a result. The production of multiple callable units that need dedicated space in memory (stack frames) to operate—seemingly stacking on top of each other—is referred to as a call stack. A call stack is a dynamic data-structure maintained inside the memory management unit, specifically the RAM, of the computing system by the OS. The purpose of a call stack is to control the way callable units call and pass parameters to each other.

Figure 1 illustrates an example call stack.

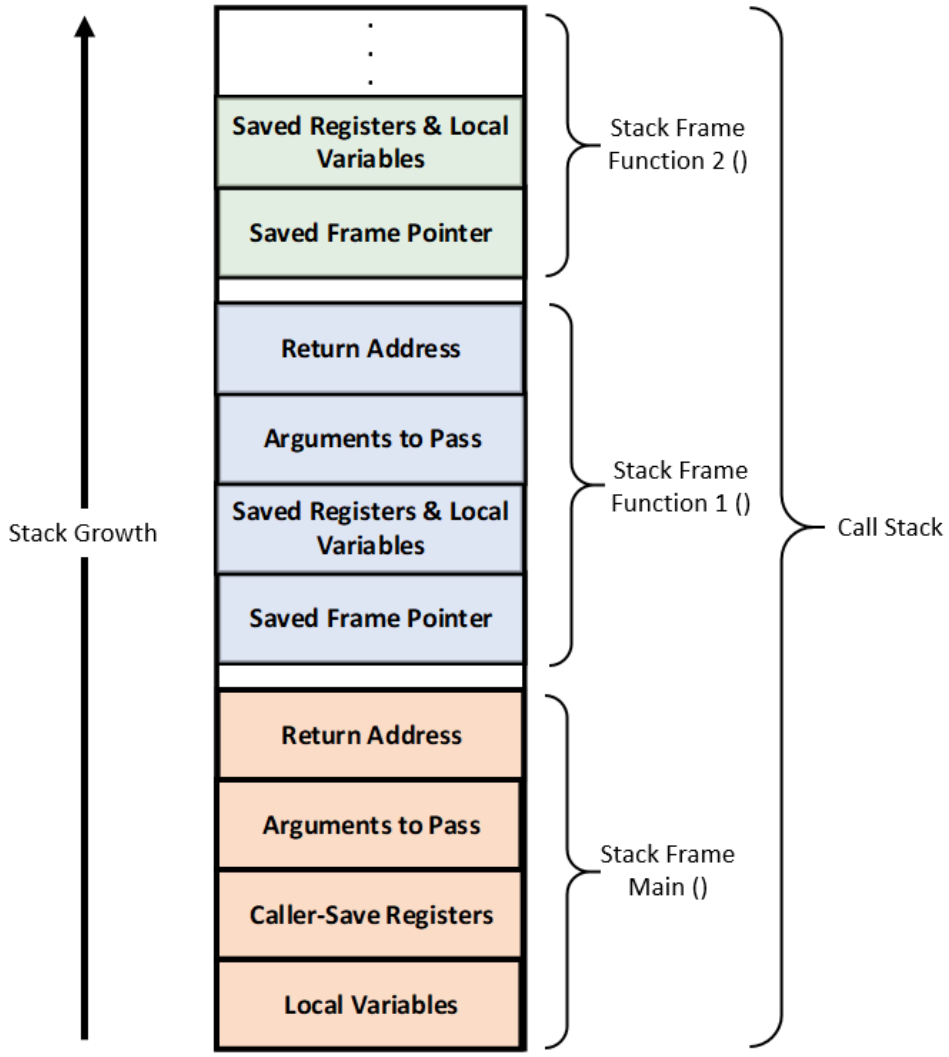

Figure 1

The call stack illustrated in Figure 1 is composed of three x86 stack frames (Main ( ), Function 1 ( ), and Function 2 ( )). A call stack can contain many stack frames, depending on the callable unit and the memory of the computing system. As illustrated, stack frames may be comprised of local variables, registers, arguments, and return (ret) addresses. Ret addresses are virtual addresses that originate from a computing system's CPU program counter. Ret addresses are later used to return back to and execute the preceding callable unit.

When a callable unit (*e.g.*, Function 1 ( )) invokes another callable unit (*e.g.*, Function 2 ( )), the parameters, local variables, etc. are pushed on the call stack in a single stack frame. When

the callable unit associated with the stack frame at the top of the call stack has fully executed, then the stack frame is eliminated (popped off) from the call stack.

Call stacks are integral to the operations of a computing system. A modern method of attack on the processes of computing systems is centered around the call stack. Return-oriented programming (ROP) attacks are a combination of stack overflow and reverse engineering existing machine instruction sequences/segments (gadgets) inside of call stacks; effectively creating new, malicious software flows that enable the attacker to execute malicious code on a computing system. Generally, these gadgets are instruction groups that end with a ret address. In short, a ROP attack would entail an attacker collecting and executing a string of reverse engineered gadgets (*e.g.*, arguments to pass).

It is desirable to detect and prevent ROP attacks. To this end, the process of comparing executed ret commands (*e.g.*, a code instruction to return to a certain address) to logged ret addresses as the stack frames were pushed onto the call stack can aid in thwarting ROP attacks.

Description:

This publication describes techniques aimed at detecting and preventing return-oriented programming (ROP) attacks. The techniques incorporate the addition of computing hardware, specifically a system on a chip (SoC), and a software adaptation to computing systems.

Figure 2 illustrates the SoC added to a computing system.

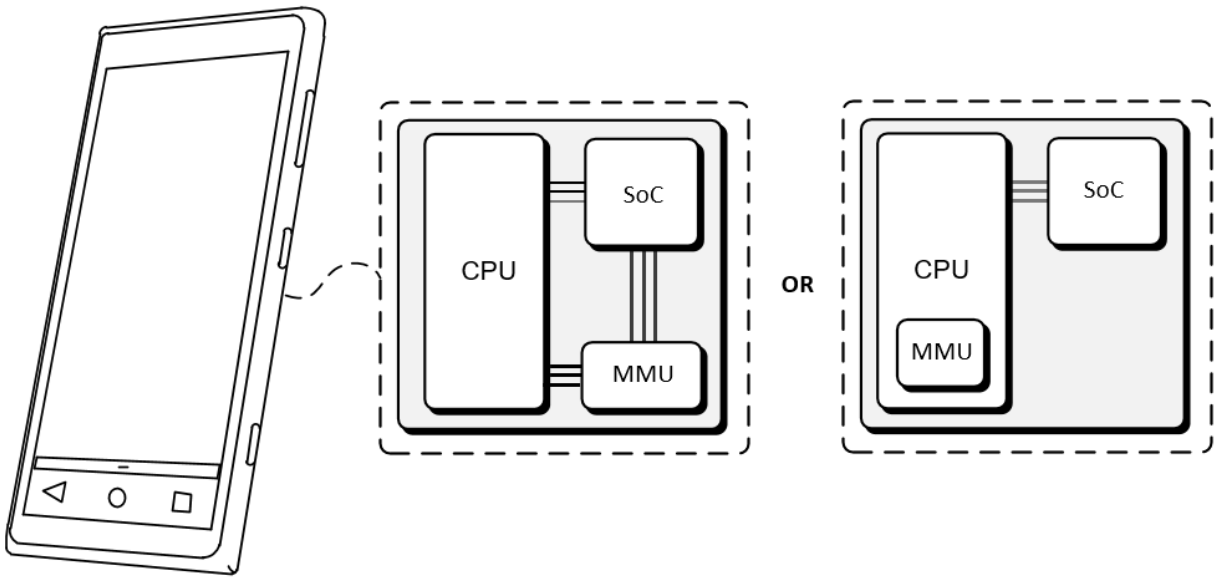

Figure 2

As illustrated in Figure 2, the computing system is a smartphone. However, other computing systems (*e.g.*, tablets, computers) can also support the techniques described in this publication. The computing system includes a native central processing unit (CPU) and a memory management unit (MMU). The MMU may be located within the native CPU or as a separate integrated chip. Regardless of location, the MMU translates addresses between the native CPU and physical memory, such as random-access memory (RAM). This translation process will be referred to herein as the Virtual to Physical addresses translation.

The computing system also includes a SoC. The SoC may include one or more processors (*e.g.*, a CPU) and a memory device such as RAM, static RAM (SRAM), dynamic RAM (DRAM), non-volatile RAM (NVRAM), read-only memory (ROM), flash memory, or the like. As illustrated, bussing connects the SoC to the CPU of the computing system. The SoC can have connectivity to the MMU either through direct or indirect bussing depending on the location of the MMU. This connectivity permits the SoC to monitor the Virtual to Physical addresses translation.

The software adaptation utilized with the SoC entails a modification to the implementation of the branch/jump command, such that when new callable units (*e.g.*, functions, procedures, subroutines) are invoked and new stack frames are pushed on top of the call stack, a ticket or a descriptor is posted to the SoC with the return address being jumped from (*i.e.*, current program counter), enabling the SoC to keep a data-structure (*e.g.*, a stack) of these addresses per process. In addition, the software adaptation permits the SoC to monitor the callable units or gadgets (existing machine instruction segments) executed by the CPU and the units' or gadgets' associated stack frame return address (ret command). In other words, with a change to the implementation to ret commands, each ret command will post the return address of the return register (*e.g.*, EBP for x86) to the SoC and the SoC can then compare the executed ret commands to the return addresses logged in the data-structure.

Figure 3 illustrates the techniques utilized to detect and prevent ROP attacks.

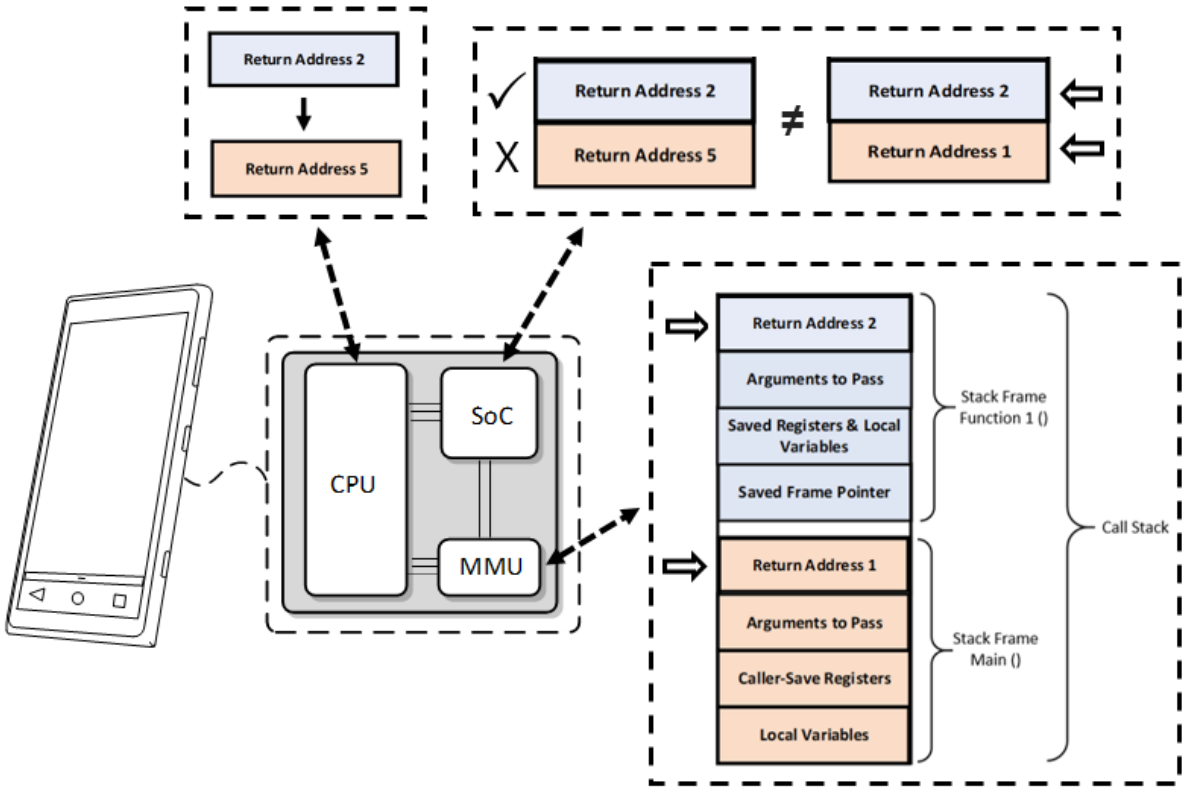

Figure 3

As illustrated in Figure 3, the MMU contains a call stack. The call stack contains two stack frames (Function 1 ( ) and Function 2 ( )), each with their respective return address: return address 1 and return address 2. For illustrative purposes, this example call stack at one time contained five stack frames. As the stack frames were pushed on top of the call stack, a ticket or descriptor with the return address of the stack frame jumped from was logged in the data-structure. As illustrated, the CPU returned to return address 2 and then returned to return address 5. In other words, the CPU executed Function 1 ( )—the stack frame with return address 2—and then executed a gadget with a return address identifying it as originating from the fifth stack frame; whereas the data-structure on the SoC indicates that the CPU should have executed the Main ( ) function immediately after Function 1 ( ). Since the SoC monitors the executions of the native CPU and compares the executed ret commands to the logged return addresses, it identified this event as a ROP attack.

These transactions can be implemented in one of two ways: a posted wait or a non-posted wait. The posted wait option avoids latency but detects a ROP attack with a slight delay. In other words, the ROP attack may execute a few gadgets, but the completion of the ROP attack is prevented. Alternatively, the non-posted wait permits the SoC to immediately check every ret command executed against the logged return addresses and thwart a ROP attack instantly after detecting deviation from the data-structure; this method, however, sacrifices computing performance. In both ways, though, ROP attacks can be successfully detected.

Additional benefits provided by these techniques can be manifest in the addition of single root input/output virtualization (SR-IOV) capabilities. For instance, adding SR-IOV capabilities to the SoC make it usable by virtual machines running on a server, allowing the same benefit to cloud-based virtual machines.

In conclusion, a software adaptation that enables a supplemental SoC to log return addresses of pushed stack frames and compare those logged return addresses to ret commands executed by the CPU of the computing system, affords computing systems a method of detecting and preventing ROP attacks.

References:

[1] Buchanan, Erik, Ryan Roemer, and Stefan Savage. "Return-Oriented Programming: Exploits Without Code Injection." August 2008. https://hovav.net/ucsd/talks/blackhat08.html.

[2] Buchanan, Erik, Ryan Roemer, Hovav Shacham, and Stefan Savage. "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC." *Proceedings of CCS 2008*, October 27-31, 2008, 1-12. https://hovav.net/ucsd/dist/sparc.pdf.

[3] Checkoway, Stephen, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-Oriented Programming without Returns." October 4-8, 2010, 1-14. https://hovav.net/ucsd/dist/noret-ccs.pdf.

[4] Shacham, Hovav. "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)." *Proceedings of ACM CCS 2007*, 2007, 1-30. https://hovav.net/ucsd/dist/geometry.pdf.