CS181S – System Security Fall 2018

Lecture 2: Vulnerabilities

September 10, 2018 *Instructor: Eleanor Birrell*

1 Traditional Stack Smashing

A buffer overflow occurs when the data written to a buffer is longer than the space allocated to that buffer. Depending on the location of the allocated buffer and the length of the overflow, additional data written to a buffer might overwrite other data, code, or return addresses. In most cases, an accidental buffer overflow will result in incorrect execution (e.g., when it overwrites other data values), a program crash (e.g., when it overwrites values the target program doesn't have permission to access), or no effect (e.g., when it overwrites values that are not accessed after that point). Malicious buffer overflows, however, can exploit buffer vulnerabilities to force the system to run exploit code. The most common form of buffer overflow attack—often called *stack smashing*—the attacker overwriting the return address pointer with a pointer to the exploit code.

1.1 Program Stacks

Before we can look at stack smashing in detail, we need to remember some of the details about how stacks are implemented. When programs are executed, the operating system stores relevant state in memory. Executable code is stored at one end of memory, static data is stored next to code, and the stack and the heap occupy the remaining memory.

The heap is used for dynamically allocated memory (ie, any time malloc is used). The stack is used primarily for managing nested calls to procedures; it also stores local variables.

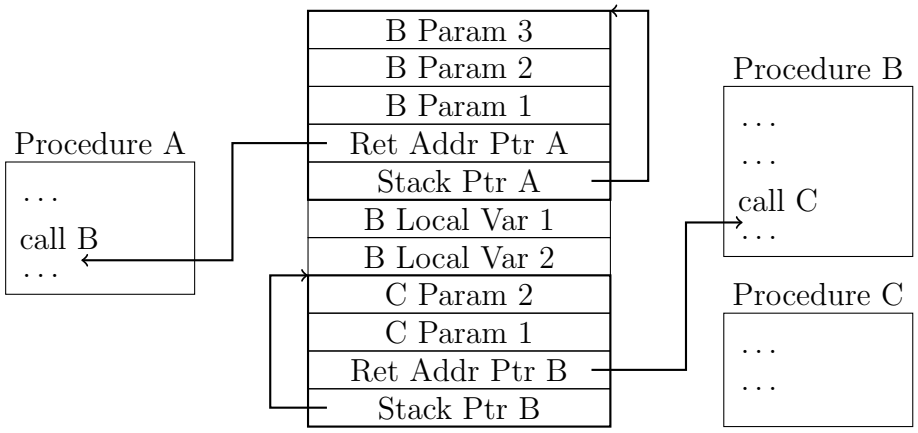


Figure 1: The stack configuration after nested procedure calls

When a procedure A calls a procedure B, A pushes a *stack frame* onto the stack; a stack frame is comprised of A's return address (the current value of the program counter), a pointer to the end of A's section of the stack (used to recover in case B has an error), as well as the parameters for B. Local variables created by B (possibly including buffers) are pushed onto the stack on top of the previous stack frame, and the process iterates when B calls its own subroutines.

1.2 Stack Smashing Attacks

In most early examples, the malicious code was written earlier in the same buffer, and the pointer was overwritten to point back to the beginning of the buffer. Note that this requires careful programming to ensure that the exploit code fits in the available space; exploit code is often written directly in assembly code to conserve space. An example is show in Figure 4.

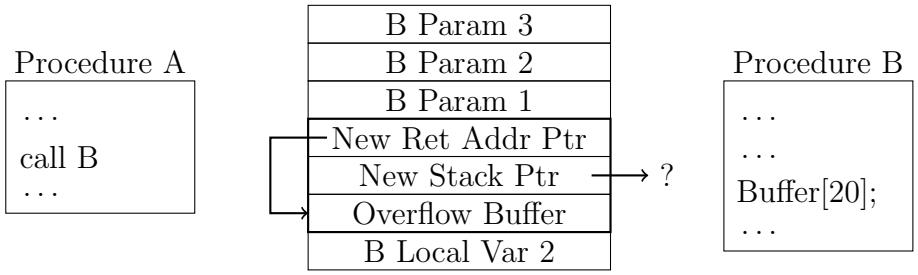


Figure 2: Stack configuration after a traditional stack smashing attack.

1.3 Countermeasure: Stack Canaries

Buffer overflows have been a known problem for a long time. The Morris worm leveraged a buffer overflow vulnerability to spread across the ARPANET in 1988. Consequently, several differently countermeasures have been developed to defend against such exploits.

In 1998, a researcher named Crispin Cowan observed that this attack pattern could be leveraged to *defend* against buffer overflow attacks. His system, called StackGuard, placed a special value called *canary value*[1] immediately below the return address pointer. When a procedure returned, the operating system would check the integrity of the canary value; if it had been modified, the system would signal an error.

Problem solved? Not quite. The canary value needs to be something that the operating system can efficiently verify. However, if the system uses something simple—say the constant value 0xf213ea08—then the defense will work (usually) until the attacker

[1]Canary values are named after the canaries traditionally carried by miners. Canaries need more oxygen than humans, so if the oxygen level in the mine dropped, the canary would die, notifying the miners in time for them to safely evacuate.

learns about the defense.[2] After learning about the defense, an attacker will modify the exploit to bypass the defense by, say, alternating the value of the malicious address with the canary value. Stackguard employed two alternative methods to prevent an attacker from successfully forging the stack canary. The first method was a stack canary comprised of the common termination symbols for C string libraries: `\0`, `CR`, `LF`, and `EOF`. An attacker couldn't use common C library functions to embed these characters in an overflow buffer because those functions would terminate when they reached their termination symbol. The second method was 32-bit random number chosen fresh each time the program was run. Since the canary value is chosen fresh each time the program is invoked and is never disclosed to anyone, an attacker is unlikely to guess the correct value to forge. Other types of canaries have also been proposed.

So do stack canaries work? They have low overhead, and they do mitigate buffer overflow attacks by making such attacks harder to successfully execute. However, a sufficiently skilled and determined adversary will often be able to bypass a stack canary.

2 Traditional Heap Smashing

In early days, defenders focused their efforts on security stack-based buffer overflows. Countermeasures like stack canaries focused exclusively on protecting the integrity of the stack. Overflows that occurred on the heap were assumed not to be exploitable. However, this assumption turned out to be inaccurate.

2.1 Heaps

Heaps contain dynamically allocated memory (e.g., memory returned by `malloc`). Ever memory allocation a program makes is represented by a data structure called a *chunk*. A chunk consists of (1) metadata and (2) the memory returned to the program. Chunks are saved to the heap. The chunk metadata structure contains the following fields:

```
INTERNAL_SIZE_T prev_size;    /* size of prev chunk (if free) */
INTERNAL_SIZE_T size;         /* size of chunk  */

struct chunk * fd;            /* double links -- used only if free */
struct chunk * bw;
```

`size` stores the size of the current chunk, in bytes. Since chunks are always 8-byte aligned, the last three bits are redundant and are actually repurposed; the first (least significant) bit is used to indicate whether the previous chunk is currently allocated. `prev_size` stores the size of the previous chunk, if the previous chunk is currently free.

[2]Relying on your defensive techniques remaining a secret a system is often called *security through obscurity*. It is generally regarded by the security community as ineffective; historically, attackers eventually find out how a system is secured, and defenses that rely on the attacker not knowing the defense strategy have repeatedly been compromised.

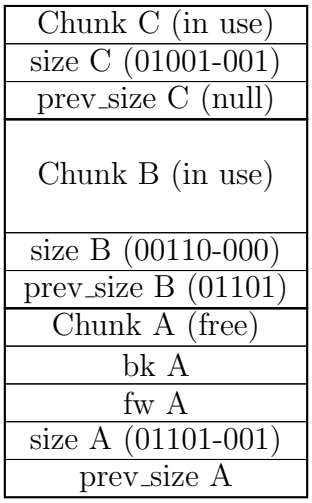


Figure 3: Example heap configuration showing one free chunk and two in-use chunks.

Free chunks are stored by size in doubly linked lists using the pointer fields `fw` and `bk`. When a chunk is freed, it checks whether the chunk in front of it is already free (by checking the least significant bit of its `size` field). If so, it merges the two chunks and move the combined chunk to the doubly-linked list for free chunks of the new (combined) size. If not, it simply adds itself to the doubly-linked list for free chunks of its own size.

Meta-data for in-use chunks does not contain the pointer fields `fw` and `bk`; the memory returned to the program starts where `fw` was stored prior to allocation.

2.2 Heap Smashing Attacks

The key observation that enables heap smashing attacks is that removing an element from a doubly-linked list involves overwriting memory locations (supposedly the `fw` and `bk` pointers of the adjacent chunks) with new values (supposedly linking those two chunks together).

A successful heap smashing attack proceeds by (1) writing a fake `fw` pointer (pointing to a targeted function pointer) to the beginning of buffer, (2) writing a fake `bk` pointer (pointing at the next memory addresss, soon to contain exploit code) to the second address of the buffer, (3) writing the exploit code starting from the third address of the buffer, (4) overwritting `prev_size` field of the next chunk to contain the size of the current chunk, and (5) overwriting the `size`field of the next chunk to indicate that the target chunk is free. When the next chunk is freed, the memory management code will (incorrectly) observe that the previous chunk is already free and will move the merged two chunks to the appropriate doubly-linked list of free chunks; the unlinking code will copy the value of `bk` (now the location of the exploit code) to the location indicated by `fw` (now the target function pointer). When the function pointer is subsequently

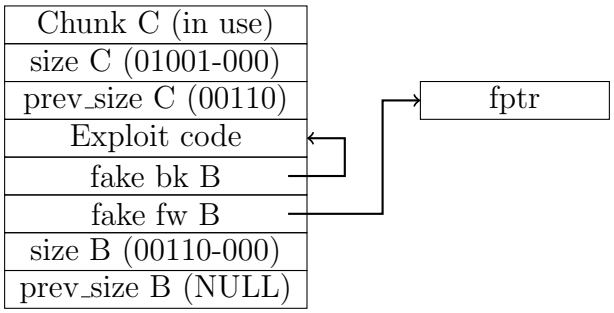


Figure 4: Heap configuration after a buffer overflow attack (before Chunk B is freed).

dereferenced, the exploit code will be run.

2.3 Countermeasure: Memory Tagging

In a tagged architecture, every machine word has one or more bits that encode the access permissions for that word. These *access bits* can be set only by priviledged (OS) instructions. The bits are tested every time the word is accessed. Attempts to access a word without appropriate permissions result in an error. Access bits can also distinguish types of access (read, write, execute) or classes of data (numeric, character, address, or pointer).

A tagged architecture could effectively mitigate buffer overflow attacks by designating return address pointers as privileged words and/or as pointer words. However, tagged architectures are not generally compatible with legacy code. And most current operating systems (including Windows, Mac OS and most Linux flavors) include legacy code dating back twenty years or more. Tagged architectures have been deployed in some systems (page-level tagging is also now available on most processors), and more are under development by major vendors like Intel, but the lack of code compatibility has precluded widespread use.

A software analog of hardware tagging is Write or Execute only ($W \oplus X$) pages, sometimes called *executable space protection*. Under this approach memory is tagged in software—at the granularity of a page—as either writable or executable, but not both. Executable space protection is widely deployed; it has shipped with Windows since XP SP2, with OX X since Leopard (10.5), and is available on most Linux flavors. When this defense is enabled, an attacker will be unable to execute code written to writable pages (e.g., code written inside the overflowed buffer) thereby nullifying traditional buffer overflow attacks targeting either the heap or the stack. However, some applications (e.g., Javascript, Flash) rely on an executable stack. Also, sophisticated attackers can sometimes trigger a memory mapping routine that marks the attack code as executable, bypassing executable space protections.

3 Code-reuse Attacks

Code-reuse attacks are a class of advanced stack smashing techniques that bypasses the protection offered by executable space protection. At a high level, instead of altering the return address pointer to point to code that has just been written on the stack, code-reuse attacks overwrite the pointer to point to code that already resides on the target system, either functions in the target program or functions in loaded libraries.

Code-reuse attacks work by overwriting the return address pointer to point to the location of the appropriate code in memory, overwriting the stack addresses beyond the new return address pointer with a fake stack frame for the new function, and overwriting the stack pointer to point to the beginning of the fake stack frame. Complex exploits can be constructed by chaining together available functions. An example code-reuse attack is depicted in Figure 7.

3.1 Return-into-`libc`

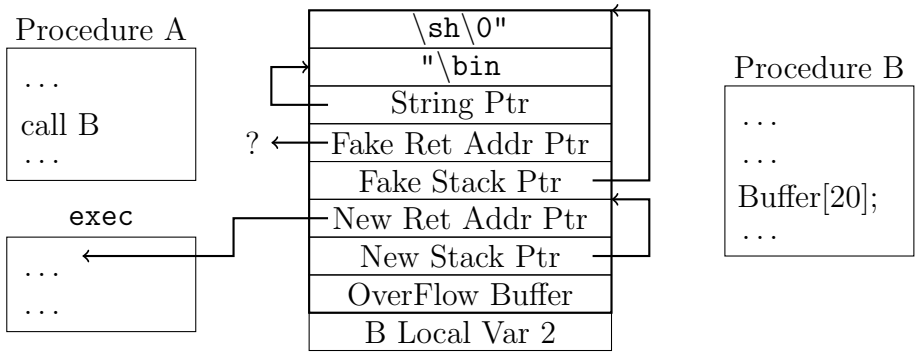


Figure 5: Stack configuration after a return-into-`libc` attack.

A common class of code-reuse attacks—known as *return-into-*`libc` attacks—targets code in the standard c library. For example, in the example shown in Figure 7, the attacker executes the function `exec("/bin/sh")`. This approach is powerful because `libc` includes the system call API and because it is loaded into every Unix program. In fact, the functions in `libc` form a Turing complete programming language.

3.2 Countermeasure: Address-Space Layout Randomization

The key observation behind address-space layout randomization (ASLR) is that many buffer overflow attacks rely on knowing the location in memory of the exploit code the attacker wishes to use. ASLR renders this difficult by randomizing the memory layout: the base addresses of the stack, heap, code, and memory mapped segments are random-

ized at load and link time. This ensures that hardcoded addresses are unlikely to point to the desired code when the attack targets a particular system.

This approach was initially highly-effective against many forms of buffer overflow attacks, although attacks that rely exclusively on relative addresses continued to be effective. ASLR was widely deployed in both Linux and OpenBSD. Derandomization techniques (Windows Vista, for example, only used 8 heap and 14 stack bits of randomness) have since eroded the effectiveness of ASLR on 32-bit machines. Derandomization attacks on 64 bit machines take several minutes and are thus often detectable, but can still be successfully executed in some contexts.

3.3 Countermeasure: Language Support

High level languages are compiled into machine code before they are executed. During this phase, the compiler has the option to introduce additional checks. For example, when presented with an array access, the compiler could introduce bounds checks, logically replacing loop (a) with loop (b):

```
int a[20];
for(int i=0; i<max; i++){
    a[i]=0;
}
```

(a) Loop without bounds checks

```
int a[20];
for(int i=0;i<max;i++){
    if(i<0) signal error;
    if(i>=20) signal error;
    a[i]=0;
}
```

(b) Loop with bound checks

Compilers can also execute *type checks*, ensuring that the data assigned to a location has the appropriate type for that location

Language support effectively eliminates vulnerabilities like buffer overflows. However, much legacy code was written in lower level languages without such checks, so vulnerabilities remain. Introducing such checks also imposes a performance cost, so programmers continue to write code—and produce vulnerabilities—in non-safe languages.

4 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.

4.1 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:

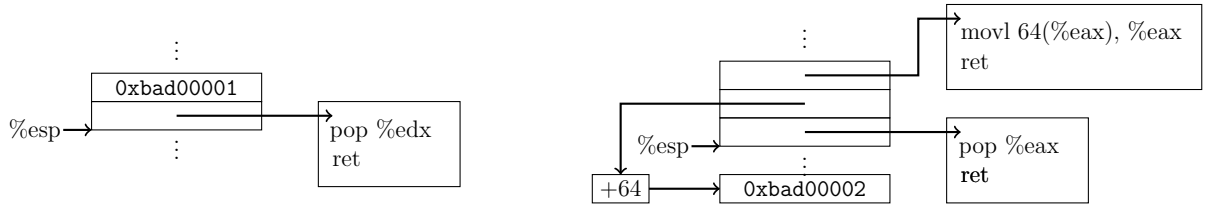
```
f7 c7 07 00 00 00     test $0x00000007, %edi
0f 95 45 c3           setnzb -61(%ebp)
```

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.



(c) Load constant gadget (d) Load from memory gadget

Figure 6: Example gadgets for loading values

4.2 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

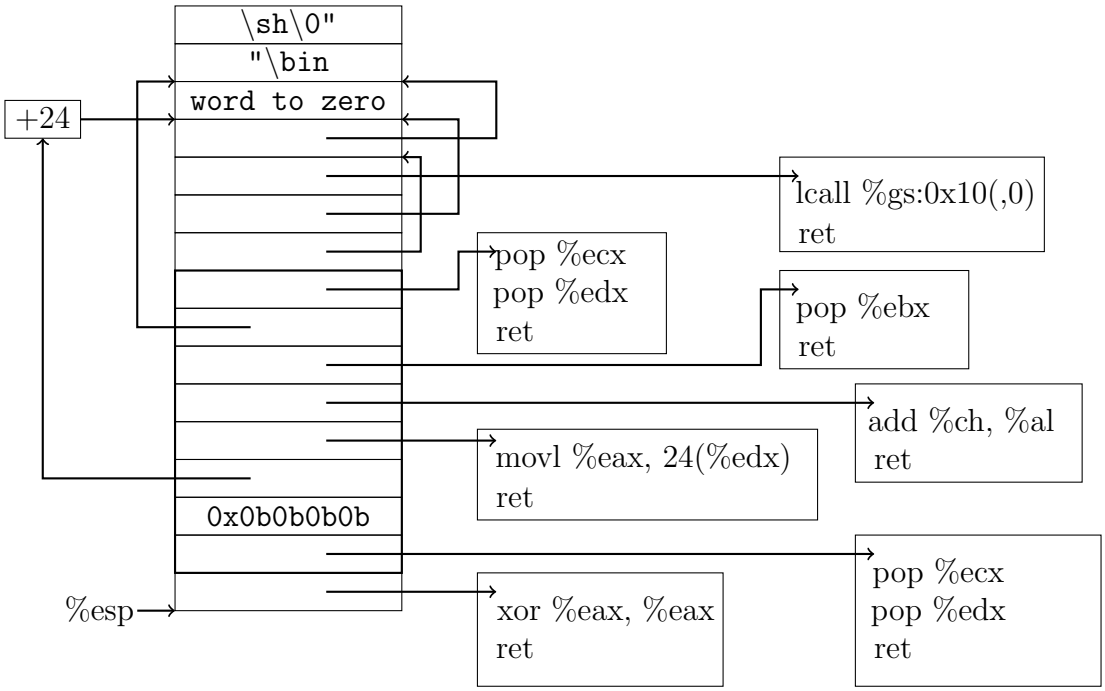


Figure 7: Return oriented shellcode.

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.

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.

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.

5 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.

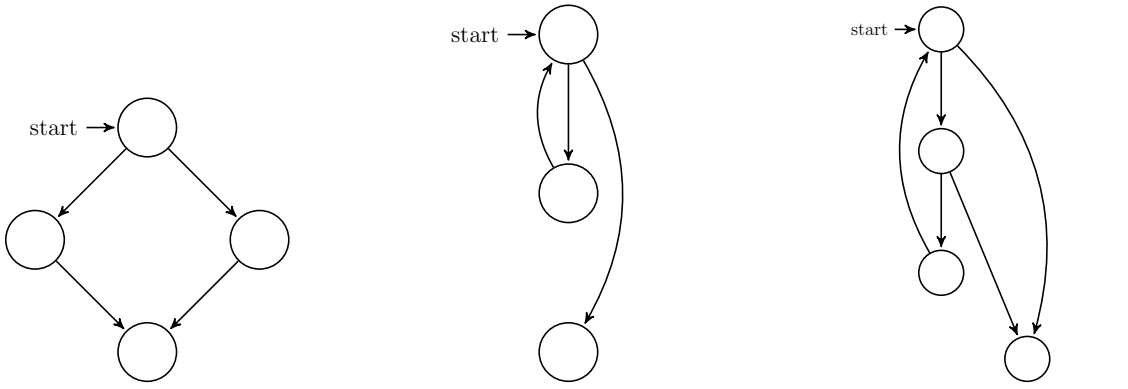
5.1 Control Flow Graphs

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.

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.

5.2 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



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

Figure 8: Example control flow graphs

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 Con-

trol 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.