ROP and CFI

Chengyu Song

Lab1 tips

- crackme: inputs to `printf` and `scanf`
  - crackme0x00: free!!
  - crackme0x01: what is the input to `scanf`?
  - crackme0x02: calculation, but really?
  - crackme0x03: which one is the correct branch?
  - crackme0x04: what does the loop in `check` do?
  - crackme0x05: one more check, what does `parell` do?

Prevent exploit against stack buffer overflow

- What are the key steps?
  1. Overwrite the return address, **sequentially** → stack canary
  2. Jump to the beginning of the shellcode → ASLR
  3. Execute the shellcode → DEP/NX

Prevention bypass

- Can we bypass these preventions?
  1. Stack canary
  2. ASLR
  3. DEP/NX

Code reuse attacks (CRA)

- Q1: if we cannot inject code, can we just reuse existing code?
- Q2: does CRA has the same capability as shellcode?
- Q3: is CRA general enough (i.e., Turing-complete)?

Return-to-libc attacks (1)

```
void start() {
  printf("IOLI Crackme Level 0x00\n");
  printf("Password:");

  char buf[32];
  memset(buf, 0, sizeof(buf));
  read(0, buf, 256);

  if (!strcmp(buf, "250382"))
    printf("Password OK :)\n");
  else
    printf("Invalid Password!\n");
}
```

Return-to-libc attacks (2)

```
int main(int argc, char *argv[])
{
  setvbuf(stdout, NULL, _IONBF, 0);
  setvbuf(stdin, NULL, _IONBF, 0);

  void *self = dlopen(NULL, RTLD_NOW);
  printf("stack   : %p\n", &argc);
  printf("printf(): %p\n", dlsym(self, "printf"));

  start();

  return 0;
}
```

Return-to-libc attacks (3)

- Task 1: exploit the buffer overflow and print out "Password OK :)"
- Challenge: with DEP, you cannot inject shellcode, so how?

```
[printf's frame]      [buf  ]
[ra]                  [.....]
[args...]             [ra   ] -> printf
[fmt]                 [dummy]
[caller's frame]      [arg1 ] -> "Password OK :)"
```

Return-to-libc attacks (4)

- Task 2: can you start a shell?

```
[buf  ]
[.....]
[ra   ] -> system
[dummy]
[arg1 ] -> "/bin/sh"
```

Return-to-libc attacks (5)

- Task 3: can you chain two function calls?

```
[buf      ]
[.....    ]
[old-ra   ] -> 1) printf
[ra       ] ------------------> 2) system
[old-arg1 ] -> 1) "Password OK :)"
[arg1     ] -> "/bin/sh"
```

Return-oriented Programming

- Can we do arbitrary computation with CRA?
- ROP gadgets: code snippets ends with a ret instruction
  - Do not need to be intended instructions (x86 instructions are variable length so jumping to the middle of an instruction could make the following byte stream interpreted differently).
- What kind of primitives do we need?
  - Load/store, arithmetic/logic, control-flow, syscall, function calls

ROP: load constant

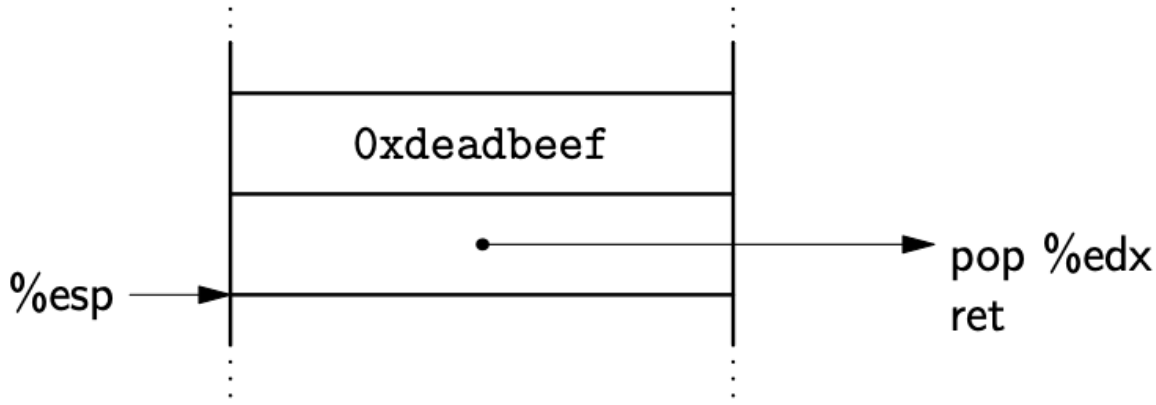

Figure 2: Load the constant `0xdeadbeef` into %edx.

ROP: load from memory

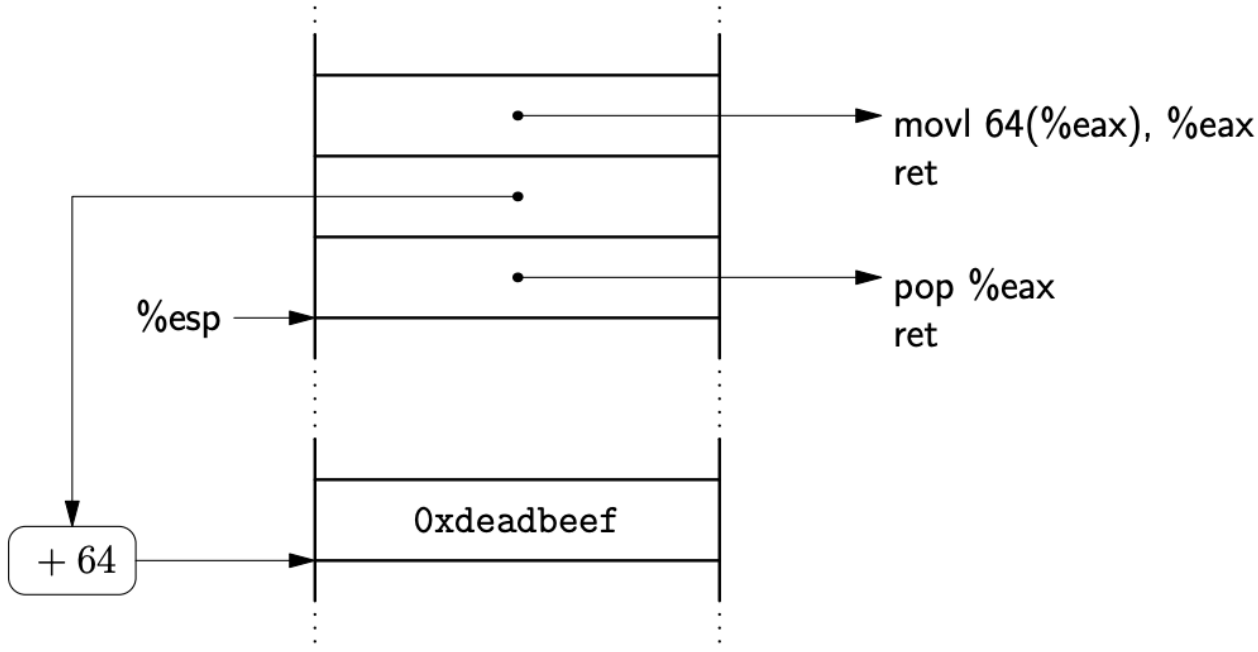

Figure 3: Load a word in memory into %eax.

ROP: store to memory

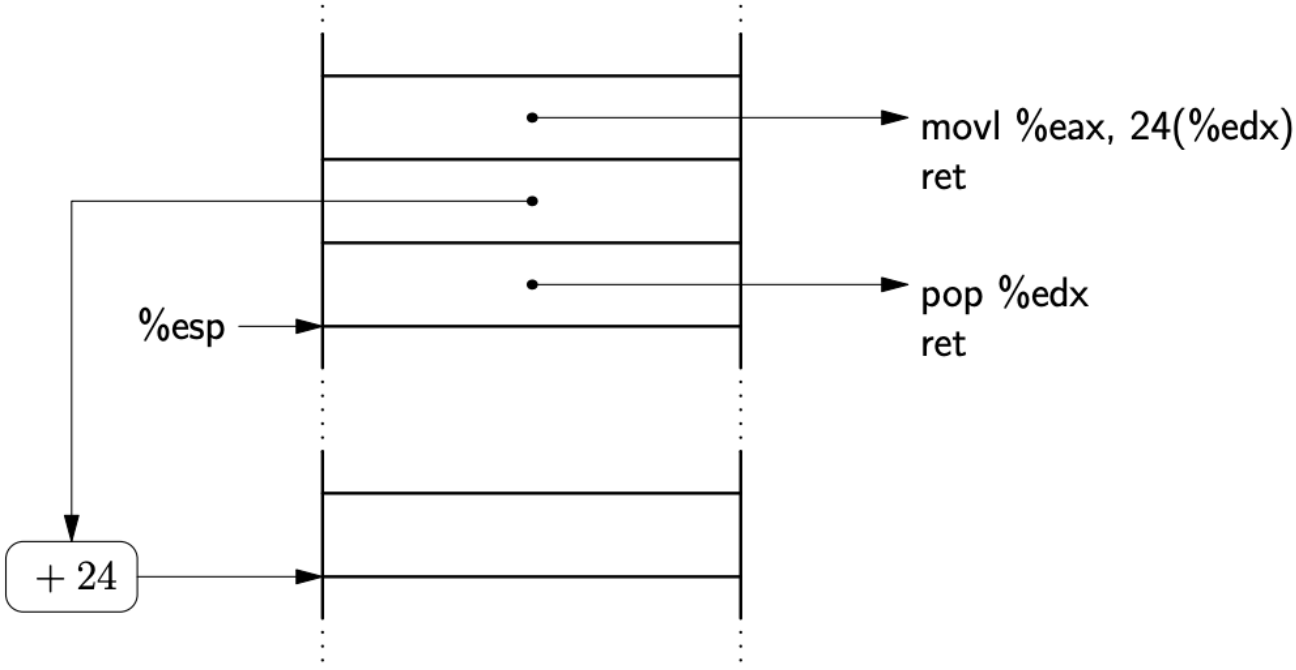

Figure 4: Store %eax to a word in memory.

ROP: simple add

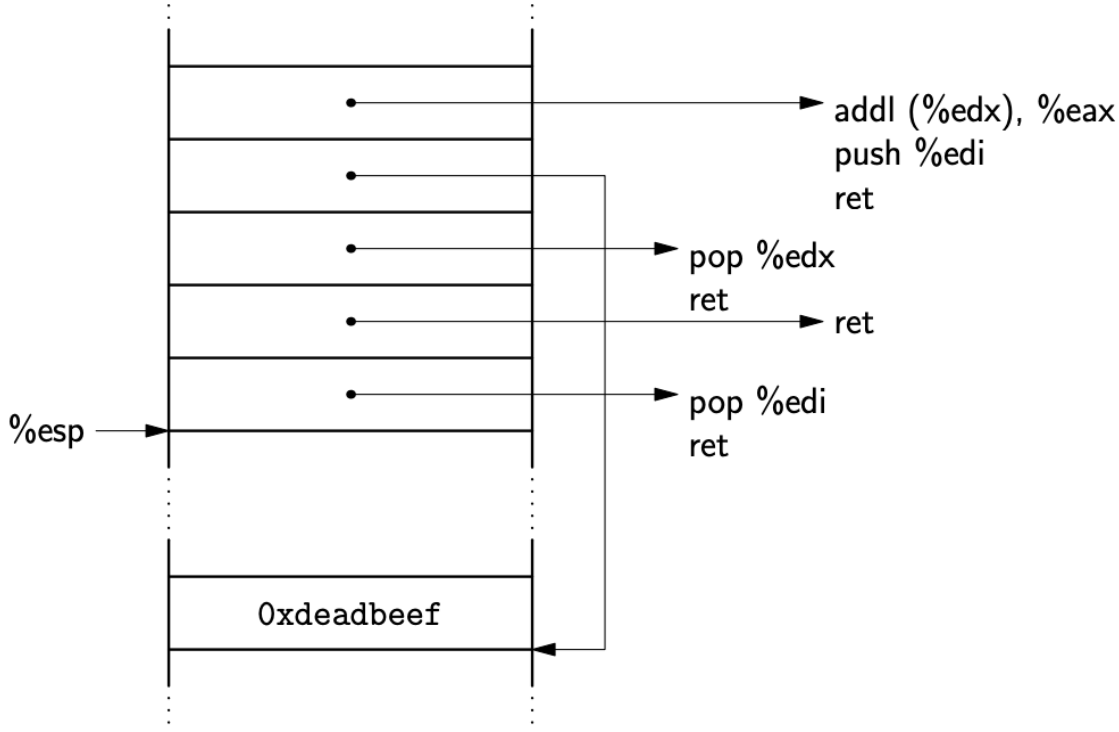

Figure 5: Simple add into %eax.

ROP: unconditional jump

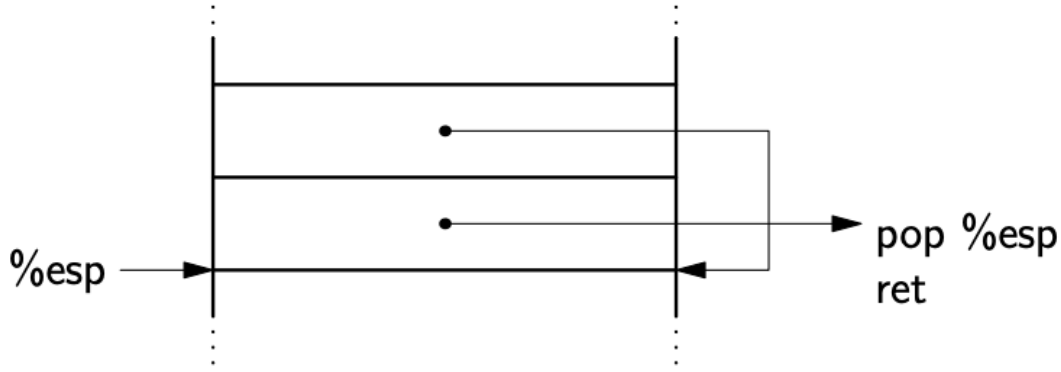

Figure 10: An infinite loop by means of an unconditional jump.

ROP: other operations

- Please refer to the paper for details.

Other flavors of CRA

- Call-oriented programming
- Jump-oriented programming
- Counterfeit object-oriented programming

Defend against ROP

- Key steps in ROP
  1. Control stack/ `ESP`
  2. Locate gadgets
- What defenses would work?

Arm race: round 0

- Defense: Stack canary → Offense: stack pivot
- Defense: ASLR → Offense: information leak

Arm race: round 1

- Defense: shadow stack → Offense: CRA without returns
- Defense: fine-grained randomization → Offense: Just-in-time CRA

Control-flow Integrity (CFI)

- One simple principle: *runtime control-flow should not deviate from the control-flow graph (CFG) derived from analysis*
  - Both forward-edge (calls/jmps) and backward-edge (ret)

CFI: CFG construction

- Binary analysis: coarse-grained, call to any valid function begins, return to any callsites
- Static source code analysis: fine-grained, many implementations
- Dynamic analysis accurate, with higher performance overhead

CFI: enforcement

- Labeling
- Shadow stack
- Finite set
- Encryption: CFI and ASLR is equivalent!!
- Hardware

CFI: challenges

- How to support dynamic linking: Modular-CFI
- How to support dynamic code generation (JIT): RockJIT

CFI availability

- Microsoft: control-flow guard (`/guard:cf`)
  - Windows 8.1 and VS 2015 and newer
  - Return flow guard
- GCC: vtable verification (VTV)
- Clang: `-fsanitize=cfi`
  - https://clang.llvm.org/docs/ControlFlowIntegrity.html
- Intel: Control-flow Enforcement Technology (CET)
- ARM: Branch Target Integrity (BTI)

Arm race: round 2

- Q: if the control-flow graph (CFG) is not accurate enough to only allow a single target, can we still launch CRA?
- A: Yes!!
  - Against coarse-grained CFI: Out-of-Control
  - Against CFI without shadow stack: Losing Control
  - Against fine-grained CFI: Control Jujutsu, COOP