Lab 4

Stack Attack!

Overview

What is Stack Attack! All About?

- The Goal
- The Targets
  - Injection Attacks (**ctarget**)
  - Return Oriented Programming (**rtarget**)
- The Tools (objdump, gdb, hex2raw)
- The Grade

The Goal

- Learn how attackers exploit vulnerabilities in code
- Better knowledge of safe and secure coding practices
- Learn stack and parameter passing
- More x86 instructions and GDB debugging

How do we get there?

Build exploit strings for two programs

- **ctarget** – 3 buffer overflow exploits
- **rtarget** – 2 return-oriented exploits

The exploits are
more difficult at each level

Injection Attacks

Use exploit strings to add executable code to an existing program, making it change behavior

- Level 1 – Change return address
- Level 2 – Inject function call w/int
- Level 3 – Inject function call w/string

Level 1

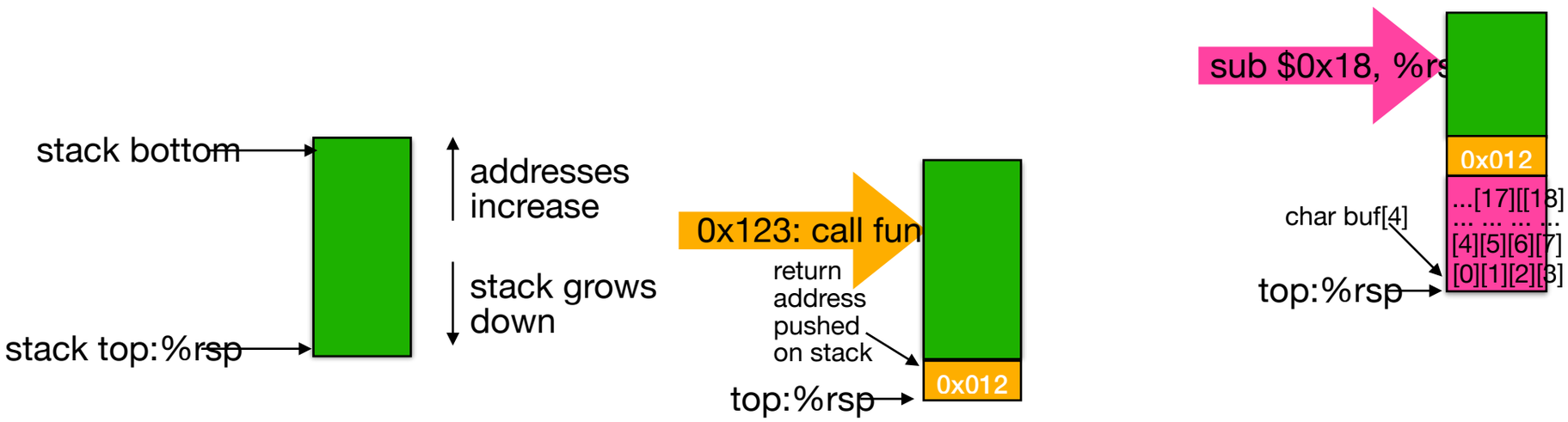
Change return of **getbuf()** to return to **touch1()** instead of **test()** in **ctarget**.

- "Code Injection" not needed
- Rewrite return address on stack
- **objdump -d ctarget** can get you there
- Review **buftest** video … same technique

buftest.c

Demonstration

buftest.c


stack bottom
addresses increase
stack grows down
stack top:%rsp
0x123: call fun
return address pushed on stack
top:%rsp
0x012
sub $0x18, %rs
0x012
char buf[4]
...[17][[18]
... ...
[4][5][6][7]
[0][1][2][3]
top:%rsp

Level 2

Inject code to pass **magic cookie** to **touch2()** instead of returning from **getbuf()** in **ctarget**.

- Build machine code from assembly
- Add code to the stack (inject)
- Force return to injected code
- Call **touch2(int x)** with correct parameter

buftest2.c

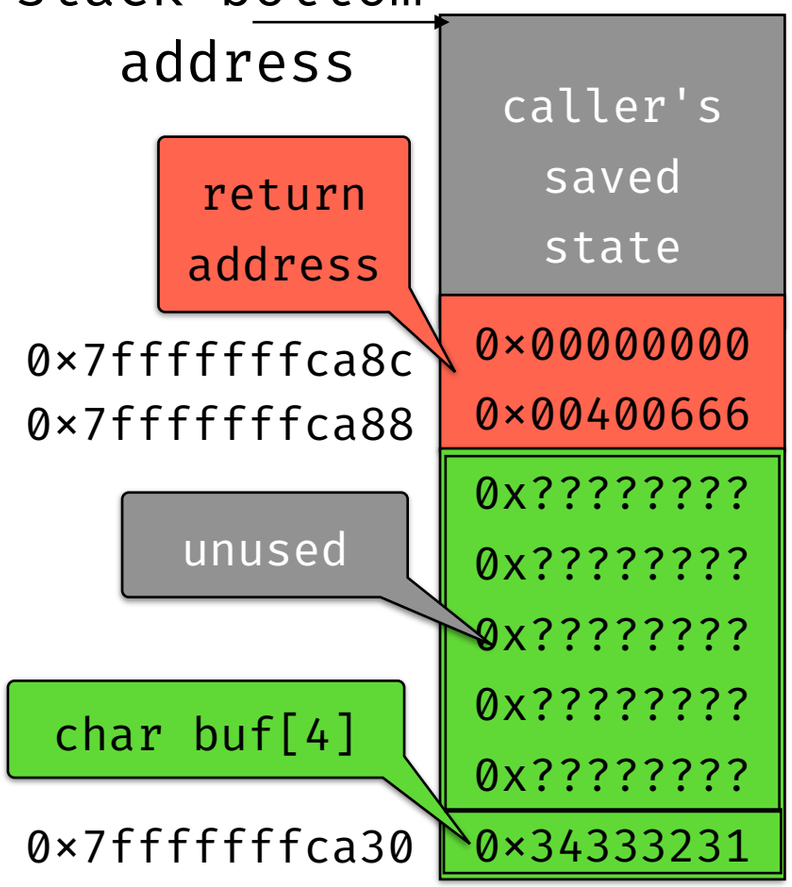
Demonstration

Local Variable Allocation on Stack

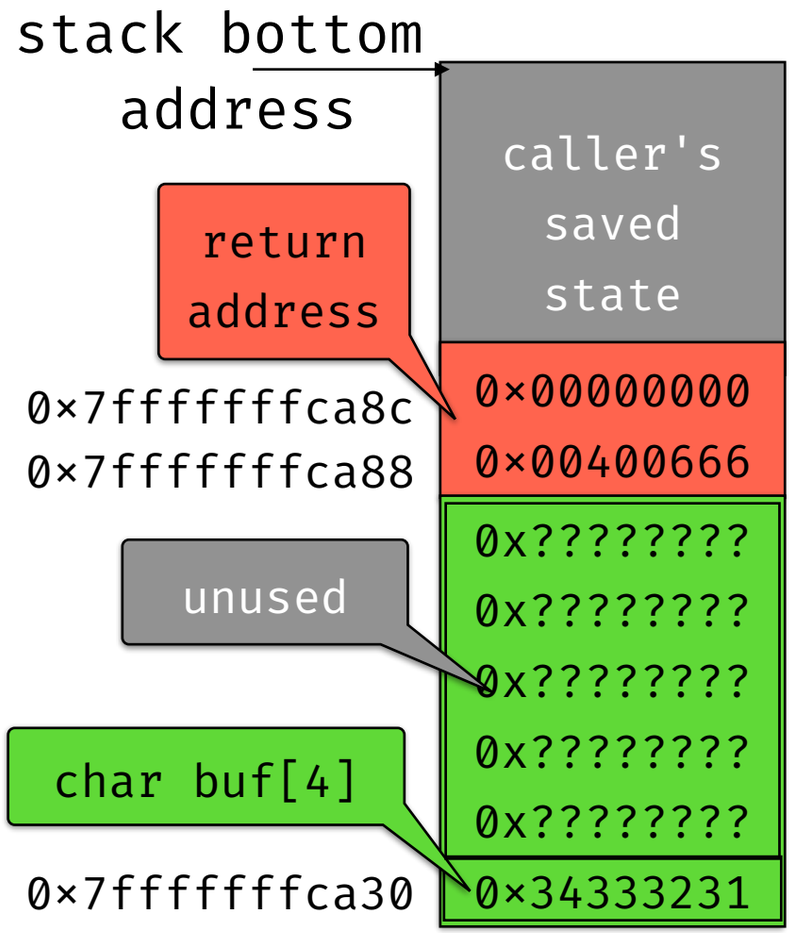
```
echo:
  0×40063f: sub $0×18,%rsp
  0×400643: mov %rsp,%rdi
   ...
  0×400653: add $0×18,%rsp
  0×400657: retq
   ...

main:
  0×40065c: mov $0,%eax
  0×400661: callq echo
  0×400666: mov $400745,%rdi
   ...
```

allocates 24 bytes for locals

pushes return address

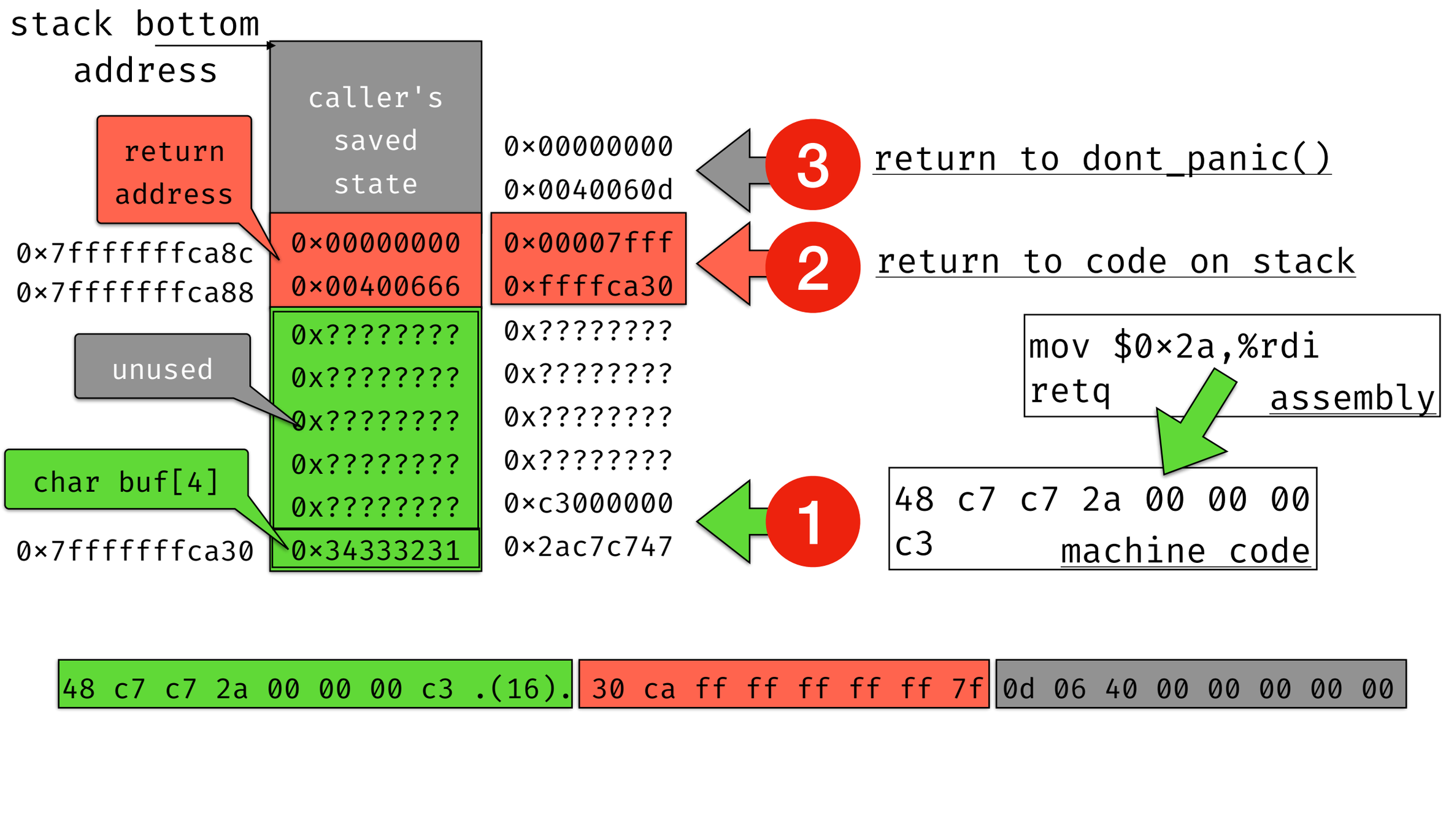


Executing Code on the Stack



1. Encode Instructions on Stack (in the buffer)
2. Change Existing Return Address (to our instructions)
3. Encode another return address (to another function)

Executing Code on the Stack



Level 3

Inject code to pass **string version of magic cookie** to **touch3()** instead of returning from **getbuf()** in **ctarget**.

- Build machine code from assembly
- Add code to the stack (inject)
- Force return to injected code
- Call **touch3(char *p)** with correct parameter

Return Oriented Programming

Injection Attacks don't work on modern software because ...

- Don't use things like **gets()**
- "data" is flagged as non-executable (the stack is data)
- Address Space Layout Randomization (ASLR)

Return Oriented Programming

But we can **jump to existing code** ...

```
400f15: c7 07 d4 48 89 c7   movl   $0×c78948d4,(%rdi)
400f1b: c3                  retq
```

This is the machine code for the `movl` instruction

This part, by itself, is a different instruction!

... all we need to do is **jmp 0×400f18** to use it!

```
400f18: 48 89 c7            movq   %rax,%rdi
400f1b: c3                  retq
```

Level 4

Repeat the same attack as Level 2 on **rtarget**

- Call **touch2(int x)** with magic cookie
- Use **gadgets** in **farm.c**

Level 5

Repeat the same attack as Level 3 on **rtarget**

- Call **touch3(char *p)** with magic cookie
- Use **gadgets** in **farm.c**
- **Significantly More Difficult**
  (add fire and brimstone effect)

The Tools

You have all the same tools you have been using thus far

- **objdump -d target** to generate asm code
- **gdb** the fabulous debugger

Plus **hex2raw** which will generate binary data files from text files for injection.

The Grade

Along with the target files you should create and commit these as your work:

- **descriptions.txt** – your attack notes, similar to Bit Bomb.

- **level1.txt**, **level2.txt**, … **levelN.txt** – your ASCII text exploit strings. One file per level.

The Grade

Your grade is based on the levels you complete and your files being committed to GitHub.

- Level 1 = 20 points
- Level 2 = 25 points
- Level 3 = 20 points
- Level 4 = 30 points
- Level 5 = 5 points

retq