StackGuard, ASLR, and NX

March 7th, 2019

Today's Goals

- Introduce several defense mechanisms, the assumptions they make, and their weaknesses
- Mechanisms cannot be empirically measured for success; human adversaries are smart, and will come up with a workaround
- Some lead up into next week's topic, ROP

Announcements

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

Reading Material

- StackGuard
  - Available on USF campus or via USF VPN
- Address Space Layout Randomization
  - Wiki article has a good coverage of the concept and history
  - Overcoming ASLR in Windows
    - Not a lot of detail
- Data Execution Prevention
  - NX Bit in Linux
- Return-Oriented Programming Overcoming Defenses
- Pwntools checksec tool
  - Prints the security mechanisms currently on the program. All green is bad for us.

Buffer Overflows—A Quick Recap

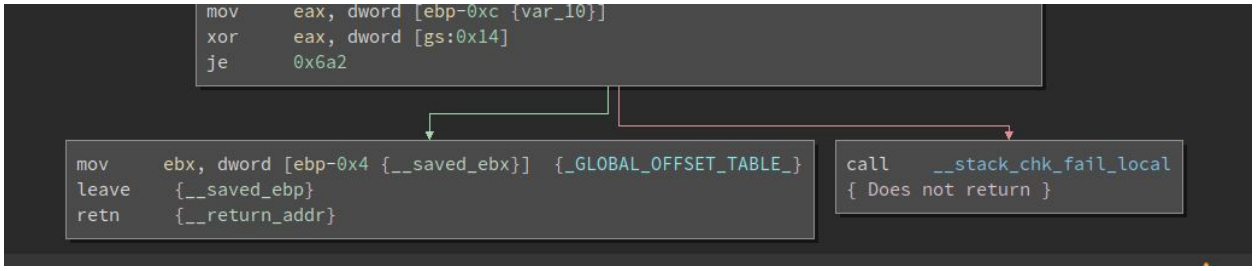
It's been a while since our first talk on buffer overflows, so let's take a look at them again. We will take a look at the old bof.c program from the buffer overflow talk (in particular, the section "What is a buffer overflow?" and "Pwning the system").

StackGuard

Stackguard is a proposed solution to buffer overflows that became quite popular. Stackguard works by placing a “canary” word between the local variables on the stack, and the return address/stored ebp.

Local Vars
Canary (4B)
EBP (4B)
Return address (4B)
Function Args

When the function is about to return, the code checks if the Canary Word has been changed. If it has been changed, the program exits. The below Binary Ninja snippet shows the stack check call code in assembly.



If we try to run our code that we used before, it will fail. Our previous code attempts to overwrite EIP, and thus kills the canary. Let’s try it.

- Run bof.py, which uses the version without any defenses. Success
- Run try_bof_all.py, which uses the version with all the defenses. Stack smashing detected.

It blocked our effort here, but the paper claims StackGuard is “a simple compiler technique that virtually eliminates buffer overflow vulnerabilities”. Eliminates buffer overflows, eh?

StackGuard Shortcomings

In the StackGuard paper, it discusses two shortcomings. The first is skipping over the canary word, and the second is that the attacker could theoretically get lucky and guess the canary word.

Stackguard doesn't prevent all buffer overflows; instead it prevents a small subsection of buffer overflow attacks that may be referred to as stack-smashing attacks. What about local variables? Does StackGuard offer any protection for local variables?

Corrupting Local Variables

StackGuard itself does not offer any protection; however, GCC is smart and will reorder local variables on the stack so that buffers are placed below other variables, and thus cannot overwrite them. Take our overwriteInt.c example. With the canary enabled, the stack is reordered so that the buffer is placed below the integer.

Random integer
String
Canary (4B)
EBP (4B)
Return address (4B)
Function Args

No matter how you change the code, the string will not be before the integer. However, structs in C do provide ordering for local variables. Let's take a look at overwriteStruct.c.

As you can see, the stack canary can do nothing to prevent this kind of buffer overflow! What would have happened if the value had been a function pointer?

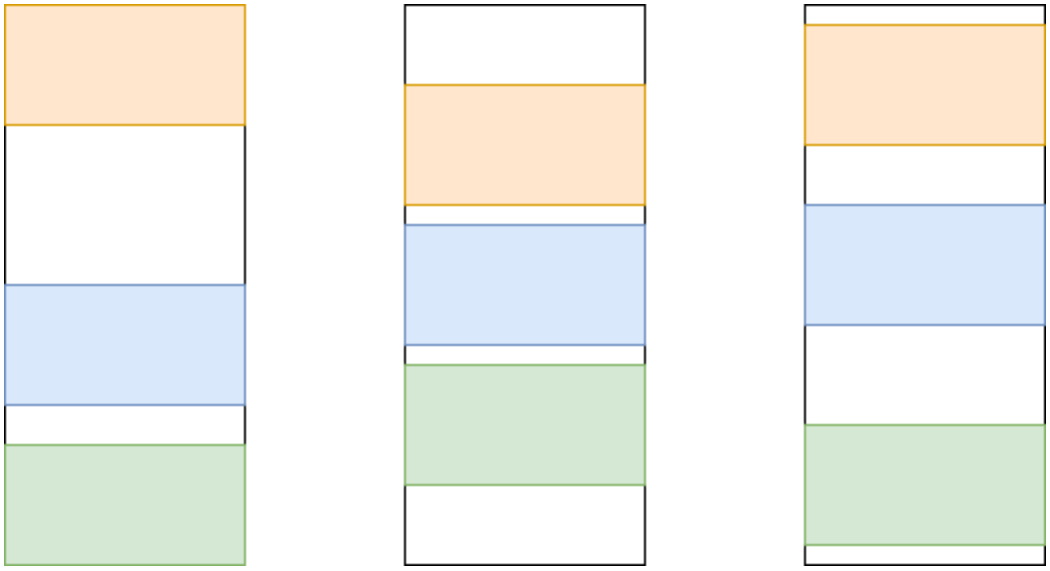
This serves as one example of how StackGuard does not prevent all buffer overflow attacks.

Why does StackGuard fail?

StackGuard makes the assumption that buffer overflows must make subsequent writes to memory that will overwrite the canary. If we work around this assumption, like in the above snippet, we can develop new exploits.

Address Space Layout Randomization

ASLR loads the different sections of a program at different offsets. This means that the addresses of variables on the stack will be changed between each run! If the code is compiled with the position independent code (PIE) option, the addresses of code will also be changed! Here is a simple diagram representing what's happening:



Note that the makefile included will disable/enable ASLR on your system if you run the following commands. If you are interested in this command, check out the contents of the makefile. To disable this in GDB, run set disable-randomization off.

```
make disable # Disable ASLR
make enable # Enable ASLR
```

Let's check this out. Run ./bof_all over and over, checking the printed address of the string each time.

Defeating ASLR

There are a few different ways to go about defeating ASLR, all of which essentially have us rendering the randomization of the stack useless. For simplicity, I assume the canary option is off and that the stack is executable.

Leaking a Stack Address

The most common way of defeating ASLR is by leaking a memory address for one or more segments of the program. For example, our buffer overflow program is kind of enough to tell us the address of our buffer on the stack. By using this address, we now know the addresses of all of the stack values.

In our buffer overflow example, we know the return address is 16 bytes after our buffer. So, if we find the buffer was at address 10, the return address must be at address 26, and so on.

Check out the python script bof_aslr_shell.py which uses shellcode and the lack of NX to obtain a shell using the technique described above.

Making Randomization Meaningless

If we could allocate many strings, we could fill memory with shellcode. Once memory is full, we can jump to a random address, and chances are high that we will jump into our shellcode. This is a technique known as spraying.

If there is interest in this, we can talk about heap exploits in the future, including heap overflows and heap spraying.

Check out this article for heap spraying. Stack spraying isn't considered practical, but it's theoretically possible to do by providing a long string of malicious arguments.

Not Really Random…

If the ASLR can be predicted, or is not properly reset between runs, it would be possible to determine the addresses. Check out this paper. In it, the ASLR offset for processes forked from a parent can be determined since the children aren't properly randomized each time.

No Execute

No execute makes the stack and other non-code sections non-executable. This breaks our example that uses shellcode! Check out bof_shell.py

This brings us to current state-of-the-art techniques. Code reuse attacks such as return-oriented programming jump to code that already exists!

Technically, our original bof solution is a reuse attack, as it jumps to the shell function! However, that's a bit silly.

There really isn't much more to say about NX… We will be doing ROP next week!