SPQR

The Roman Xpl0it

RopGun

Detecting ROP attacks using modern HW facilities

by anticlockwise

Let’s begin

DEMO

Outline

1. What is a ROP exploit?

2. How do Modern Processors work?

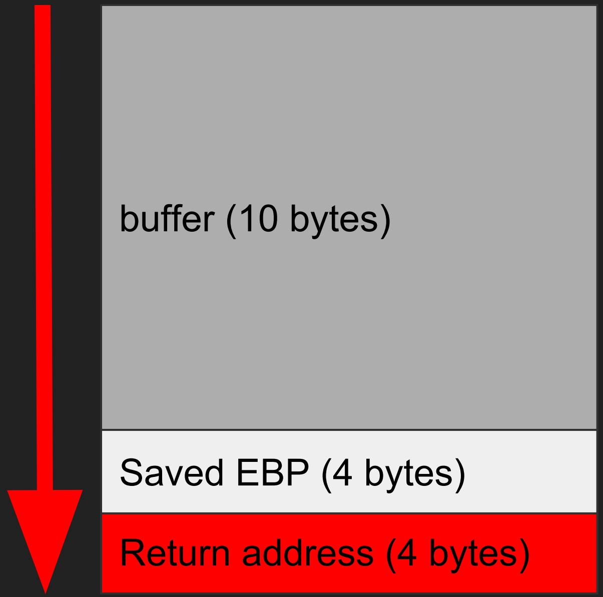
3. Let’s use HW to save the world from ROP!

Outline

1. What is a ROP exploit?

Buffer overflow

```
void foo() {

        char buffer[10];

        scanf("%s", buffer); //DOH

}
```



buffer (10 bytes)

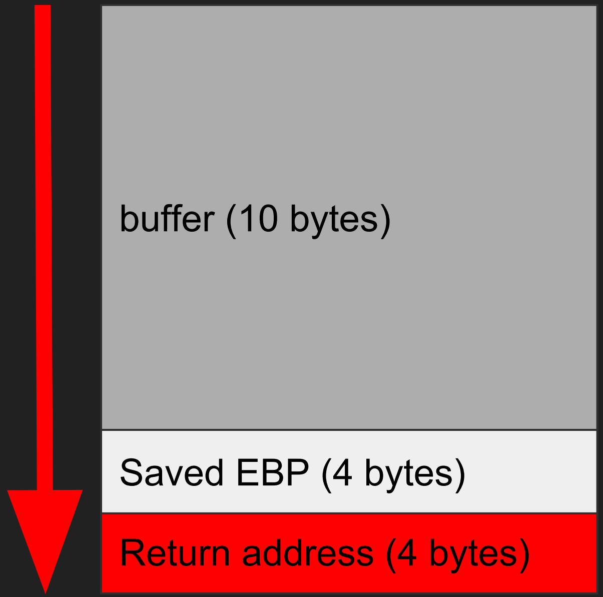
Saved EBP (4 bytes)

Return address (4 bytes)

Spawn a Shell!

Let’s override the return address content!

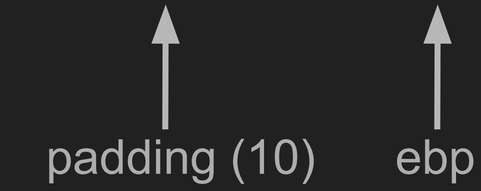
After foo() the execution will be resumed from the return address.

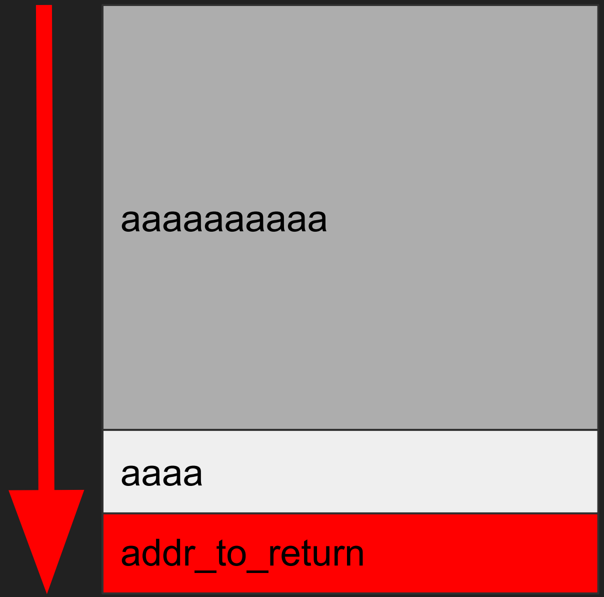


Spawn a Shell!

For example:

“aaaaaaaaaa” + “aaaa” + addr_to_return



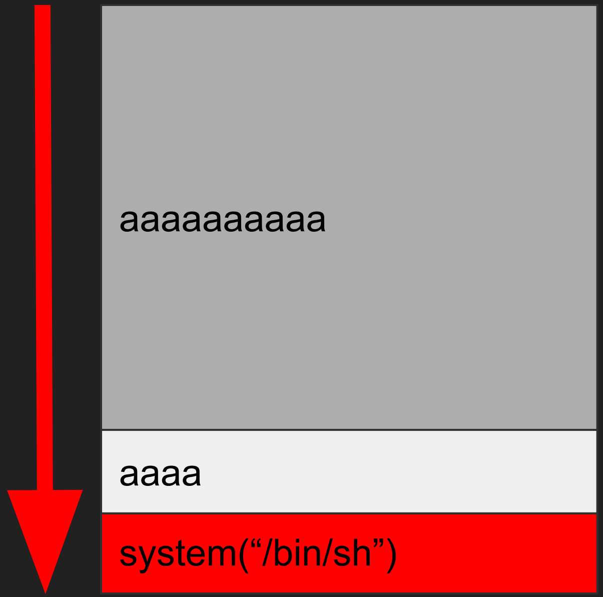


Spawn a Shell!

What to put in addr_to_return?

Calling system("/bin/sh") will execute a shell in the current context

But what if we don't have system available to call?



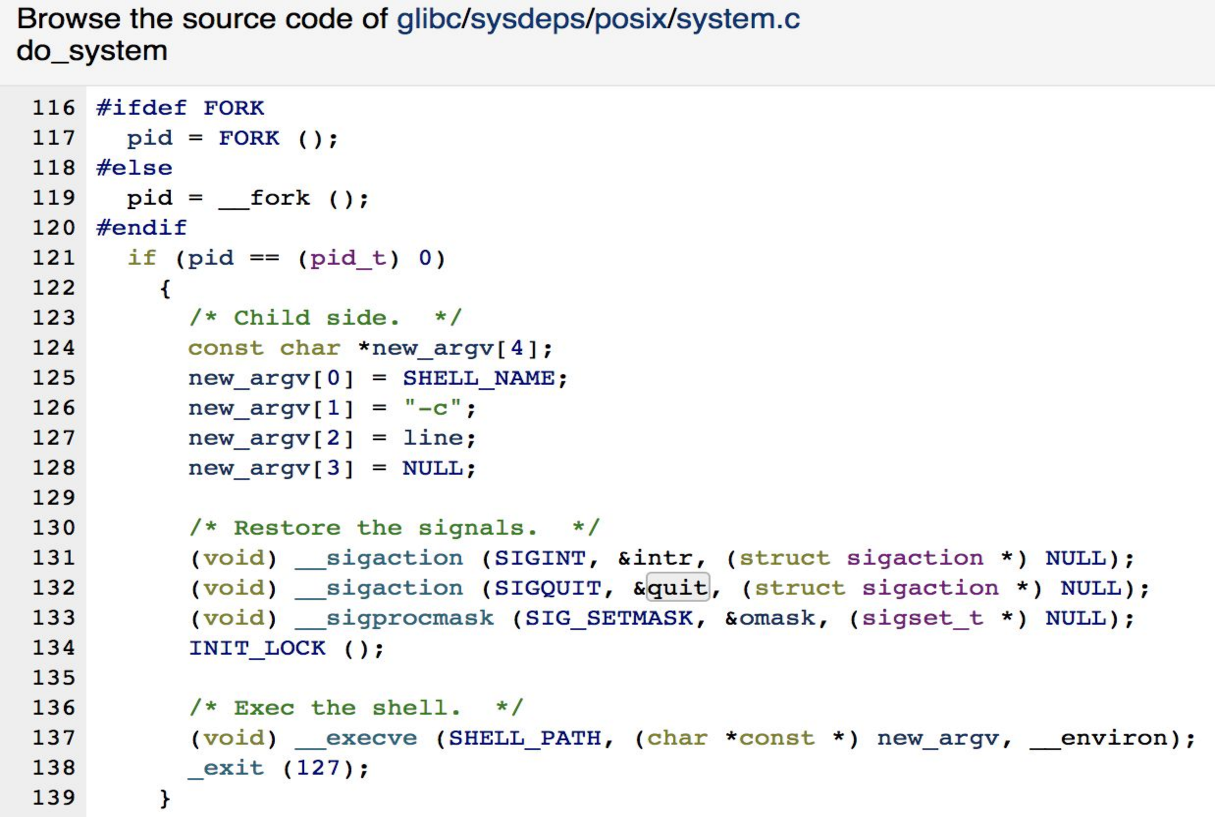
aaaaaaaaaaa

aaaa

system("/bin/sh")

Spawn a Shell!

Let’s dissect system!



Browse the source code of glibc/sysdeps/posix/system.c
do_system

```
#ifdef FORK
  pid = FORK ();
#else
  pid = __fork ();
#endif
  if (pid == (pid_t) 0)
    {
      /* Child side.  */
      const char *new_argv[4];
      new_argv[0] = SHELL_NAME;
      new_argv[1] = "-c";
      new_argv[2] = line;
      new_argv[3] = NULL;

      /* Restore the signals.  */
      (void) __sigaction (SIGINT, &intr, (struct sigaction *) NULL);
      (void) __sigaction (SIGQUIT, &quit, (struct sigaction *) NULL);
      (void) __sigprocmask (SIG_SETMASK, &omask, (sigset_t *) NULL);
      INIT_LOCK ();

      /* Exec the shell.  */
      (void) __execve (SHELL_PATH, (char *const *) new_argv, __environ);
      _exit (127);
    }
```

Spawn a Shell!

__execve is the stub for the execve system call: the system call that transforms the calling process into a new process to execute.

It sets:

eax -> 0x0b
ebx -> address of “/bin/sh” (line to execute)
ecx -> NULL
edx -> NULL

then executes:
int $0x80

Spawn a Shell!

We need to execute something similar to:

```
mov    ebx, /bin/sh_string_address
mov    ecx, 0x0
mov    edx, 0x0
mov    eax ,0xb
int    0x80
```

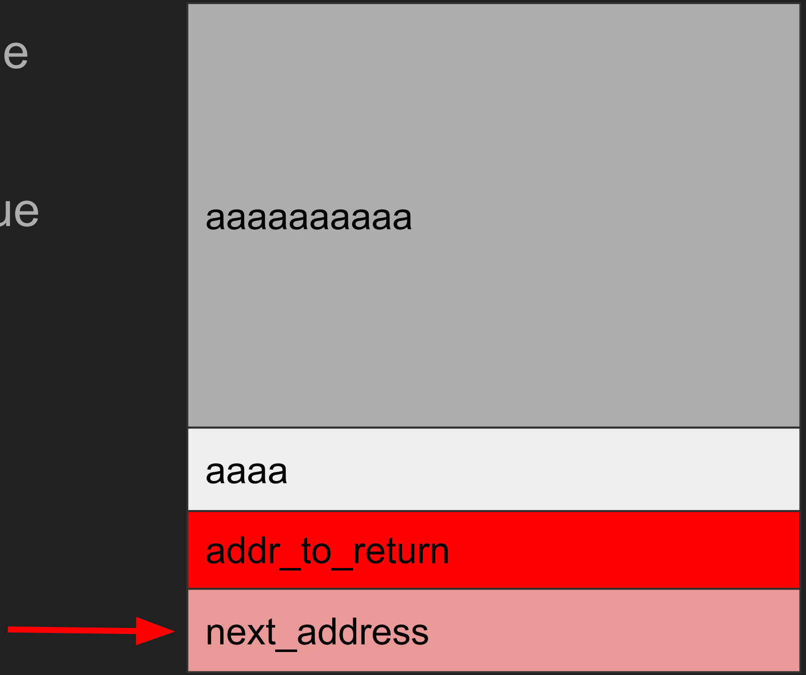
But how?

… Let’s notice something

Composing a chain

If addr_to_return point to some code that ends in “ret”

the execution will then continue from the address after



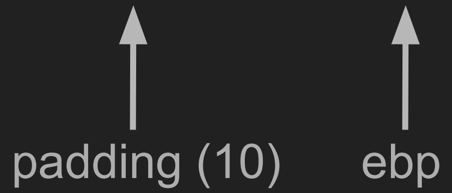
...and we can iterate again and again

Composing a chain

So why we don’t insert more than one address to return to?

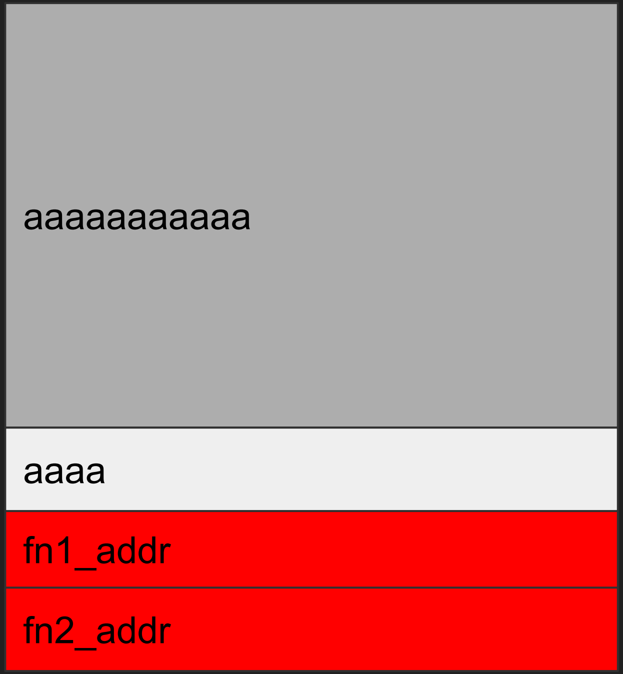
For example:

“aaaaaaaaaa” + “aaaa” + fn1_addr + fn2_addr



padding (10)

ebp



aaaaaaaaaaa

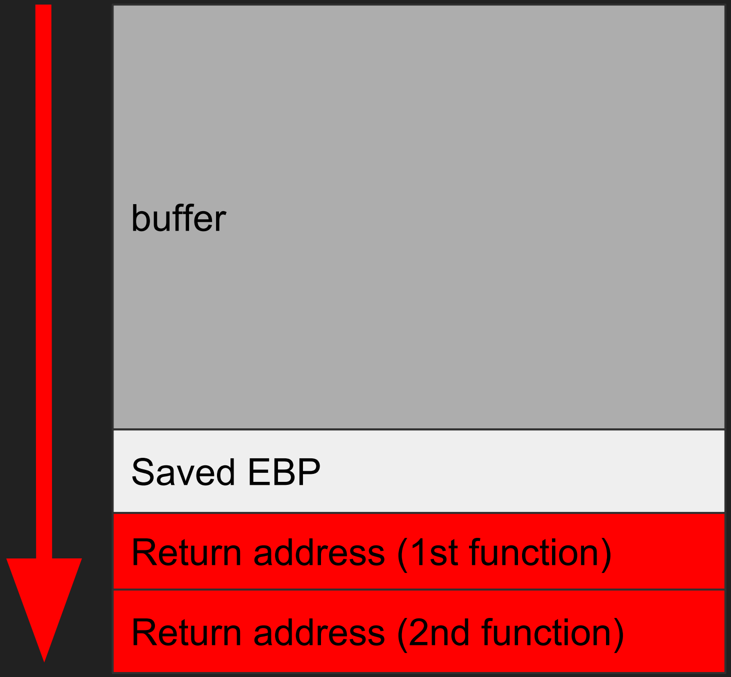
aaaa

fn1_addr

fn2_addr

Composing a chain

```
fn1:
    mov eax, 0xabadcafe
    ret
fn2:
    mov ecx, eax
    ret
```



buffer

Saved EBP

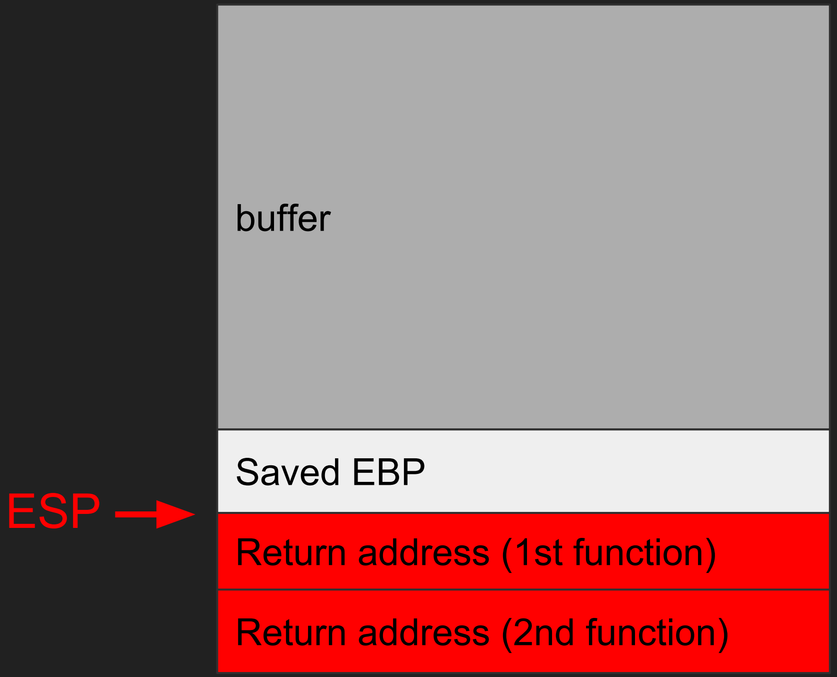
Return address (1st function)

Return address (2nd function)

Composing a chain

```
foo:

    …

    ret    ← EIP
```

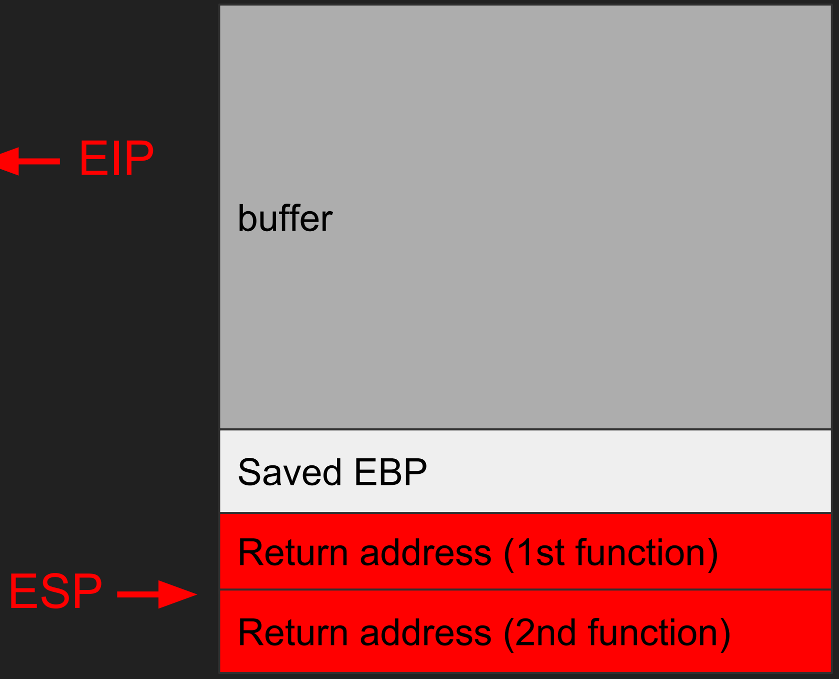
EAX: ?
ECX: ?



Composing a chain

```
fn1:

    mov eax, 0xabadcafe  ← EIP

    ret
```

EAX: ?
ECX: ?



ESP →

buffer

Saved EBP

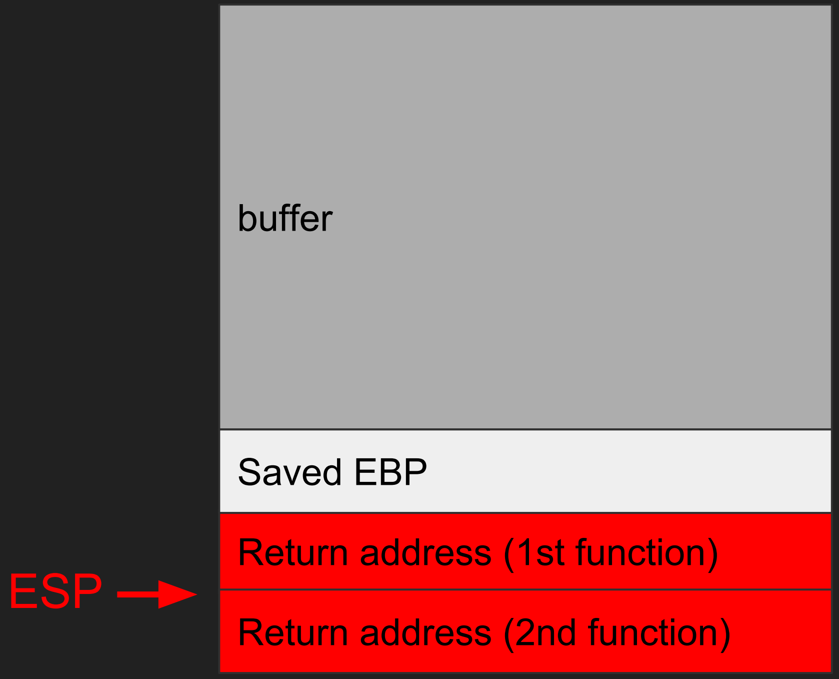
Return address (1st function)

Return address (2nd function)

Composing a chain

```
fn1:

    mov eax, 0xabadcafe

    ret    ← EIP
```

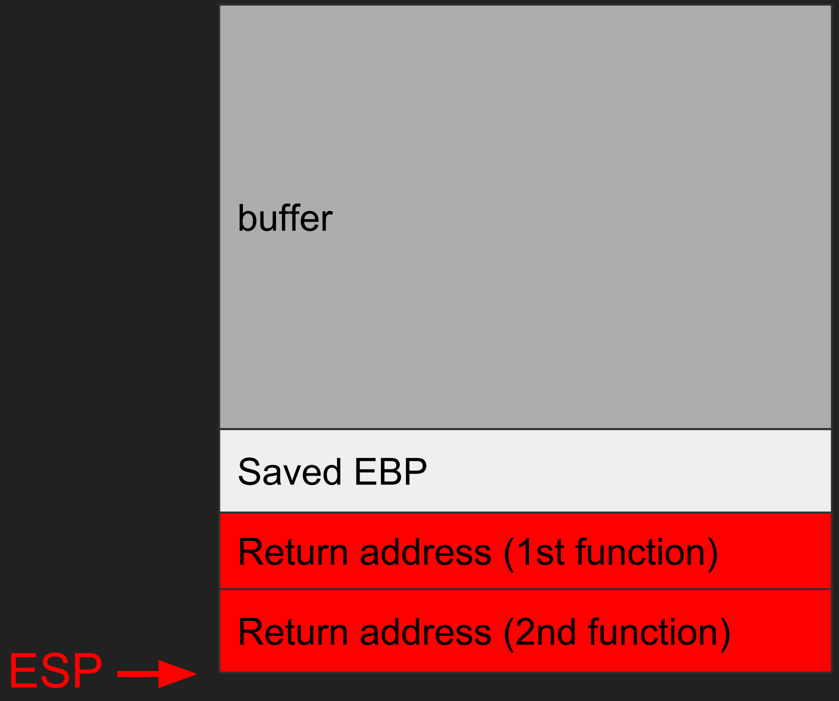
EAX: 0xabadcafe
ECX: ?



Composing a chain

```
fn2:

    mov ecx, eax  ← EIP

    ret
```

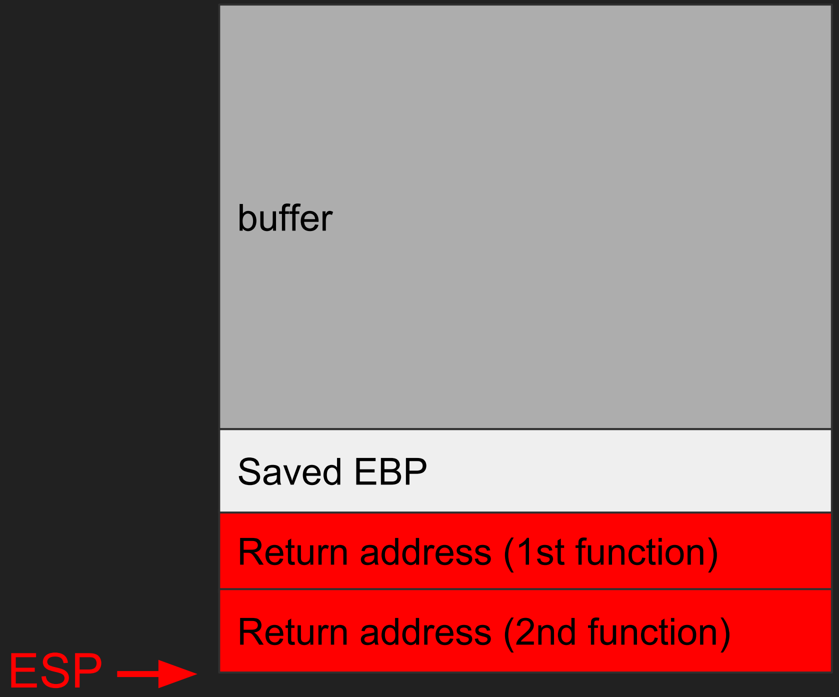
EAX: 0xabadcafe
ECX: ?



Composing a chain

```
fn2:

    mov ecx, eax

    ret  ← EIP
```

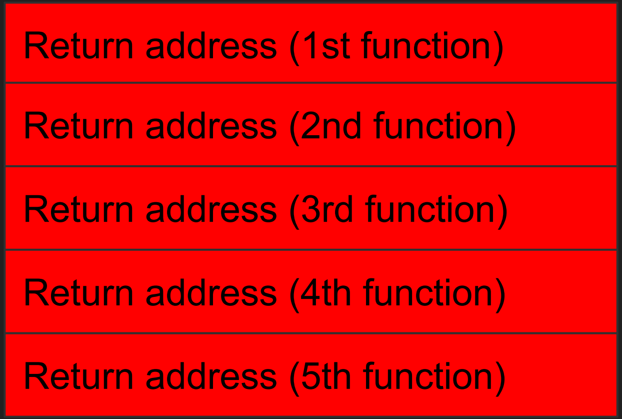
EAX: 0xabadcafe
ECX: 0xabadcafe



Composing a chain

We can insert addresses to return to execute, as long as we want!

BUT: every piece of code we execute must end with a ret instruction, to continue the chain



Return address (1st function)

Return address (2nd function)

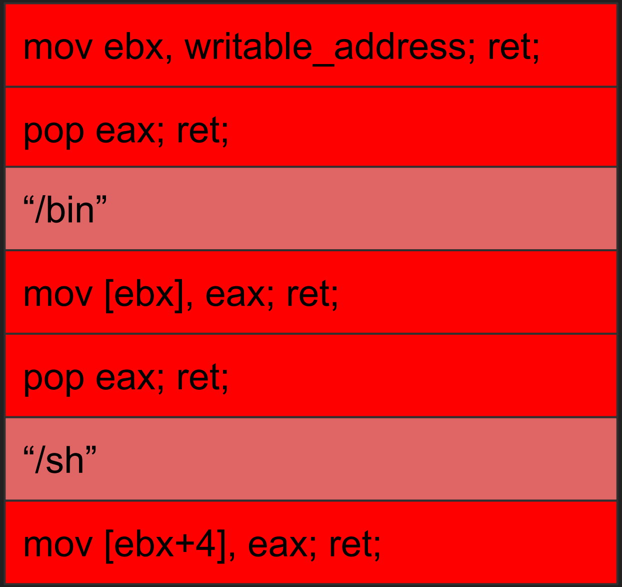
Return address (3rd function)

Return address (4th function)

Return address (5th function)

Composing a chain

For example, let’s load ebx with the address of a /bin/sh string



mov ebx, writable_address; ret;

pop eax; ret;

“/bin”

mov [ebx], eax; ret;

pop eax; ret;

“/sh”

mov [ebx+4], eax; ret;

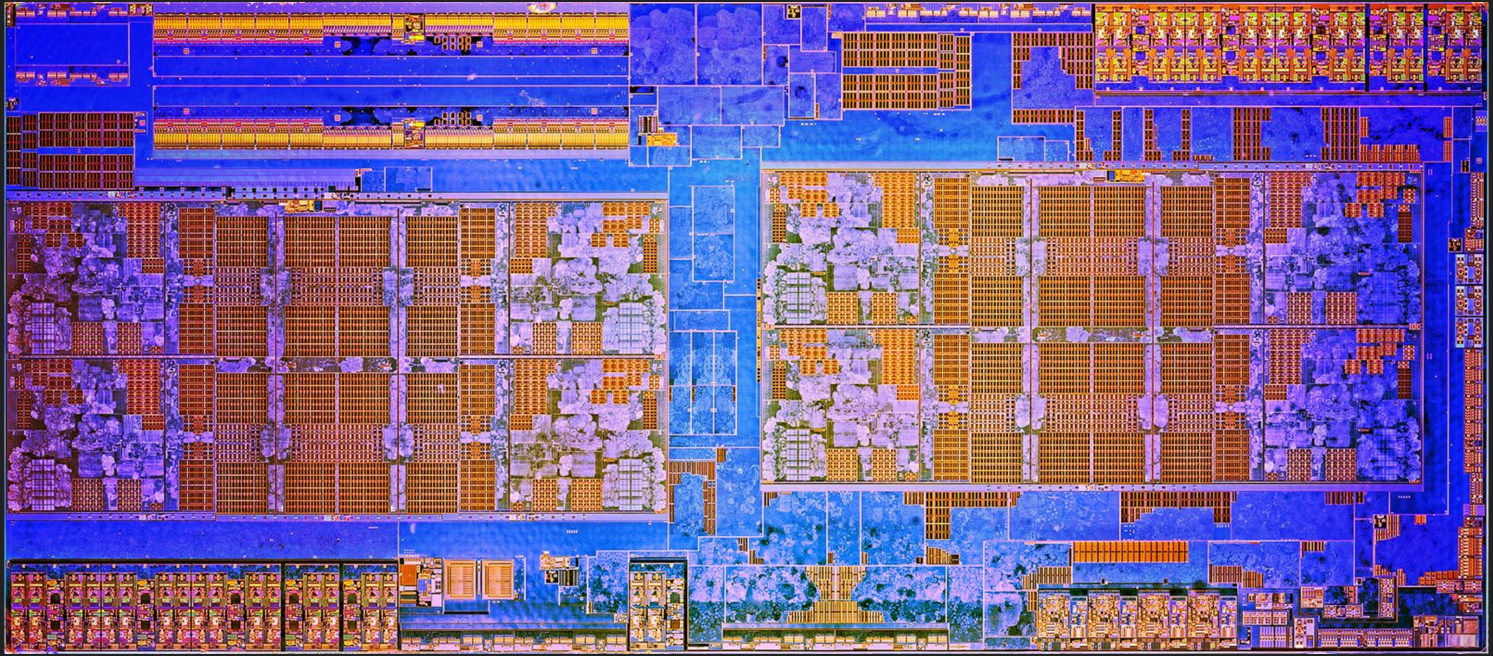
Now ebx points to the /bin/sh string

Outline

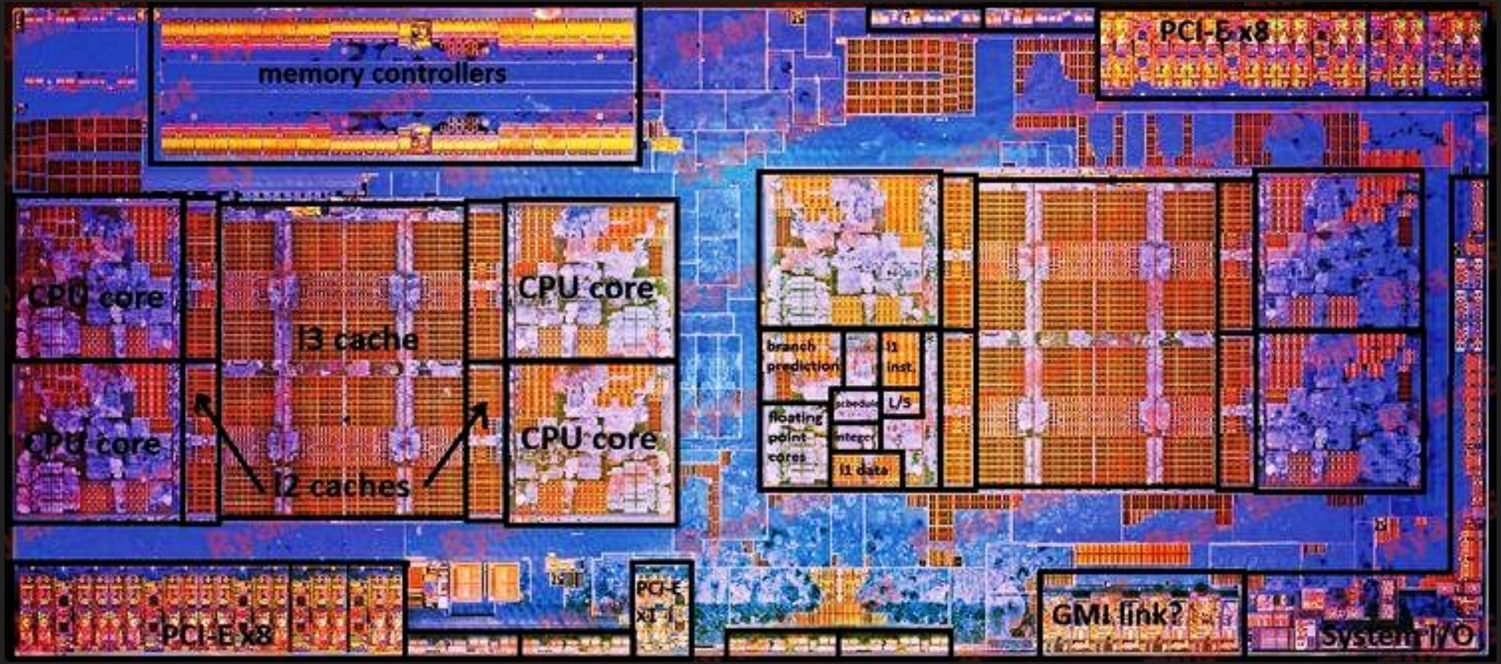
1. What is a ROP exploit?

2. How do Modern Processors work?

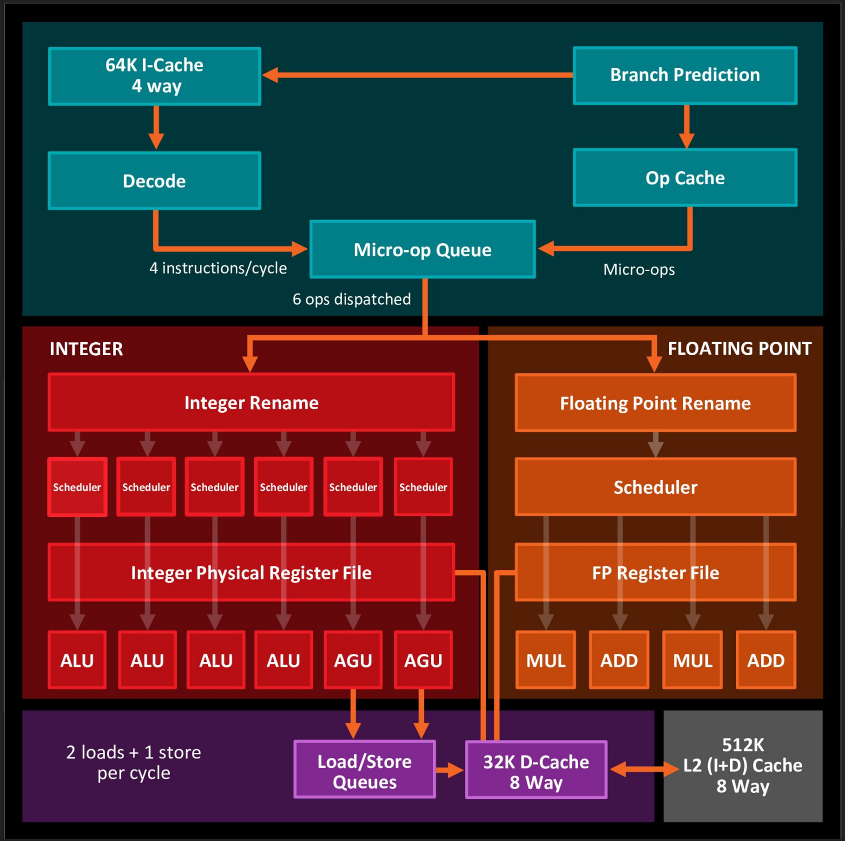
AMD Ryzen



AMD Ryzen

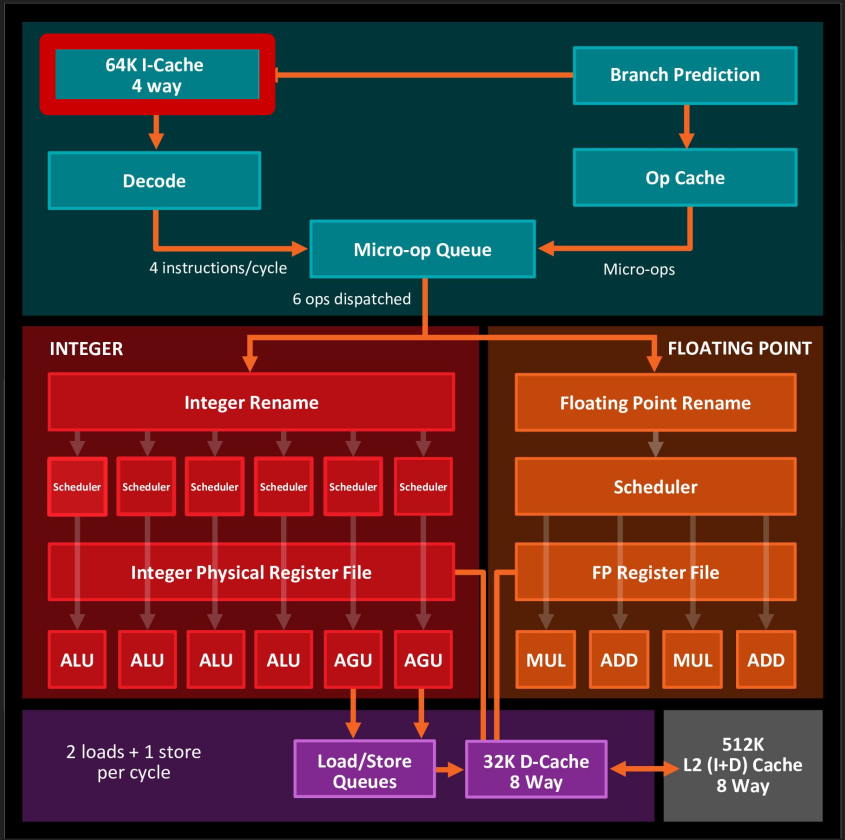

memory controllers
PCI-E x8
CPU core
CPU core
l3 cache
CPU core
CPU core
l2 caches
branch prediction
l1 inst.
scheduler
L/S
floating point cores
integer
l1 data
PCI-E x8
PCI-E x1
GMI link?
System I/O

AMD Ryzen


64K I-Cache
4 way
Branch Prediction
Decode
Op Cache
Micro-op Queue
4 instructions/cycle
Micro-ops
6 ops dispatched
INTEGER
FLOATING POINT
Integer Rename
Floating Point Rename
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler
Integer Physical Register File
FP Register File
ALU
ALU
ALU
ALU
AGU
AGU
MUL
ADD
MUL
ADD
2 loads + 1 store
per cycle
Load/Store
Queues
32K D-Cache
8 Way
512K
L2 (I+D) Cache
8 Way

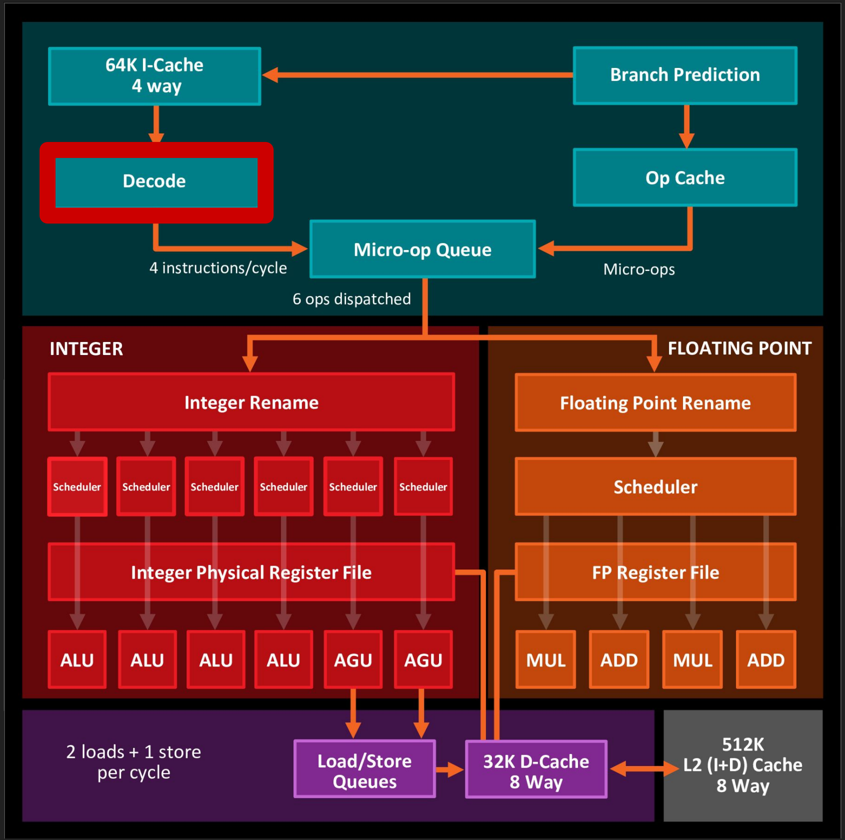
AMD Ryzen

add qword ptr [rax], rbx



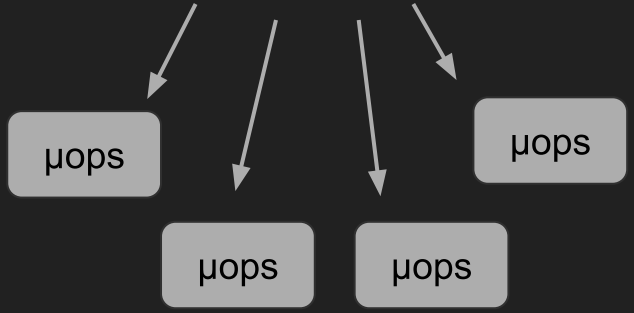
AMD Ryzen

add qword ptr [rax], rbx



AMD Ryzen

```
add qword ptr [rax], rbx
```

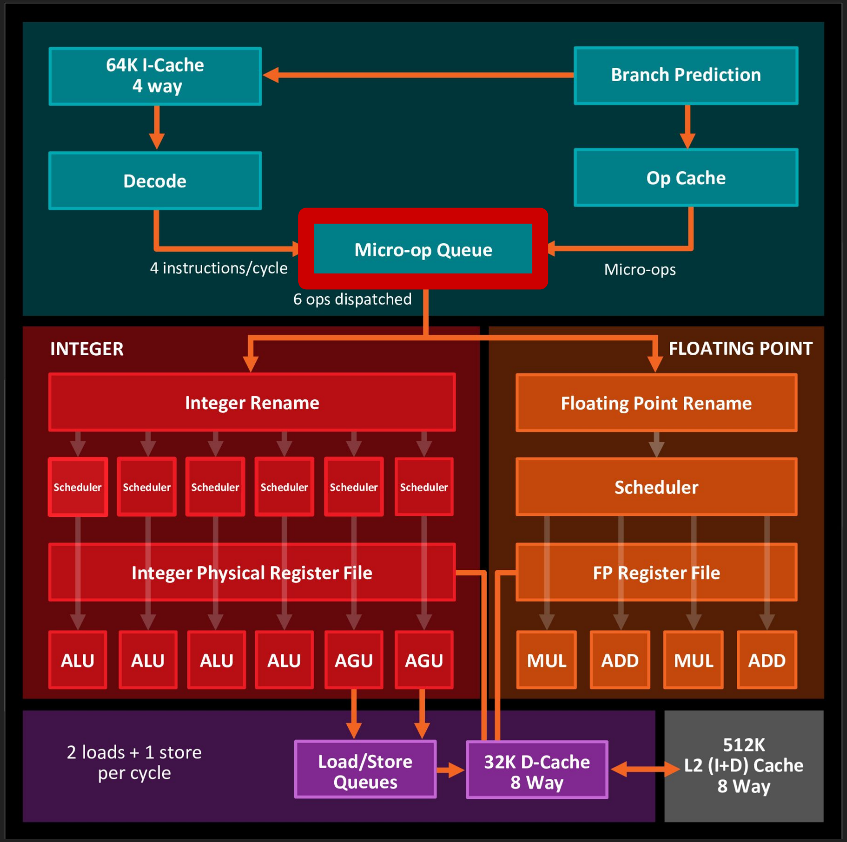


μops

μops

μops

μops



64K I-Cache
4 way

Branch Prediction

Decode

Op Cache

Micro-op Queue

4 instructions/cycle

Micro-ops

6 ops dispatched

INTEGER

FLOATING POINT

Integer Rename

Floating Point Rename

Scheduler

Scheduler

Scheduler

Scheduler

Scheduler

Scheduler

Scheduler

Integer Physical Register File

FP Register File

ALU

ALU

ALU

ALU

AGU

AGU

MUL

ADD

MUL

ADD

2 loads + 1 store
per cycle

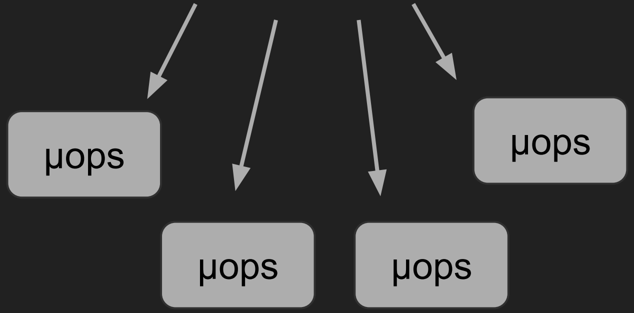
Load/Store
Queues

32K D-Cache
8 Way

512K
L2 (I+D) Cache
8 Way

AMD Ryzen

```
add qword ptr [rax], rbx
```

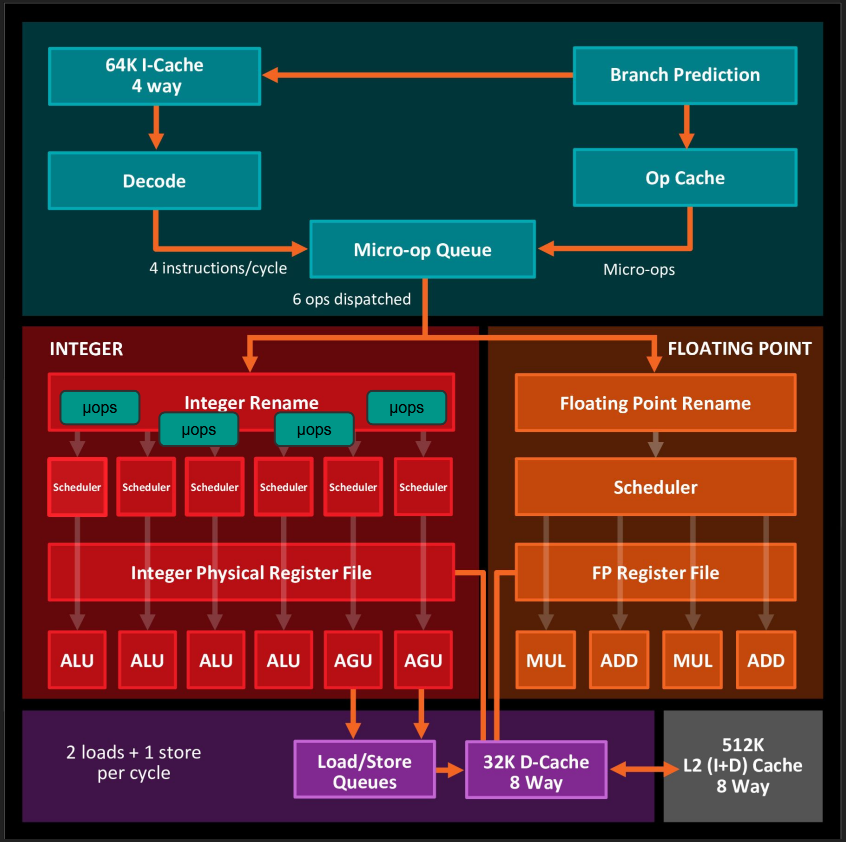


µops

µops

µops

µops



64K I-Cache 4 way

Branch Prediction

Decode

Op Cache

Micro-op Queue

4 instructions/cycle

Micro-ops

6 ops dispatched

INTEGER

FLOATING POINT

Integer Rename

µops

µops

µops

µops

Scheduler

Scheduler

Scheduler

Scheduler

Scheduler

Scheduler

Integer Physical Register File

ALU

ALU

ALU

ALU

AGU

AGU

Floating Point Rename

Scheduler

FP Register File

MUL

ADD

MUL

ADD

2 loads + 1 store per cycle

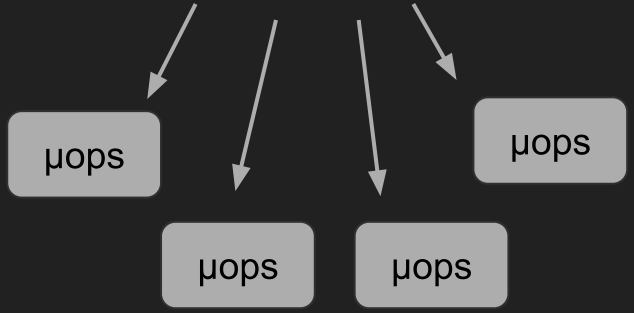
Load/Store Queues

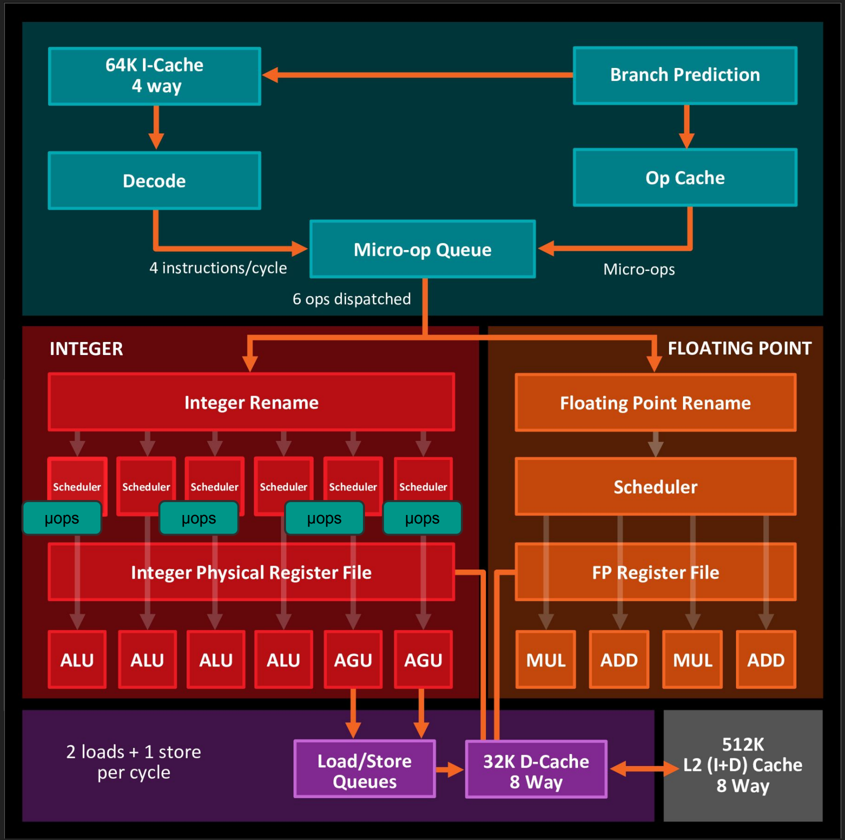
32K D-Cache 8 Way

512K L2 (I+D) Cache 8 Way

AMD Ryzen

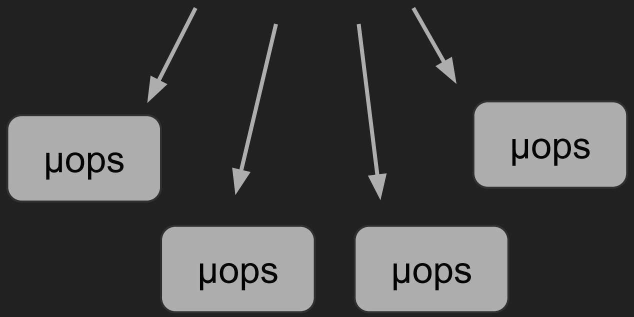
add qword ptr [rax], rbx

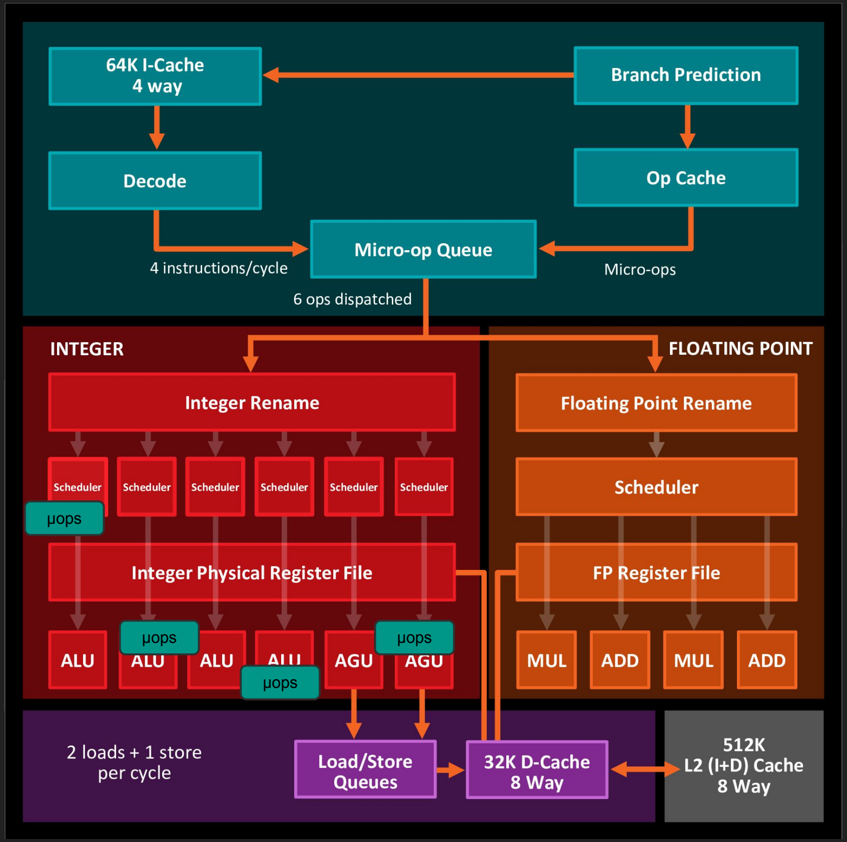




AMD Ryzen

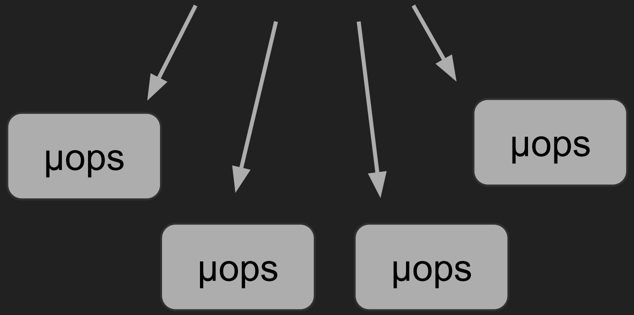
```
add qword ptr [rax], rbx
```





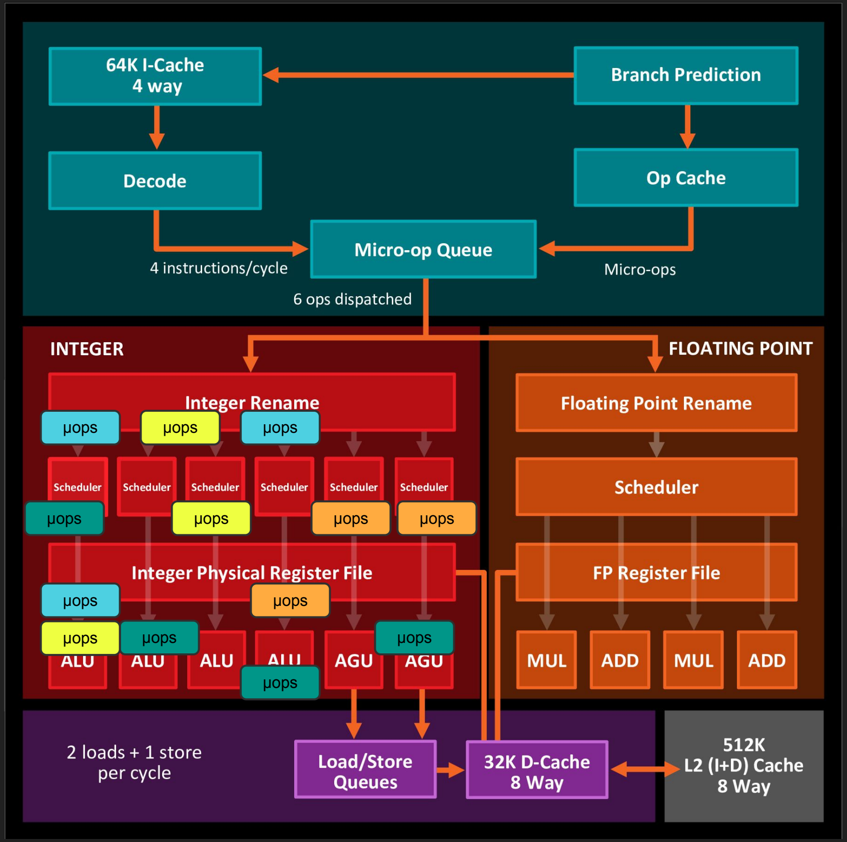
AMD Ryzen

add qword ptr [rax], rbx



mov rdx, 1

cmp rdx, qword ptr [rax]

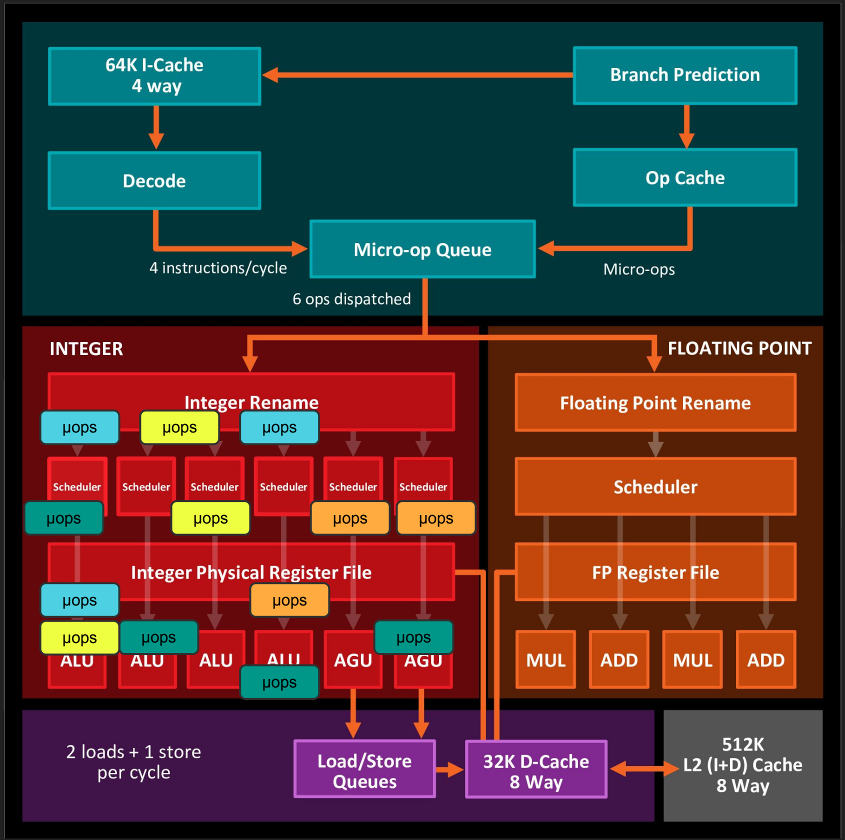


AMD Ryzen

```
add qword ptr [rax], rbx
mov rdx, 1
cmp rdx, qword ptr [rax]
jne 0xdeadbeef
```

…   …

…   …



64K I-Cache
4 way

Branch Prediction

Decode

Op Cache

Micro-op Queue

4 instructions/cycle

Micro-ops

6 ops dispatched

INTEGER

Integer Rename

µops µops µops

Scheduler Scheduler Scheduler Scheduler Scheduler Scheduler

µops µops µops µops

Integer Physical Register File

µops µops

µops µops µops

ALU ALU ALU ALU AGU AGU

µops

FLOATING POINT

Floating Point Rename

Scheduler

FP Register File

MUL ADD MUL ADD

2 loads + 1 store
per cycle

Load/Store
Queues

32K D-Cache
8 Way

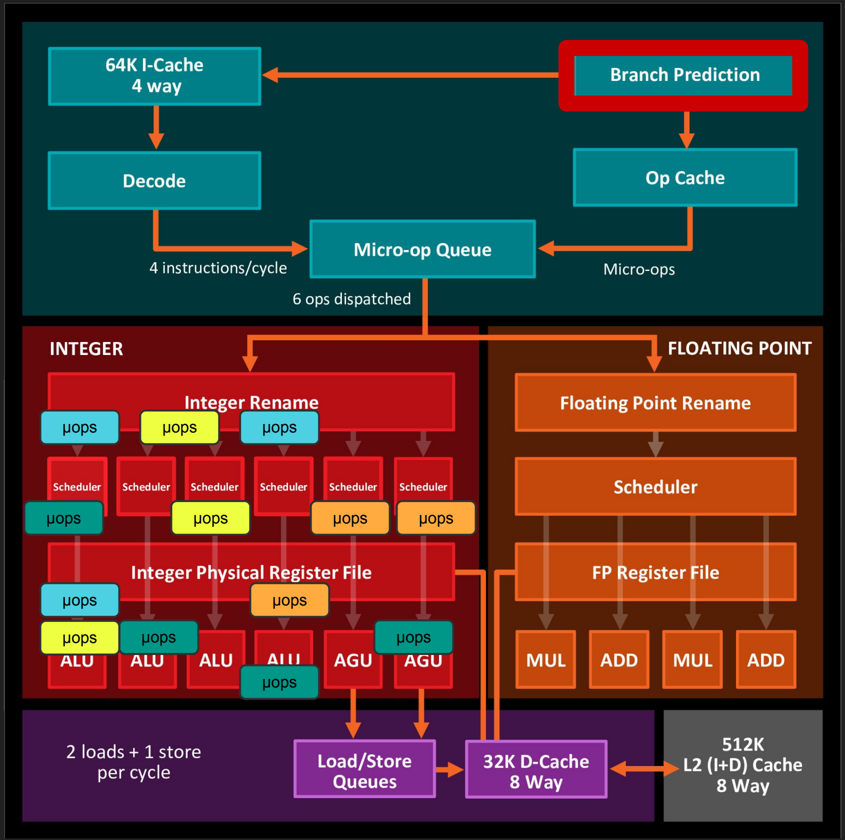
512K
L2 (I+D) Cache
8 Way

AMD Ryzen

```
add qword ptr [rax], rbx
mov rdx, 1
cmp rdx, qword ptr [rax]
jne 0xdeadbeef
```

... ...

... ...



Branch Prediction

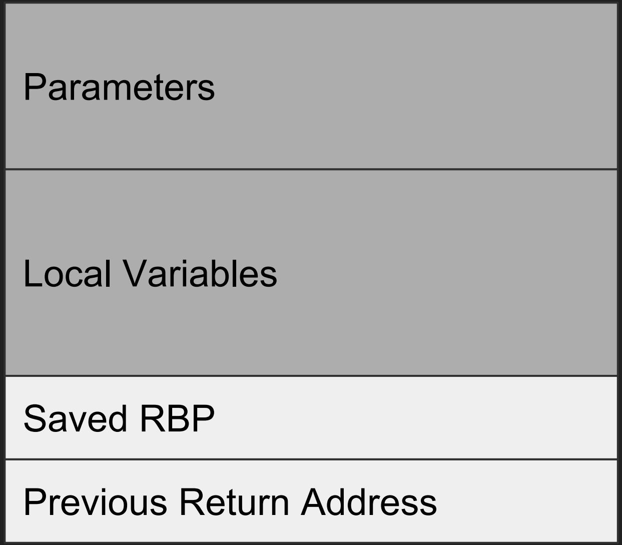
```
void foo(int i) {
        int a = 3;
        int b = 5;
        if(i > 0) ←
                bar(a, b); ←
}

int bar(int x, int y) {
        return x+y; ←
}
```

Branch Prediction

```
void foo(int i) {
        int a = 3;
        int b = 5;
        if(i > 0)
                bar(a, b);
}

int bar(int x, int y) {
        return x+y;
}
```

Return Stack Buffer

```
void foo(int i) {
        int a = 3;
        int b = 5;
        if(i > 0)
                bar(a, b);    ←
}

int bar(int x, int y) {
        return x+y;
}
```



Parameters

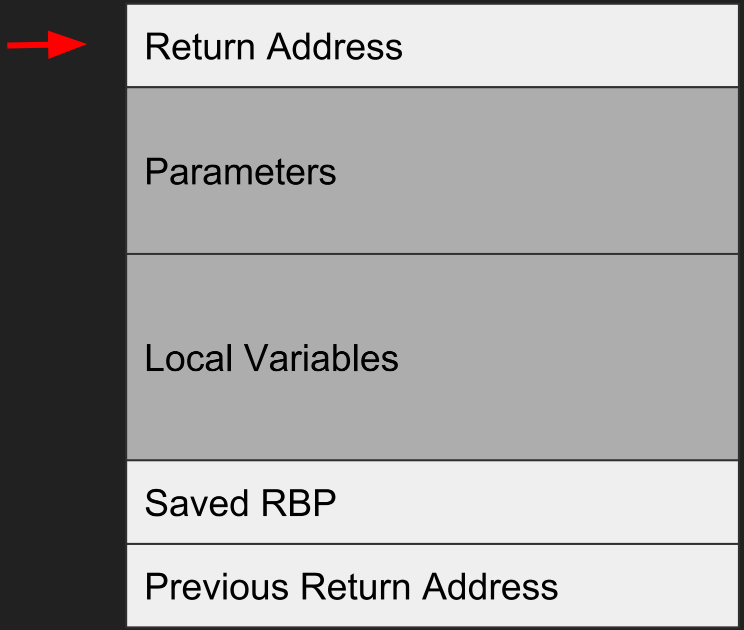
Local Variables

Saved RBP

Previous Return Address

Return Stack Buffer

```
void foo(int i) {
        int a = 3;
        int b = 5;
        if(i > 0)
                bar(a, b);
}

int bar(int x, int y) {
        return x+y;
}
```



Return Address

Parameters

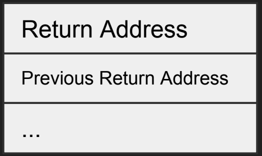
Local Variables

Saved RBP

Previous Return Address

Return Stack Buffer

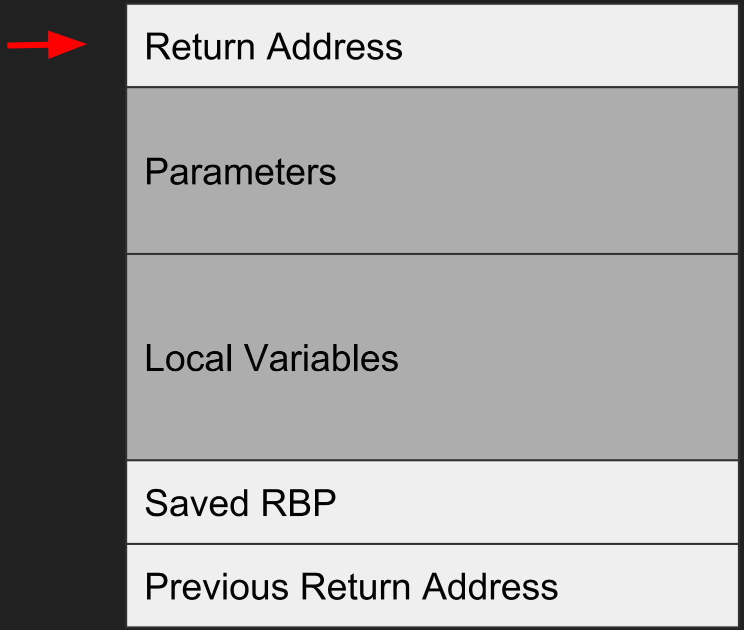
```
void foo(int i) {
        int a = 3;
        int b = 5;
        if(i > 0)
                bar(a, b);
}

int bar(int x, int y) {
        return x+y;
}
```



Return Address

Previous Return Address

...



Return Address

Parameters

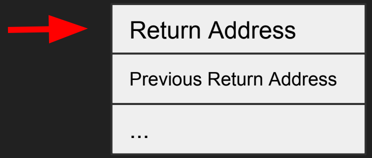
Local Variables

Saved RBP

Previous Return Address

Return Stack Buffer

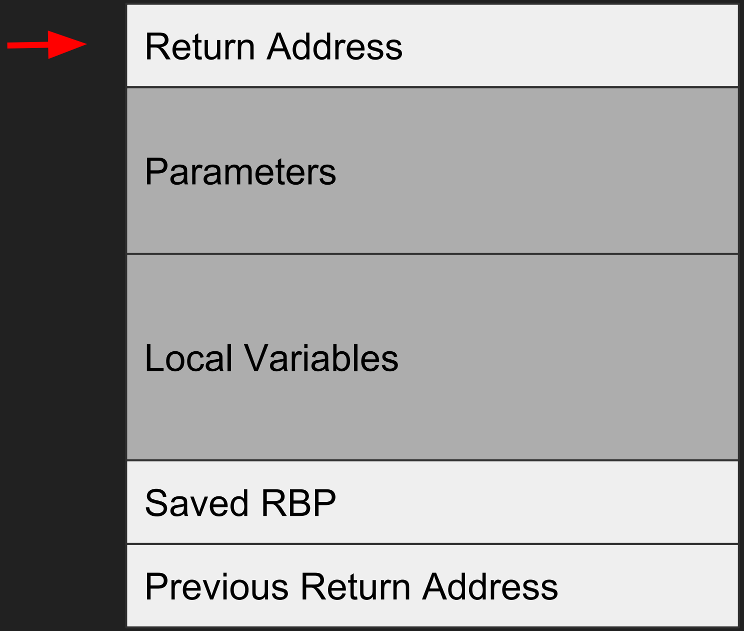
```
void foo(int i) {
        int a = 3;
        int b = 5;
        if(i > 0)
                bar(a, b);
}

int bar(int x, int y) {
        return x+y;
}
```



Return Address

Previous Return Address

...



Return Address

Parameters

Local Variables

Saved RBP

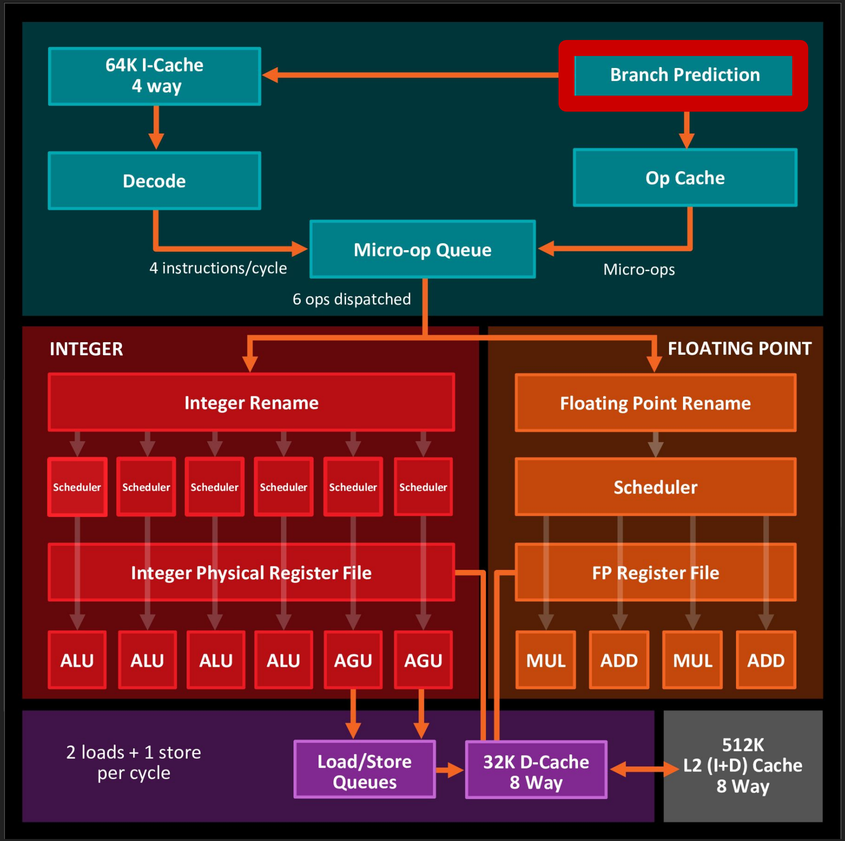
Previous Return Address

Return Stack Buffer

ret

↓

?



Return Address
...
...

→ Stack

Outline

1. What is a ROP exploit?

2. How do Modern Processors work?

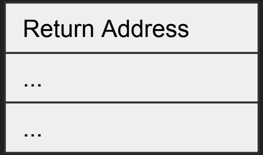
3. Let’s use HW to save the world from ROP!

Let’s notice something...

We will write our ROPchain on the stack

It will execute a lot of ret

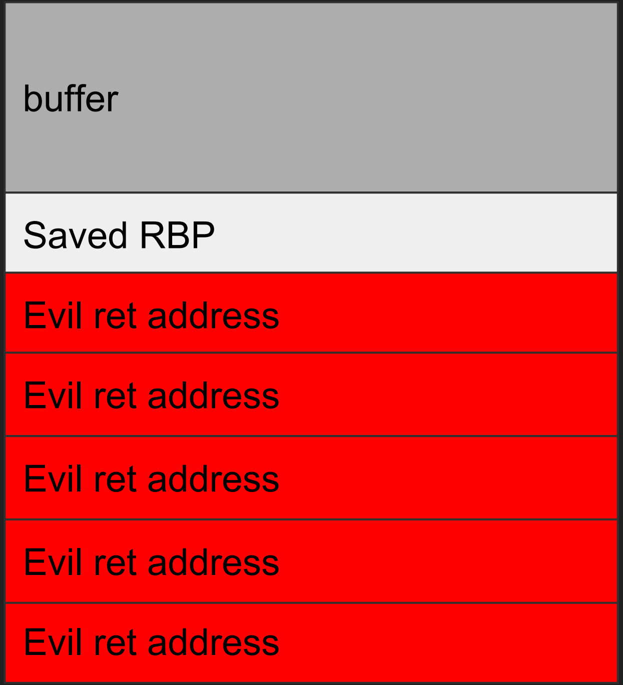
But no call has ever inserted any right address in the RAS



Return Address

...

...



buffer

Saved RBP

Evil ret address

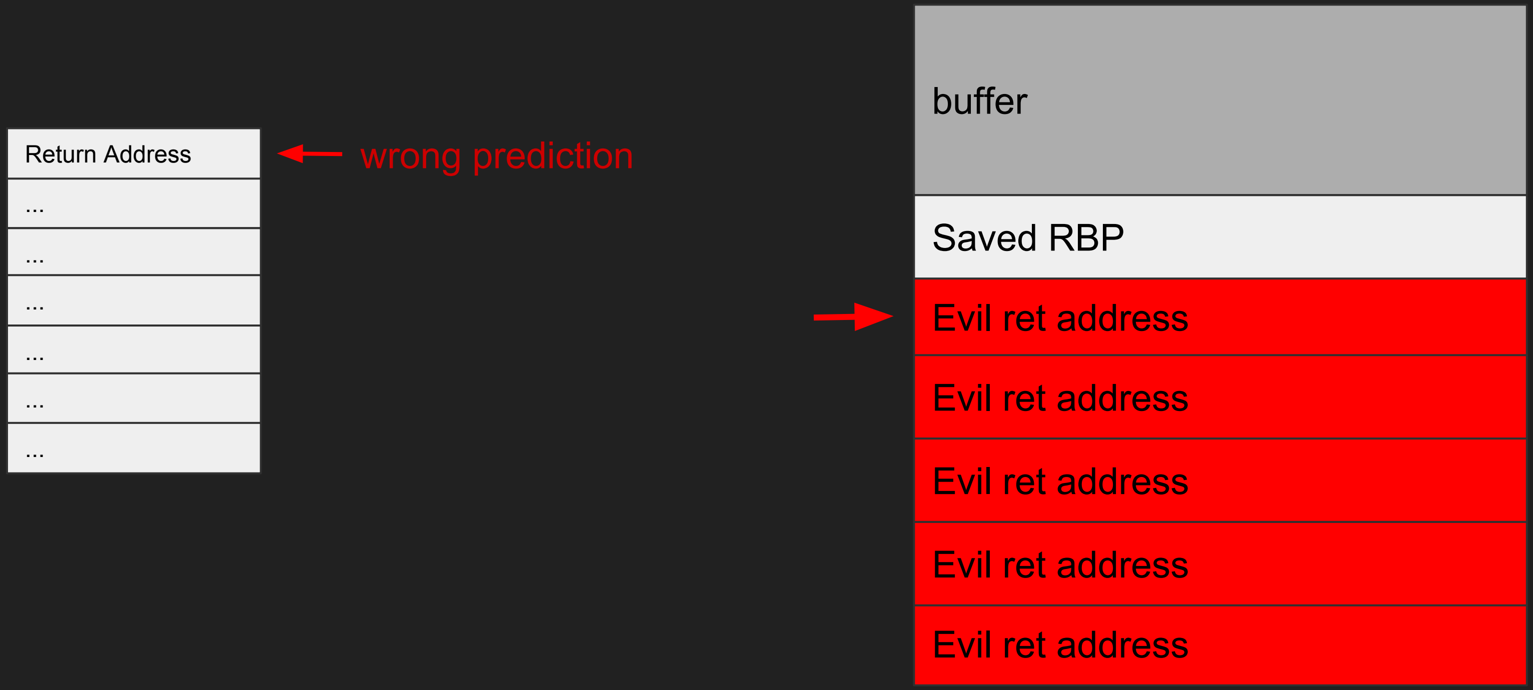
Evil ret address

Evil ret address

Evil ret address

Evil ret address

Let’s notice something...



Return Address

...

...

...

...

...

...

wrong prediction

buffer

Saved RBP

Evil ret address

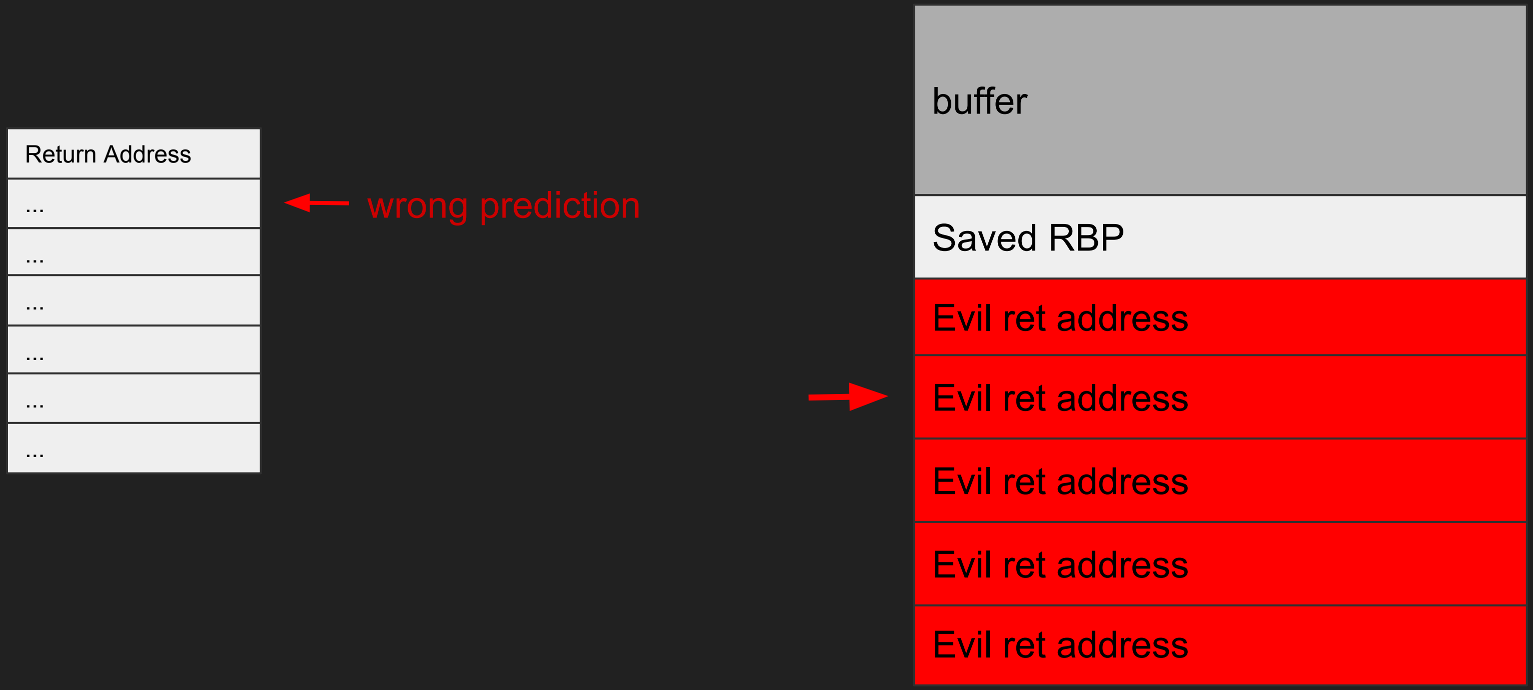
Evil ret address

Evil ret address

Evil ret address

Evil ret address

Let’s notice something...



Return Address

...

...

...

...

...

...

wrong prediction

buffer

Saved RBP

Evil ret address

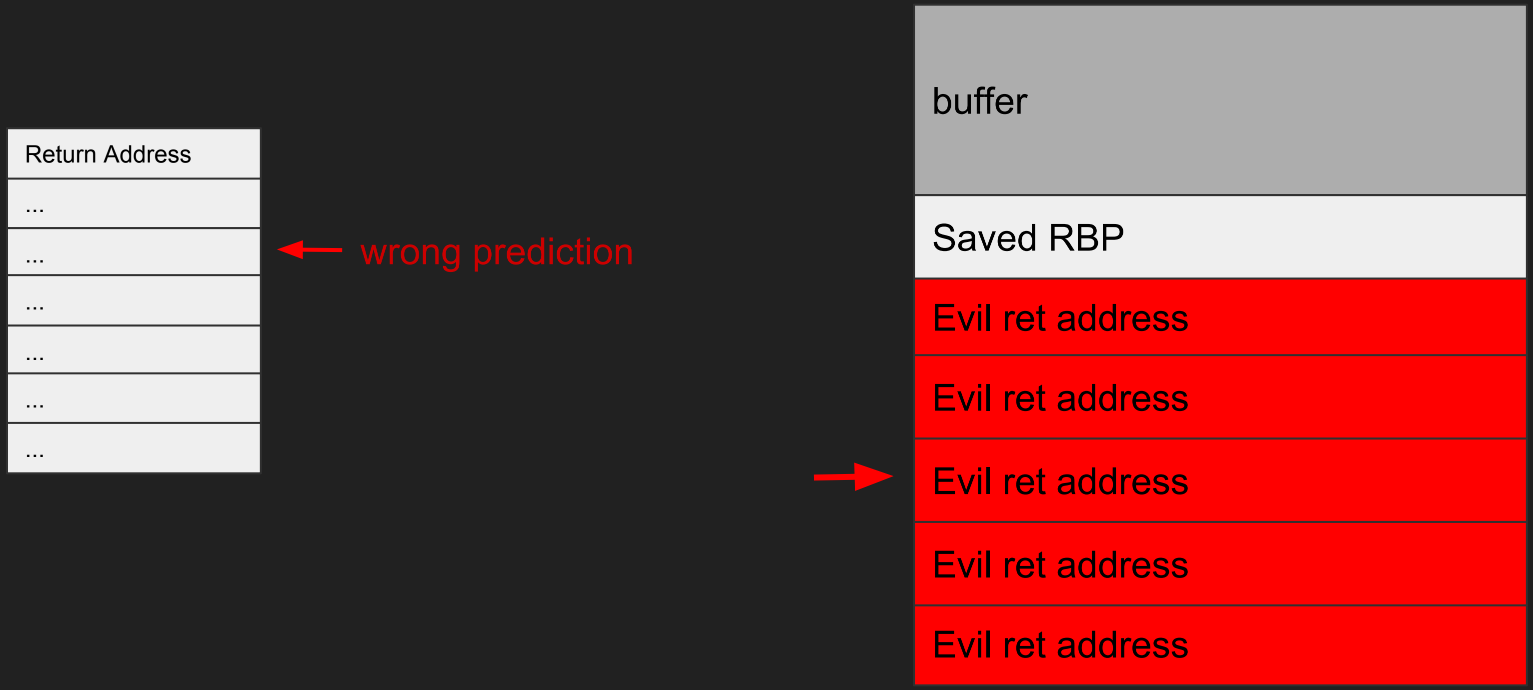
Evil ret address

Evil ret address

Evil ret address

Evil ret address

Let’s notice something...



Return Address

...

...

wrong prediction

...

...

...

...

buffer

Saved RBP

Evil ret address

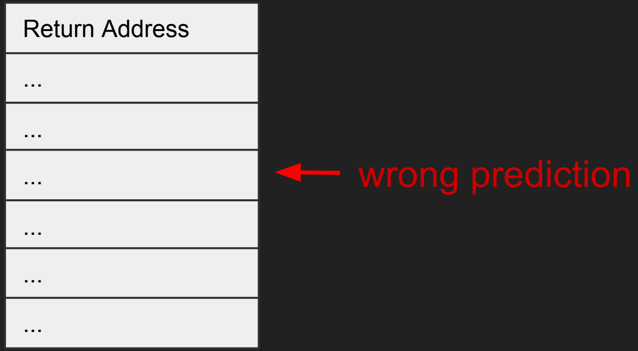
Evil ret address

Evil ret address

Evil ret address

Evil ret address

Let’s notice something...



Return Address

...

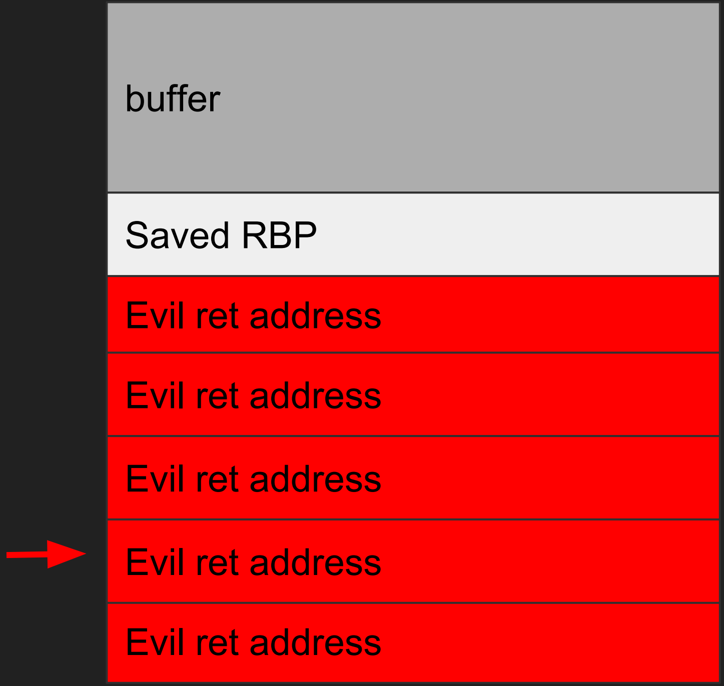
...

...

...

...

...



buffer

Saved RBP

Evil ret address

Evil ret address

Evil ret address

Evil ret address

Evil ret address

High rate of misprediction in the system!

RopGun

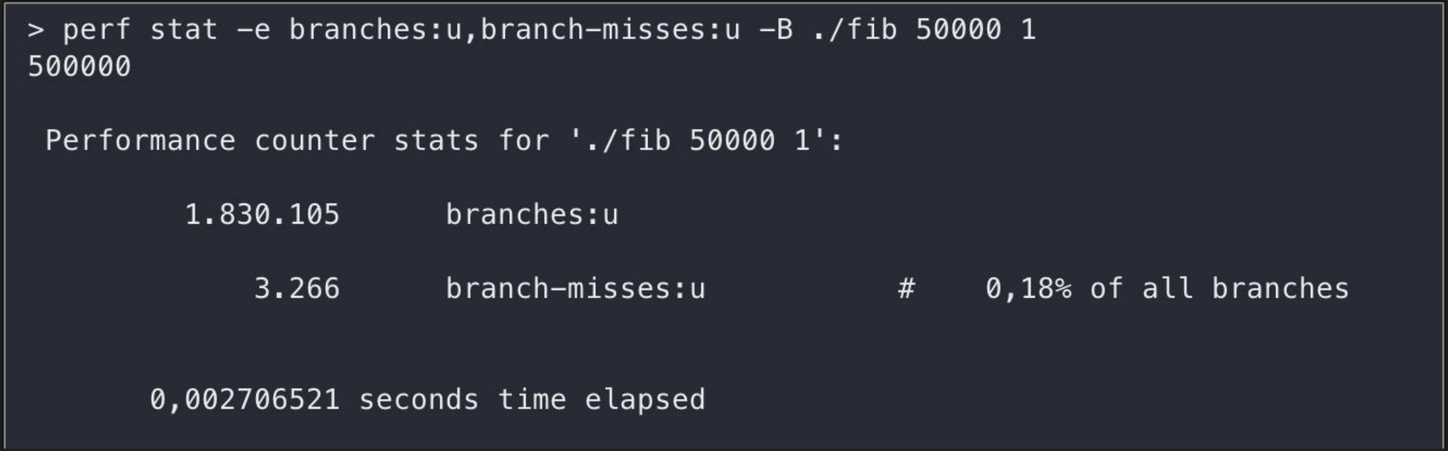
Now the idea is simple!

Just monitor the misprediction rate (through performance counters) in the system to detect ROP payloads executing!

Kill a process when the misprediction rate is too high

How?

Use Performance Monitoring Counters!



```
> perf stat -e branches:u,branch-misses:u -B ./fib 50000 1
500000

 Performance counter stats for './fib 50000 1':

         1.830.105      branches:u

             3.266      branch-misses:u           #    0,18% of all branches


       0,002706521 seconds time elapsed
```

RopGun

DEMO 2

Thank you!

Questions?