Code Reuse

G. Lettieri

2 November 2023

1 Introduction

One of the most important discoveries in binary exploitation is the realization that arbitrary computation is possible even *without injecting any new code*. This means that non-executable data cannot completely block all attacks: it only makes them (slightly) more difficult.

In this lecture we will examine the most common techniques that reuse the code of the attacked binary to perform computations chosen by the attacker.

1.1 Return to libc

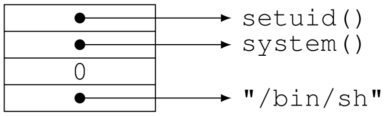
Probably the first public demonstration of how to defeat NX is Solar Designer's 1997 *return-into-libc* exploit[1].

The bug exploited by this technique is a standard buffer overflow that allows the attacker to overwrite part of a stack frame in a vulnerable program. The idea is to overwrite the saved return address with the entry point of another function. When the vulnerable function returns, the process starts executing the attacker's chosen function.

What function can the attacker choose? A very useful one, found in the C library, is of course `system()`. The attacker must also pass an argument to this function, which should be a pointer to a string containing the shell command to execute. On 32-bit systems, the arguments are passed up the stack, which the attacker controls, so the only problem is finding the address of an appropriate string. Strings are just data and can be injected into the stack as part of the buffer overflow attack. However, very useful strings (such as `"/bin/sh"`) can already be found in the C library itself. Having a string already in the binary has the added advantage that it can ve used even if the exploitable bug doesn't allow the attacker to inject null bytes, because otherwise it would be very difficult to terminate an injected string.

Solar Designer's exploit also shows how the attacker can chain two function calls, such as calling `setuid(0)` before calling `system("/bin/sh")`, to to bypass the shell's set-uid check. The attacker can prepare the following stack:

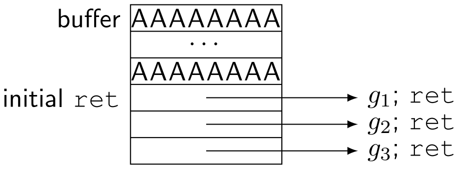
[1] `https://seclists.org/bugtraq/1997/Aug/63`



This takes adavantage of the fact that on function enter, the stack top of the stack should contain the return address of the function, followed by the function arguments. Thus, `setuid()` will use the `0` argument and, upon completion, "return" into `system()`, which will then use the pointer to `"/bin/sh"`. However, this method severely limits the number and type of functions that can be chained. In the following years, more general ways of performing longer computations were proposed, culminating in the Return Oriented Programming technique[2].

1.2 Return Oriented Programming

The idea of Return Oriented Programming (ROP for short) is to chain existing code using `ret` instructions, similar to the example above, but the chained code does not have to be complete functions. Rather, any existing code fragment that ends in a `ret` instruction can be used. These fragments are commonly called *ROP gadgets*, or simply gadgets when ROP is understood.[3] The ROP gadgets are *chained* using the `ret` instructions at their end. Suppose that a program contains a classic stack-based buffer overflow and the attacker wants to exploit it to execute gadget g_1;`ret`, followed gadget g_2;`ret`, followed by gadget g_3;`ret`. Assume, for the moment, that the gadgets do nothing with the stack pointer other than popping a return address with their final `ret`. Then the attacker can arrange the stack as follows:



The buffer is overflown to reach the saved return address, where the attacker places the address of the first gadget, followed by the addresses of the other gadgets. When the vulnerable function executes it `ret`, it jumps to gadget g_1, at the same time popping the address of g_1 off the stack. The stack pointer now points to the line below, which contains the address of g_2. When g_1 completes and executes its own `ret`, the execution will jump to g_2 and the stack pointer will be moved to g_3, and so on.

[2] The name is a joke on Object Oriented Programming.

[3] In the original proposal the term "gadget" referred to a stack arrangement that used one or more of these small fragments to perform a recognizable function, such as an if-then-else. With repeated use, however, the meaning of the term has become simpler and now refers to the code fragments themselves.

Probably the best way to think about this technique, is to forget the intended meaning of "`ret`" and imagine that we are programming yet another weird machine, which we can call the ROP machine, implemented by the underlying "normal" machine. The instruction pointer of the ROP machine is the stack pointer of the normal machine. The instruction set of the ROP machine contains all available gadgets, each gadget "microcoded" using the instructions of the normal machine. The opcode of each gadget is the address of the gadget in the normal machine memory. A ROP chain is a program for the ROP machine: it starts execution with a first `ret`, which must be executed by a normal program running on the normal machine, and then continues on its own.

What kind of gadgets can you expect to find, and which ones are useful? In a large codebase (think of the C library, for example) there are indeed many useful gadgets. In fact, it is very often the case that the ROP machine is Turing complete. In principle, then, the attacker can do whatever she wants by simply chaining gadgets. For example, she could implement a shell in this way. However, this would require a very large stack. In practice, it is better to just do what is needed to `execve()` a normal shell.

We can make some general observations about the technique:

- We need the absolute addresses of the gadgets; if we are attacking a remote server, we need a copy of its binary;
- we need to be able to inject these addresses, which can be a problem if they contain illegal bytes.

The latter is especially a problem (for attackers, that is) on 64 bit systems, where addresses tend to contain a lot of null bytes. If the bug that the attacker wants to exploit is based on some misuse of the string functions, it is usually not possible to inject null bytes.

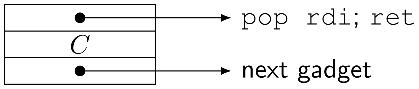
1.3 Notable gadgets

Let us now look at some notable gadgets (many more can be found in the exercises). Of particular interest are the very simple ones that perform elementary operations.

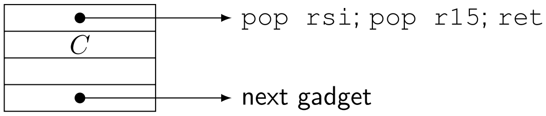
1.3.1 Loading registers

Consider a gadget that consists only of the two instructions `pop rdi` and `ret`[4]. This can be used to load a constant C into the `rdi` register, by arranging the stack as follows:

[4] This gadget is very common and you might wonder why: after all, `rdi` is a scratch register and the complier doesn't need to restore its contents before a function returns. In fact, this a prime example of an *unintended instruction* found in a binary: the original instruction was most likely `pop r15`, encoded as 0x41 0x5f, but this becomes `pop rdi` if we skip the first byte.



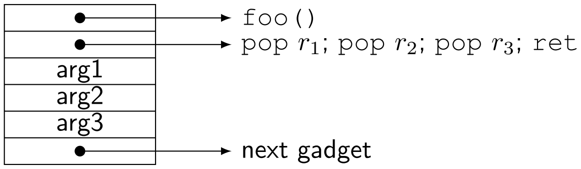
It may happen that we need to load a constant C into a register, say `rsi`, but we can only find gadgets that also load other registers. For example, the program may only contain the `pop rsi`; `pop r15`; `ret` gadget. This is often not a problem, if we can simply ignore the contents of `r15`: we only need to account for the additional `pop`:



In general, the gadgets can do something else in addition to what the attacker needs, and this is only a problem if the additional actions don't allow the process to continue (for example, a memory access to a random address that could crash the process).

1.3.2 Calling functions

Calling functions in 32 bit systems works much like the return-into-libc technique, but we can use the ROP idea to chain as many calls as we want. Suppose we want to call `foo(arg1, arg2, arg3)`. We have to arrange the stack as follows:



Instead of jumping directly from `foo()` into the next function (called "next gadget" above), we first jump into any gadget that moves the stack pointer past the arguments, like the 3-pops gadget above. After that we are again in a "clean" state again and we can continue in any way we want.

On 64-bit Intel/AMD systems, the first 6 arguments are passed in registers, which can be loaded using appropriate gadgets. It is sometimes difficult to find a gadget that loads the third register (`rdx`)[5], but one can use the general "return-to-csu" technique to achieve that[6]

[5] This is because `rdx` is scratch, and adding the 0x41 REX prefix to `pop rdx` we obtain `pop r10`, but `r10` is also scratch.

[6] You can find the details by following the links from here: `http://hmarco.org/`.

1.3.3 NOP

A gadget consisting only of a `ret` instruction is a NOP instruction for the ROP machine. This can actually be useful if we need to change the alignment of the stack pointer. For example, some SSE instructions raise an exception if the stack is not 16 byte aligned when they are executed, and these instructions can be found in some binaries that have been compiled with advanced optimization options (notably, the Ubuntu GNU libc). For example, on 64-bit systems the stack is aligned to either 8 or 16. A `ret` gadget will add 8 to `rsp`, changing the alignment from 8 to 16 or vice versa.

1.4 Finding gadgets

There are several tools and libraries that can analyze a binary and find potentially useful gadgets. Some of these tools are also able to automatically build generally useful ROP chains, such as chains that will `execve` a shell.

One such tool is `ropper`[7]. If we want to analyze a file named `binary`, we can call it like this:

```
ropper -f binary
```

This will print all the gadgets that `ropper` has found, with their absolute addresses. Note that not all gadgets found will end in `ret`: this is because other techniques have been developed that use other kinds of gadgets (for example, Jump Oriented Programming and Call Oriented Programming). These other gadgets may occasionally be useful even in an otherwise standard ROP chain, but you can pass `--type rop` if you don't want to see them.

The `ropper` tool also implements a search command that uses the “`%`” character as a jolly, much like the shell's “`*`” meta character. For example,

```
ropper -f binary --search ’pop %’
```

will find all gadgets that pop something off the stack, while

```
ropper -f binary --search ’mov [%], %’
```

will search all the gadgets that write to memory (note that Intel syntax is used).

The tool can automatically generate ROP chains for common tasks, but the binary should contain enough gadgets for it to succeed. For example, if `/lib/libc.so.6` is the path of the system C library, the following command

```
ropper -f /lib/libc.so.6 --chain execve
```

will print on stdout a (Python 2) program that outputs the generated ROP chain. The output can be redirected to a file and then edited.

Other useful options can be found by studying the output of `ropper --help`.

[7] `https://github.com/sashs/Ropper`

2 One gadgets

In many cases, the C library already contains fragments of code that do everything that the attacker needs. Finding these "one gadgets" is very useful for attackers, since sometimes they can only overwrite a single function pointer in memory. The most common one-gadgets that can be found are, again, fragments that `execve()` the shell. The `one_gadget`[8] utility is able to find such fragments in a binary. Typically, the binary is the C library: the trick is that there are at least two places in the library where `/bin/sh` is spawned: in the implementation of `system()` and in the implementation of `popen()`. Of course these functions will want to pass their own arguments to `/bin/sh`, but jumping at the right place inside of them may end up running a shell without arguments, or with arguments chosen by the attacker. These jump targets in the middle of these functions are the one-gadgets. However, there is catch: the tool may show a set of *constraints* for each one-gadget it finds. These are sufficient conditions that must be true just before the jump, to guarantee that the one-gadget will be successful. These constraints typically require some register or some stack line to contain a particular value (typically 0). If the attacker can only redirect execution to the one-gadget, with no way to execute anything else, she must choose a one-gadget whose constrains are all satisfied.

Note, however, that the conditions shown by the tool are only sufficient, not necessary: it is often the case that the gadget will work, even if the constrains are not satisfied. For example, Linux `execve()` will work even if the second argument is `NULL`. This is not a standard behavior, and the man page warns against relying on it if you want your program to be portable to other Unix-like systems. Of course, none of this is of concern to an attacker.

[8] `https://github.com/david942j/one\_gadget`