CS5431 – Computer Security Practicum Spring 2017

Lecture 2: Control Flow Integrity

February 3, 2017 *Instructor: Eleanor Birrell*

1 Control Flow Hijacking

Last week we talked about how attackers could use overflow-vulnerable buffers on the stack or on the heap to execute exploit code. We also discussed several countermeasures: canaries, address space layout randomization (ASLR), and $W \oplus X$ pages. We discussed how advanced attackers could bypass canaries and/or ASLR defenses. And finally, we concluded by introducing return-into-`libc`. Recall that return-into-`libc` attacks work by overwriting the stack with a sequence of fake stack frames that appear to return into functions in the standard library `libc`.

Return-into-`libc` can successfully target systems that deploy $W \oplus X$ countermeasures because unlike traditional stack- or heap-smashing attacks, return-into-`libc` attacks work by running code that was already in the system rather than by writing new code into an overflow buffer. A successful return-into-`libc` attack is therefore an example of *control-flow hijacking*; it achieves its goals by gaining control of the control flow of a system and then executing existing executable code in a different order than the system's owner intended. Today we will introduce *return-oriented programming*—another class of control flow hijacking attacks—and we will discuss countermeasures that defend against control flow hijacking.

2 Return-oriented programming

Like return-into-`libc` attacks, return-oriented programming works by gaining control of the control flow of a program and causing it to execute a sequence of carefully selected program segments that implement the desired exploit functionality. Unlike in the attacks we've looked at earlier, however, the code segments used in return-oriented programming are not complete functions. Instead, return-oriented programming constructs exploits from short code segments or *gadgets* that are usually just two or three instructions long.

2.1 Gadgets

Recall that the x86 architecture uses variable-length instructions, and that these instructions are not necessarily word-aligned. Moreover, the x86 ISA is extremely dense, so a random byte string will often be interpreted as a valid sequence of instructions. By starting at a different location, it is therefore possible to interpret the same sequence of bytes in multiple different ways.

Consider, the following example, drawn from one implementation of the standard `libc` library. The two instructions that appear at the entrypoint `ecb_crypt` are encoded as follows:

```
f7 c7 07 00 00 00     test $0x00000007, %edi
0f 95 45 c3           setnzb -61(%ebp)
```

Starting one byte later, the sequence is instead interpreted as:

```
c7 07 00 00 00 0f     movl $0x0f0000000, (%edi)
95                    xchg %ebp, %eax
45                    inc %ebp
c3                    ret
```

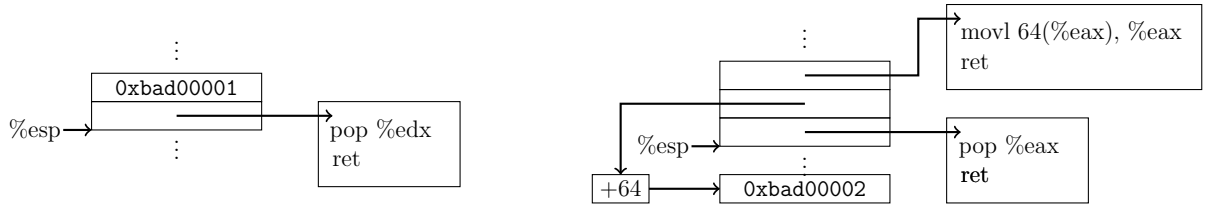
In its simplest form, any sequence of bytes that ends in `c3` could potentially be useful to an attacker. In any substantial piece of code (e.g., in `libc`), it is therefore likely that an attacker can find a sequence of gadgets that will implement the desired exploit.

The set of possibly-useful gadgets in a particular binary can be efficiently discovered through static analysis. Briefly, the analysis constructs a prefix tree by recursively searching the binary backwards from bytes that can be interpreted as a `ret` instruction; if a sequence of bytes can be interpreted as a valid instruction, it is added to the prefix tree. One analysis of an implementation of the `libc` library yielded a prefix tree with 15,121 nodes; the set of gadgets discovered formed a Turing complete language.

2.2 Programming with Gadgets

All gadgets expect to be entered the same way: the processor executes a `ret` instruction when the stack point points to the bottom of the gadget. Since all gadgets end with a `ret` instruction, gadgets can be strung together by placing one on top of another on the stack; the first gadget is placed so that its bottom word overwrites a return address pointer, trigger the exploit.

Lets consider a simple example gadget: loading a constant value into a register can be accomplished simply by the sequence `pop %edx; ret`. This example is given in Figure 1a: the `ret` instruction that enters the gadget will cause the gadget's address to be poped off the stack and the gadget to execute; the pop instruction in the gadget will cause the constant value `0xbad0001` to be popped off the stack and stored in the register `%edx`, then the ret instruction will cause the processor to proceed.



(a) Load constant gadget

(b) Load from memory gadget

Figure 1: Example gadgets for loading values

In the case of the load from memory gadget (shown in Figure 1b), the value loaded from memory into a register (now register`%eax`) is instead the memory location 64 bytes lower than the location of the value the attacker wants to load from memory. When the first (load constant) gadget returns, it will return into the second gadget, which will copy the value offset from `%eax` by 64 bytes (`0xbad00002`) into the register `%eax` and then return into the next gadget.

The gadgets for loading values are relatively simple, but gadgets for storing values, implementing arithmetic, and implementing control flow (e.g., conditional jumps) can be constructed in a similar manner from a sequence of smaller gadgets.

A successful return-oriented programming attack is implemented by overwriting the stack with the locations of a series of gadgets. The first gadget overwrites the return address pointer of the target frame stack; when the target function returns, it will trigger the sequence of gadgets that implement the exploit code. An example exploit that opens a shell is shown in Figure 2.

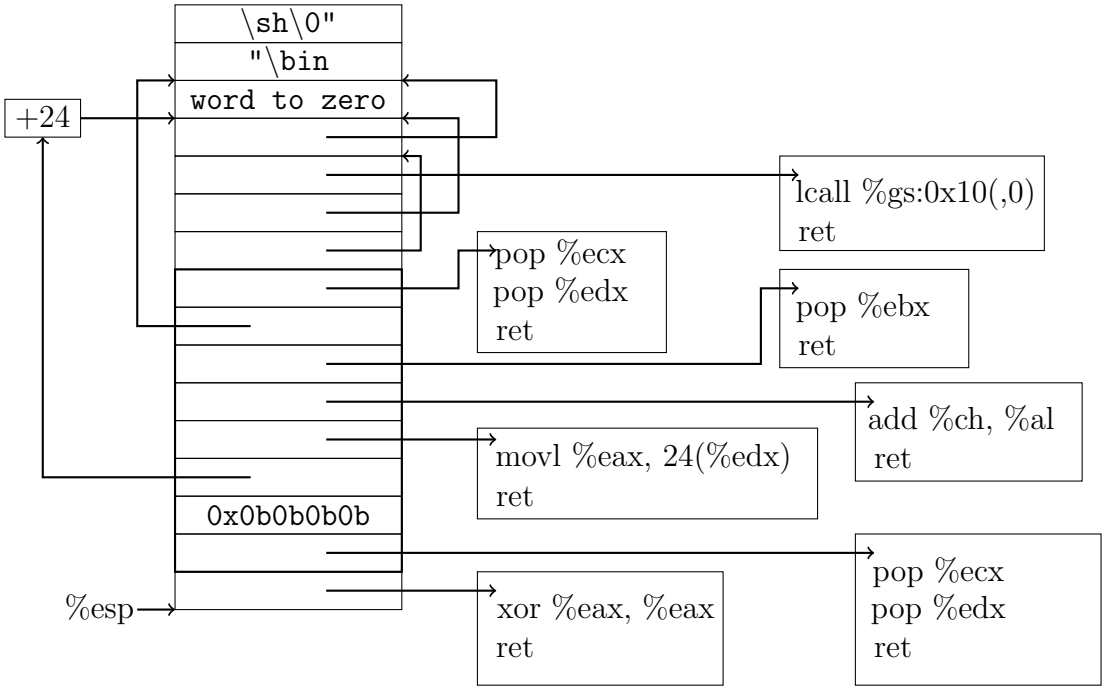


Figure 2: Return oriented shellcode.

In the example exploit invokes the `execve` system call to open a shell. This is achieved by (1) setting the system call index in register `%eax` to `0x0b` by first setting it to zero (word 1) and then updating the last byte (word 6), (2) setting the path-to-run in register `%ebx` to the string "\bin\sh" using the pop instruction in word 7, (3) setting the argument vector `argv` in register `%ecx` to an array of two pointers—the first of which points to "\bin\sh" and the second of which is null—by using the pop instruction in

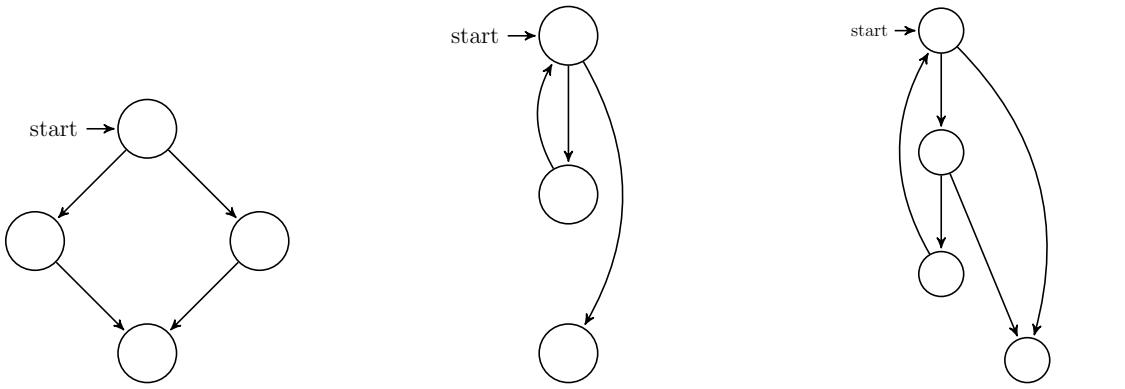
word 9 after setting the second pointer to null (zero) in word 5, and (4) setting the environment vector `envp` stored in register `%edx` to a length-one array—containing a single null pointer—using the second pop instruction in word 9, again after setting the same pointer to null in word 5. Finally, the shellcode traps to the kernel in word 12.

3 Control Flow Integrity

Control Flow Integrity is a general approach to mitigating all control flow hijacking attacks, including both return-to-`libc` attacks and return-oriented programming. The key observation is that programs have an intended control flow–that is, the programmer intended for the program to execute one of a few particular sequences of code. For example, a programmer who writes a loop is intending to allow the code to run through the loop some number of times; the programmer is not intending the code to jump from one iteration of the loop to a function that a shell. If the set of intended patterns can be concisely defined ahead of time, then all control jumps can be checked against intended behavior and control flow hijacking attempts can be detected.

3.1 Control Flow Graphs

Control flow integrity is implemented by constructing a *control flow graph* for the program. A control-flow graph is a directed graph in which each node corresponds to a straight-line code segment without any jumps. There edges represent intended jumps in the control flow. Example control flow graphs for standard programming constructs are given in Figure 3.



(a) If-then-else statement (b) A while loop (c) A loop with two exits

Figure 3: Example control flow graphs

The control flow graph for a program can be defined by statically analyzing a program binary, by execution profiling, or by construction at compile time.

3.2 Enforcing Control Flow Integrity

At a high level, control flow integrity enforces that when ever an instruction transfers control (e.g., calls a sub-procedure or returns from a function), it must target a valid destination in the CFG. In most cases, the control-transfer instruction targets a constant destination, so the validity check can be added statically by modifying the program binary. In other cases, the valid destination is determined at runtime (and thus the validity must be checked) at runtime.

The effectiveness of a control flow integrity enforcement mechanism depends on the precision of the control graph. However, large graphs tend to impose unacceptable performance overheads. Existing implementations of control flow integrity must choose a balance between these two competing constraints. The original CFI system, for example, assumes that if the control flow graph contains edges to two destinations from a common source, then the destinations are equivalent. This assumption optimizes performance overhead, but it is not always true; unintended control flow jumps may be permitted due to this approximation. In fact, practical code-reuse attacks have been demonstrated against this approximate implementation of control flow integrity.

Control Flow Guard. First introduced in Windows 8.1, Control Flow Guard is perhaps the most broadly deployed implementation of control flow integrity on the market today. When Control Flow Guard is enabled by the compiler, it injects target address checks before every indirect call during program compilation. These checks trigger a function called ntdll!LdrpValidateUserCallTarget—located at a particular location defined by the compiler—which implements the target address check.

Control Flow Guard implements control flow integrity by storing a bitmap of valid function start addresses, at the granularity of 8 bytes. A jump into a function (e.g., returning control flow to the return address pointer on the stack) is only permitted if the location is a valid function start location in the bitmap. Control Flow Guard implements an approximation of full control flow integrity—by allowing jumps to any permitted function start point and by considering locations at 8 byte granularity—in order to reduce the overhead of full precision CFI. However, this imprecision has permitted some successful attacks to bypass this defensive measure.

In a particularly memorable example of defensive failures, the initial version of Control Flow Guard was quickly discovered to have a severe vulnerability: the location of the guard function was written in read-only memory, however, attackers discovered they were able to make the read-only memory writable and then overwrite the address check function with a trivial function that accepts all addresses. They were then able to successfully implement a standard control flow hijacking attack targeting a vulnerable buffer despite the defense. This vulnerability was quickly patched, and it is unknown whether or to what extent additional vulnerabilities introduced by the approximate nature of Control Flow Guard remain exploitable.