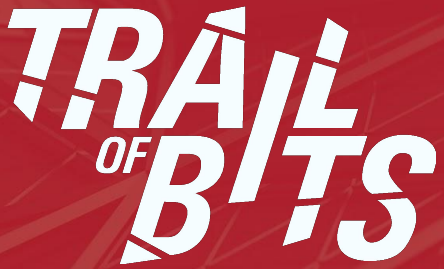


Lifting program binaries with McSema

Peter Goodman, Akshay Kumar

Introductions

Peter Goodman
Senior Security Engineer

peter@trailofbits.com

Akshay Kumar
Senior Security Engineer

akshay.kumar@trailofbits.com

Overview of this workshop (1)

- **Introduction to McSema**
  - Background on LLVM and McSema
  - Machine code, instructions, and registers
  - How machine code is lifted to LLVM bitcode
- **A vulnerable program, part 1**
  - A simple C program
  - Return-oriented-programming: taking control of execution
  - Lifting a binary to bitcode with McSema
  - Reproducing the vulnerability… or not

Overview of this workshop (2)

- **Is it safe yet? A vulnerable program, part 2**
  - Return to the same simple C program
  - Privilege escalation: how a stack-based buffer overflow vulnerability manifests at the machine code level
- **Abstraction recovery**
  - From local variables in C to stack frames
  - Lifting stack frames and recovering local variables
  - Reproducing the vulnerability… or not??

Overview of this workshop (3)

- **Wrapping it up**
  - Use-cases for McSema: when McSema is the right tool and when it's not
  - The future of McSema

Introduction to LLVM and McSema

What is LLVM bitcode?

- **Bitcode is the LLVM compiler's intermediate representation (IR)**
  - Abstract virtual machine language that is slightly lower level than C
  - Mostly architecture neutral
- **Source code translated to machine code via LLVM IR**
  - Front-ends lower source code (C, C++, etc.) to LLVM bitcode
  - Back-ends lower bitcode to machine code (x86, ppc, mips, etc.)

Why is LLVM (and its bitcode) so popular?

- **LLVM IR / bitcode is a great target for…**
  - Optimization
  - Instrumentation (AddressSanitizer, UBSan, TSan, XRay, etc.)
  - Program analysis (KLEE, S2E, Phasar, etc.)
- **LLVM is always improving**
  - Momentum!
  - Actively maintained/improved by big companies (Google, Apple, Qualcomm, etc.)
  - Actively used in research projects, which we can benefit from

From source code, to bitcode, to machine code

```
char *concat(char *a, char *b) {
  size_t a_len = strlen(a);
  size_t b_len = strlen(b);
  char *cat = malloc(a_len + b_len + 1);
  strcpy(cat, a);
  strcpy(&(cat[a_len]), b);
  return cat;
}
```

```
define i8* @concat(i8*, i8*) #0 {
  %3 = call i64 @strlen(i8* %0) #3
  %4 = call i64 @strlen(i8* %1) #3
  %5 = add i64 %3, 1
  %6 = add i64 %5, %4
  %7 = call noalias i8* @malloc(i64 %6) #4
  %8 = call i8* @strcpy(i8* %7, i8* %0) #4
  %9 = getelementptr inbounds i8, i8* %7, i64 %3
  %10 = call i8* @strcpy(i8* %9, i8* %1) #4
  ret i8* %7
}
```

```
; char *__fastcall concat(char *, char *)
public _concat
_concat proc near
push    rbp
mov     rbp, rsp
push    r15
push    r14
push    r12
push    rbx
mov     r14, rsi
mov     r15, rdi
call    _strlen
mov     r12, rax
mov     rdi, r14        ; char *
call    _strlen
lea     rdi, [r12+rax+1] ; size_t
call    _malloc
mov     rbx, rax
mov     rdi, rbx        ; char *
mov     rsi, r15        ; char *
call    _strcpy
lea     rdi, [r12+rbx]  ; char *
mov     rsi, r14        ; char *
call    _strcpy
mov     rax, rbx
pop     rbx
pop     r12
pop     r14
pop     r15
pop     rbp
retn
_concat endp
```

… and back again with McSema and FCD!

```
char *concat(char *a, char *b) {
  size_t a_len = strlen(a);
  size_t b_len = strlen(b);
  char *cat = malloc(a_len + b_len + 1);
  strcpy(cat, a);
  strcpy(&(cat[a_len]), b);
  return cat;
}
```

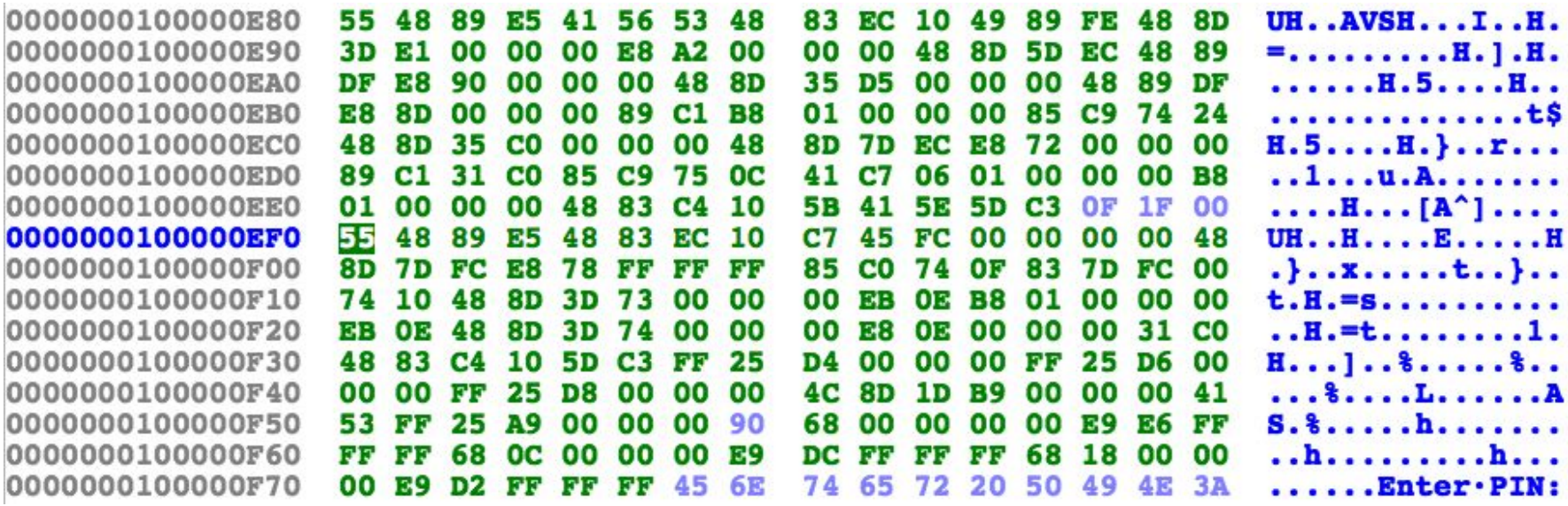
```
define i8* @concat(i8*, i8*) #0 {
  %3 = call i64 @strlen(i8* %0) #3
  %4 = call i64 @strlen(i8* %1) #3
  %5 = add i64 %3, 1
  %6 = add i64 %5, %4
  %7 = call noalias i8* @malloc(i64 %6) #4
  %8 = call i8* @strcpy(i8* %7, i8* %0) #4
  %9 = getelementptr inbounds i8, i8* %7, i64 %3
  %10 = call i8* @strcpy(i8* %9, i8* %1) #4
  ret i8* %7
}
```

```
; char *__fastcall concat(char *, char *)
public _concat
_concat proc near
push    rbp
mov     rbp, rsp
push    r15
push    r14
push    r12
push    rbx
mov     r14, rsi
mov     r15, rdi
call    _strlen
mov     r12, rax
mov     rdi, r14        ; char *
call    _strlen
lea     rdi, [r12+rax+1] ; size_t
call    _malloc
mov     rbx, rax
mov     rdi, rbx        ; char *
mov     rsi, r15        ; char *
call    _strcpy
lea     rdi, [r12+rbx]  ; char *
mov     rsi, r14        ; char *
call    _strcpy
mov     rax, rbx
pop     rbx
pop     r12
pop     r14
pop     r15
pop     rbp
retn
_concat endp
```

McSema lifts machine code to bitcode

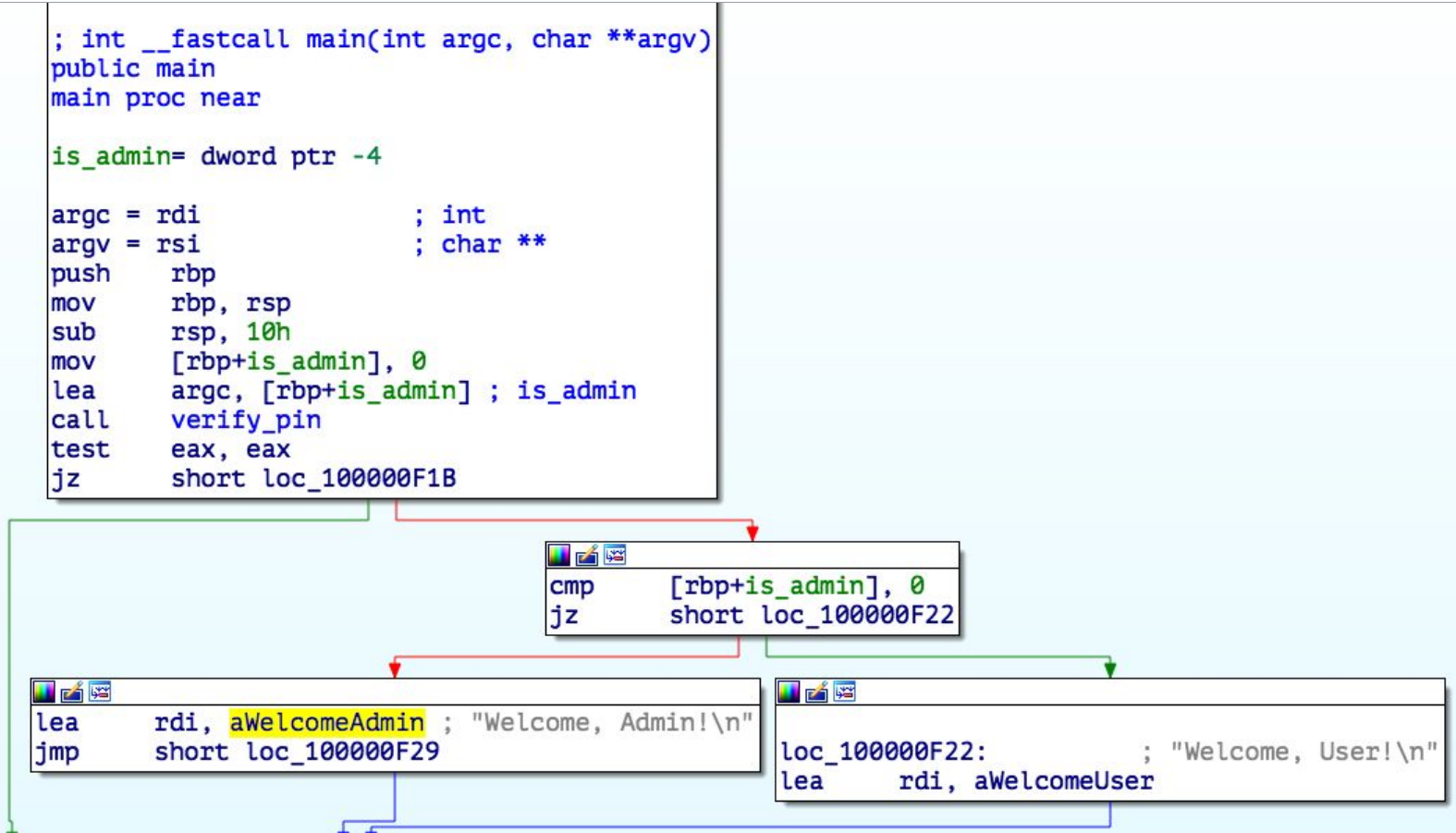
- **Why bitcode?**
  - We want to do with binaries what compilers lets us do with source code: analyze, optimize, modify, harden, etc.
- **What kinds of machine code?**
  - x86 (32-bit), amd64 (64-bit x86, a.k.a. x86-64, what most servers use)
    - Including x87 FPU, MMX, SSE, AVX, eventually AVX512
  - aarch64 (64-bit ARMv8, used by modern smartphones)
- **Works on real programs**
  - Apache httpd web server
  - Linux, macOS, and Windows

McSema lifts this stuff to bitcode

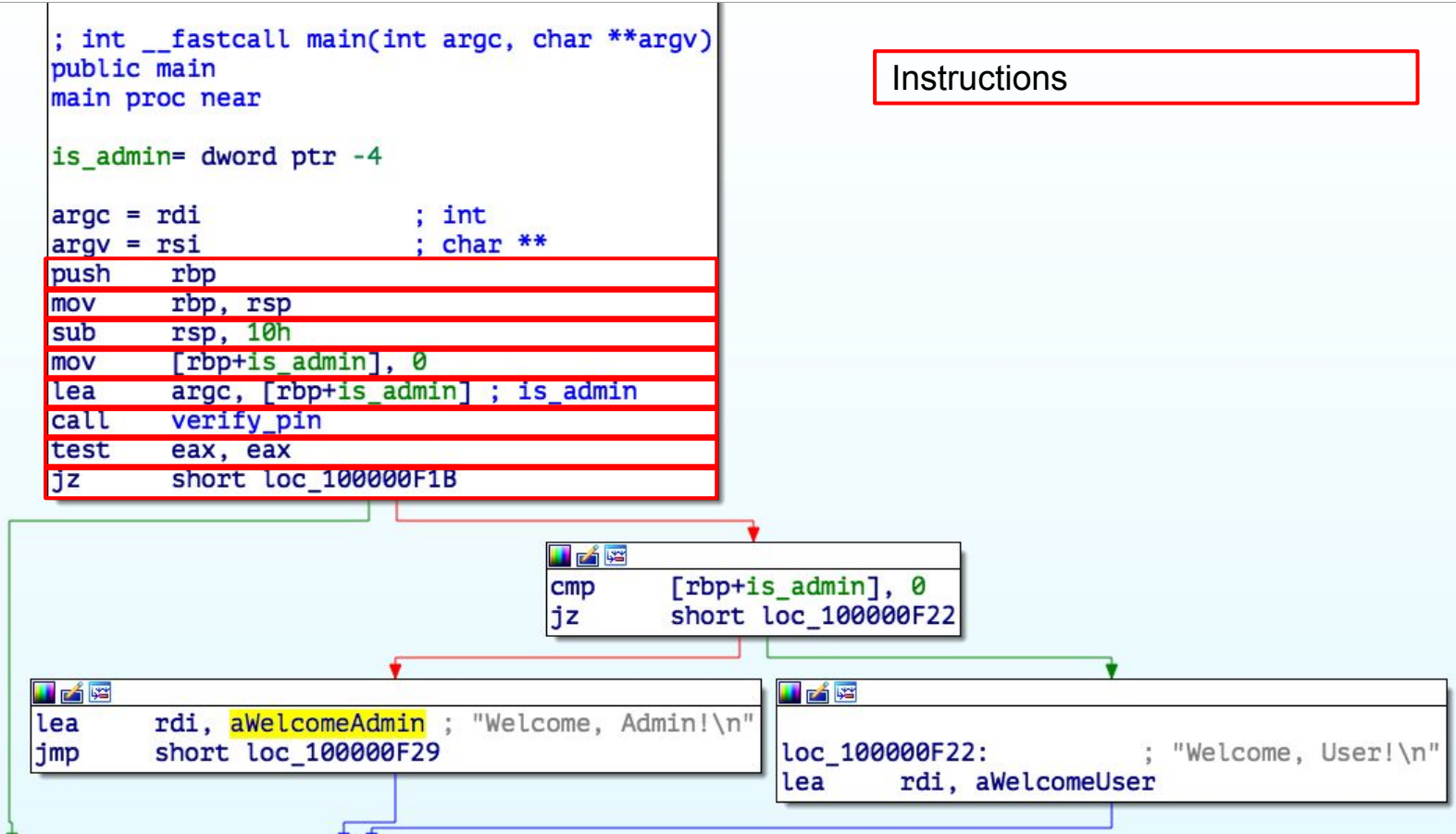


```
0000000100000E80  55 48 89 E5 41 56 53 48  83 EC 10 49 89 FE 48 8D  UH..AVSH...I..H.
0000000100000E90  3D E1 00 00 00 E8 A2 00  00 00 48 8D 5D EC 48 89  =.........H.].H.
0000000100000EA0  DF E8 90 00 00 00 48 8D  35 D5 00 00 00 48 89 DF  ......H.5....H..
0000000100000EB0  E8 8D 00 00 00 89 C1 B8  01 00 00 00 85 C9 74 24  ..............t$
0000000100000EC0  48 8D 35 C0 00 00 00 48  8D 7D EC E8 72 00 00 00  H.5....H.}..r...
0000000100000ED0  89 C1 31 C0 85 C9 75 0C  41 C7 06 01 00 00 00 B8  ..1...u.A.......
0000000100000EE0  01 00 00 00 48 83 C4 10  5B 41 5E 5D C3 0F 1F 00  ....H...[A^]....
0000000100000EF0  55 48 89 E5 48 83 EC 10  C7 45 FC 00 00 00 00 48  UH..H....E.....H
0000000100000F00  8D 7D FC E8 78 FF FF FF  85 C0 74 0F 83 7D FC 00  .}..x.....t..}..
0000000100000F10  74 10 48 8D 3D 73 00 00  00 EB 0E B8 01 00 00 00  t.H.=s..........
0000000100000F20  EB 0E 48 8D 3D 74 00 00  00 E8 0E 00 00 00 31 C0  ..H.=t........1.
0000000100000F30  48 83 C4 10 5D C3 FF 25  D4 00 00 00 FF 25 D6 00  H...]..%.....%..
0000000100000F40  00 00 FF 25 D8 00 00 00  4C 8D 1D B9 00 00 00 41  ...%....L......A
0000000100000F50  53 FF 25 A9 00 00 00 90  68 00 00 00 00 E9 E6 FF  S.%.....h.......
0000000100000F60  FF FF 68 0C 00 00 00 E9  DC FF FF FF 68 18 00 00  ..h.........h...
0000000100000F70  00 E9 D2 FF FF FF 45 6E  74 65 72 20 50 49 4E 3A  ......Enter·PIN:
```

What a binary looks like in a disassembler

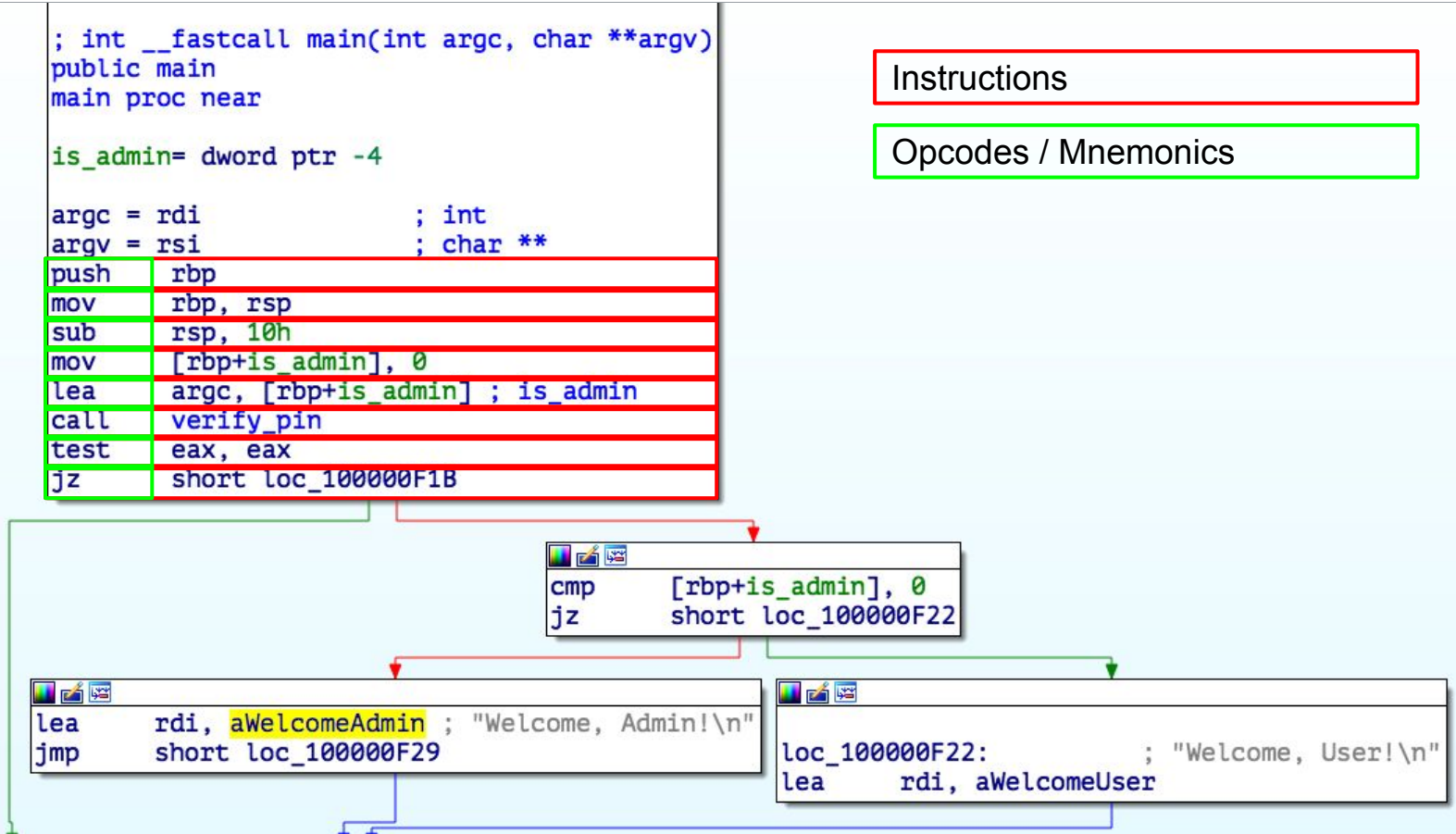


What a binary looks like in a disassembler

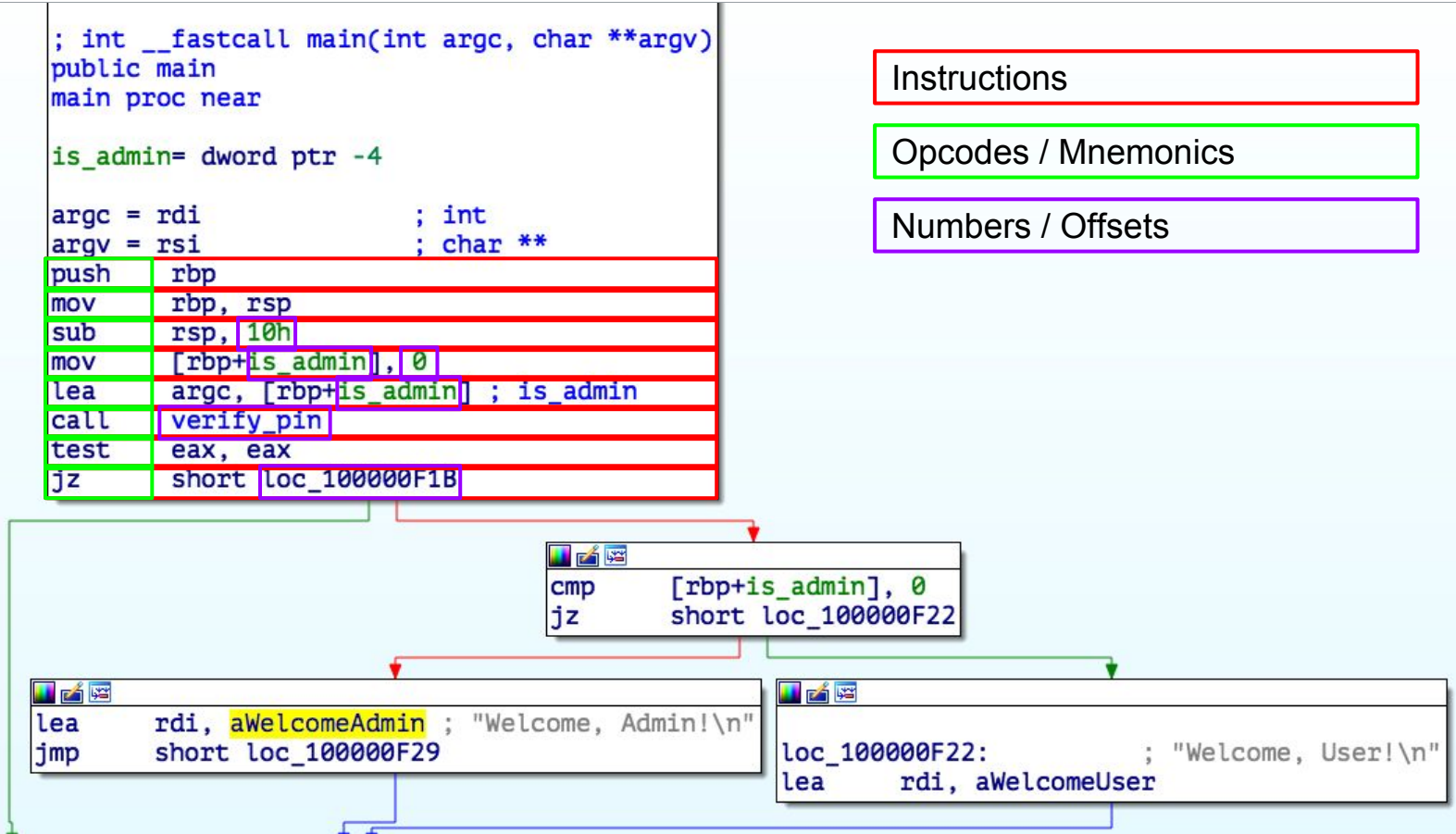


Instructions

What a binary looks like in a disassembler



What a binary looks like in a disassembler

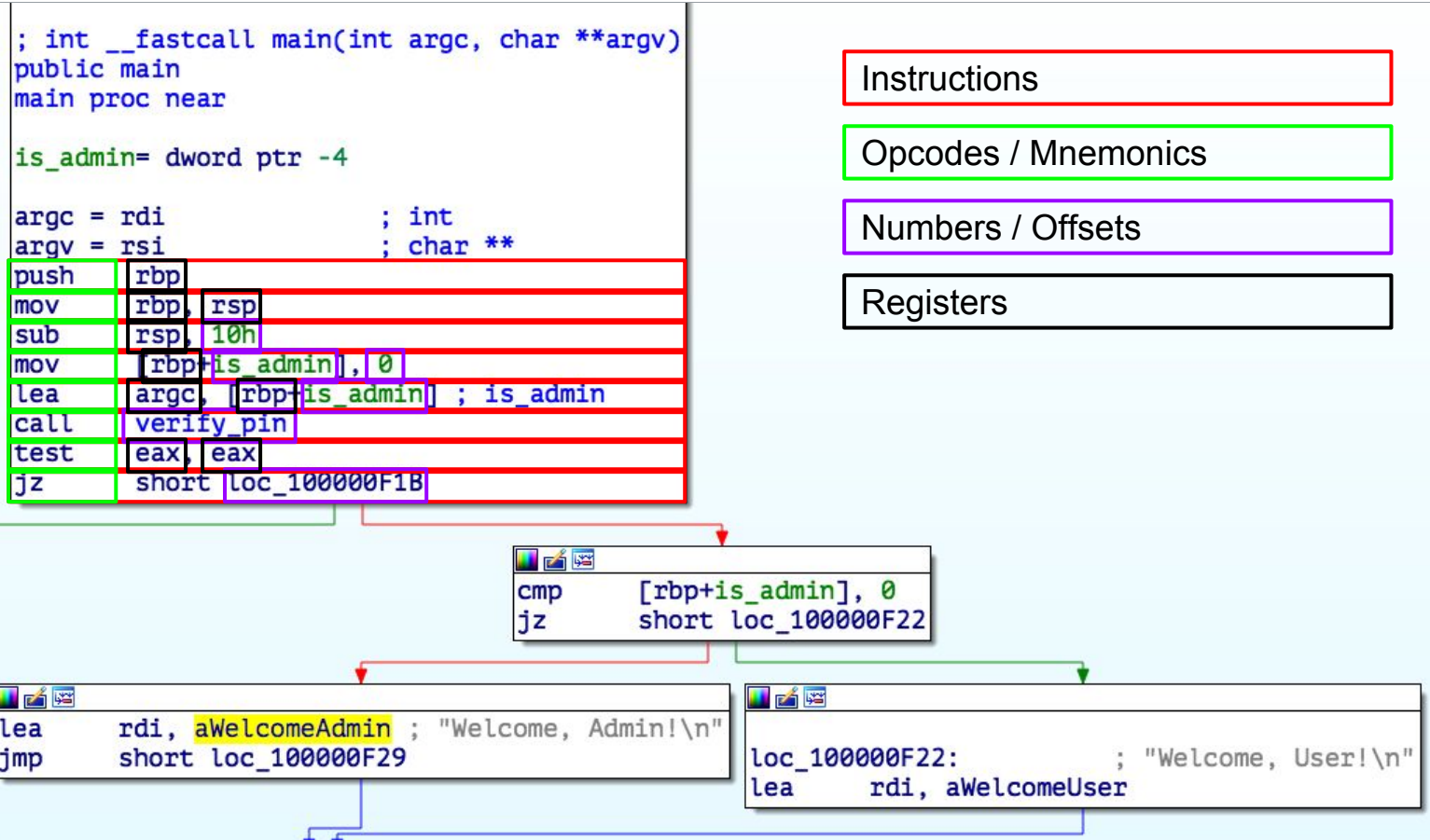


Instructions

Opcodes / Mnemonics

Numbers / Offsets

What a binary looks like in a disassembler



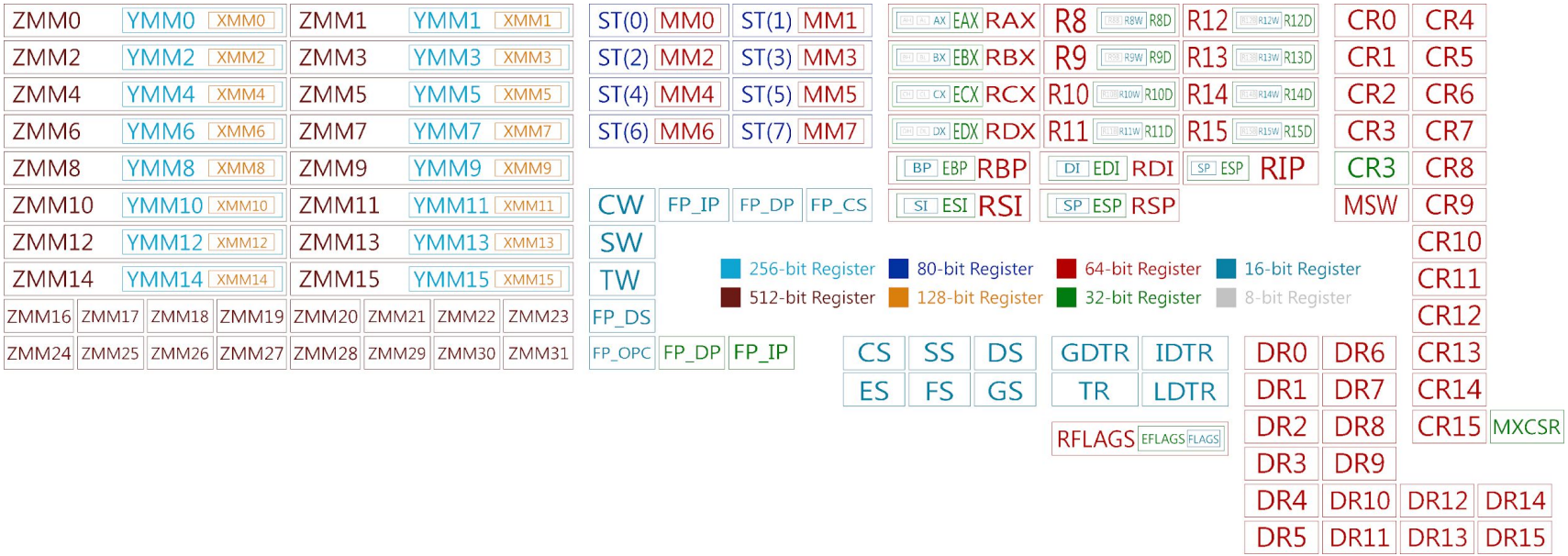
Instructions

Opcodes / Mnemonics

Numbers / Offsets

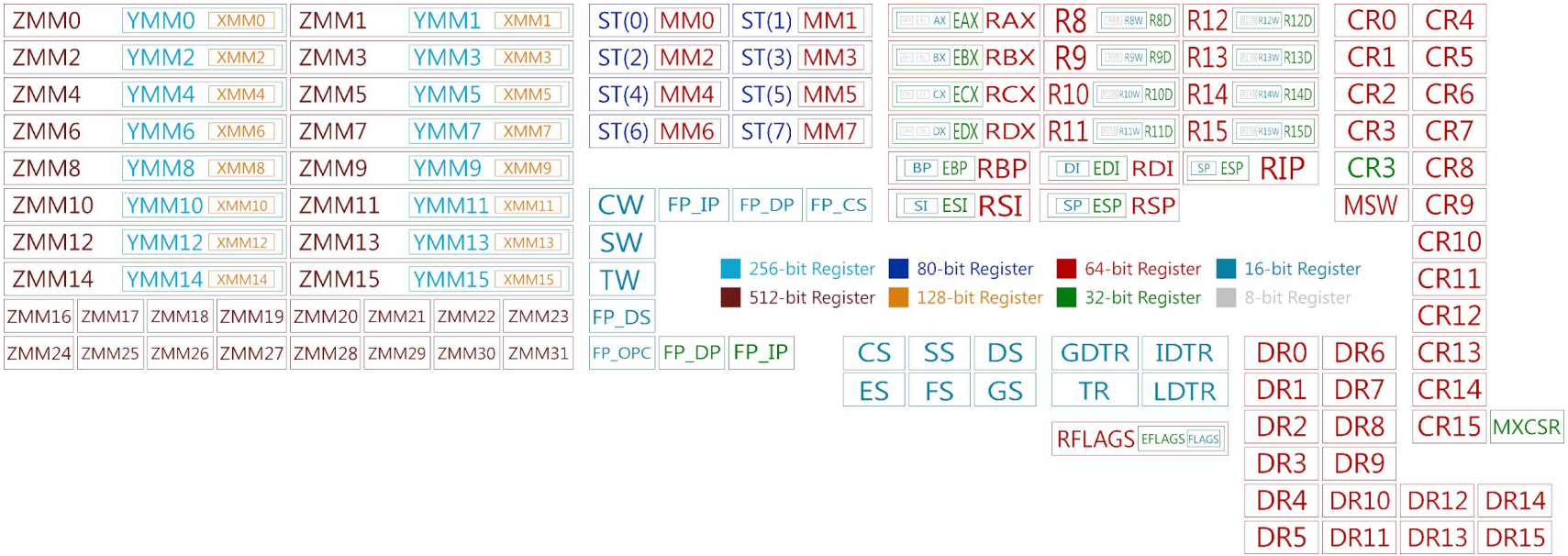
Registers

How registers are lifted to bitcode (1)



How registers are lifted to bitcode (2)

```
struct State {
```




```
};
```

How registers are lifted to bitcode (3)

```
Memory *__remill_basic_block(State &state, addr_t curr_pc, Memory *memory) {
  bool branch_taken = false;
  auto &BRANCH_TAKEN = branch_taken;

  auto &AH = state.gpr.rax.byte.high;
  auto &AL = state.gpr.rax.byte.low;
  auto &AX = state.gpr.rax.word;
  auto &EAX = state.gpr.rax.dword;
  auto &RAX = state.gpr.rax.qword;

  ...
```

How instructions are lifted to bitcode (1)

```
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-4], 0
lea     argc, [rbp-4]    ; is_admin
call    verify_pin
test    eax, eax
jz      short loc_100000F1B
```

How instructions are lifted to bitcode (2)

```
Memory *lifted_main(State &state, addr_t curr_pc, Memory *memory) {
  bool branch_taken = false;
  auto &BRANCH_TAKEN = branch_taken;
  auto &RDI = state.gpr.rdi.qword;
  auto &RBP = state.gpr.rbp.qword;
  auto &RSP = state.gpr.rsp.qword;
  auto &EAX = state.gpr.rax.dword;
  memory = PUSH<R64>(memory, state, RBP);
  memory = MOV<R64W, R64>(memory, state, &RBP, RSP);
  memory = SUB<R64W, R64, I64>(memory, state, &RSP, RSP, 0x10);
  memory = MOV<M32W, I32>(memory, state, RBP - 0x4, 0x0);
  memory = LEA<R64W, M8>(memory, state, &RDI, RBP - 0x4);
  memory = CALL<PC>(memory, state, 0x…);
  memory = lifted_verify_pin(state, …, memory);
  memory = TEST<R32, R32>(memory, state, EAX, EAX);
  memory = JZ<R8W, PC, PC>(memory, state, &BRANCH_TAKEN, …, …);
  if (BRANCH_TAKEN) {
    …
  } …
```

How instructions are lifted to bitcode (3)

```
Memory *lifted_main(State &state, addr_t curr_pc, Memory *memory) {
  bool branch_taken = false;
  auto &BRANCH_TAKEN = branch_taken;
  auto &RDI = state.gpr.rdi.qword;
  auto &RBP = state.gpr.rbp.qword;
  auto &RSP = state.gpr.rsp.qword;
  auto &EAX = state.gpr.rax.dword;
  memory = PUSH<R64>(memory, state, RBP);
  memory = MOV<R64W, R64>(memory, state, &RBP, RSP);
  memory = SUB<R64W, R64, I64>(memory, state, &RSP, RSP, 0x10);
  memory = MOV<M32W, I32>(memory, state, RBP - 0x4, 0x0);
  memory = LEA<R64W, M8>(memory, state, &RDI, RBP - 0x4);
  memory = CALL<PC>(memory, state, 0x…);
  memory = lifted_verify_pin(state, RIP, memory);
  memory = TEST<R32, R32>(memory, state, EAX, EAX);
  memory = JZ<R8W, PC, PC>(memory, state, &BRANCH_TAKEN, …, …);
  if (BRANCH_TAKEN) {
    …
  } …
```

```
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     dword ptr [rbp-4], 0
lea     argc, [rbp-4]   ; is_admin
call    verify_pin
test    eax, eax
jz      short loc_100000F1B
```

Instructions

Opcodes / Mnemonics

Numbers / Offsets

Registers

How instructions are lifted to bitcode (4)

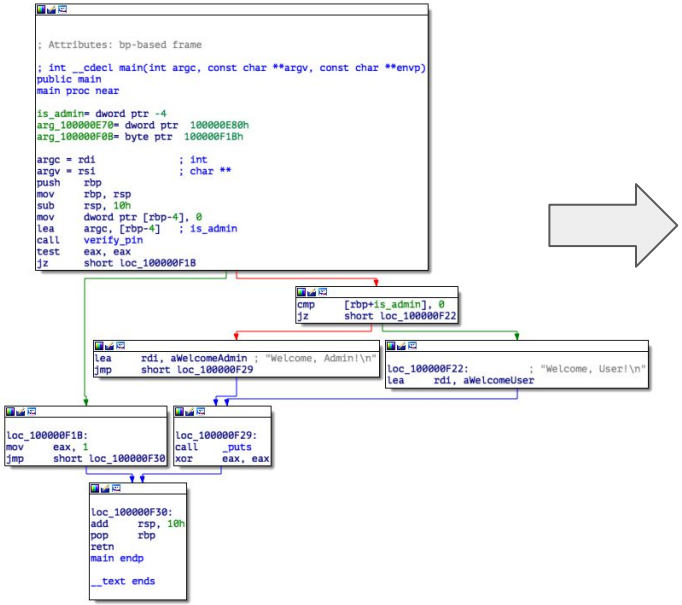
```
Memory *lifted_main(State &state, addr_t curr_pc, Memory *memory) {
  auto &RBP = state.gpr.rbp.qword;
  auto &RSP = state.gpr.rsp.qword;

  // memory = PUSH<R64>(memory, state, RBP);
  memory = __remill_write_memory(memory, RSP, RBP);
  RSP -= 8;

  // memory = MOV<R64W, R64>(memory, state, &RBP, RSP);
  RBP = RSP;

  // memory = SUB<R64W, R64, I64>(memory, state, &RSP, RSP, 0x10);
  RSP = RSP - 0x10;
  ZF = RSP == 0x0;  // Result is zero flag.
  … // More flags computations.

  // memory = MOV<M32W, I32>(memory, state, RBP - 0x4, 0x0);
  memory = __remill_write_memory_32(memory, RBP - 0x4, 0x0);
```

How instructions are lifted to bitcode (5)

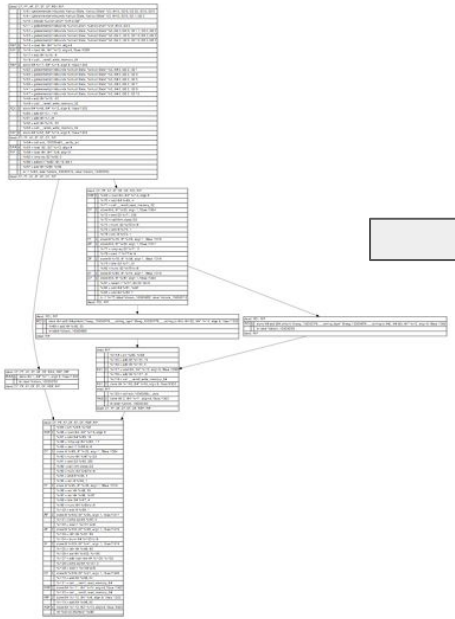
```
define %struct.Memory* @lifted_main(%struct.State*, i64, %struct.Memory*) #2 {
entry:
 …
 %10 = load i64, i64* %9, align 8
 %11 = load i64, i64* %8, align 8, !tbaa !1303
 %12 = add i64 %11, -8
 %13 = inttoptr i64 %12 to i64*
 store i64 %10, i64* %13
 store i64 %12, i64* %9, align 8, !tbaa !1299
 …
 %20 = add i64 %11, -12
 %21 = inttoptr i64 %20 to i32*
 store i32 0, i32* %21
 store i64 %20, i64* %7, align 8, !tbaa !1299
 %22 = add i64 %1, -112
 %23 = add i64 %1, 24
 %24 = add i64 %11, -32
 %25 = inttoptr i64 %24 to i64*
 store i64 %23, i64* %25
 store i64 %24, i64* %8, align 8, !tbaa !1299
 %26 = tail call %struct.Memory* @lifted_verify_pin(%struct.State* %0, i64 %22, %struct.Memory* %2)
 …
```

How instructions are lifted to bitcode (6)

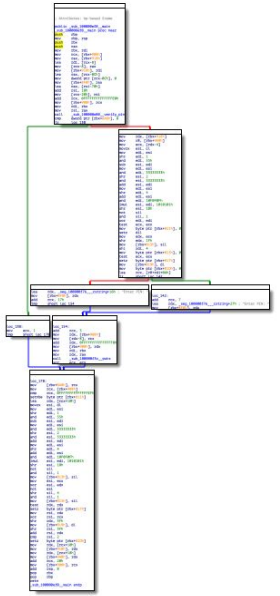
Original Binary



Lifted Bitcode



Compiled Bitcode



Now you can lift binaries too!

- **McSema does all this**
  - Interfaces with interactive disassemblers IDA Pro and Binary Ninja
  - Exports all information relevant to lifting (instructions, data, cross-references) to a “CFG file”
  - Reads CFG file and produces bitcode
- **… and more**
  - Support for C++ exceptions
  - Can lift global and local variables
  - Can compile bitcode back into runnable binaries

A vulnerable program

Time to apply our newfound knowledge

We'll start with a simple authentication program

```
$ cd ~/mcsema
$ git clone git@github.com:trailofbits/issisp-2018.git
$ cd issisp-2018
$ cat authenticate.c
```

```c
void admin_control(void);
void user_control(void);

int main(int argc, char *argv[]) {
  bool is_admin = false;
  bool is_logged = verify_pin(&is_admin);
  if (is_admin) {
    admin_control();
  } else if (is_logged) {
    user_control();
  } else {
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

```c
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
    return true;
  } else if (!strcmp(pin, "w00t")) {
    *is_admin = true;
    return true;
  } else {
    return false;
  }
}
```

What is done right, and what is wrong? (1)

BAD: Never use `gets`, no way to limit how much input is read

```
void admin_control(void);
void user_control(void);

int main(int argc, char *argv[]) {
  bool is_admin = false;
  bool is_logged = verify_pin(&is_admin);
  if (is_admin) {
    admin_control();
  } else if (is_logged) {
    user_control();
  } else {
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
    return true;
  } else if (!strcmp(pin, "w00t")) {
    *is_admin = true;
    return true;
  } else {
    return false;
  }
}
```

What is done right, and what is wrong? (2)

GOOD-ish: Make sure there's room for `gets` to replace the `\n` with a `\0` (NUL char)

```
void admin_control(void);
void user_control(void);

int main(int argc, char *argv[]) {
  bool is_admin = false;
  bool is_logged = verify_pin(&is_admin);
  if (is_admin) {
    admin_control();
  } else if (is_logged) {
    user_control();
  } else {
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
    return true;
  } else if (!strcmp(pin, "w00t")) {
    *is_admin = true;
    return true;
  } else {
    return false;
  }
}
```

What is done right, and what is wrong? (3)

BAD-ish: Not checking `is_logged && is_admin`

```
void admin_control(void);
void user_control(void);

int main(int argc, char *argv[]) {
  bool is_admin = false;
  bool is_logged = verify_pin(&is_admin);
  if (is_admin) {
    admin_control();
  } else if (is_logged) {
    user_control();
  } else {
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}
```

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
    return true;
  } else if (!strcmp(pin, "w00t")) {
    *is_admin = true;
    return true;
  } else {
    return false;
  }
}
```

Let's see the binary (1)

Back in the terminal, please compile the program

```
$ cd ~/mcsema
$ git clone git@github.com:trailofbits/issisp-2018.git
$ cd issisp-2018
$ cat authenticate.c
$ gcc -fno-stack-protector -O1 -g3 authenticate.c
```

Let's see the binary (2)

Back in the terminal, please compile the program

```
$ cd ~/mcsema
$ git clone git@github.com:trailofbits/issisp-2018.git
$ cd issisp-2018
$ cat authenticate.c
$ gcc -fno-stack-protector -O1 -g3 authenticate.c
```

Let’s see the binary (3)

Let’s test it out

```
$ cd ~/mcsema
$ git clone git@github.com:trailofbits/issisp-2018.git
$ cd issisp-2018
$ cat authenticate.c
$ gcc -fno-stack-protector -O1 -g3 authenticate.c
$ ./a.out
```

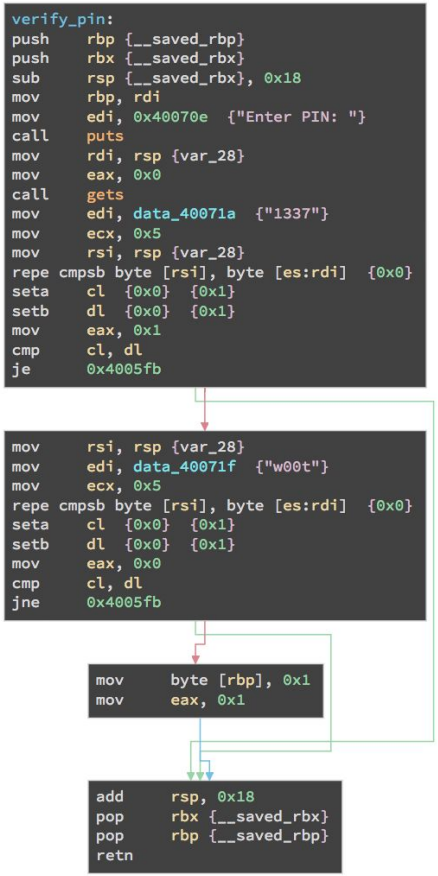
```
ehlo
1337
w00t
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
```

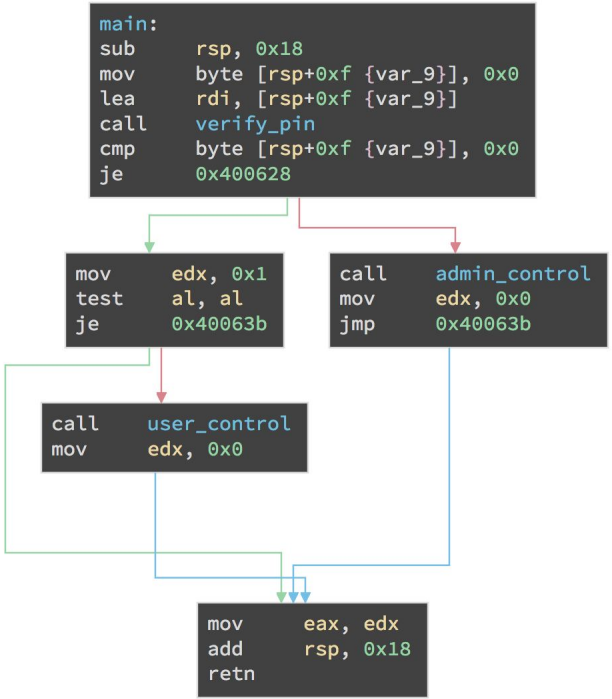
Let's see the binary (4)

Open a.out in Binary Ninja

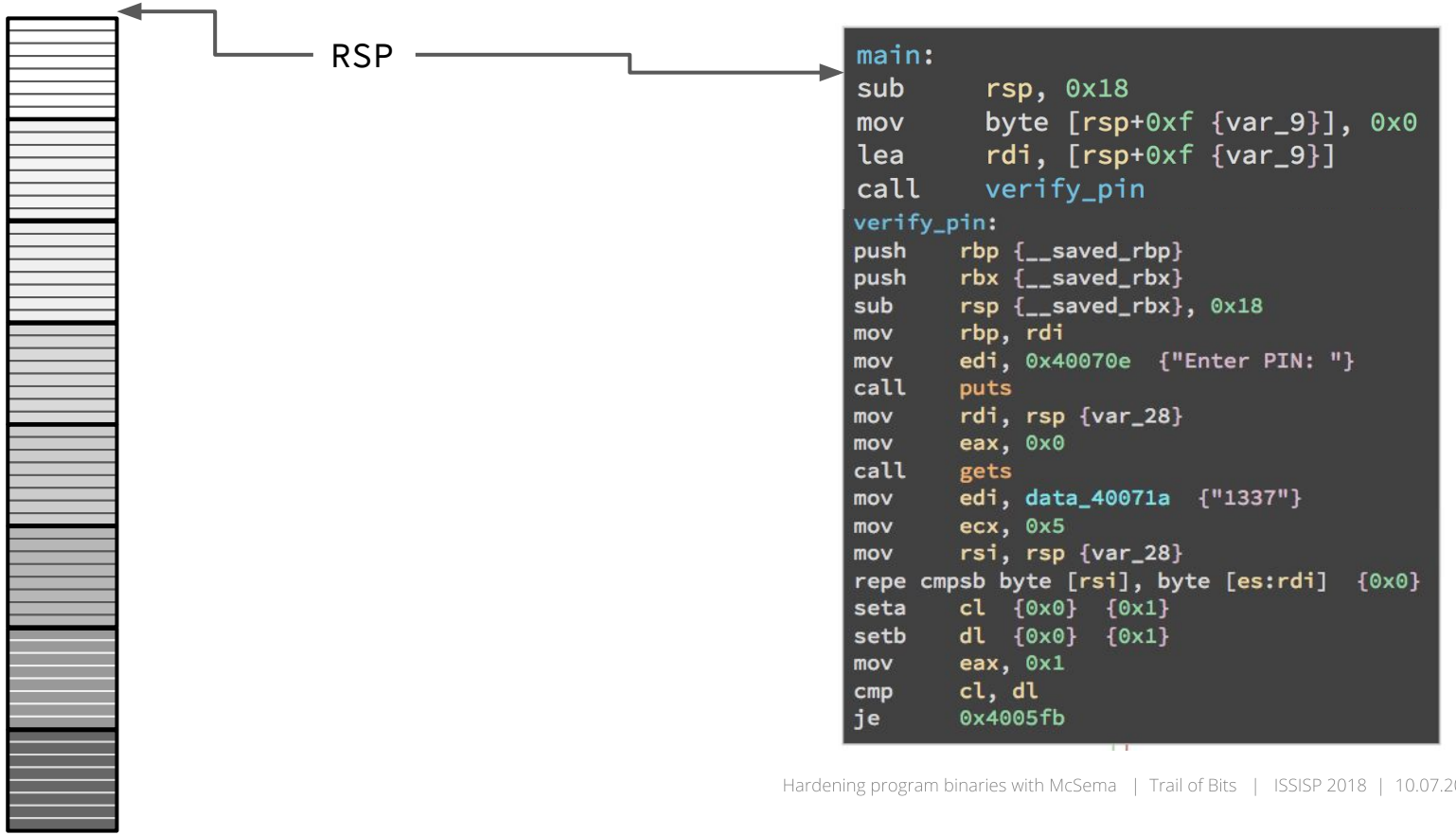
```
$ cd ~/mcsema
$ git clone git@github.com:trailofbits/issisp-2018.git
$ cd issisp-2018
$ cat authenticate.c
$ gcc -fno-stack-protector -O1 -g3 authenticate.c
$ ./a.out
$ /opt/binaryninja/binaryninja ~/mcsema/issisp-2018/a.out
```

Let's see the binary (4)

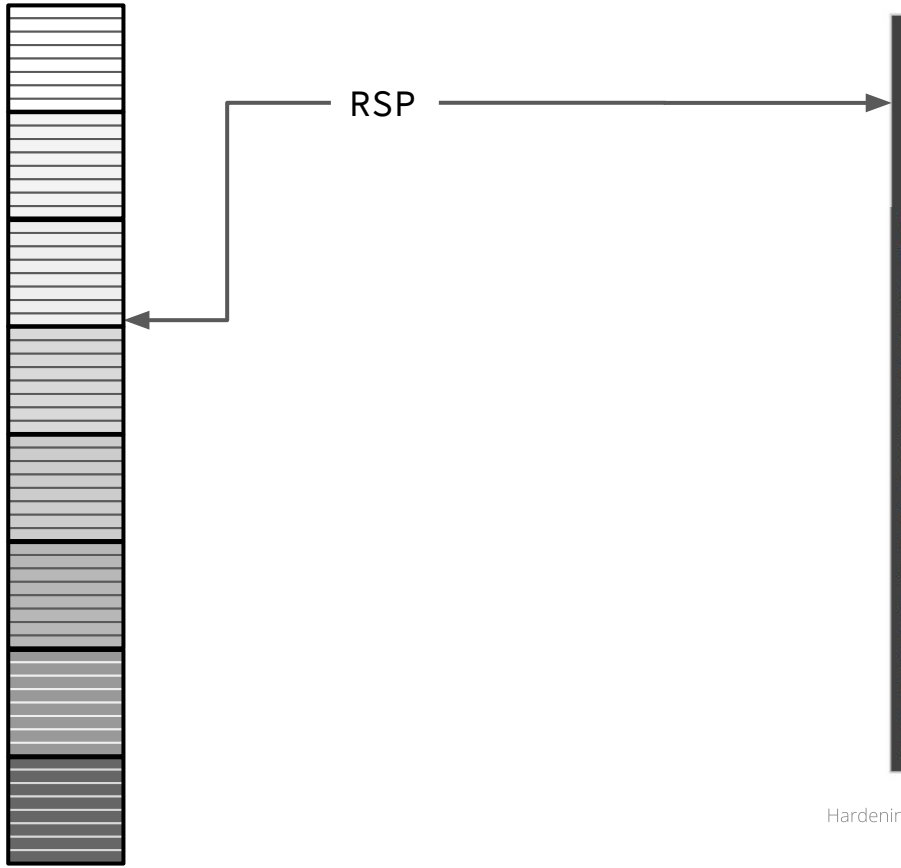




What happens when this code executes (1)

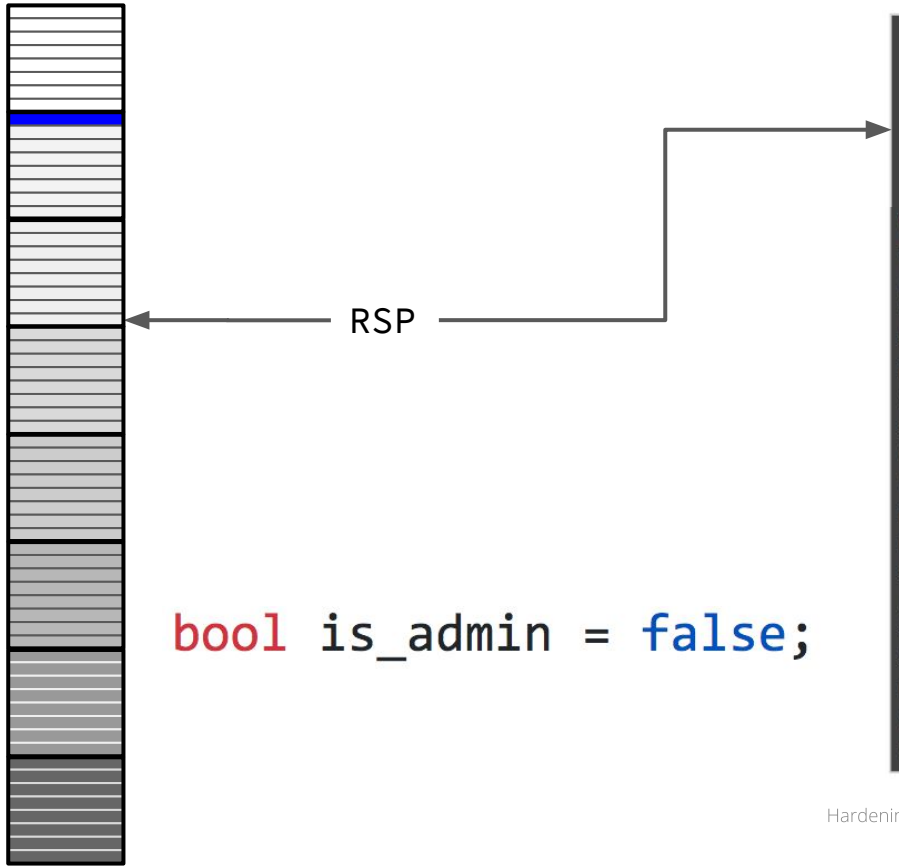


What happens when this code executes (2)




```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

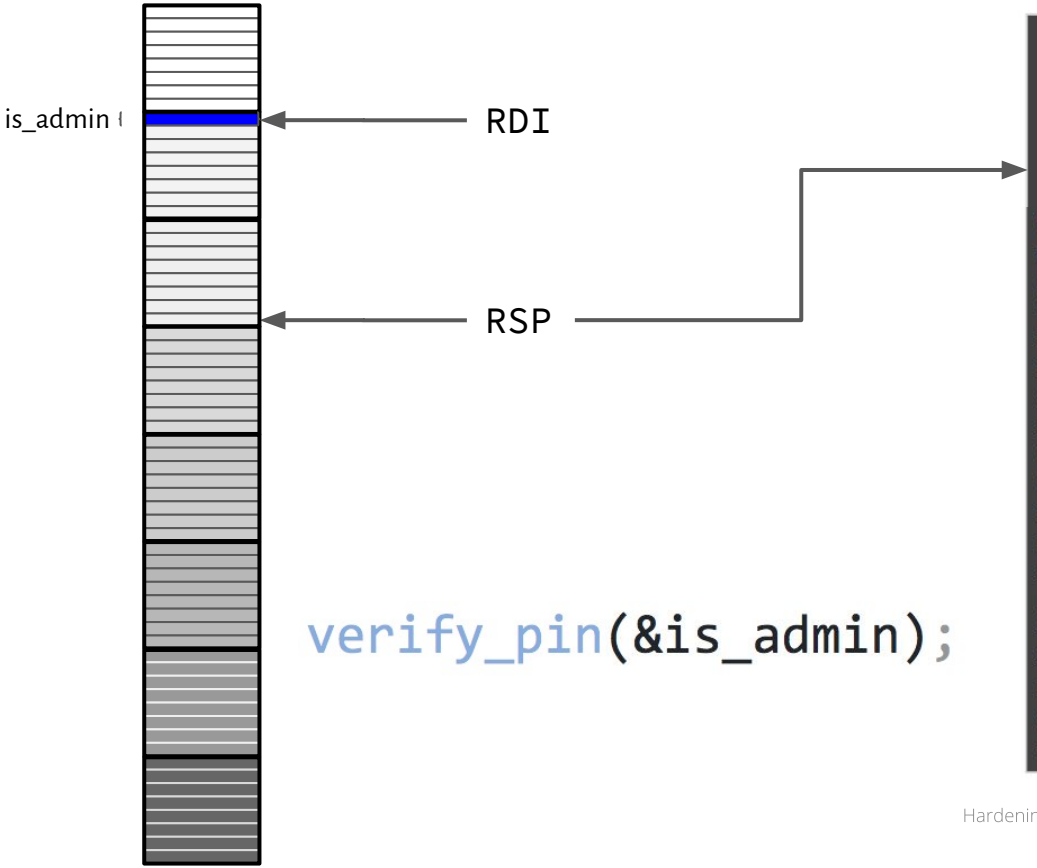
What happens when this code executes (3)




```
bool is_admin = false;
```

```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

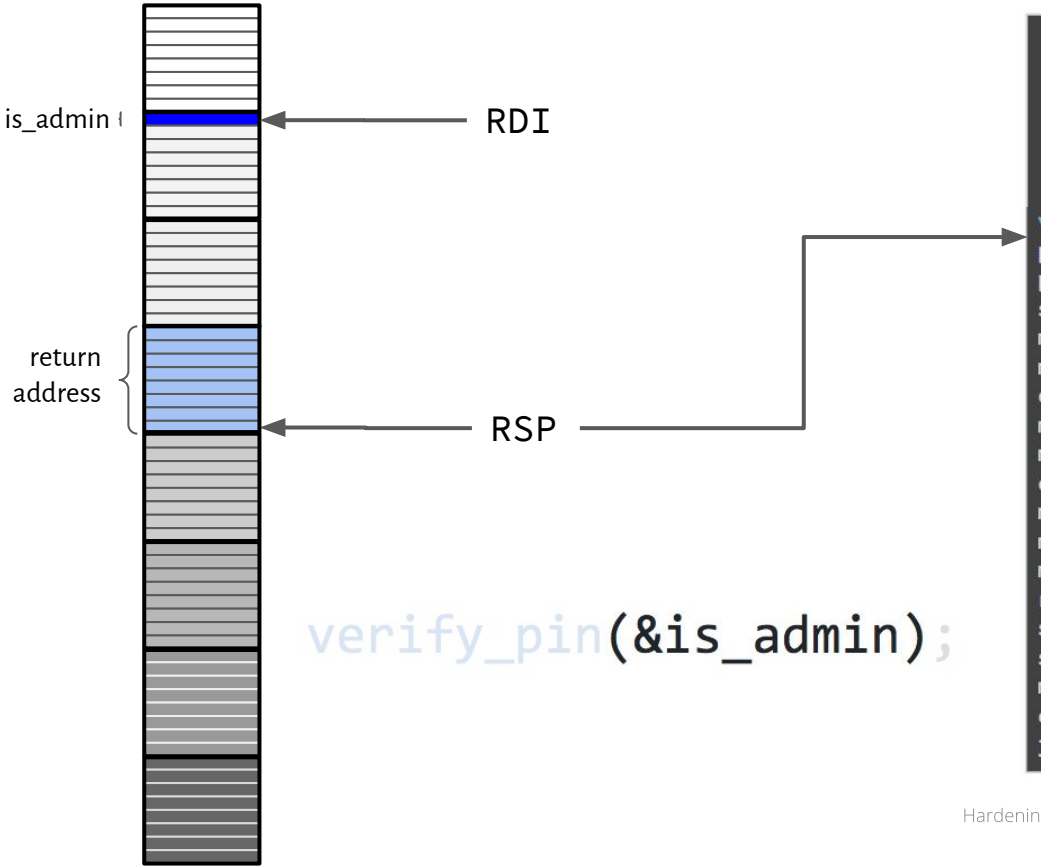
What happens when this code executes (4)




```
verify_pin(&is_admin);
```

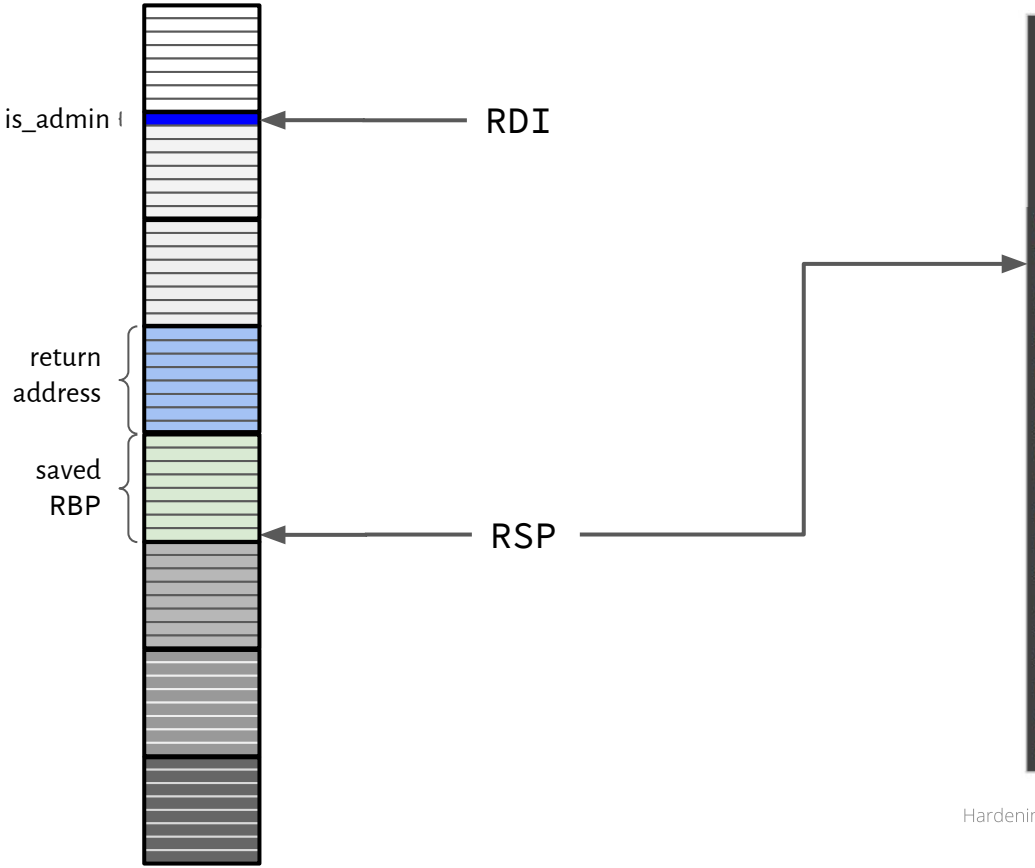
```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

What happens when this code executes (5)



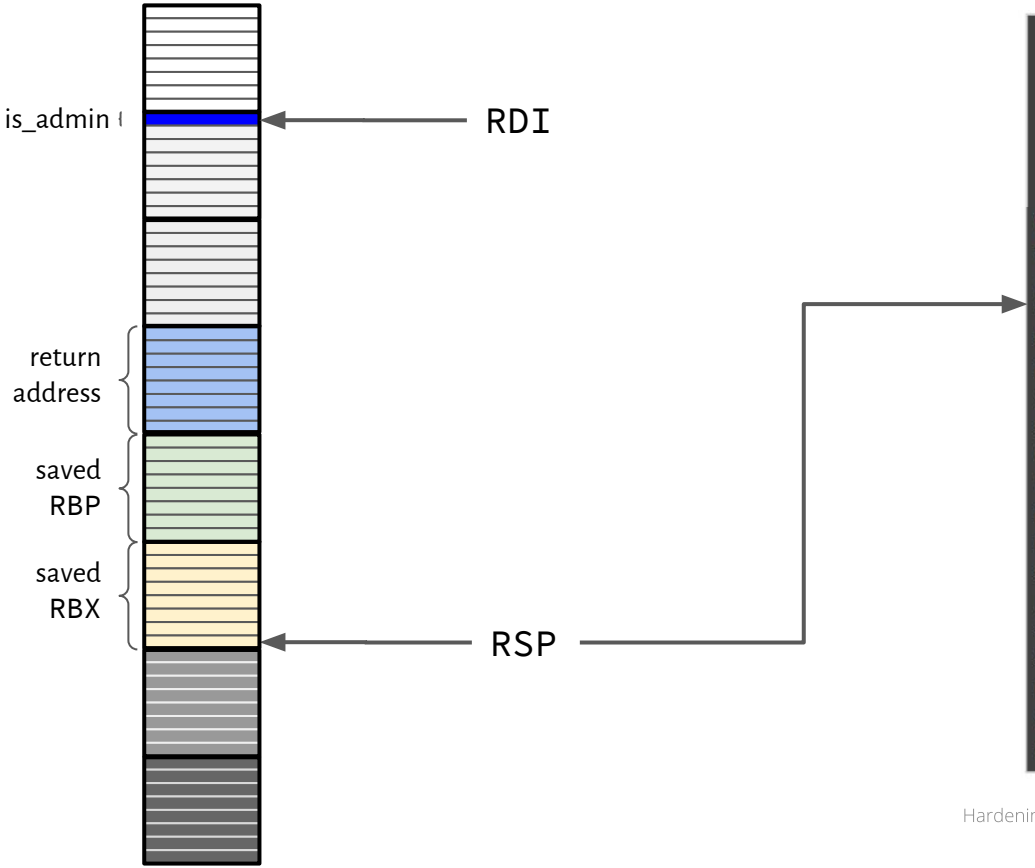

```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

What happens when this code executes (6)



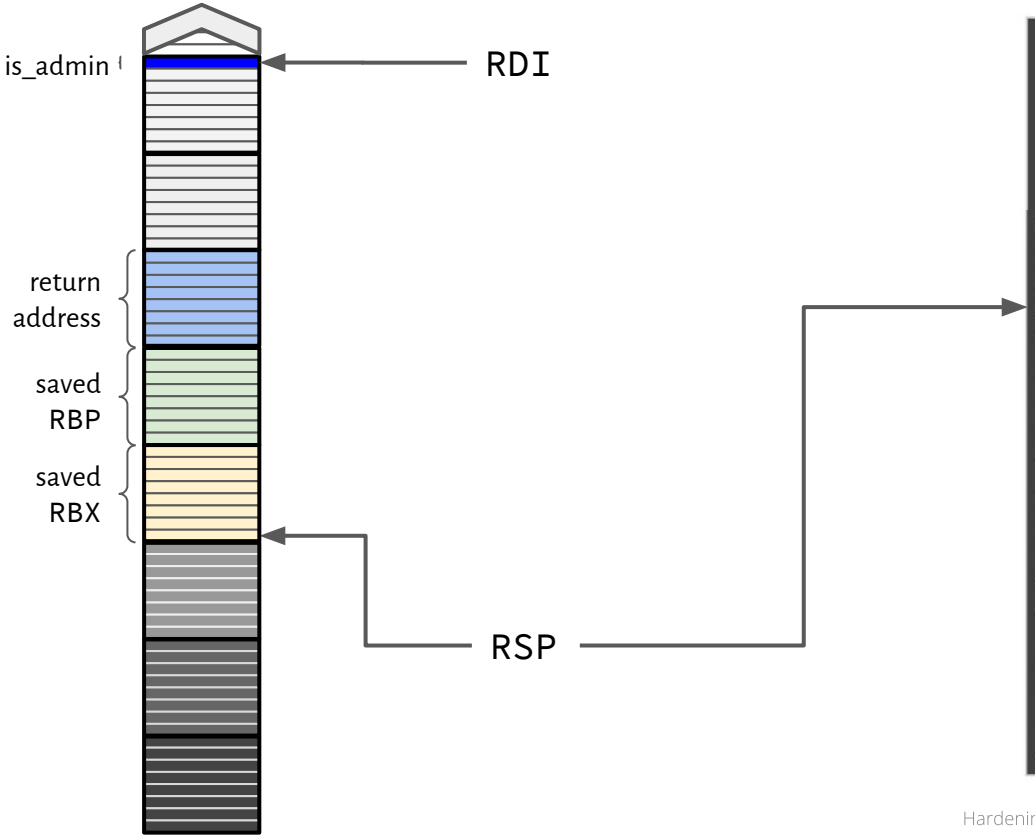

```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

What happens when this code executes (7)



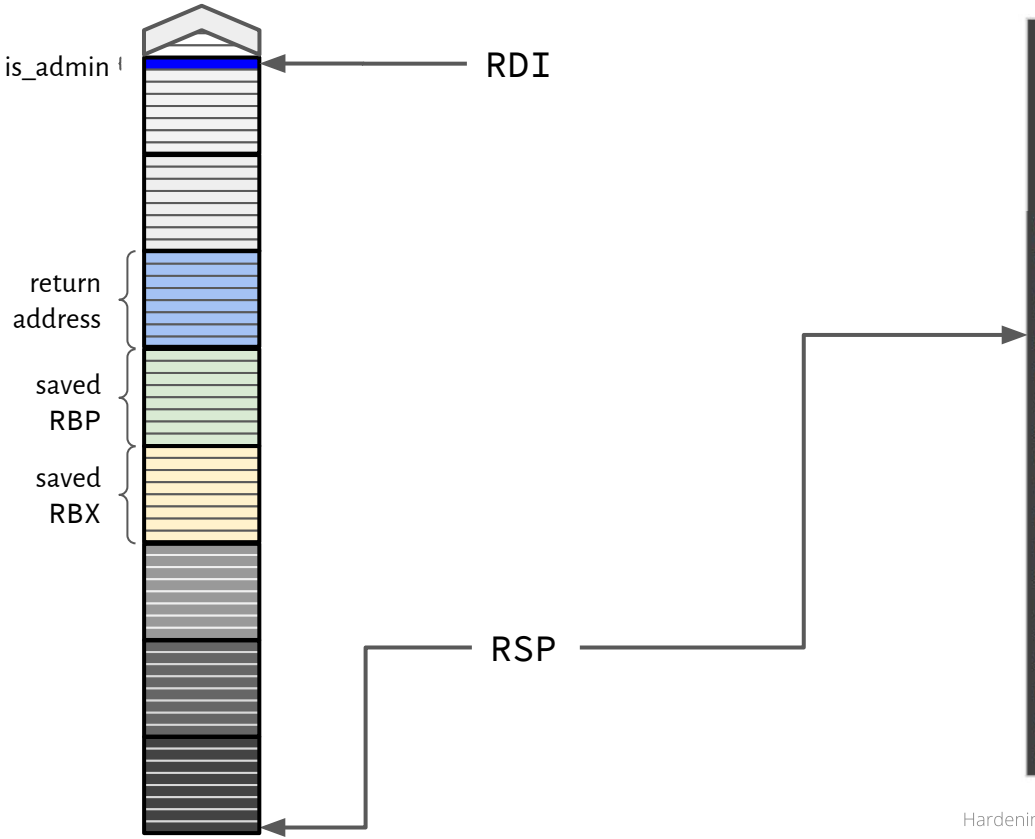

```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

What happens when this code executes (8)



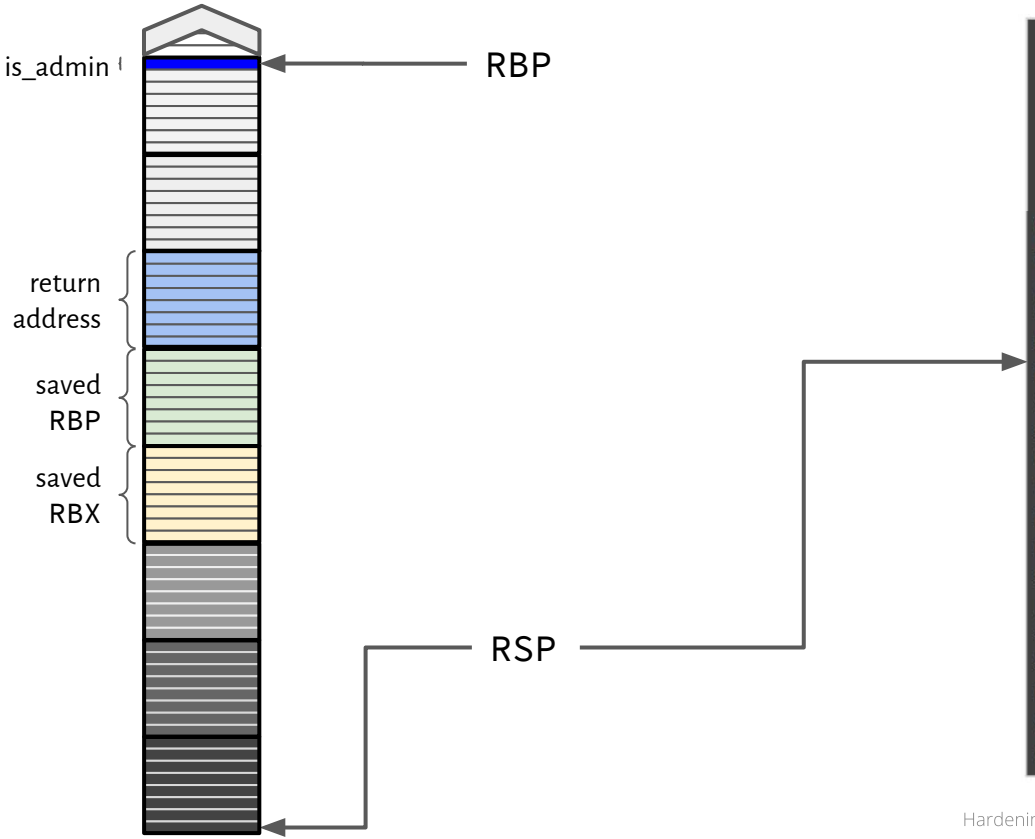

```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

What happens when this code executes (9)



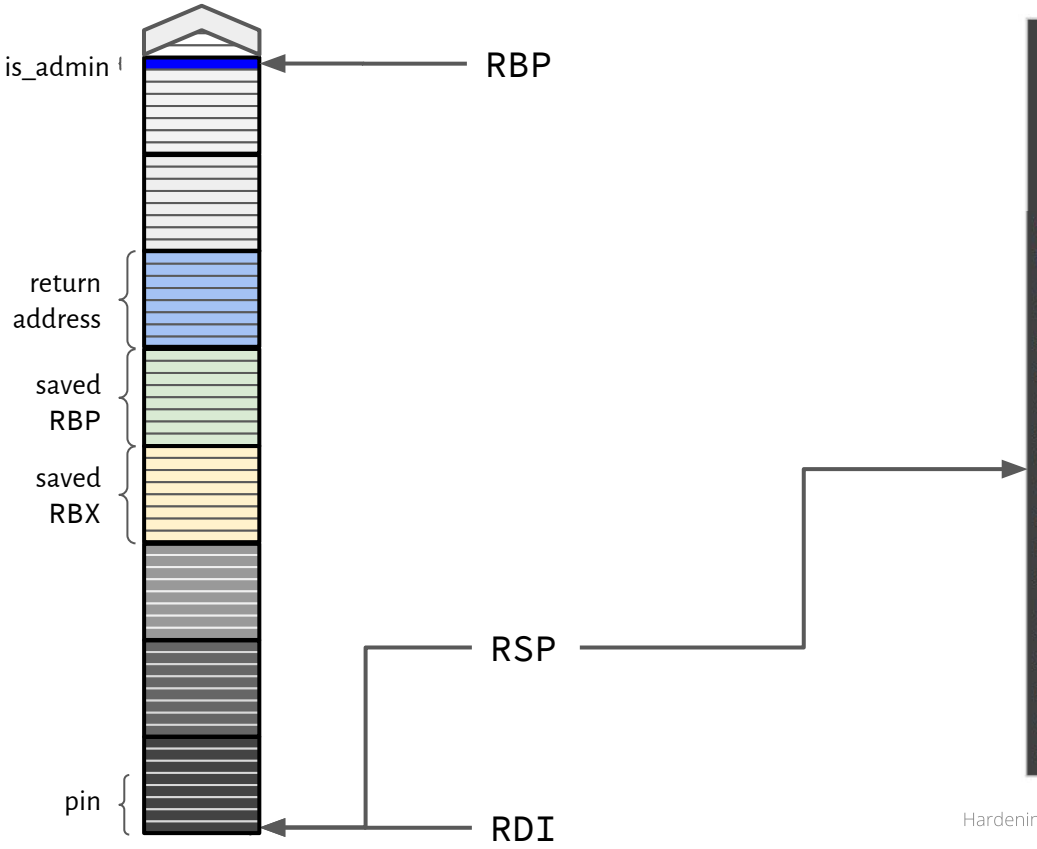

```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

What happens when this code executes (10)




```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

What happens when this code executes (11)




```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

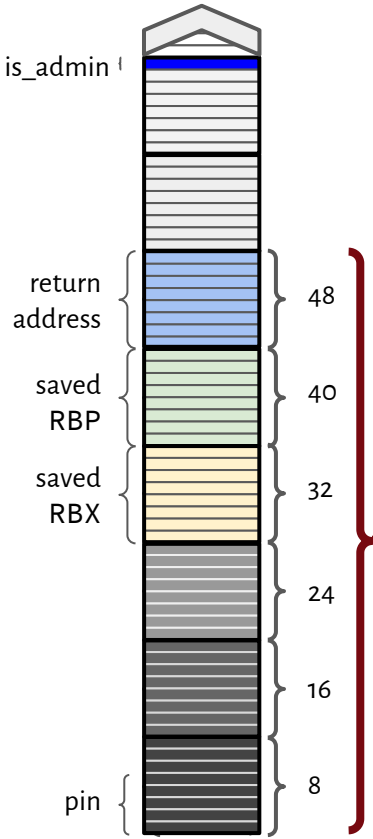
Return-oriented-programming (1)

- **Procedures are linked together by return addresses on the stack**
  - Calling a function pushes a “return to here” address on the stack; return instruction pops the saved address off the stack and jumps to it
- **What if we overwrote a return address?**
  - We could take control of execution when the function returns!

Return-oriented-programming (2)

- **How can we overwrite a return address?**
  - Key idea: Need a way to get enough user-controller input written onto the stack so that eventually that input overwrites a saved return address
- **`gets(pin)` is our unsafe weapon of choice**
  - `pin` is a stack-allocated `char` array, so it decays into a pointer into the stack
  - `gets(pin)` can write an *unbounded* amount of data to the stack

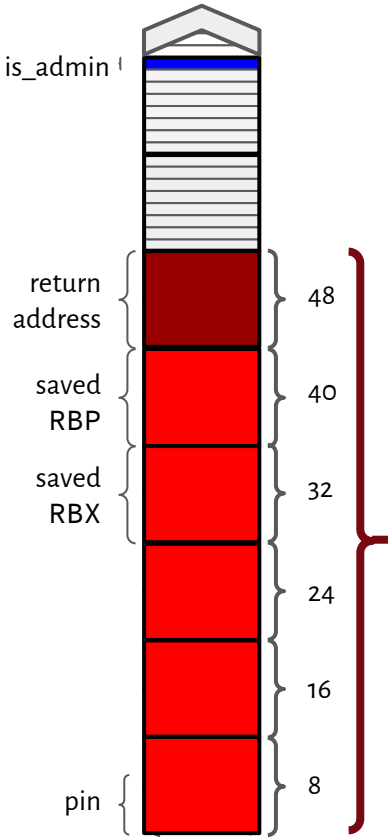
Attack of the binary ninja: theory



Goal: Overwrite these bytes

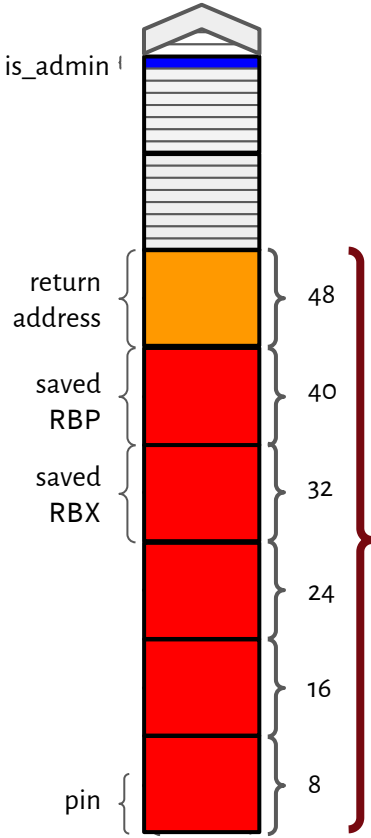
```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
        eax, 0x0
        gets
        edi, data_40071a  {"1337"}
        ecx, 0x5
        rsi, rsp {var_28}
        psb byte [rsi], byte [es:rdi]  {0x0}
        cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

Attack of the binary ninja: first strike




```
$ python
>>> import struct
>>> chr(0) * 40 + struct.pack("<Q", 0x41414141)
'\x00\x00\x00…\x41\x41\x41\x41\x00\x00\x00\x00'
>>> open("input", "w").write(_)
>>> exit()

$ ./a.out <input
Enter PIN:
Segmentation fault

$ dmesg | tail -n 1
[…] a.out[…]: segfault at 41414141 ip
0000000041414141 sp …
```

Attack of the binary ninja: fatal blow



Address of `admin_control`

```
$ python
>>> import struct
>>> chr(0) * 40 + struct.pack("<Q", 0x400602)
'\x00\x00\x00…\x02\x06\x40\x00\x00\x00\x00\x00'
>>> open("input", "w").write(_)
>>> exit()

$ ./a.out <input
Enter PIN:
Welcome, Admin!

You have the power!!!!!

Segmentation fault
```

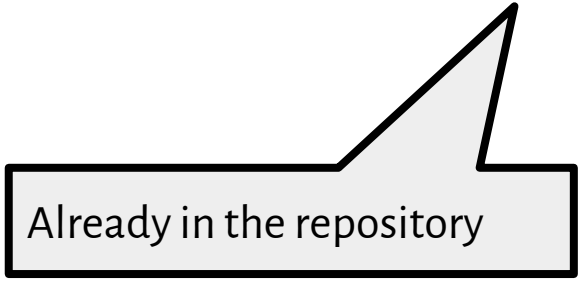
Mitigating ROP with McSema

- **McSema automatically (automagically?) mitigates this vulnerability**
  - Lifted bitcode emulates all operations, including stack memory accesses
  - Lifted functions call lifted functions… so where are return addresses between (compiled) lifted functions stored?
- **(Compiled) Lifted bitcode executes on a second stack**
  - Lifted bitcode will emulate the return address overwrite on the *emulated* stack
  - Lifted `verify_pin` will return to the lifted `main`

Let's lift the binary (1)

Disassemble a.out (using Binary Ninja) into a CFG file

```
$ mcsema-disass --arch amd64 --os linux \
                --disassembler /opt/binaryninja/binaryninja \
                --binary a.out --entrypoint main \
                --output authenticate.cfg
```



Let's lift the binary (2)

Lift authenticate.cfg to LLVM bitcode

```
$ mcsema-disass --arch amd64 --os linux \
                --disassembler /opt/binaryninja/binaryninja \
                --binary a.out --entrypoint main \
                --output authenticate.cfg
$ mcsema-lift-6.0 \
    --arch amd64 --os linux --cfg authenticate.cfg \
    --output authenticate.bc --abi_libraries libc
```

Let's lift the binary (3)

Compile authenticate.bc to a new program with Clang

```
$ mcsema-disass --arch amd64 --os linux \
                --disassembler /opt/binaryninja/binaryninja \
                --binary a.out --entrypoint main \
                --output authenticate.cfg
$ mcsema-lift-6.0 \
    --arch amd64 --os linux --cfg authenticate.cfg \
    --output authenticate.bc --abi_libraries libc
$ remill-clang-6.0 -o a.out.lifted authenticate.bc \
                   /lib/libmcsema_rt64-6.0.a
```

Run before you can walk!

Run a.out.lifted with the exploit input

```
$ mcsema-disass --arch amd64 --os linux \
                --disassembler /opt/binaryninja/binaryninja \
                --binary a.out --entrypoint main \
                --output authenticate.cfg
$ mcsema-lift-6.0 \
    --arch amd64 --os linux --cfg authenticate.cfg \
    --output authenticate.bc --abi_l
$ remill-clang-6.0 -o a.out.lifted a
                   /lib/libmcsema_rt
$ ./a.out.lifted <input
Enter PIN:
$
```

What happened?

The crashing input uses zeroes for its PIN, which doesn't log the user in, so the lifted program exits normally!

Mitigated, but not fixed

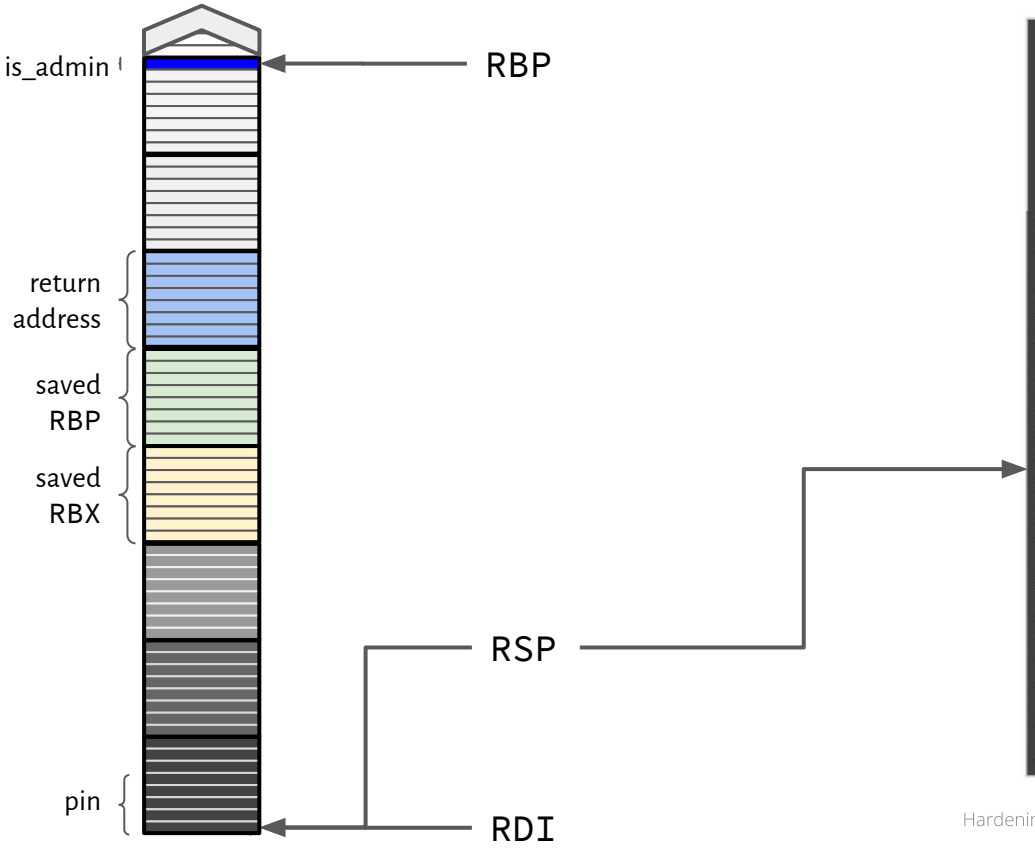
- **Lifting the program with McSema mitigated the ROP exploit**
  - Return addresses on the emulated stack are ignored, so an exploit that overwrites the return addresses is ineffectual
  - Stack on which lifted bitcode executes behaves like a safe or shadow stack
- **The program still uses `gets`**
  - Is it program still vulnerable?

The program is still vulnerable

We're safe, right?

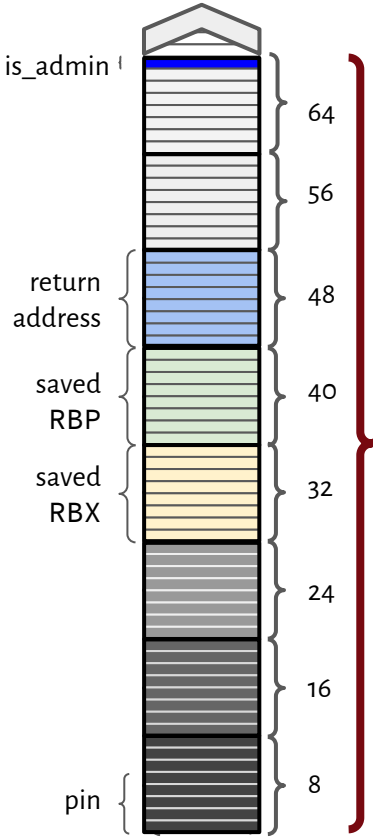
- **McSema mitigates the ROP vulnerability**
  - Lifted bitcode emulates all operations, including the stack operations
  - The return address overwrite happens on the emulated stack
  - Lifted `verify_pin` returns to the lifted `main` safely
- **What can go wrong?**
  - McSema allocates the emulated stack as flat memory array
  - Lifted local variable `is_admin` and `pin` gets allocated on the same stack
  - Buffer overflow on the emulated stack will still overwrite `is_admin`

There is another exploitation opportunity (1)




```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
mov     edi, 0x40070e  {"Enter PIN: "}
call    puts
mov     rdi, rsp {var_28}
mov     eax, 0x0
call    gets
mov     edi, data_40071a  {"1337"}
mov     ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

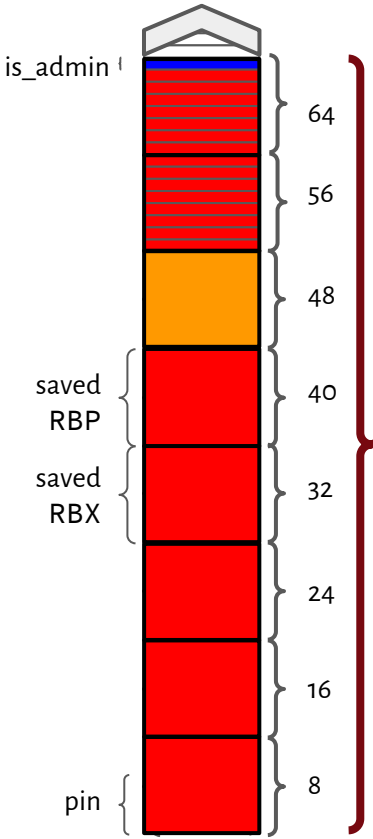
There is another exploitation opportunity (1)



What if we overwrite `is_admin`?

```
main:
sub     rsp, 0x18
mov     byte [rsp+0xf {var_9}], 0x0
lea     rdi, [rsp+0xf {var_9}]
call    verify_pin
verify_pin:
push    rbp {__saved_rbp}
push    rbx {__saved_rbx}
sub     rsp {__saved_rbx}, 0x18
mov     rbp, rdi
        edi, 0x40070e  {"Enter PIN: "}
        puts
        rdi, rsp {var_28}
        eax, 0x0
        gets
        edi, data_40071a  {"1337"}
        ecx, 0x5
mov     rsi, rsp {var_28}
repe cmpsb byte [rsi], byte [es:rdi]  {0x0}
seta    cl  {0x0}  {0x1}
setb    dl  {0x0}  {0x1}
mov     eax, 0x1
cmp     cl, dl
je      0x4005fb
```

Attack of the binary ninja: second strike




```
$ python
>>> import struct
>>> chr(0) * 40 + struct.pack("<Q", 0x0400615) +
chr(0) * 15 + chr(1)
'\x00\x00\x00…\x00\x15\x06\x40…\x00\x00\x00\x01'
>>> open("input", "w   write(_)
>>> exit()

$ ./a.out <input
Enter PIN:
Welcome, Admin!

You have the power!!!!!
```

Return address from `verify_pin`

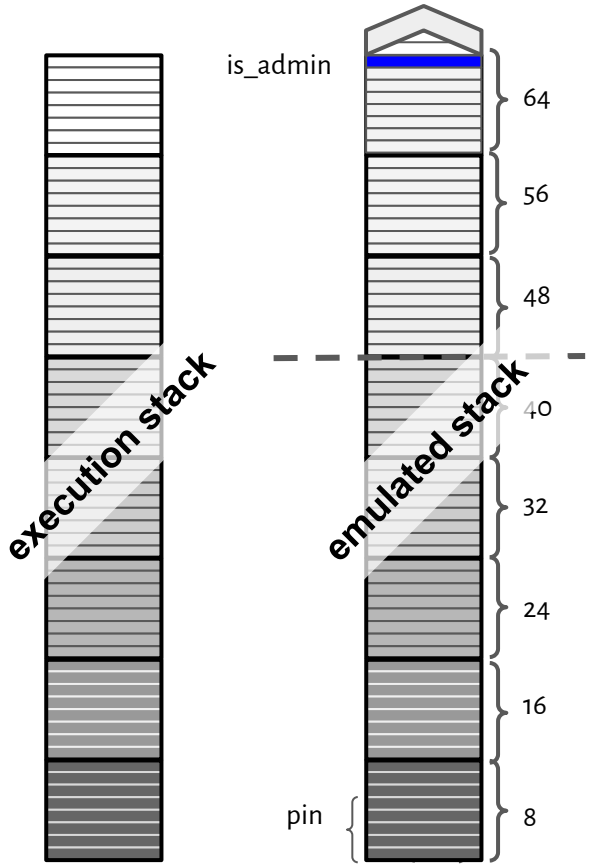
Set `is_admin = true`

The lifted program is also vulnerable!

```
define %struct.Memory* @sub_400602_main(%struct.State*, i64, %struct.Memory*) #2 {
entry:
  %8 = getelementptr inbounds %struct.State, %struct.State* %state, i64 0, i32 6, i32 13, i32 0, i32 0
  %RSP = load i64, i64* %8, align 8
  %16 = add i64 %RSP, -9
  %17 = intto
  store i8 0,
  %20 = add
  store i64 %
  %22 = call                                                          %18, %struct.Memory* %2)
  …

define %struc                                                         {
entry:
  …
  %13 = getelementptr inbounds %struct.State, %struct.State* %state, i64 0, i32 0, i32 13, i32 0, i32 0
  %RSP = load i64, i64* %13, align 8, !tbaa !1282
    …
  %39 = inttoptr i64 %22 to i8*
  %40 = tail call i8* @gets(i8* %39), !noalias !1286
```

```
$ ./a.out.lifted <input
Enter PIN:
Welcome, Admin!

You have the power!!!!!
```

Why is the lifted program vulnerable?




```
define %struct.Memory*
%struct.Memory*) #2 {
entry:
  %8 = getelementptr %struct                truct.State* %state, i64 0, …
  %RSP = load i64, i64* %8                  gn 8
  %is_admin = add i64 %RS , -9
  %17 = inttoptr i64 %16 to i8*
  store i8 0, i8* %17
  %20 = add i64 %RSP, -32
  store i64 %20, i64* %8, align 8,
  %22 = tail call %struct.Memory* @sub_400596_verify_pin(%struct.State*
nonnull %state, i64 %18, %struct.Memory* %2)
  …
define %struct.Memory*
%struct.Memory*) #3 {
entry:
  …
  %13 = getelementptr %struct.              truct.State* %state, i64 0, …
  %RSP = load i64, i64* %13,                8, !tbaa !1282
  ...
  %pin = add i64 %RSP, -40
  …
  %39 = inttoptr i64 %22 to i8*
  %40 = tail call i8* @gets(i8* %39), !noalias !1286
```

`is_admin` allocated on emulated stack

Adjust the emulated stack

`pin` buffer allocated on emulated stack

Let's review what happened

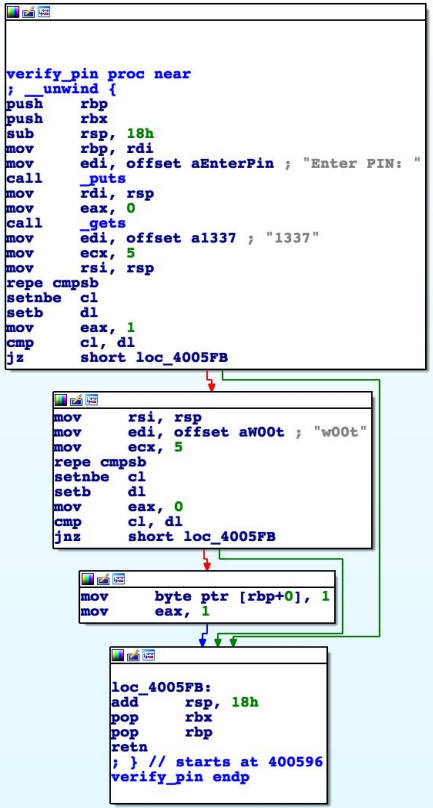
- **McSema mitigated the ROP because lifted bitcode ignores emulated return addresses, and "trusts" return addresses stored on its own execution stack**
- **The lifted bitcode emulates the allocation variables `pin` and `is_admin` on the emulated stack**
  - In fact, the lifted bitcode doesn't know anything about `pin` or `is_admin`
- **Can we lift `is_admin` and `pin` onto our "trusted" execution stack, i.e recover the local variables?**

Abstraction recovery: Lifting stack variables

Local variables are lost

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
    return true;
  } else if (!strcmp(pin, "w00t")) {
    *is_admin = true;
    return true;
  } else {
    return false;
  }
}
```

No types, no variables :-(



Where are the accesses to local variables? (1)

```
verify_pin      proc near               ; CODE XREF: main+E↓p
; __unwind {
                push    rbp
                push    rbx
                sub     rsp, 18h
                mov     rbp, rdi
                mov     edi, offset aEnterPin ; "Enter PIN: "
                call    _puts
                mov     rdi, rsp
                mov     eax, 0
                call    _gets
                mov     edi, offset a1337 ; "1337"
                mov     ecx, 5
                mov     rsi, rsp
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 1
                cmp     cl, dl
                jz      short loc_4005FB
                mov     rsi, rsp
                mov     edi, offset aW00t ; "w00t"
                mov     ecx, 5
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 0
                cmp     cl, dl
                jnz     short loc_4005FB
                mov     byte ptr [rbp+0], 1
                mov     eax, 1

loc_4005FB:                             ; CODE XREF: verify_pin+3C↑j
                                        ; verify_pin+5A↑j
                add     rsp, 18h
                pop     rbx
                pop     rbp
                retn
; } // starts at 400596
verify_pin      endp
```

make a stack frame

puts("Enter PIN: ")

gets(pin)

strcmp(pin, "1337")

strcmp(pin, "w00t")

*is_admin = true

un-make a stack frame

return

Where are the accesses to local variables? (2)

```
verify_pin      proc near               ; CODE XREF: main+E↓p
; __unwind {
                push    rbp
                push    rbx
                sub     rsp, 18h
                mov     rbp, rdi
                mov     edi, offset aEnterPin ; "Enter PIN: "
                call    _puts
                mov     rdi, rsp
                mov     eax, 0
                call    _gets
                mov     edi, offset a1337 ; "1337"
                mov     ecx, 5
                mov     rsi, rsp
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 1
                cmp     cl, dl
                jz      short loc_4005FB
                mov     rsi, rsp
                mov     edi, offset aW00t ; "w00t"
                mov     ecx, 5
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 0
                cmp     cl, dl
                jnz     short loc_4005FB
                mov     byte ptr [rbp+0], 1
                mov     eax, 1

loc_4005FB:                             ; CODE XREF: verify_pin+3C↑j
                                        ; verify_pin+5A↑j
                add     rsp, 18h
                pop     rbx
                pop     rbp
                retn
; } // starts at 400596
verify_pin      endp
```

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
    return true;
  } else if (!strcmp(pin, "w00t")) {
    *is_admin = true;
    return true;
  } else {
    return false;
  }
}
```

Where are the accesses to local variables? (3)

```
verify_pin      proc near               ; CODE XREF: main+E↓p
; __unwind {
                push    rbp
                push    rbx
                sub     rsp, 18h
                mov     rbp, rdi
                mov     edi, offset aEnterPin ; "Enter PIN: "
                call    _puts
                mov     rdi, rsp
                mov     eax, 0
                call    _gets
                mov     edi, offset a1337 ; "1337"
                mov     ecx, 5
                mov     rsi, rsp
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 1
                cmp     cl, dl
                jz      short loc_4005FB
                mov     rsi, rsp
                mov     edi, offset aW00t ; "w00t"
                mov     ecx, 5
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 0
                cmp     cl, dl
                jnz     short loc_4005FB
                mov     byte ptr [rbp+0], 1
                mov     eax, 1

loc_4005FB:                             ; CODE XREF: verify_pin+3C↑j
                                        ; verify_pin+5A↑j
                add     rsp, 18h
                pop     rbx
                pop     rbp
                retn
; } // starts at 400596
verify_pin      endp
```

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
    return true;
  } else if (!strcmp(pin, "w00t")) {
    *is_admin = true;
    return true;
  } else {
    return false;
  }
}
```

Where are the accesses to local variables? (4)

```
verify_pin      proc near               ; CODE XREF: main+E↓p
; __unwind {
                push    rbp
                push    rbx
                sub     rsp, 18h
                mov     rbp, rdi
                mov     edi, offset aEnterPin ; "Enter PIN: "
                call    _puts
                mov     rdi, rsp
                mov     eax, 0
                call    _gets
                mov     edi, offset a1337 ; "1337"
                mov     ecx, 5
                mov     rsi, rsp
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 1
                cmp     cl, dl
                jz      short loc_4005FB
                mov     rsi, rsp
                mov     edi, offset aW00t ; "w00t"
                mov     ecx, 5
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 0
                cmp     cl, dl
                jnz     short loc_4005FB
                mov     byte ptr [rbp+0], 1
                mov     eax, 1

loc_4005FB:                             ; CODE XREF: verify_pin+3C↑j
                                        ; verify_pin+5A↑j
                add     rsp, 18h
                pop     rbx
                pop     rbp
                retn
; } // starts at 400596
verify_pin      endp
```

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
    return true;
  } else if (!strcmp(pin, "w00t")) {
    *is_admin = true;
    return true;
  } else {
    return false;
  }
}
```

Where are the accesses to local variables? (5)

```
verify_pin      proc near               ; CODE XREF: main+E↓p
; __unwind {
                push    rbp
                push    rbx
                sub     rsp, 18h
                mov     rbp, rdi
                mov     edi, offset aEnterPin ; "Enter PIN: "
                call    _puts
                mov     rdi, rsp
                mov     eax, 0
                call    _gets
                mov     edi, offset a1337 ; "1337"
                mov     ecx, 5
                mov     rsi, rsp
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 1
                cmp     cl, dl
                jz      short loc_4005FB
                mov     rsi, rsp
                mov     edi, offset aW00t ; "w00t"
                mov     ecx, 5
                repe cmpsb
                setnbe  cl
                setb    dl
                mov     eax, 0
                cmp     cl, dl
                jnz     short loc_4005FB
                mov     byte ptr [rbp+0], 1
                mov     eax, 1

loc_4005FB:                             ; CODE XREF: verify_pin+3C↑j
                                        ; verify_pin+5A↑j
                add     rsp, 18h
                pop     rbx
                pop     rbp
                retn
; } // starts at 400596
verify_pin      endp
```

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
    return true;
  } else if (!strcmp(pin, "w00t")) {
    *is_admin = true;
    return true;
  } else {
    return false;
  }
}
```

That's where they are! (1)

```
verify_pin      proc near               ; CODE XREF: main+E↓p
; __unwind {
                push    rbp
                push    rbx
                sub     rsp, 18h
                mov     rbp, rdi
                mov     edi, offset aEnterPin ; "Enter PIN: "
                call    _puts
                mov     byte ptr [rbp+0], 1
                mov     eax, 1
loc_4005FB:                             ; CODE XREF: verify_pin+3C↑j
                                        ; verify_pin+5A↑j
                add     rsp, 18h
                pop     rbx
                pop     rbp
                retn
; } // starts at 400596
verify_pin      endp
```

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
                    pin, "w00t")) {
```

Observation

All uses of stack variables use the RSP (stack pointer) register, or the RBP (base pointer, or stack frame pointer) register

That's where they are! (2)

```
verify_pin      proc near               ; CODE XREF: main+E↓p
; __unwind {
                push    rbp
                push    rbx
                sub     rsp, 18h
                mov     rbp, rdi
                mov     edi, offset aEnterPin ; "Enter PIN: "
                call    _puts
                mov     byte ptr [rbp+0], 1
                mov     eax, 1
loc_4005FB:                             ; CODE XREF: verify_pin+3C↑j
                                        ; verify_pin+5A↑j
                add     rsp, 18h
                pop     rbx
                pop     rbp
                retn
; } // starts at 400596
verify_pin      endp
```

```
bool verify_pin(bool *is_admin) {
  char pin[5];
  puts("Enter PIN: ");
  gets(pin);
  if (!strcmp(pin, "1337")) {
                    pin, "w00t")) {
```

Key idea

If we know where local variables are, then we can rewrite all uses of the RSP and the RBP registers to instead reference local variables defined using LLVM `alloca` instructions

How do we recover stack variables?

- **McSema disassembler does the variable recovery**
  - Uses Binary Ninja or IDA Pro to identify the stack frame structure
    - Collects the stack frame members
  - Identifies instructions that reference the stack frame
    - Identify the uses of stack and frame pointer registers
    - Associate uses with stack variables

```
$ mcsema-disass --arch amd64 --os linux \
                --disassembler /opt/binaryninja/binaryninja \
                --binary a.out --entrypoint main \
                --output authenticate.vars.cfg --recover-stack-vars
```

How do we lift stack variables?

- **Lift the recovered stack variables into LLVM IR**
  - Deconstruct the emulated stack
  - Allocate variables onto execution stack using `alloca`
  - Associate type information with the lifted variables: `i32`, `i16`, `[16 * i8]`, etc.
  - Rewrite operands to the lifted instruction to instead reference stack variables

```
$ mcsema-lift-6.0 \
    --arch amd64 --os linux --cfg authenticate.vars.cfg \
    --output authenticate.bc --abi_libraries libc --stack_protector

$ remill-clang-6.0 -o a.out.lifted authenticate.bc \
                   /lib/libmcsema_rt64-6.0.a
```

How do we lift stack variables?

- **Lift the recovered stack variables into LLVM IR**
  - ion stack
  - * i8], etc.
  -

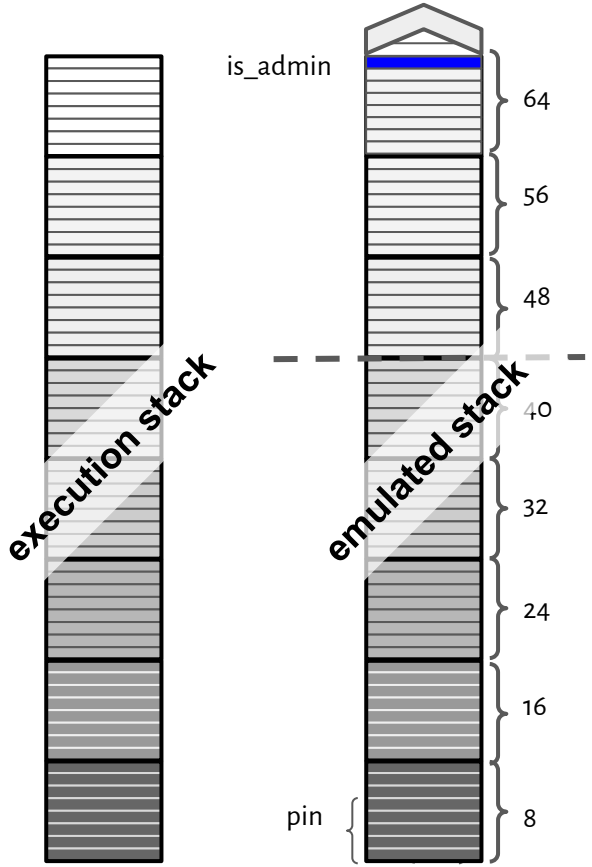
Challenges

Identifying the indirect access of stack frames not using the stack (frame) pointers

Special cases where the frame members can't be lifted, i.e. variable args functions

```
$ mcse
    --
    --
$ remi                    /lib/libmcsema_rt64-6.0.a
```

Before lifting stack variables

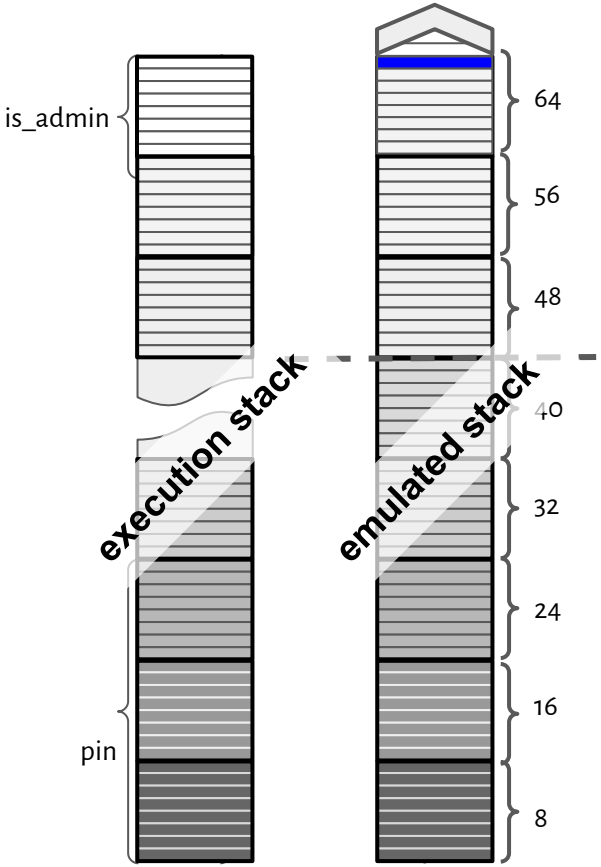



```
define %struct.Memory* @sub_400602_main(%struct.State*, i64,
%struct.Memory*) #2 {
entry:
  %8 = getelementptr %struct.State, %struct.State* %state, i64 0, i32 6, …
  %RSP = load i64, i64* %8, align 8
  %is_admin = add i64 %RSP, -9
  %17 = inttoptr i64 %16 to i8*
  store i8 0, i8* %17
  %20 = add i64 %RSP, -32
  store i64 %20, i64* %8, a
  %22 = call %struct.Memory
i64 %18, %struct.Memory* %2)
  …
define %struct.Memory* @sub_400596_verify_pin(%struct.State*, i64,
%struct.Memory*) #3 {
entry:
  …
  %13 = getelementptr %stru
  %RSP = load i64, i64* %13,
  ...
  %pin = add i64 %RSP, -40
  …
  %39 = inttoptr i64 %22 to i8*
  %40 = tail call i8* @gets(i8* %39), !noalias !1286
```

`is_admin` allocated on emulated stack

`pin` buffer allocated on emulated stack

After lifting stack variables




```
define %struct.Memory* @sub_400602_main(%struct.State*, i64, %struct.Memory*)
#2 {
entry:
  %8 = getelementptr %struct.State, %struct.State* %state, i64 0, …
  %is_admin = alloca [9 x i8], align 1
  %RSP = load i64, i64* %8, align
  %17 = ptrtoint [9 x i8]* %is_adm
  %18 = add i64 %17, 24
  %19 = inttoptr i64 %18 to i8*
  store i8 0, i8* %is_admin
  %20 = add i64 %RSP, -32
  store i64 %20, i64* %8, align 8, !tbaa !1261
  %22 = tail call %struct.Memory* @sub_400596_verify_pin(%struct.State* nonnull
%state, i64 %18, %struct.Memory* %2)
  …
define %struct.Memory* @sub_400596_verify_pin(%struct.State*, i64,
%struct.Memory*) #3 {
entry:
  %13 = getelementptr %struct.State, %struct.State* %state, i64 0, i32 6, …
  %pin = alloca [24 x i8], align 1
  %RSP = load i64, i64* %1
  …
  %20 = ptrtoint [24 x i8]* %pin t
  %21 = add i64 %20, 40
  %22 = add i64 %RSP, -40
  %39 = inttoptr i64 %21 to i8*
  %40 = tail call i8* @gets(i8* %39), !noalias !1286
```

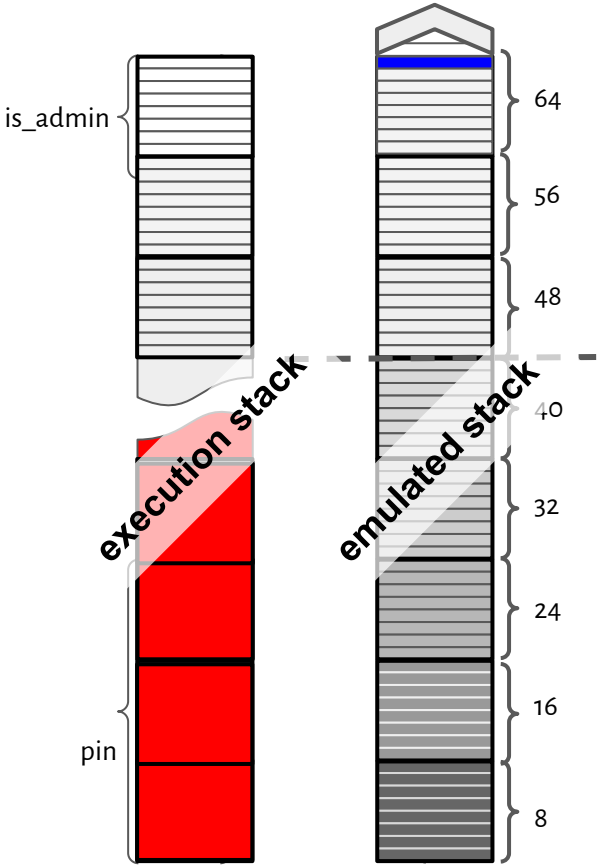
`is_admin` allocated on lifted stack

`pin` buffer allocated on lifted stack

Will this mitigate the overflow of `is_admin`?

- **Maybe**
- **Lifting stack variables leaves our program *differently* exposed**
  - With no additional modifications, overflows of `pin` now have the possibility of exploiting the native stack, not the emulated stack
  - But lifting stack variables also enables us to build-in new security measures that may not have been present in the original binary
    - SafeStack, stack protector/canaries, etc.
    - AddressSanitizer

Does the exploit still work?




```
$ python
>>> import struct
>>> chr(0) * 40 + struct.pack("<Q", 0x0400615) +
chr(0) * 15 + chr(1)
'\x00\x00\x00…\x00\x15\x06\x40…\x00\x00\x00\x01'
>>> open("input", "w").write(_)
>>> exit()

$ ./a.out.lifted <input
Enter PIN:
```

Set is_admin = true

In summary, we conclude

McSema is a 95% solution

- **Key challenge with lifting is *not* production of bitcode, accuracy of instruction semantics, or coverage of ISA features**
  - Key problem is accurate disassembly: figuring out if a number embedded in a code/data segment represents a reference to other code/data or not
  - Getting cross-references wrong means all sorts of problems :-(
- **The other 5% of the problem is undecidable**
  - Figuring out when heuristics cause regressions is also challenging
  - But we're persistent and we can work with you to get that extra 5%

You should try McSema

- **Your binaries might not belong to that 5%!**
- **McSema generally used in one of two ways**
  - Lifting for recompiling (what we talked about, uses libmcsema_rtNN.a)
  - Lifting for symbolic execution by KLEE (use `--explicit_args` flag)
- **It's open-source, permissively licensed**
  - McSema for lifting programs: https://github.com/trailofbits/mcsema
  - Remill for lifting instructions: https://github.com/trailofbits/remill
  - We pay bounties for open-source contributions, we are looking for interns, and we are interested in academic/industrial partnerships

Farewells

Peter Goodman
Senior Security Engineer

peter@trailofbits.com

Akshay Kumar
Senior Security Engineer

akshay.kumar@trailofbits.com


TRAIL
OF
BITS


TRAIL
OF
BITS