Chapter 28
Control Flow Integrity Checking

Revision 1.0, October 2025.

Objectives

- Understand the need for control flow integrity checking.
- Learn about hardware-based solutions to CFI checking
- Learn about software-based solutions to CFI checking.

28.1 Motivation

The goal of control flow integrity checking is to add another layer of defense against advanced exploitation techniques such as Return Oriented Programming (ROP). This exploitation technique is used by attackers to bypass modern security defenses like Address Space Layout Randomization (ASLR) and W⊕X described in the previous two chapters. ROP is a code-reuse attack that uses existing, legitimate code within a program to perform malicious actions.

ROP attacks try to control a program's control flow by chaining together small snippets of existing, executable instructions called *gadgets*. The attacker typically uses a memory corruption vulnerability (such as a buffer overflow) to overwrite the call stack with a sequence of addresses pointing to these gadgets. When the program reaches a return instruction, the address on the stack directs execution to the first gadget.

Each gadget usually ends with a return instruction. The return instruction pops the address of the next gadget from the stack, effectively linking them in a malicious chain The attacker finds a sequences of instructions in the code, the gadgets, when assembled in order, produces the goal of the attack. Since each gadget ends in a return instruction, it transfers control the next gadget. By carefully selecting and ordering gadgets, an attacker can perform arbitrary operations or execute a malicious payload without injecting new code into the system.

ROP a are effective because all executed code is from existing, trusted, and executable memory regions, making it difficult for traditional defenses to detect the malicious intent.

Figure 1 shows a ROP attack in progress. The attacker, perhaps through a buffer overflow, has managed to push a sequence of code addressed onto the stack. When the program reaches the next return instruction, it will use the address on the top of the stack as its return address, jumping to the first gadget selected by the attacker. That gadget is a sequence of carefully

selected instructions in the code. Importantly, the last instruction in any gadget is a return instruction, which will cause execution to jump to the next address on the stack, i.e., to the next gadget. Researchers have developed ways to help find useful gadget sequences in the code of the program being attacked, making useful gadget selection feasible.

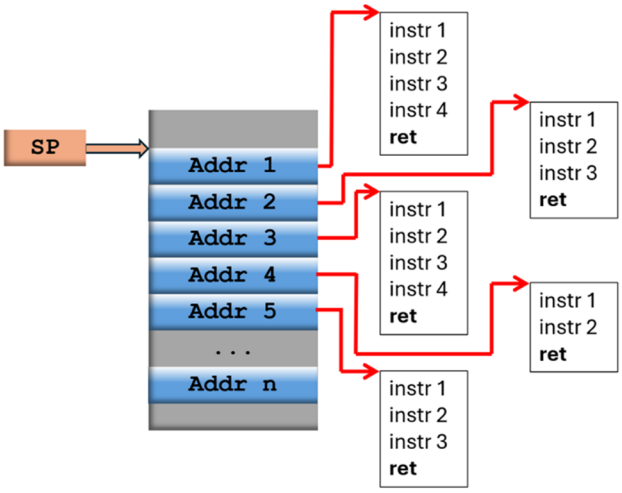

Figure 1: Diagram of ROP Attack
The stack points to a group of gadgets, each of which is terminated by a return instruction.

As result of ROP attacks, and other similar types of control flow attacks such as JOP (Jump Oriented Programming), both CPU designers and software developers have developed techniques to make these attacks more difficult.

28.2 Hardware Solutions

The designers of modern CPUs have been actively developing techniques to try to prevent control flow attacks.

28.2.1 Shadow Stacks

A shadow stack is a second stack used exclusively for control transfer operations to store and to retrieve the return address pointers. The shadow stack is distinct from the ordinary data stack, and holds no data, only return addresses. There is a separate control stack pointer register (CSP) that cannot be used in normal move or arithmetic instructions. This makes it extremely difficult to find a gadget that includes an instruction that modifies the CSP.

When shadow stacks are enabled, a call instruction will push the return address onto the regular stack, as normal. It will also push the return address onto the shadow stack.

The return instruction pops the return address from both the shadow stack

and the regular stack. If the return address values popped from the two stacks are not equal then the processor causes a control flow exception.

Shadow stacks are found in Intel processors as part of their Control-flow Enforcement Technology (CET). The RISC-V processor also implements shadow stacks in their Zicfiss instruction set extension. In addition to the second stack, it defines new memory page protections for the shadow stack such that only explicit shadow stack instructions can access the memory.

The ARM's implementation of shadow stacks is called *guarded control stacks* and works quite similarly to the RISC-V implementation.

IBM took a somewhat different approach starting with the Power10 processor. When a function is called, the return address is cryptographically hashed, and this hash is saved in memory (not in a separate shadow stack). Just before the function returns, the processor pops the return address from the stack and recomputes the hash. This recomputed hash is compared to the hash value stored in memory. If these values do not match, then the return is flagged as a security violation, generating an exception.

28.2.2 Landing Pad Instructions

Some control flow attacks, such as JOP, are based on causing indirect jump instructions to go to an address of the attacker's choosing. An indirect jump instruction is one where the destination address of the jump is in a register. If the attacker can control the control the contents of the register, they can redirect the program's execution without modifying any of the code.

Landing pad instructions are special instructions that have no execution purpose other than to mark the destination of a jump. When the compiler generates code for the program, any location in the code that potentially will be reached by an indirect jump must be ones of these special landing pad instructions. If an indirect branch is directed to a non-landing pad instruction, then the processor causes a control flow exception.

Intel introduced the `ENDBR32` and `ENDBR64` instructions to mark landing pads. Landing pad checking can be disabled for individual jump instructions by adding the `NO-TRACK` prefix to that instruction.

RISC-V introduced the `LPAD` landing pad instructions as part of CET. This instruction is implemented in their Zicfilp instruction set extension.

ARM implements landing pads as part of their Branch Target Identification. Indirect jumps must arrive at a `BTI_J` instruction. Additionally, indirect call instructions must arrive at a `BTI_C` instruction.

28.3 Software Solutions

As we saw in the previous two chapters, compiler writers have been busy developing techniques that will make the code that their compilers generate more resistant to attack. Here, we describe two such efforts based on trying to prevent control flow attacks.

28.3.1 Microsoft Control Flow Guard

Microsoft introduced Control Flow Guard (CFG) in Windows 8 to provide software protection against control flow attacks. The compiler identifies valid function entry points and builds a bitmap of memory with a "1" bit for addresses that are the start of a valid function. The compiler also inserts a call to a special function, `_guard_check_icall`, before each indirect function call. This special function will check that the call address refers to a memory location with a "1" in the bitmap. If this is true, then the function returns; otherwise it terminates the program.

This feature is enabled with the `/guard:cf` compiler option in Visual Studio.

28.4 LLVM/Clang CFI Option

The Clang C and C++ compilers implement a variety of control flow checks that happen at runtime. These compiler can check for such things as:

Indirect call instruction that is calling a function of the wrong type: This can happen if there was an attack that tried to direct the call instruction to a different location than intended in the code.

Method calls based on a modified virtual function table pointer: For C++ programs, method calls in classes that go through an object's virtual function table. Calls to such functions are based on a pointer the virtual function table, the vptr. If the vptr was modified, then the control flow could be redirected. The compiler inserts checks that the type of the call destination match the type of the method call.

These control flow checks, and a variety of other related ones, are enabled by the `-fsanitize=cfi` compiler option. Note that systems such as Android have been using this feature for many years to improve their security.

28.5 Summary

There is a constant battle between attackers and defenders. In response, compiler writers and CPU designers have been trying to prevent such attacks. The goal is to provide automated techniques that allow code to be more secure without requiring the programmer to do anything special. In addition, these techniques need to have low enough runtime overhead so that

programmers will not disable them. The challenge for the defenders is to try to anticipate and stay ahead of the attackers.

28.6 Exercises

1. Write a C program that uses a function pointer and compile it with Clang. Modify the program so that you assign to that pointer the address of a function of the wrong type. Compile the program with and without the `-fsanitize=cfi` option. When you run the two versions of the program, what difference do you see in its behavior?