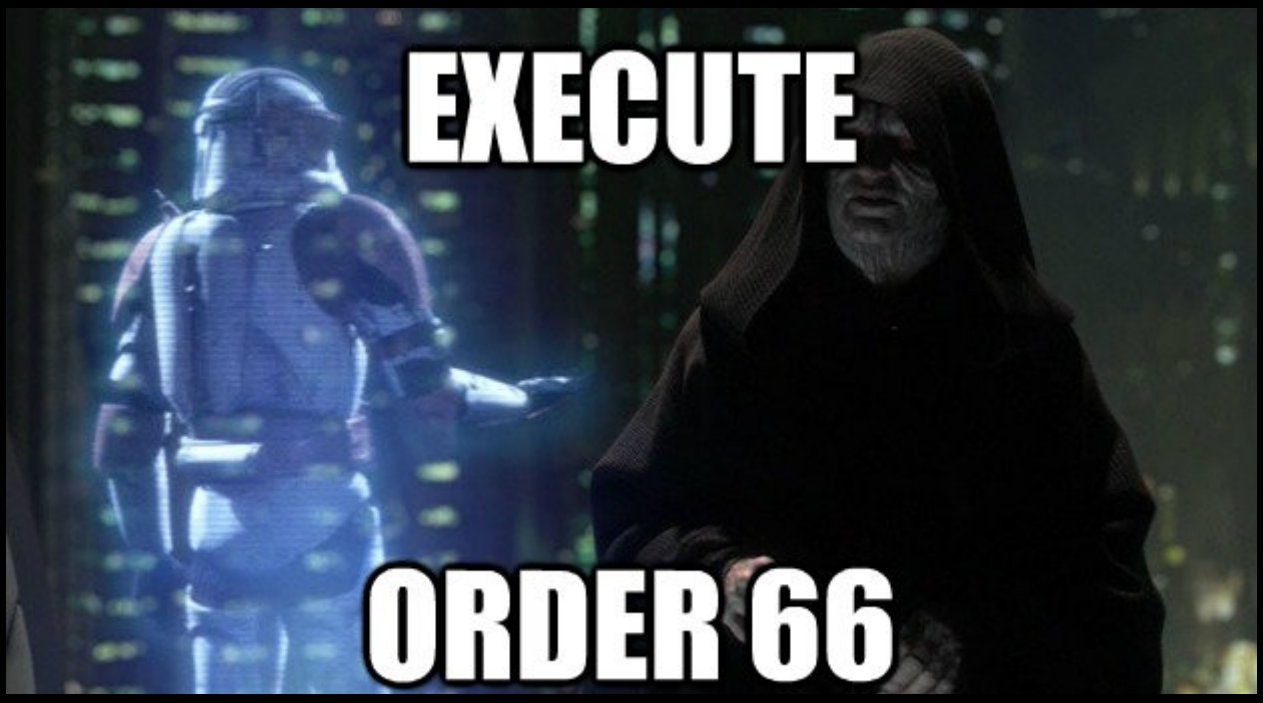
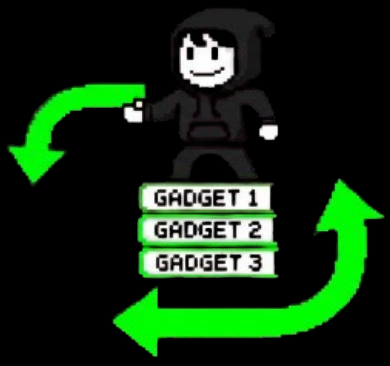
NX Memory



Introduction

The NX bit is a feature of the CPU that mark certain areas of memory as non-executable. Thankfully, all modern operating systems support the NX bit, and by default, the stack is set to NX. In this paper, we'll explore some techniques to bypass the NX bit in order to exploit the stack. We'll dive into how functions are called at the assembly level, touch on a concept known as Return Oriented Programming (ROP), and even create some new shellcodes along the way.
LET'S GET STARTED!

ROP



Return Oriented Programming (ROP) is the technique based on fact that we can change return address also to the different functions. libc as you remember is included in a process as a shared library. So instead of jumping on our shellcode we can call functions from libc directly. The only thing we need is to keep Calling Convention.

Calling Convention is a scheme for how functions receive parameters and how they return a result. Each CPU architecture have his own calling convention, and even for x86 and separately for x86-64.

You'll find some extra resources to dive deeper into this topic at the end of the paper. For now, let's you should remember before we move on.

1. Adjust RSP before everything
2. Returned value is in RAX
3. Arguments stored in:
   RDI, RSI, RDX, RCX, R8, R9
4. 128 bytes below RSP is safe

Stick to those rules, and you'll save more neurons in your brain.

Let's try ROP in practice.

```
#include <stdio.h>

void  win(void)
{
  puts("That's a good one");
}

int main(void)
{
  unsigned char buffer[16];

  puts("Say something interesting");
  gets(buffer);
}
```

Turn off ASLR and compile with `-no-pie` flag. Test in GDB for now.

In GDB, you can find the address of the win() function by using the command `p win`. From there, try to overflow the buffer and change the return address.

To ensure our program exits correctly, we can use the exit() function from libc. While the program is running, we need to locate the address of the exit() function and place it right after the address of the win() function. Try to spawn a shell using ROP along with the system() function. You can easily find a pointer to /bin/sh in libc using `find "/bin/sh"` in GDB.

Making Stack Executable

Since the stack is set to NX by default, let's change that up. The libc library has this fantastic function called mprotect() (man mprotect for more). mprotect() lets us modify the permissions of almost any memory region. For instance, we can make it executable. The catch is that we need to pass arguments to the function through registers. But don't worry, that's pretty straightforward, we can manipulate the return address to point to some instruction using ROP technique. For example, we can put our variables after the ret instruction, to place at the at the top of the next stack frame after ret, pop them into the registers, and then return to the next address, and keep going from there.

```c
#include <stdio.h>

int main(void)
{
   unsigned char buffer[64];
   printf("main: %p\n", main);
   gets(buffer);
}
```

In the VM, I left a tool called **ropper**. Since our main code is a bit too small to include all the necessary gadgets, we'll need to look up the libc instead. Use the command **ldd file_name** to find the path to the libc library file, load it into ropper with **file /path/to/file** command, and then **search instruction** to locate your gadget offsets. Just a heads up, ropper will only provide you with the gadget offset, so you'll need to calculate it yourself by taking first address of libc and adding the offset of gadget. You can find the address of the loaded libc using the **vmmap** command in GDB while the program is running, then you can calculate distance from main().

The important part is, that you need to change permissions of whole stack. That's very important. From previous papers, you need to remember (I hope so) about memory pages. mprotect() works per page, so to make it simple, just take whole stack which is already adjusted, and work with that.

For the third argument in mprotect(), simply set it to 0x7 to allow all permissions for the memory region.

Also you can use GDB's ropsearch, sometimes it can find necessary ones, sometimes with the junk but anyway.

Allocating New Memory

Sometimes, when shellcode gets a bit too large, we can take matters into our own hands by manually allocating memory with the right permissions to hold it. You could save the shellcode's bytecode in a file and have the target program read from it to execute it. Or, feel free to get creative and think outside the box!

I won't go into the specifics of how to do that here. You already understand how a ROP chain operates, and I'm confident you can figure out how to implement your exploit in that manner (`man mmap`).

Just remember, in the future, you'll definitely need to allocate new memory—not just for storing shellcode, but also for reading and saving other data. Plus, allocating new memory can help you fly under the radar and avoid detection, since it's less suspicious than just changing stack permissions.

References

https://en.wikipedia.org/wiki/Calling_convention

https://en.wikipedia.org/wiki/Return-oriented_programming

https://ret2rop.blogspot.com/2018/08/make-stack-executable-again.html