Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Return-oriented programming without returns

S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi,
H. Shacham, M. Winandy

Dresden, 2010-10-20

Fundamental problem with stacks

- User input gets written to the stack.
- x86 allows to specify only read/write rights.

- Idea:
  - Create programs so that memory pages are either writable or executable, never both.
  - ***W ^ X paradigm***

- Software: OpenBSD *W^X*, PaX, RedHat *ExecShield*
- Hardware: Intel XD, AMD NX, ARM XN

A perfect W^X world

- User input ends up in writable stack pages.
- No execution of this data possible – problem solved.
- But: existing code assumes executable stacks
  - Windows contains a DLL function to disable execution prevention – used e.g. for IE <= 6
  - Nested functions: GCC generates trampoline code on stack

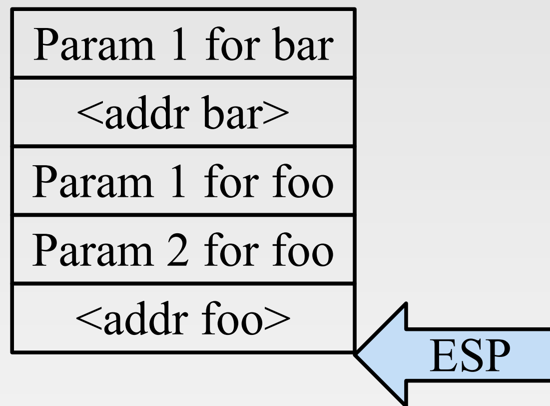
Circumventing W^X

- We cannot anymore: execute code on the stack directly
- We still can: Place data on the stack
  - Format string attacks, non-stack overflows, …
- Idea: modify return address to start of function known to be available
  - e.g., a libC function such as execve()
  - put additional parameters on stack, too

return-to-libC attack

Chaining returns

- Not restricted to a single function:
  - Modify stack to return to another function after the first:



  - And why only return to function beginnings?

Return anywhere

- x86 instructions have variable lengths (1 – 16 bytes)
  - → x86 allows jumping (returning) to an ***arbitrary address***
- Idea: scan binaries/libs and find all possible ret instructions
  - Native RETs: **0xC3**
  - RET bytes within other instructions, e.g.
    - MOV %EAX, %EBX
      0x89 **0xC3**
    - ADD $1000, %EBX
      0x81 **0xC3** 0x00 0x10 0x00 0x00

Return anywhere

- Example instruction stream:

```
.. 0x72 0xf2 0x01 0xd1 0xf6 0xc3 0x02 0x74 0x08 ..

0x72 0xf2             jb <-12>
0x01 0xd1             add %edx, %ecx
0xf6 0xc3 0x02        test $0x2, %bl
0x74 0x08             je <+8>
```

- Three byte forward:

```
.. 0x72 0xf2 0x01 0xd1 0xf6 0xc3 0x02 0x74 0x08 ..

0xd1 0xf6             shl, %esi
0xc3                  ret
```

Many different RETs

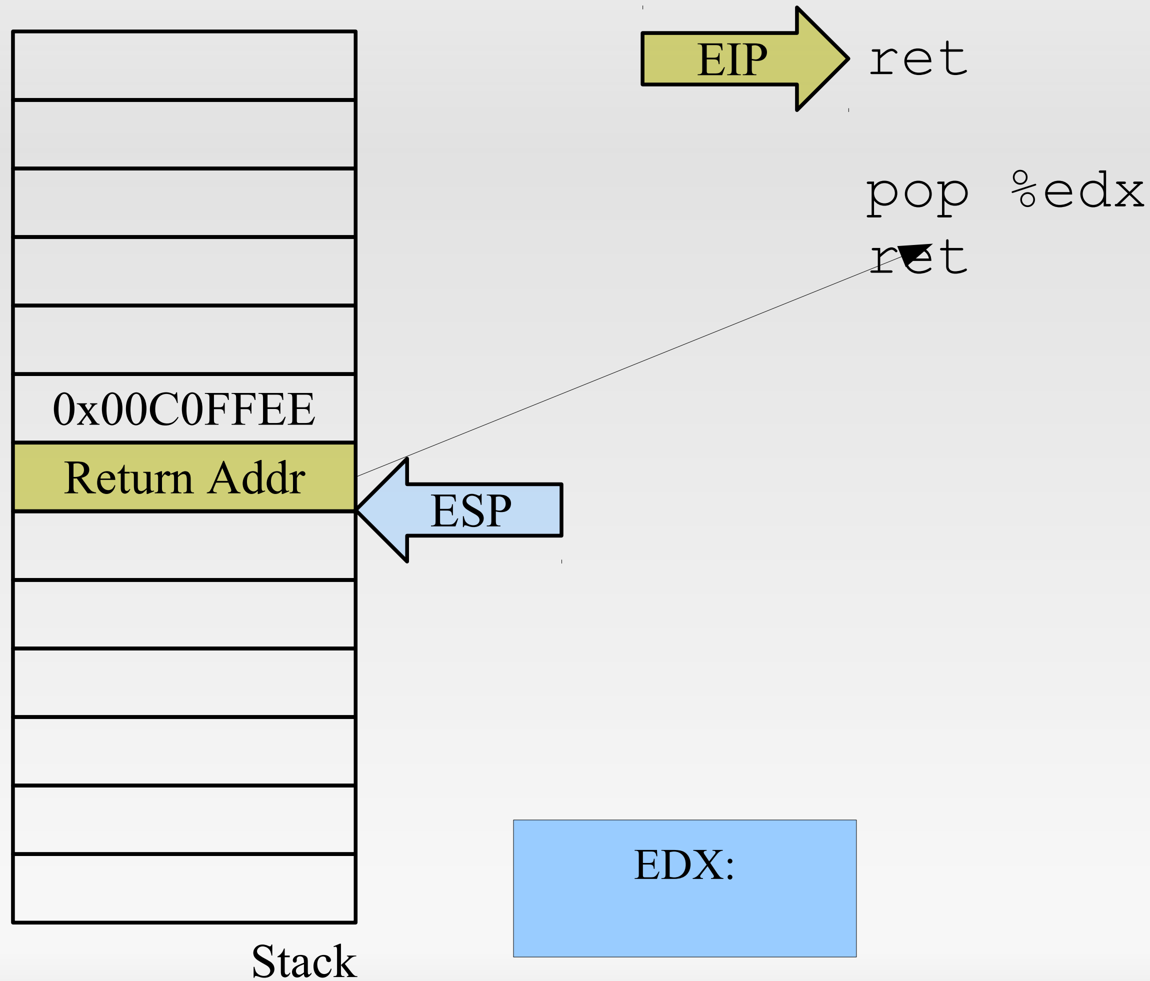
- Claim:
  - Any sufficiently large code base
    e.g. libC, libQT, ...
  - consists of 0xC3 bytes
    == RET
  - with sufficiently many different prefixes
    == a few x86 instructions terminating in RET
    (in [Sha07]: ***gadget***)
- *"sufficiently many"*: `/lib/libc.so.6` on Ubuntu 10.4
  - ~17,000 sequences (~6,000 unique)

Return-Oriented Programming

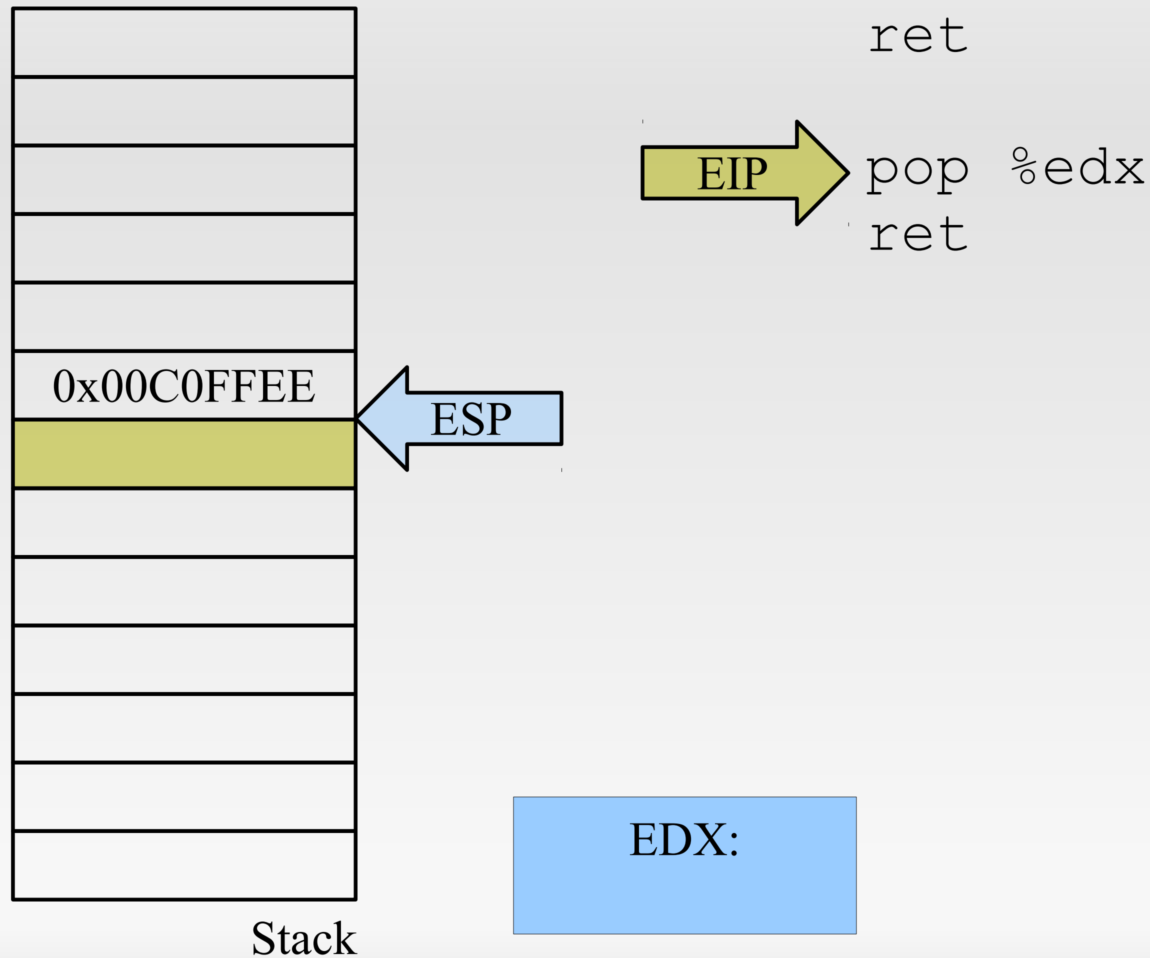
- Return addresses jump to code **gadgets** performing a small amount of work
- Stack contains
  - Data arguments
  - Chain of addresses returning to gadgets
- Claim: This is enough to write arbitrary programs (and thus: shell code).

Return-oriented Programming

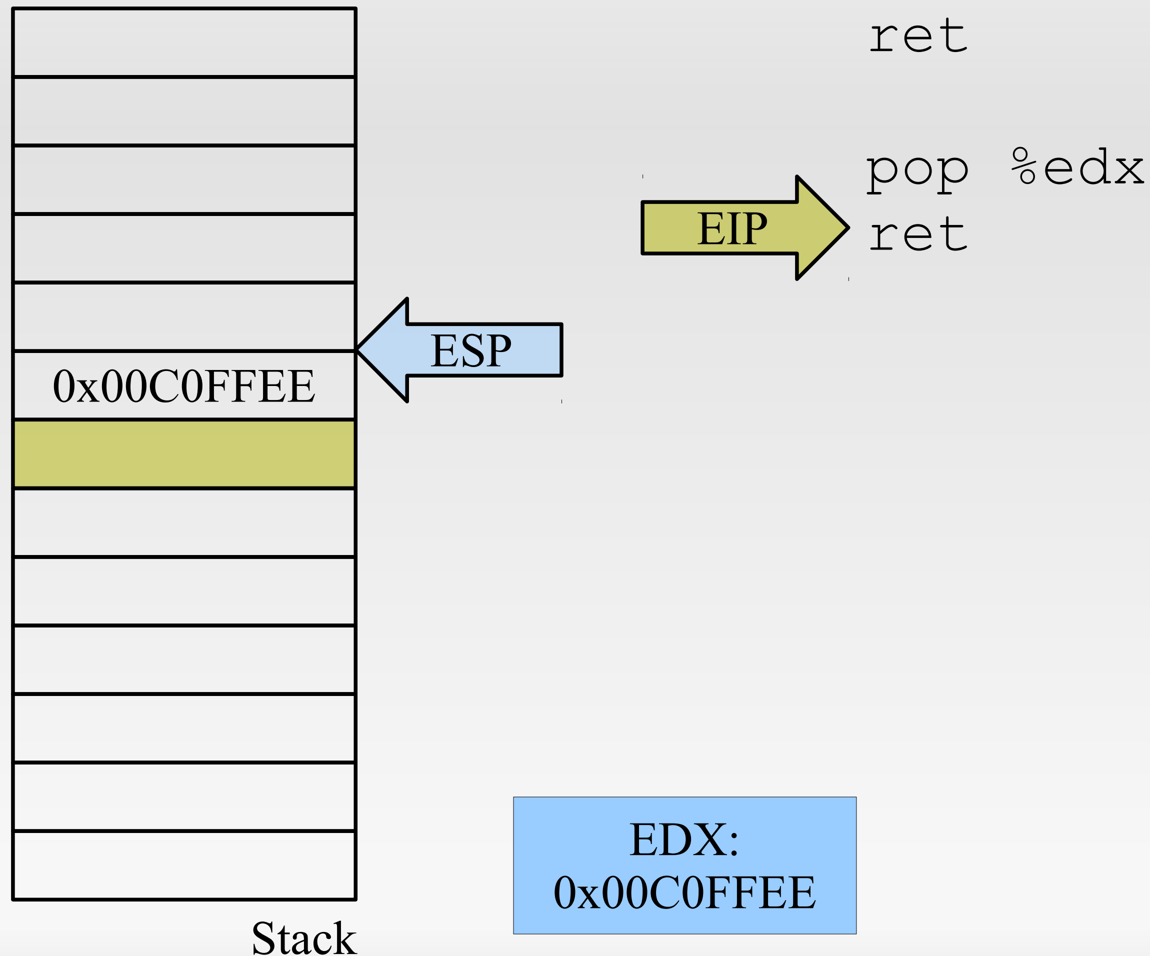
ROP: Load constant into register



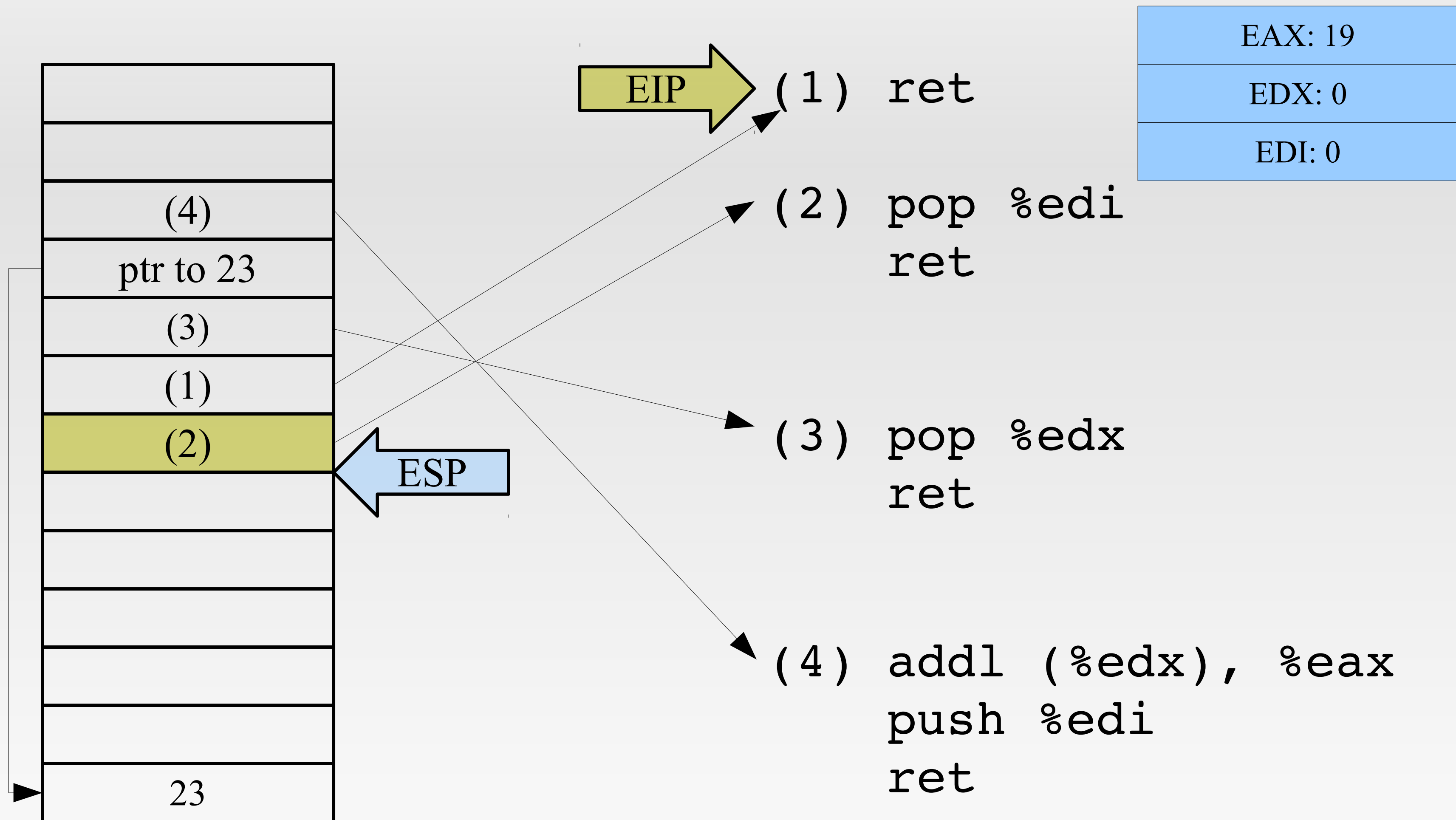
ROP: Load constant into register



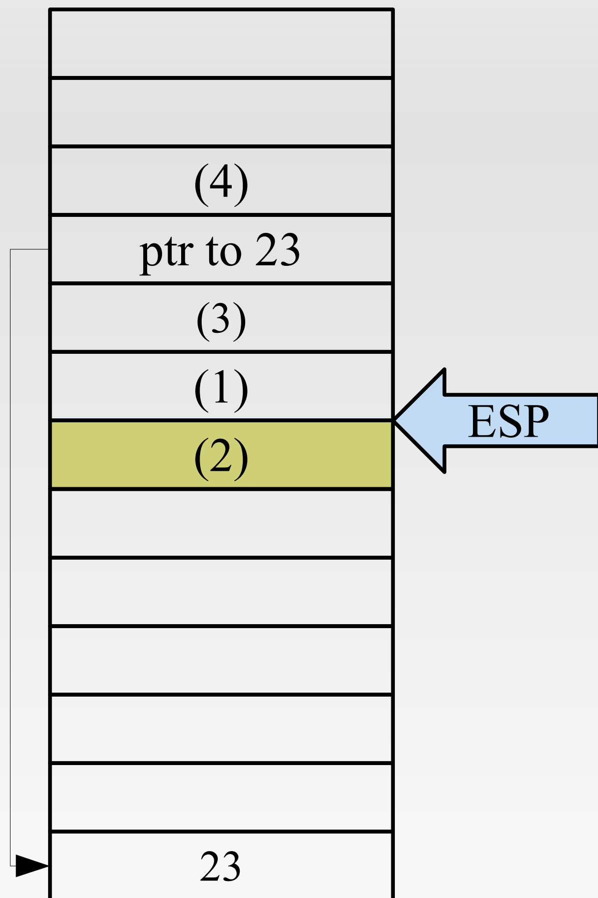
ROP: Load constant into register

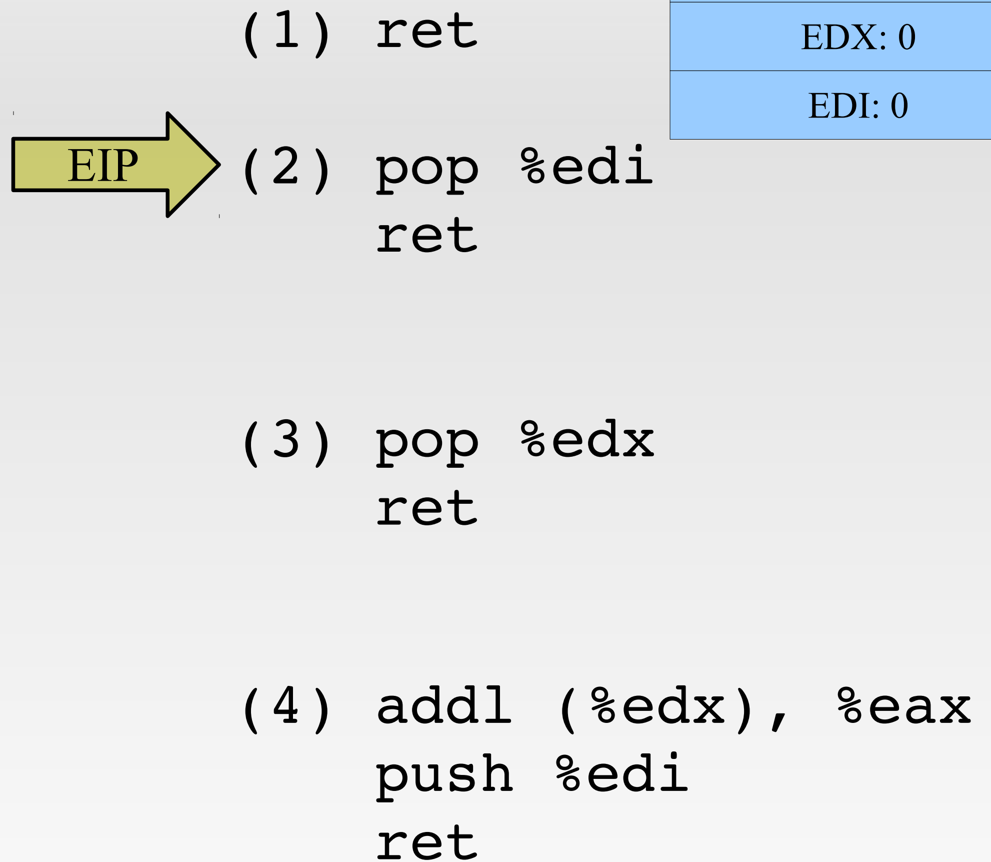


ROP: Add 23 to EAX

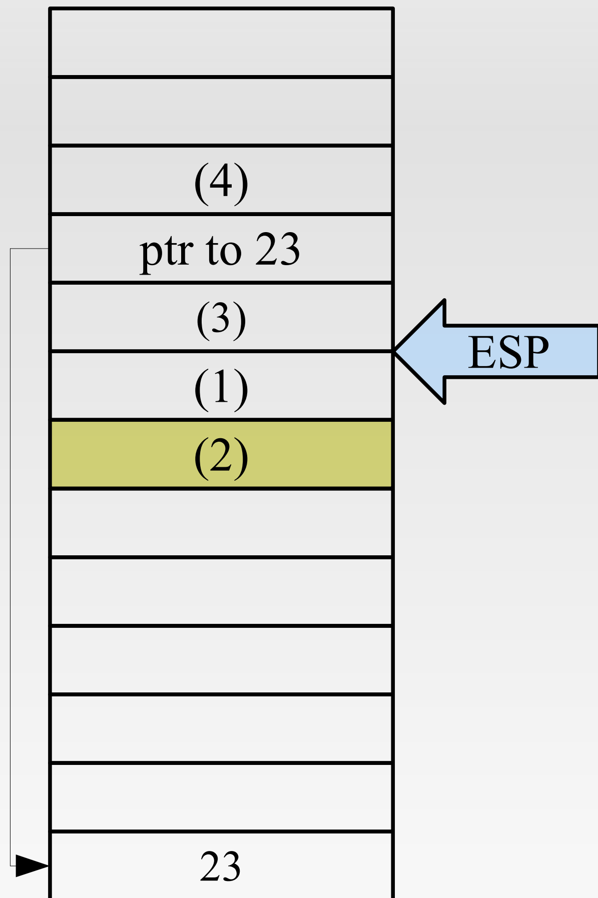


ROP: Add 23 to EAX





ROP: Add 23 to EAX



EAX: 19
EDX: 0
EDI: addr of (1)

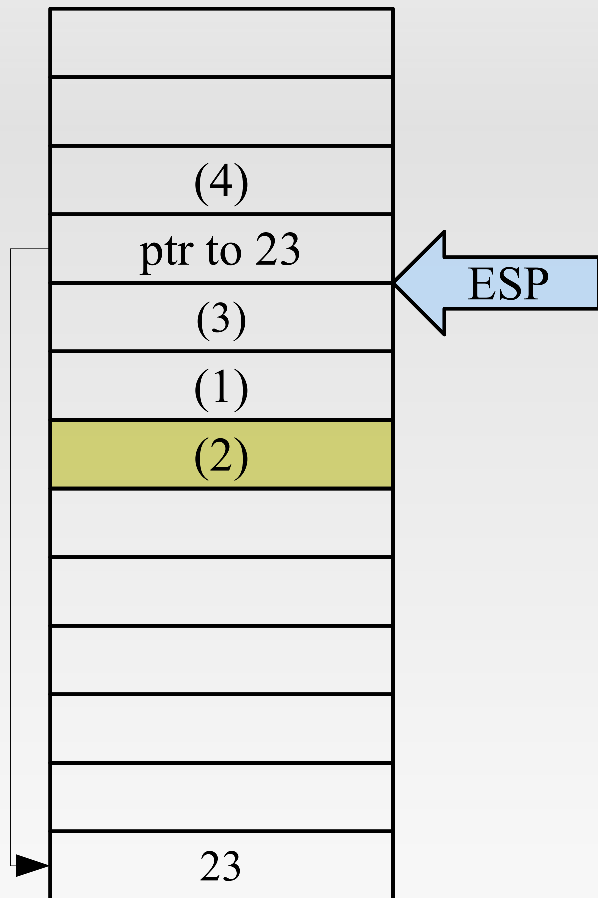
```
(1) ret

(2) pop %edi
    ret

(3) pop %edx
    ret

(4) addl (%edx), %eax
    push %edi
    ret
```

ROP: Add 23 to EAX



EIP

EAX: 19
EDX: 0
EDI: addr of (1)

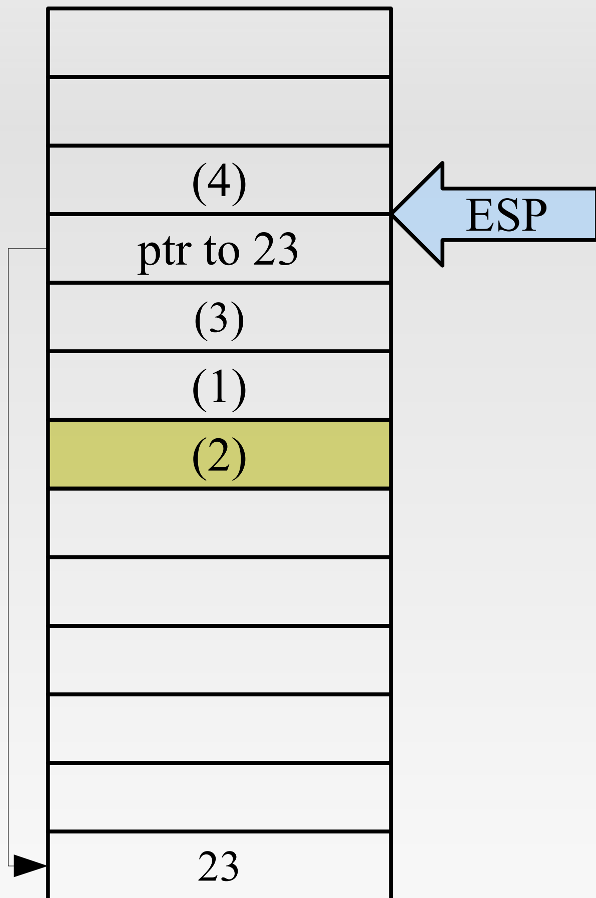
```
(1) ret

(2) pop %edi
    ret

(3) pop %edx
    ret

(4) addl (%edx), %eax
    push %edi
    ret
```

ROP: Add 23 to EAX



EAX: 19
EDX: addr of '23'
EDI: addr of (1)

EIP

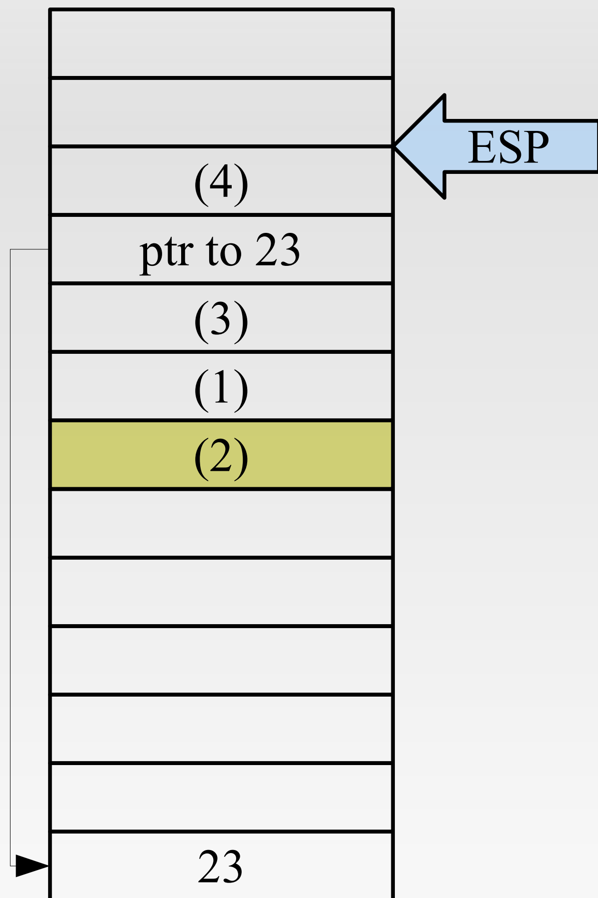
```
(1) ret

(2) pop %edi
    ret

(3) pop %edx
    ret

(4) addl (%edx), %eax
    push %edi
    ret
```

ROP: Add 23 to EAX



EAX: 19
EDX: addr of '23'
EDI: addr of (1)

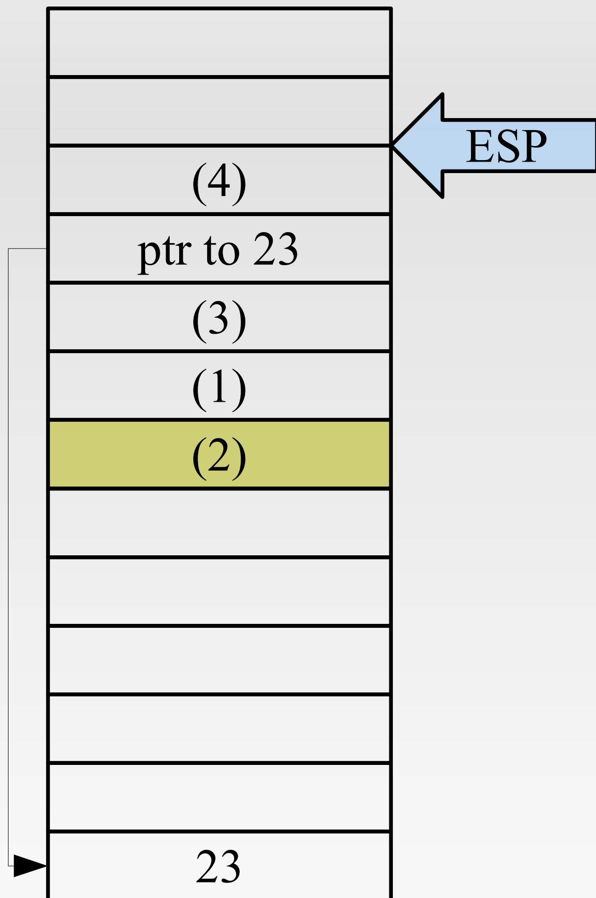
```
(1) ret

(2) pop %edi
    ret

(3) pop %edx
    ret

(4) addl (%edx), %eax
    push %edi
    ret
```

ROP: Add 23 to EAX



EAX: 42
EDX: addr of '23'
EDI: addr of (1)

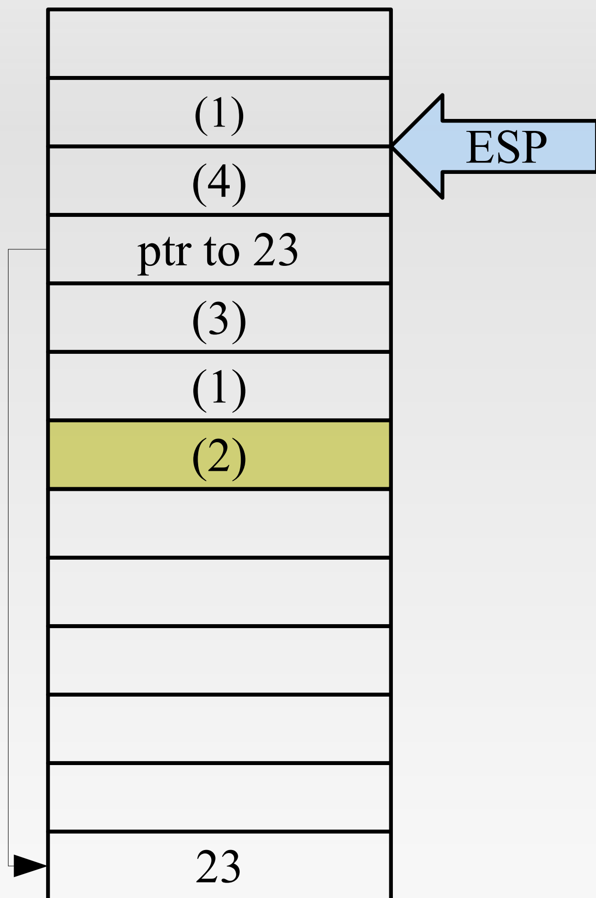
```
(1) ret

(2) pop %edi
    ret

(3) pop %edx
    ret

(4) addl (%edx), %eax
    push %edi
    ret
```

ROP: Add 23 to EAX



EAX: 42
EDX: addr of '23'
EDI: addr of (1)

```
(1) ret

(2) pop %edi
    ret

(3) pop %edx
    ret

(4) addl (%edx), %eax
    push %edi
    ret
```

Return-oriented programming

- More samples in the paper – it is assumed to be Turing-complete.
- Problem: need to use existing gadgets, limited freedom
  - Yet another limitation, but no show stopper.
- Good news: Writing ROP code can be automated, there is a C-to-ROP compiler.

ROP protection

- Assuming use of RETs:
  - Detect abnormal frequency of executed RETs
  - Ensure LIFO principle for stack pointer
  - Compile binaries without 0xC3 bytes
  - Shadow return stack
- Other:
  - Address-space layout randomization
  - Runtime CFI checking

ROP without RETs

- Dissecting RET: 2 operations at once
  - Memory-indirect JMP (modifies control flow)
  - Update processor state (stack pop on x86, register load on ARM)

- Is it necessary to use it?
  - No! RET-less compilers show exactly this.
  - Just use some sequence that does exactly the same:

```
pop %edx       // modifies stack
jmp *(%edx)    // indirect jump
```

Update-load-branch

- **Update**: update control structures to point to next gadget
- **Load**: load next gadget's address
- **Branch**: Jump

- Problem: occurs much less frequent than RET
- Solution:
  - use exactly **one** Update-Load-Branch sequence as a trampoline
  - reserve a register as pointer to trampoline
  - then: all sequences ending in indirect jmp through register can serve as gadgets

The many faces of update-load-branch

- Any `pop X; jmp *X` sequence suffices.
- Doubly indirect jump
  - JMP on x86 can have register or memory operand
  - Use memory operand: adversary data can contain a table of usable gadgets → **sequence catalog**
  - May even contain immediate operands, such as `jmp *4(%edx)`
  - Both, `jmp` and `ljmp` are valid.

ROP gadgets without RET

- Debian libC
  - contains **no** ULB sequence!
  - add Mozilla's libxul and libphp or customize attack to target application
- Trampoline from libxul uses %ebx
  - Trampoline address stored in %edx
  - Gadgets must end with `jmp *(%edx)`
- Chose 34 sequences to construct 19 gadgets to show Turing-completeness of approach.
  - Only a subset of possible sequences
  - Still far fewer than the 6,000 RET sequences in my libC

Load register / Store memory

```
pop %eax
sub %dh, %bl
jmp *(%edx)
```

```
mov 4(%eax), %ecx
jmp *(%edx)
```

```
mov %esi, -0xb(%eax)
jmp *(%edx)
```

Not-so-difficult gadgets

- Move within memory: combine
  - Load from memory to register
  - Store from register to memory
- Arithmetic negate, phase 1: (Goal: %esi := - <val>)

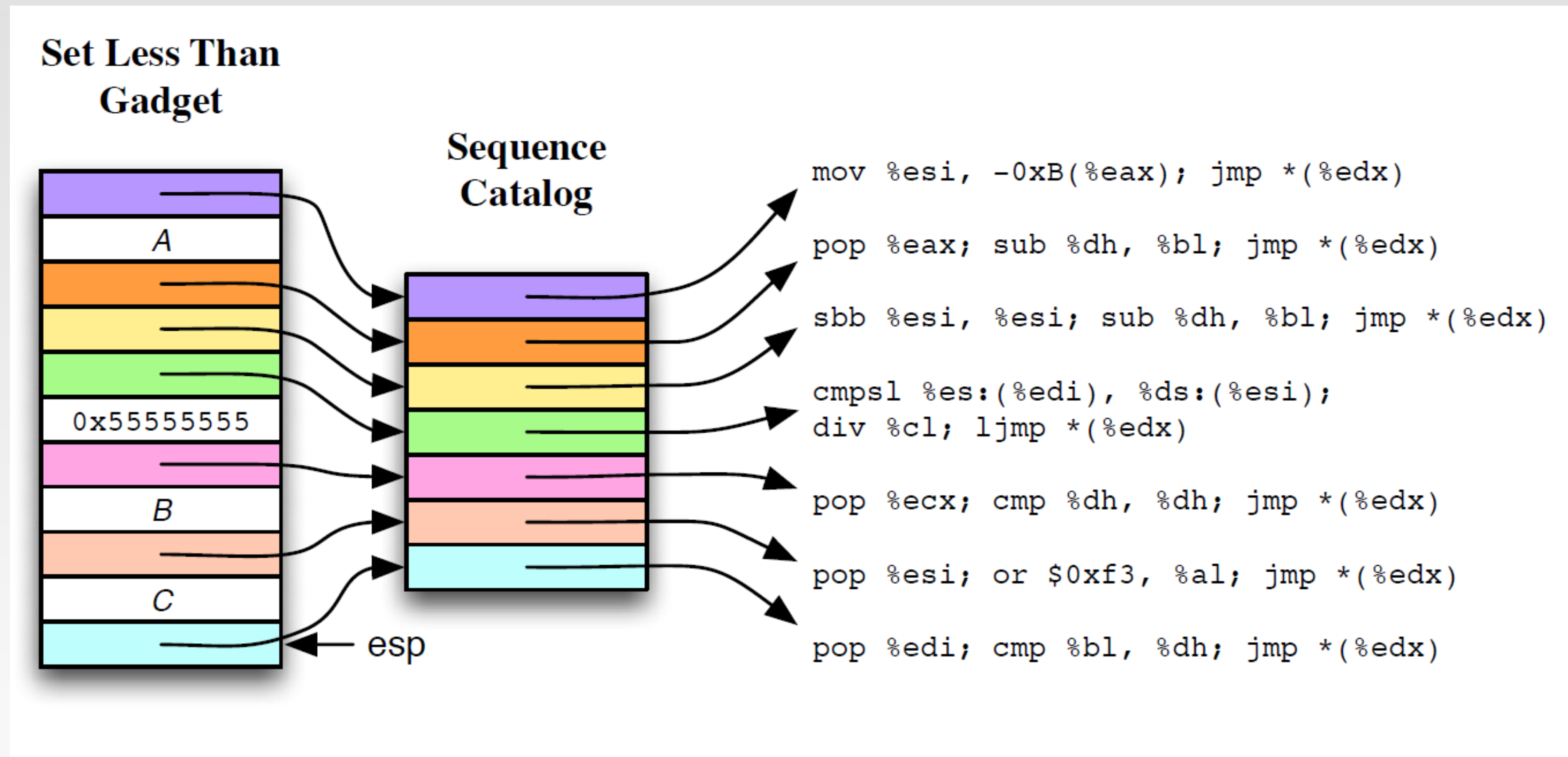
```
xor %esi, %esi   // %esi := 0
jmp (%edx)       // trampoline
```

- Arithmetic negate, getting tricky:

```
subl -0x7D(%ebp, %ecx, 1), %esi
   // %esi := - (%ebp + 1*%ecx - 0x7D)
   // requires
   // %ebp == <val> + 0x7D - <jmp target>
jmp (%ecx)      // next gadget
```

Set-less-than

- Goal: if (a < b) result = -1; else result = 0;



Getting the attack to run

- Need attacks that don't require any RET
- Stack overflow:
  - Don't overflow RET address (would violate LIFO order)
  - Instead overwrite higher-level function's local data, especially if this is later used for determining where to branch
- Overwrite SETJMP buffers
- Overwrite C++ vtables and function pointers
  - Deemed practically impossible without use of RET

Discussion

- Is CFI the ultimate solution?
  - Overhead
  - More code → more gadgets? – but all jmp sequences look identical
  - CFI vs. JIT compilation???
- Allowing JNI on Android (or in any JVM) is obviously broken.
- Is everything lost?