Return-Oriented Programming

- Return-Oriented Programming (ROP) is an attack technique that injects no code, yet can induce arbitrary behavior in the targeted system.

Return-Oriented Programming

- Return-Oriented Programming (ROP) is an attack technique that injects no code, yet can induce arbitrary behavior in the targeted system.
- The code aggregates malicious computing by linking together short code snippets already present in the program's address space.
  - Each snippet ends with a `ret` instruction, which allows an attacker controlling the stack to chain them together.
  - Because the executed code is stored in memory marked executable, page protection mechanisms won't help.

Return-Oriented Programming

- Return-Oriented Programming (ROP) is an attack technique that injects no code, yet can induce arbitrary behavior in the targeted system.
- The code aggregates malicious computing by linking together short code snippets already present in the program's address space.
  - Each snippet ends with a `ret` instruction, which allows an attacker controlling the stack to chain them together.
  - Because the executed code is stored in memory marked executable, page protection mechanisms won't help.
- Still needs a buffer overflow or similar mechanism to subvert the control flow of the program.

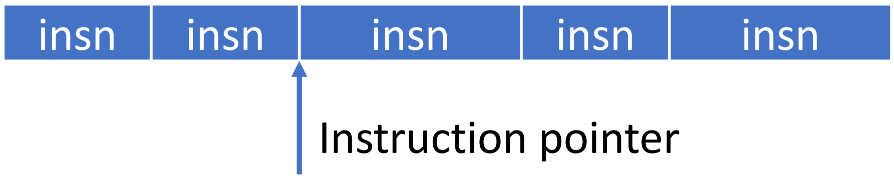
Program Layout

- An ordinary program (OP) is a sequence of machine instructions (MIs) laid out in the *text* segment of the program.
  - Each MI is a byte pattern that, when interpreted by the processor, induces a small amount of change in machine state.
  - The instruction pointer `%rip` governs the MI that is fetched and executed next.

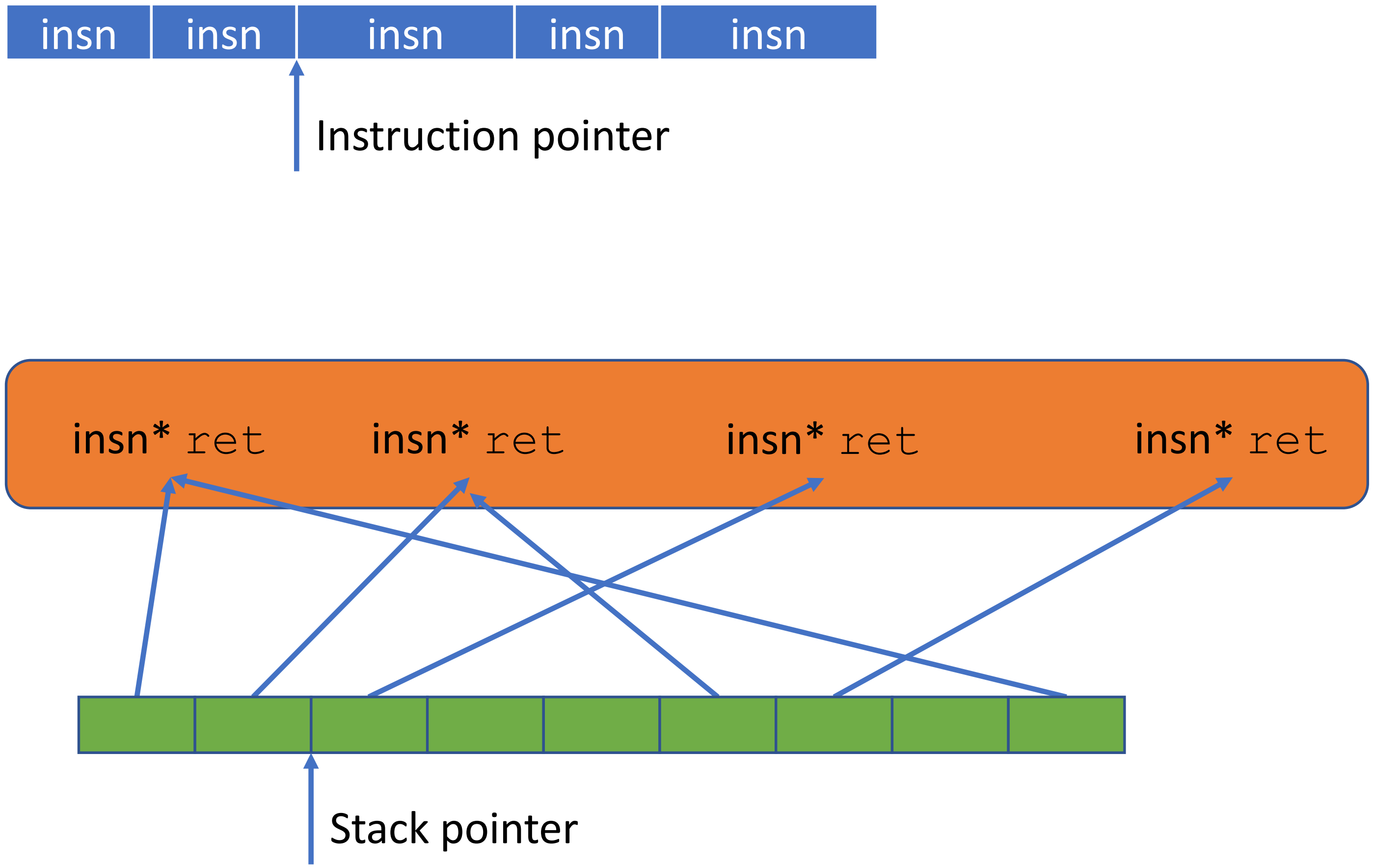
Program Layout

- An ordinary program (OP) is a sequence of machine instructions (MIs) laid out in the *text* segment of the program.
  - Each MI is a byte pattern that, when interpreted by the processor, induces a small amount of change in machine state.
  - The instruction pointer `%rip` governs the MI that is fetched and executed next.
- A return-oriented program (RP) is a sequence of return-oriented instructions (RIs) laid out in the *stack* segment of the exploited program.
  - Each RI is a sequence of words on the stack pointing to a sequence of MIs ending in a `ret`, somewhere in the address space of the exploited program.
  - The stack pointer governs what MI sequence is to be fetched next in the following way. The execution of a `ret` instruction has two effects: first, the word to which `%rsp` points is read and used as the new value for `%rip`; second, `%rsp` is incremented by 8 bytes to point to the next word on the stack.

OP vs RP

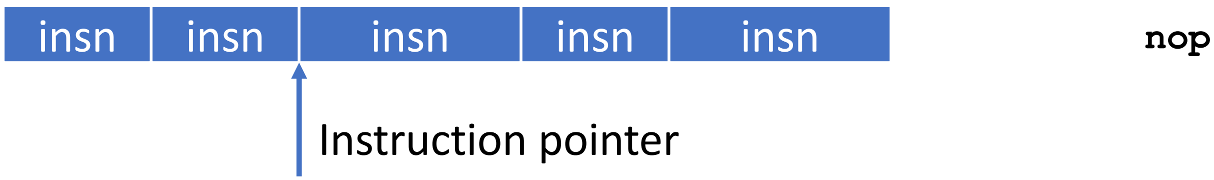


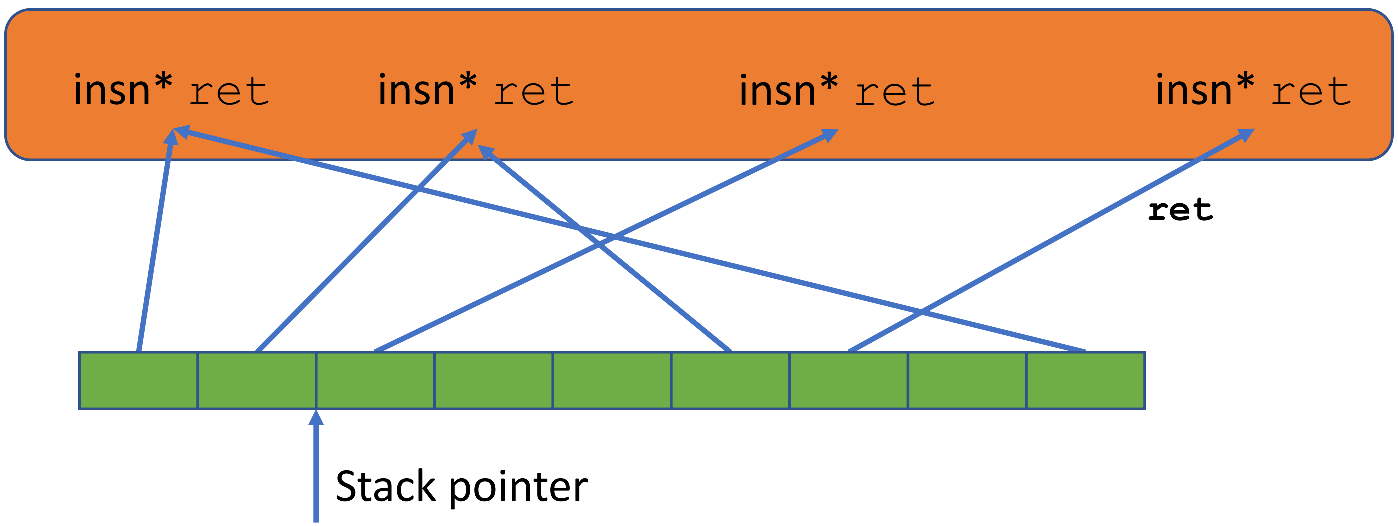
OP vs RP



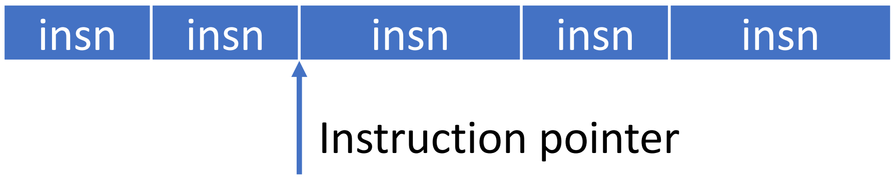
insn
insn
insn
insn
insn
Instruction pointer
insn* ret
insn* ret
insn* ret
insn* ret
Stack pointer

OP vs RP

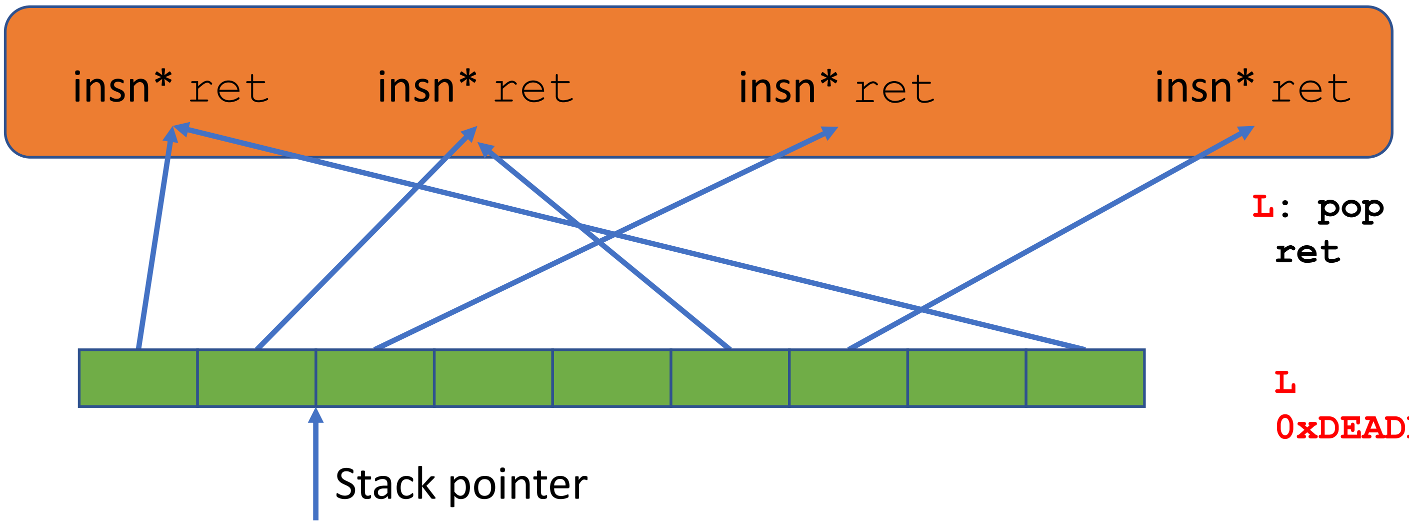

insn
insn
insn
insn
insn
Instruction pointer
nop


insn* ret
insn* ret
insn* ret
insn* ret
ret
Stack pointer

OP vs RP



`mov $0xDEADBEEF, %eax`




```
L: pop %eax
 ret
```

```
L
0xDEADBEEF
```

Gadgets

- A gadget is an arrangement of words on the stack, including one or more instruction sequence pointers and associated immediate values, that encodes a logical unit of work in the RP.
    - Gadgets act like a return-oriented instruction set and are the natural target of a return-oriented compiler’s assembler.

Gadgets

- A gadget is an arrangement of words on the stack, including one or more instruction sequence pointers and associated immediate values, that encodes a logical unit of work in the RP.
  - Gadgets act like a return-oriented instruction set and are the natural target of a return-oriented compiler's assembler.
- Example: A memory-load gadget.
  - MI sequences
    - S_1: `pop %eax; ret`
    - S_2: `mov (%eax), %ebx; ret`
  - On stack

    [Pointer to S_1, address of word to load, pointer to S_2]

Gadget Execution

- Correct execution of a gadget
  - Precondition: `%rsp` points to the first word in the gadget and the processor executes a ret instruction.
  - Postcondition: when the ret instruction in its last instruction sequence is executed, `%rsp` points to the next gadget to be executed.

Gadget Execution

- Correct execution of a gadget
  - Precondition: `%rsp` points to the first word in the gadget and the processor executes a ret instruction.
  - Postcondition: when the ret instruction in its last instruction sequence is executed, `%rsp` points to the next gadget to be executed.
- Together, these conditions guarantee that the return-oriented program will execute correctly, one gadget after another.

Gadget Execution

- Correct execution of a gadget
  - Precondition: `%rsp` points to the first word in the gadget and the processor executes a ret instruction.
  - Postcondition: when the ret instruction in its last instruction sequence is executed, `%rsp` points to the next gadget to be executed.
- Together, these conditions guarantee that the return-oriented program will execute correctly, one gadget after another.
- Exploitation
  - Place the payload containing these gadgets in the memory of the program to be exploited, and redirect the stack pointer so that it points to the first gadget.
  - The easiest way to accomplish these tasks is by means of a buffer overflow on the stack; the gadgets are placed on the overflowed stack so that the first has overwritten the saved instruction pointer of some function. When that function tries to return, the return-oriented program is executed instead.

Finding Useful Instruction Sequences

- Every instruction sequence ending in a `ret` instruction (represented by the byte `0xc3` on x86) is potentially useful.
- The suffix of any useful instruction sequence is also useful.
- The frequency of occurrence of some sequence is not relevant; only its existence is.

Finding Useful Instruction Sequences

- Every instruction sequence ending in a `ret` instruction (represented by the byte `0xc3` on x86) is potentially useful.
- The suffix of any useful instruction sequence is also useful.
- The frequency of occurrence of some sequence is not relevant; only its existence is.

- On variable-length instruction sequences, there can also be unintended instruction sequences.

```
 f7 c7 07 00 00 00      test $0x00000007, %edi
 0f 95 45 c3            setnzb -61(%ebp)

c7 07 00 00 00 0f      movl $0x0f000000, (%edi)
95                     xchg %ebp, %eax
45                     inc %ebp
c3                     ret
```