Return Oriented Programming

March 21st, 2019

Today's Goals

- Introduce ROP!

Announcements

- See https://sites.google.com/whitehatters.org/wcsc/announcements for CTFs we will be playing in this week

Reading Material

- My Intro to Buffer Overflows
- Dive into ROP
- Return-Oriented Programming Overcoming Defenses

Buffer Overflows—A Quick Recap

It's been a while since our first talk on buffer overflows, so let's take a look at them again (again, because we did this last week too). We will take a look at the old bof.c program from the buffer overflow talk (in particular, the section "What is a buffer overflow?" and "Pwning the system").

Code-Reuse Attacks

Last week, we worked on an example with shellcode, but with DEP implemented. This didn't work, because the stack was not executable.

To get around this, we need to reuse existing code that is marked as executable. In our buffer overflow example, we actually already reused existing code in the binary! Instead of writing shellcode, we simply jumped to somewhere in the binary. Of course, this is a trivial example.

Let's take a look at a more advanced code-reuse techniques, return-oriented programming.

Return-Oriented Programming

In return-oriented programming, our goal is to use the stack frame that we overwrote to call new bits of code. To do this, we will create fake stack frames! In other words, we will repeatedly use return to go to new code by adding consecutive stack frames that can be returned to. Sounds confusing… but let's go through an example to clear things up!

Let's think about the stack before we dive into an example. Here is our stack before a buffer overflow.

Local 1 (string)
Stored EBP
Stored EIP
Arg 1
Arg 2

After the buffer overflow

AAAA
AAAA
0xdeadbeef
0xcafebabe
0x1337dad0

Using the above stack, what will happen when it reaches a return statement? The code will jump to 0xdeadbeef! Let's assume that 0xdeadbeef was a function; the stack now looks like:

Local Vars
Stored EBP
0xcafebabe
0x1337dad0

Now what will happen when we reach a return? The code will jump to 0xcafebabe! Assuming that was also a function, we now have:

Local Vars
Stored EBP
0x1337dad0

And when this reaches a return statement? So, we have now called multiple functions using return-oriented programming! This is a pretty simple example again, but it helps us ease into the concept. Check out noargs.c for an example of this.

Calling functions with arguments

What does the stack look like when the call instruction is used? Where are arguments stored on the stack? When a function is called, the arguments are pushed onto the stack in reverse order, and the return address is then pushed on by the call instruction. What if we wanted to call 0xdeadbeef with arguments 1, 2, 3? What does this look like?

Return for 0xdeadbeef
1
2
3

So, using our rop form from before, we need to write the following to run 0xdeadbeef:

0xdeadbeef
Return for 0xdeadbeef
1
2
3

What happens when this stack reaches a return statement?

The return instruction returns to 0xdeadbeef, beginning the function there.

Return for 0xdeadbeef
1
2
3

Now, 0xdeadbeef is running, and uses the arguments. When 0xdeadbeef is ready to finish, it will return to the return address set above.

Let’s take a look at this in onearg.c!

Multiple Functions with Arguments

Alright, but how do we call multiple functions with arguments? Let’s take a look at that. We will start with our ROP from the previous example. Let’s say we want to call 0xdeadbeef, then call 0xcafebabe. The stack below will return us to 0xdeadbeef, run 0xdeadbeef with 1, 2, 3, and then return to 0xcafebabe. However, is the stack correctly setup for 0xcafebabe? Let’s step through it.

0xdeadbeef
0xcafebabe
1
2
3

First, we return to 0xdeadbeef:

0xcafebabe
1
2
3

0xdeadbeef runs, and is now complete. So, the code now returns to 0xcafebabe:

1
2
3

What are the return address and arguments now for 0xcafebabe? 1 is now the return address, 2 is the 1st argument, and 3 is the 2nd argument. Well, that is not good. What if we needed to pass arguments to 0xcafebabe. We need to pop off these arguments first, and provide the arguments for 0xcafebabe. The stack will look something like this then:

0xdeadbeef	Return to 0xdeadbeef
****	Jump addr to remove args
1	0xdeadbeef arg 1
2	0xdeadbeef arg 2
3	0xdeadbeef arg 3
0xcafebabe	Return to 0xcafebabe
****	Return address for 0xcafebabe
4	0xcafebabe arg 1
5	0xcafebabe arg 2
6	0xcafebabe arg 3

The file multiarg.c requires us to do this, and multiarg_sol.py solves it. So, we somehow need to remove those arguments off the stack. How would we normally go about removing values from the stack?

In other cases, we may use pop to remove values, or we might just subtract from esp. Either of these would work. So, we need to search the code for some address that does (in the above) pop, pop, pop, ret. Why do we need that last ret?

0xdeadbeef	Return to 0xdeadbeef
0x10901090	Pop pop pop ret
1	0xdeadbeef arg 1
2	0xdeadbeef arg 2
3	0xdeadbeef arg 3
0xcafebabe	Return to 0xcafebabe
****	Return address for 0xcafebabe
4	0xcafebabe arg 1
5	0xcafebabe arg 2
6	0xcafebabe arg 3

So, assuming 0x10901090 points to pop pop pop ret, lets walk through this stack now.

First, we return to 0xdeadbeef:

0x10901090	Pop pop pop ret
1	0xdeadbeef arg 1
2	0xdeadbeef arg 2
3	0xdeadbeef arg 3
0xcafebabe	Return to 0xcafebabe
****	Return address for 0xcafebabe
4	0xcafebabe arg 1
5	0xcafebabe arg 2
6	0xcafebabe arg 3

Now 0xdeadbeef runs with arguments 1, 2, 3, and is ready to return. It returns to address 0x10901090, which points to pop pop pop ret.

1	0xdeadbeef arg 1
2	0xdeadbeef arg 2
3	0xdeadbeef arg 3
0xcafebabe	Return to 0xcafebabe
****	Return address for 0xcafebabe
4	0xcafebabe arg 1
5	0xcafebabe arg 2
6	0xcafebabe arg 3

Alright, so now the pop pop pop runs. What does the stack look like now?

0xcafebabe	Return to 0xcafebabe
****	Return address for 0xcafebabe
4	0xcafebabe arg 1
5	0xcafebabe arg 2
6	0xcafebabe arg 3

So the arguments have been popped off, and we are now at a ret. This should look familiar now; we return to 0xcafebabe with arguments 4, 5, and 6, just like we wanted. We could call as many functions as we wanted in this fashion.

Rop Gadgets

How do we go about finding that pop pop pop ret we used in the previous example? That bit of code is called a rop gadget. A rop gadget is a small bit of code used in a rop chain to perform some operation. Typically this is followed by a return, call or jump that allows the attacker to continue controlling the stack.

These gadgets may occur anywhere in the executable code. To help find them, there are many different tools. For example, we have:

- RopGadget

- Ropper
- Rop-tool
- And many more!

Fortunately for us, both pwntools and pwndbg provide support for these tools already. If we open GDB with pwndbg installed, we can simply type rop to have RopGadget dump all potential rop gadgets. If we want to search for a specific instruction, we can use rop --grep <instr> to find search through the gadgets.

For example, when writing multiargs_sol.py, I used rop --grep ret to find gadgets ending in ret. Let’s check that out as well.