CSE127 Scribe Notes Lecture 4

Professor Nadia Heninger

These lecture notes were scribed by students in CSE 127 Winter 2021. They have been lightly edited but may still contain errors.

1 Return-Oriented Programming

Return-Oriented Programming (ROP) is a type of exploit that allows attackers to make shellcode out of existing code. This is done by using *gadgets*, which are small sequences of instructions that generally end in a return. By utilizing a buffer overflow, an attacker can overwrite the saved return addresses on the stack to point to different gadgets. Doing so essentially "strings together" small sequences of existing instructions in order to construct an entirely new program.

The chief benefit of ROP is that it allows attackers to circumvent control-flow defenses, such as W^X, which prevent them from injecting and executing their own shellcode. ROP is also very flexible in the sense that it allows the attacker to construct shellcode with exactly their desired functionality, something that attacks like return-into-libc are not capable of.

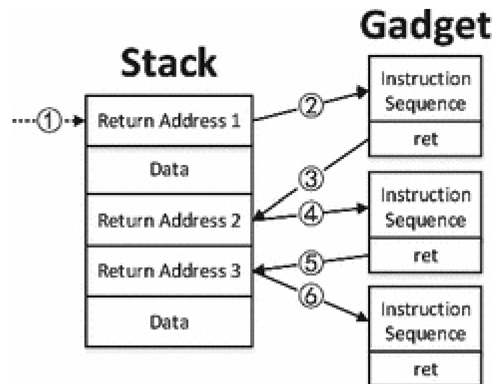

Figure 1: Interactions between gadgets and the stack[1]

[1]Si L., Yu J., Luo L., Ma J., Wu Q., Li S. (2016) ROP-Hunt: Detecting Return-Oriented Programming Attacks in Applications. In: Wang G., Ray I., Alcaraz Calero J., Thampi S. (eds) Security, Privacy, and Anonymity in Computation, Communication, and Storage. SpaCCS 2016. Lecture Notes in Computer Science, vol 10066. Springer, Cham. `https://doi.org/10.1007/978-3-319-49148-6_12`

We should note that later works have developed ROP without ending in returns, we will focus on gadgets that ends in a *ret* instruction, or 0xC3 in byte-code. This is because the ret instruction is the mechanism that is used to chain gadgets together. In the x86 architecture, the ret instruction pops the return address off the stack and into the instruction pointer $eip, which causes execution to continue at the specified return address. Figure 1 shows how ret instructions are used to transfer execution flow from one gadget to another, which allows attackers to create entirely new programs.

In the x86 architecture, instructions are variable length, and can begin on any byte boundary. Attackers can utilize this to construct multiple gadgets, each with different functionality, out of the same set of sequential instructions. Take, for example, the following instructions at the memory addresses 0x10 and 0x14:

0x10: b8 01 00 00

0x14: 00 5b c9 c3

These instructions yield the following gadget:

```
mov $0x1, %eax
pop %ebx
leave
ret
```

Simply changing the byte where execution begins can cause entirely different gadget functionality. Starting execution at $eip = 0x12, rather than $eip = 0x10, results in the following gadget, which performs an add instead of a mov!

```
add %al, (%eax)
pop %ebx
leave
ret
```

There exists software such as *ropper* that can identify gadgets in programs with the desired functionality and aid attackers in writing programs that compile to the necessary gadgets. Given tools such as this, and a program which includes a non-trivially large number of libraries, it is entirely possible that some functionally complete set of gadgets exist that can be exploited in some way with a buffer overflow. This makes ROP extremely difficult to prevent and defend against.

Mitigation of ROP deals with the observation that these attacks overwrite return addresses and function pointers to jump around targets in memory. Defending against this requires controlling the flow of a program and restricting execution to valid targets, which will be discussed in the next section.

2 Control-Flow Integrity

Control flow integrity (CFI) is a defense mechanism against control flow hijacking. In particular, control flow integrity focuses on restricting control flow to legitimate paths. That is, we try to ensure that jumps, calls, and returns can only go to allowed target destinations. Note that attackers may still have the potential to write malicious statements into memory, but the goal of control flow integrity is to prevent the attacker from jumping to arbitrary locations (such as jumping to arbitrary gadgets).

When we talk about restricting control flow, it is important to note that it is sufficient to restrict indirect transfers of control. Direct jumps or calls are hard coded in their instructions after compilation, so these transfers of control would not be under attacker control. There are two main groups of indirect transfers of control: the *forward path* and the *reverse path*. An indirect transfer of control via the forward path refers to jumping to or calling a function at an address in a register or in memory. Examples include function calls, interrupt handlers, and virtual calls. The reverse path refers to returning from a function using an address on the stack. In several of the targets of our buffer overflow assignment, our exploits rely on the indirect transfer of control via the reverse path by overflowing the buffer and overwriting the return address on the stack.

To find legitimate targets of control flow in a given program, we examine the program's *control flow graph*. By statically analyzing the program, we can determine the locations of control flow transfers, as well as their targets. An example program and corresponding control flow graph are shown below.

```
void sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}
```

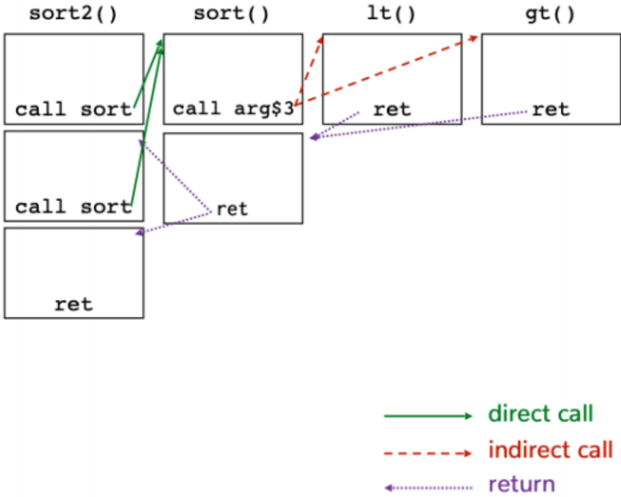

Figure 2: Control Flow Graph of `sort2`

After analyzing the control flow graph of the program, we now know the set of all legitimate indirect

transfers of control, where each transfer of control is defined by a pair consisting of the address of the indirect jump/call and its target. To restrict jumps to control flow, we assign labels to all indirect jumps and their targets. In execution, before taking an indirect jump, we first validate that the target label matches the jump site. Note that to implement this efficiently, we would require hardware support, as control flow integrity implemented only in software will have a heavy trade off of precision for performance.

A *coarse-grained CFI* implementation may be done as follows. We create labels for all destinations of indirect calls. In doing so, we ensure that every indirect call lands on a function entry. This would attempt to prevent attacks that jump to the middle of functions, such as in return-oriented programming. Next, we create labels for destination of `ret` instructions and indirect jumps. This attempts to ensure that every indirect jump lands at the start of a basic block of code, preventing returns to arbitrary locations. We call this implementation of CFI coarse grained, as an attacker could still be able to find a location in the code with a label that allows for returning to an exploitable location. The control flow graph figure below illustrates the implementation of coarse grained CFI on the `sort2` function presented earlier. Note the addition of labels as the means of control flow.

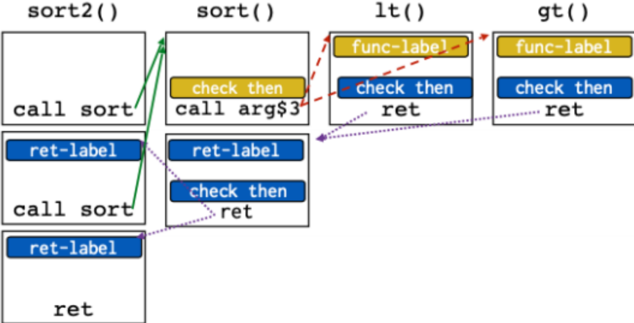

Figure 3: Coarse Grained CFI of `sort2`

Because *course-grained CFI* is vulnerable to function entry ROP gadgets, *fine-grained CFI* is a more precautionary, expensive measure. This implementation involves statically computing a control flow graph as well as integrating dynamic programming to assign labels to each target of an indirect transfer instead of imprecise function entries in *course grained CFI*. There will still be comparison of labels to ensure indirect transfers are valid.

For the cost of control flow integrity, the overhead becomes expensive. Each indirect branch instruction involves more instructions to calculate target labels and perform label comparisons. In spite of these protections, CFI is still vulnerable to data-only attacks and is dependent on accurate WˆX stack implementation, otherwise an attacker can execute the shellcode written onto the stack. *Coarse-grained CFI* enables attackers to return to multiple sites, so use of a shadow stack, which protects a function's stored return address by copying the program's call stack, can offer defense. This non-writable stack would then compare the caller's stack return address with the shadow's stack return address to check for attacks.

3 Heap Corruption

C/C++ uses explicit memory management which means that the data is allocated and freed dynamically and the dynamic memory is accessed via pointers. Since the system does not track memory liveness and does not ensure that pointers are live to valid, programmers need to be aware themselves.

Dynamically allocated data is stored on the *heap*. The heap is organized in contiguous chunks of memory which are stored in doubly linked list. A chunk is the basic unit of memory which can be allocated, freed, split or coalesced. Each chunk contains some metadata with information about the size and flags.

The heap manager provides API for allocating and deallocating memory using malloc() and free(). Each chunk of memory allocated by malloc() has to be released by a corresponding call to free(). The heap layout evolves with calls to malloc() and free().

Things can go wrong in the following ways.

- Forgot to free memory. The attacker can still access that portion of memory.
- Write /read memory we shouldn't have access to. This may allow attackers to overflow code pointer on the heap - similar to overwriting on the stack.
- Use after free. After a objected is freed, attackers can still use the dangling pointers that point to the freed object to point to other locations.
- Double free. If you try to free an already freed object it may corrupt the heap data structure. This can be used by the attacker to overwrite the metadata in a heap data structure itself.

There are many implications of heap corruption attacks. Using data-only attacks we can bypass security checks and modify the checks, flags and tags in the data. Additionally, attackers can overwrite heap management data to corrupt metadata in free chunks and program the heap weird machine. Similarly, attackers can overwrite function pointers to initiate a direct transfer of control when the function is called. In this case, C++ virtual tables are especially good targets since they are stored on heap and executable.

Here is an example of a heap exploitation performed on C++ vtables

```
1  class Base {
2      public:
3          uint32_t x;
4          Base(uint32_t x) : x(x) {};
5          virtual void f() {
6              cout << "base: " << x;
7          }
8  };
9
10 class Derived: public Base{
11     public:
12         Derived(uint32_t x) : Base (x) {};
13         void f() {
14             cout << "derived: " << x;
15         }
```

```
};

void bar(Base* obj){
    obj->f();
}

int main(int argc, char* argv[]){
    Base *b = new Base(42);
    Derived *d = new Derived(42);
    bar(b);
    bar(d);
}
```

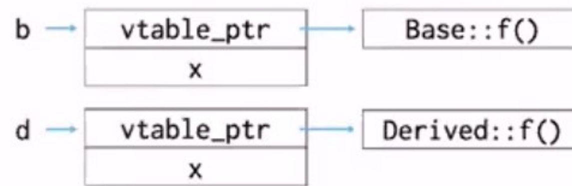

Figure 4: The memory diagram of b and d

In the above example, from the memory diagram we can see that objects b and d are vtable pointers which point to a virtual function. Then, in a use-after-free case, after freeing the objects, this chunk will be marked as free heap memory and will later be allocated to other variables. Thus, an attacker can overwrite the vtable so that the entry of the vtable points to the attacker's gadget. After that when the program calls this object's virtual function again it will transfer control to the attacker's malicious code.

In order to protect the integrity of heap data and the program, several heap exploitation mitigation have been proposed. Implementations to maintain the safety of the heap include introducing safe un-linking, adding cookies or canaries on the heap to prevent corruption and doing heap integrity check on malloc and free. However, the best mitigation remains the use of Rust or a safe garbage collected language. But this method comes with significant overhead and all these safety measures are still written in C which means they can be exploited themselves, and thus, vulnerabilities may sill exist even when using these languages.

4 Integer Overflows

One mitigation for buffer overflow attacks is to check bounds. However, doing so haphazardly leads to another class of vulnerabilities: integer overflows. Consider the following code.

```
void vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
```

```
    memcpy(buf, data, len);
}
```

This code thwarts naive attempts at a buffer overflow, as trying to copy more than 64 bytes of data will cause the function to do nothing. The issue here is that the length is stored in an *int* instead of a *size_t*. Calling this function with len = 0xffffffff, for example, will pass the bounds check because len will be interpreted as a signed int (0xffffffff = -1), but then the be interpreted as an unsigned int (0xffffffff = 4294967295) in memcpy. Fixing the code as below removes this vulnerability.

```
void safe(size_t len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

Below is another function with a vulnerability.

```
void f(size_t len, char *data) {
    char *buf = malloc(len+2);
    if (buf == NULL)
        return;
    memcpy(buf, data, len);
    buf[len] = "\n";
    buf[len+1] = "\0";
}
```

Again, calling this function with len = 0xffffffff will cause the argument to malloc to overflow, causing only 1 byte to be allocated. Meanwhile, the argument to memcpy will still be 0xffffffff. The way to guard against this input for the above code would be to make sure that len $\leq$ 0xffffffff - 2. That way, if the value of len were to cause an overflow in malloc, it is going to just return instead.

There are three main kinds of integer overflow bugs. Truncation, where a larger integer type is cast to a smaller one, arithmetic overflow bugs, where the result of of an operation is too large to fit in the datatype, and sign bugs, where signed numbers are treated as unsigned.

Integer overflow bugs are still a relevant class of bugs, as seen in this recent issue. This issue is discussing a possible integer overflow in chrome_sqlite_3_malloc when given a width of 32 and when memcpy's width is 64.

Issue 952406: Security: Possible OOB related to chrome_sqlite3_malloc
Reported by mlfbr...@stanford.edu on Fri, Apr 12, 2019, 1:59 PM PDT

Code

VULNERABILITY DETAILS
Possible OOB with chrome_sqlite3_malloc

REPRODUCTION CASE
There's a pattern of using sqlite malloc functions that call chrome_sqlite3_malloc in combination with traditional memory operations (e.g., memcpy). There may be invariants that make this ok, or a principle here that I am not aware of. Thanks for your time.

chrome_sqlite3_malloc takes an int size argument, while memcpy takes a size_t size argument. On x86-64 this means that chrome_sqlite_3_malloc's size argument is width 32, while memcpy's is width 64. This can lead to potentially concerning wrapping behavior for extreme allocation sizes (depending on the compiler, optimizations, etc).

For example:

Function fts3UpdateDocTotals
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&q=fts3UpdateDocTotals&g=0&l=3399)

(1) a = sqlite3_malloc((sizeof(u32)+10)*nStat);
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&q=fts3UpdateDocTotals&g=0&l=3416)
...
(2) memset(a, 0, sizeof(u32)*(nStat));
(https://cs.chromium.org/chromium/src/third_party/sqlite/patched/ext/fts3/fts3_write.c?type=cs&q=fts3UpdateDocTotals&g=0&l=3434)

Depending on optimization level etc, this may turn into:

```
(1)
size = mul i32 nstat 14
chrome_sqlite3_malloc(size)

(2)
tmp = sign extend nstat to i64
size = shl tmp 2
memset(size)
```

If nstat is a very large i32, the multiplication in step (1) *may* wrap. Nothing in (2) will wrap because of the sign extend, leading to an OOB.

There are a few ways to mitigate integer overflow vulnerabilities. One way is to use a language with arbitrary-length integers in a properly typed language with memory management. Another is to keep the common flavors of integer overflows in mind when programming.