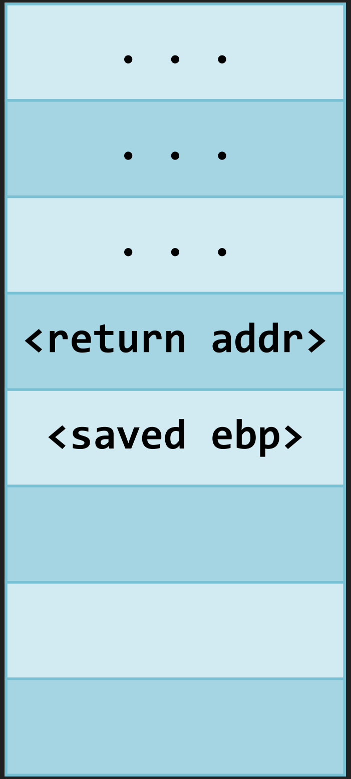
Stopping Hackers 101: DEP and ASLR

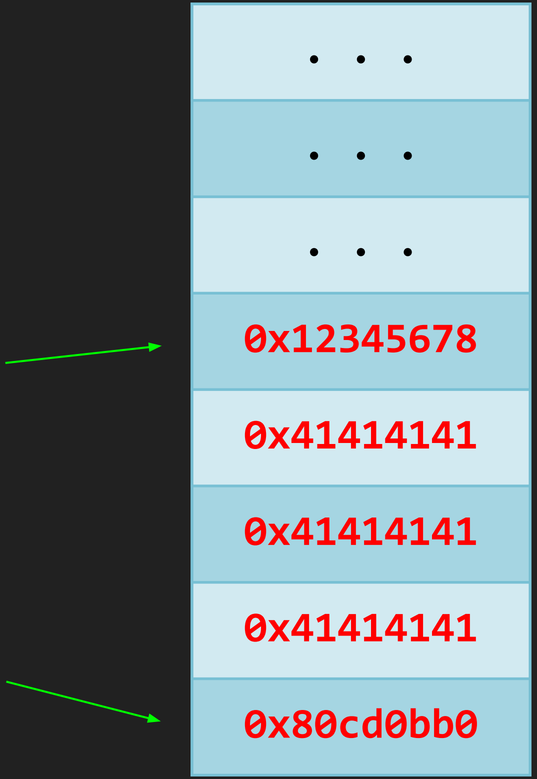
MST - 2/6

Review: Buffer Overflows



1. Write our shellcode to the buffer
2. Pad it with “A”s so it’s long enough
3. Replace the return address with the start of the buffer
4. Execute our shellcode!

Review: Buffer Overflows

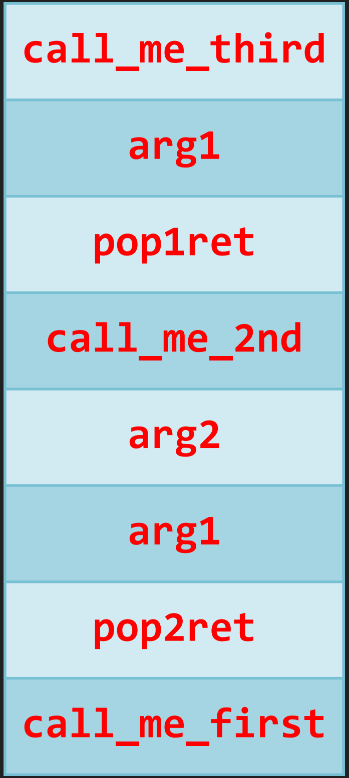


1. Write our shellcode to the buffer
2. Pad it with “A”s so it’s long enough
3. Replace the return address with the start of the buffer
4. Execute our shellcode!

Review: ROP attacks

- **Return Oriented Programming** - jumping to arbitrary locations to properly execute code
- We put multiple return addresses on the stack by creating fake stack frames
- **ROP Gadgets** are little snippets of instructions that we can use to build up a larger **ROP chain**

Review: ROP attacks



call_me_third

arg1

pop1ret

call_me_2nd

arg2

arg1

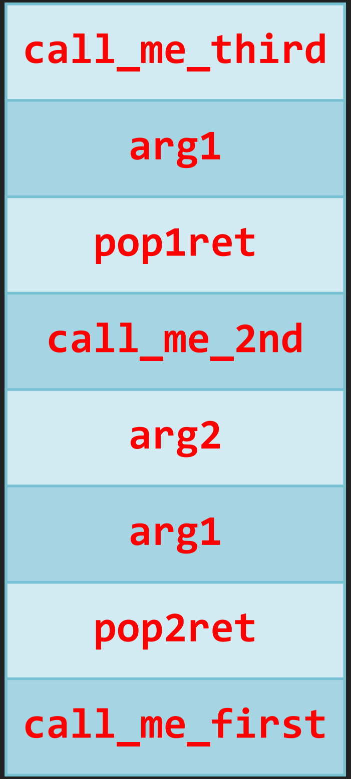
pop2ret

call_me_first

```
pop1ret:
    pop ebp
    ret

pop2ret:
    pop ebx
    pop ebp
    ret
```

Review: ROP attacks



call_me_third

arg1

pop1ret

call_me_2nd

arg2

arg1

pop2ret

call_me_first

1. `call_me_first` is called with `arg1` and `arg2`
2. `pop2ret` removes 2 arguments from the stack
3. `call_me_2nd` is called with `arg1`
4. `pop1ret` removes the argument from the stack
5. etc

DEP (or W^X or NX)

- Buffer overflows rely on putting code on the stack and executing it
- **Data Execution Prevention**: mark the stack as **non-executable** memory
    - Also known as **W^X**: memory can be Writable or eXecutable, but not both at the same time
- DEP prevents us from just writing code to protected memory and executing it

DEP in practice

Bypassing DEP

- This is where ret2libc and ROP come in
    - These are directly motivated by the rise in DEP protected programs
- Some exploits will use a short ROP chain to disable memory protections, then write code and execute it directly

ASLR

- ROP attacks and ret2libc rely on **knowing addresses** in memory of code
- **Address space layout randomization** combats this by randomly shifting code, libraries, and data around in memory on each execution
- ret2libc won’t work if we can’t find library functions
- ROP attacks fail if we don’t know where our target gadgets will be in memory!

ASLR in the wild

```c
#include <stdio.h>
int main() {
    char *mystr = "Hello\n";
    printf("%p\n", mystr);
    return 0;
}
```

```
$ ./hello
0x105772f9a
$ ./hello
0x10d435f9a
$ ./hello
0x102b0bf9a
$ ./hello
0x108014f9a
$ ./hello
0x107b58f9a
$ ./hello
0x103483f9a
$ ./hello
0x1070cef9a
```

ASLR in practice

- When the program loads, the OS will apply a random shift to each segment of the program (code, data, stack, heap, etc)
- Shared libraries can be automatically protected by the OS
- To protect the executable itself, it must be compiled as a “position independent executable”
- Not all bits of the address are randomized
    - On the earlier slide, only the middle 16 bits of the address were randomized

Bypassing ASLR

- **Key insight:** Each section has a linear shift applied to it at the start of execution
- If we can get the address of anything in a section, **we can calculate the shift and undo it**
- Shared library functions (like the standard library) are shifted as a whole
- Another avenue of attack relies on bruteforcing addresses (feasible on 32-bit systems)
- ASLR bypasses are sometimes referred to as “leaked pointers” or “dangling pointers”

Bypassing ASLR

```c
// Compiled with ASLR enabled

#include <stdio.h>
#include <stdlib.h>

int main() {
    unsigned int x = printf;
    unsigned int y = system;
    printf("print is at %x\n", x);
    printf("system is at %x\n", y);
    printf("difference: %x\n", x-y);
    return 0;
}
```

```
$ ./addrs
print is at 7c667800
system is at 7c657390
difference: 10470
$ ./addrs
print is at b4f41800
system is at b4f31390
difference: 10470
$ ./addrs
print is at 2852c800
system is at 2851c390
difference: 10470
```

Bypassing ASLR

```
// Compiled with ASLR enabled

#include <stdio.h>
#include <stdlib.h>

int main() {
        int (*func)(char *) = (int
(*)(char *)) (printf - 0x10470);
        func("echo hello world");

        return 0;
}
```

```
$ ./indirect
hello world
$ ./indirect
hello world
$ ./indirect
hello world
```

ASLR and DEP deployment

- Windows + Linux both got DEP in 2004/2005
  - Almost always enabled by default, so very common
- Linux has had a basic form of ASLR since 2005
  - Kernel ASLR was only enabled in 2014!
- Windows began widescale deployment of ASLR with Windows Vista (~2007)
- ASLR is weakened if any library doesn’t have it enabled
  - Attacker can just look for ROP gadgets there
- Many systems still run without full ASLR on everything out of the box
  - Third party packages are even less likely to have ASLR
- 2/15/2017 ASLR is broken on all modern CPUs
  - Even from JavaScript

Useful command: checksec in gdb-peda

```
Reading symbols from a.out...(no debugging symbols found)...done.
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled ; (this indicates ASLR)
RELRO     : Partial
```

Homework

Homework

- Hack an “embedded system”
- https://microcorruption.com/
- Play through the tutorial and the New Orleans level
- And then continue to play because it’s fun

Tips

- **Don’t hesitate to ask questions! (Slack, email, etc)**
  - **Or the official Microcorruption IRC channel: irc.freenode.net #uctf**

Homework grading

- Submit your profile link to cm7bv@virginia.edu with the subject “MST Assignment 4 - <YOUR_UVA_ID>”
    - eg: “MST Assignment 4 - cm7bv”
    - eg: https://microcorruption.com/profile/7552
- Also, include a brief (1-paragraph) description of what you did and how it went

Useful Resources

- Useful ROP tutorial (http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html)
- https://security.stackexchange.com/questions/22989/how-leaking-pointers-to-bypass-dep-aslr-works
- Baby's first NX+ASLR bypass (https://www.trustwave.com/Resources/SpiderLabs-Blog/Baby-s-first-NX-ASLR-bypass/)