NANYANG TECHNOLOGICAL UNIVERSITY

# Tutorial 3: ASLR & JOP

*presented by*

**Li Yi**
*Assistant Professor*
*SCSE*

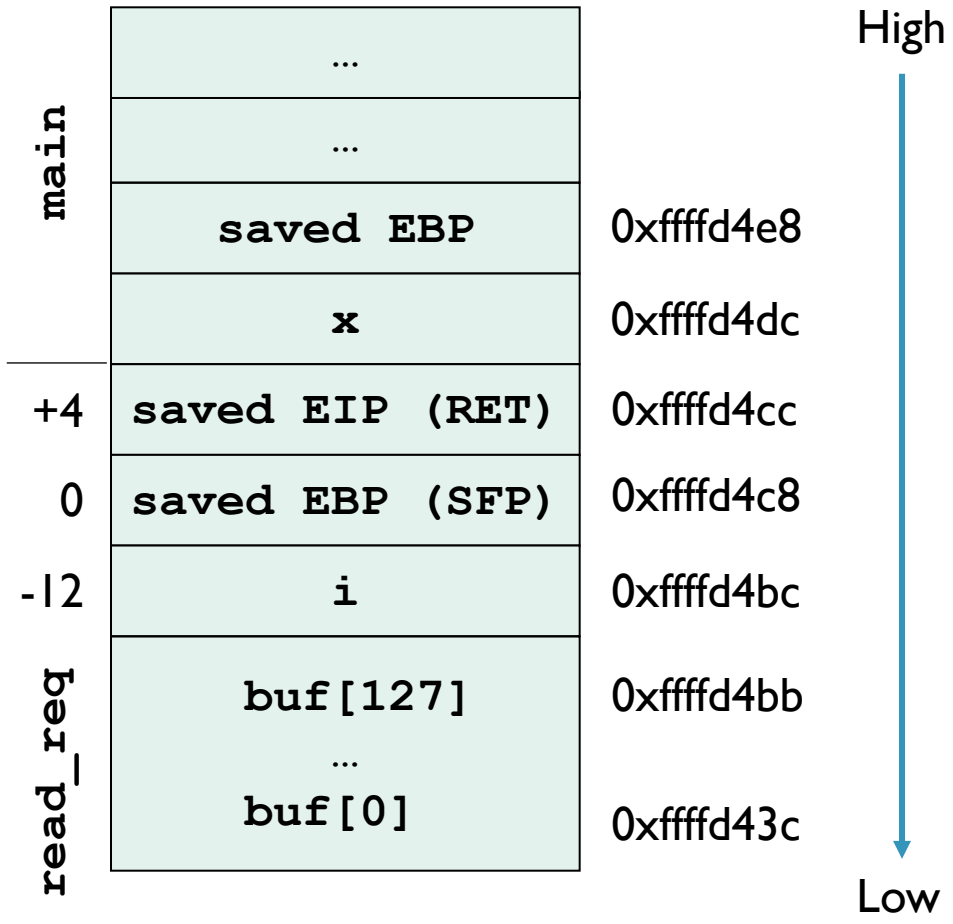
*N4-02b-64*
yi_li@ntu.edu.sg

# ASLR & Jump-Oriented Programming

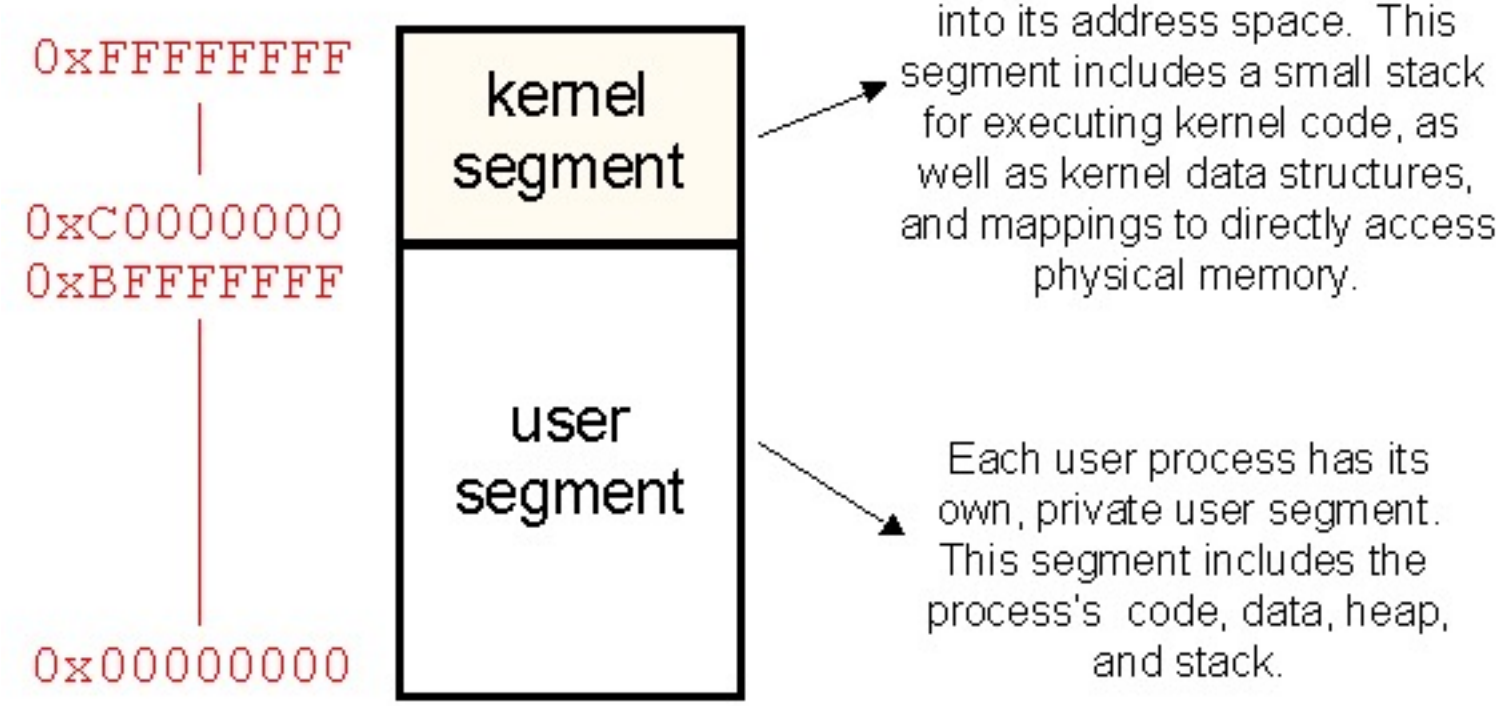
- Collect information on ASLR in Linux and in Windows
  - What is being randomized? How random is randomization?
- Jump-oriented programming creates its own trampoline for linking gadgets instead of using the call stack
  - Jump-Oriented Programming: A New Class of Code-Reuse Attack
- Read up on JOP; you should be able to explain the fundamentals of the attack method

# Stack Analysis with GDB: Example from Lecture 3

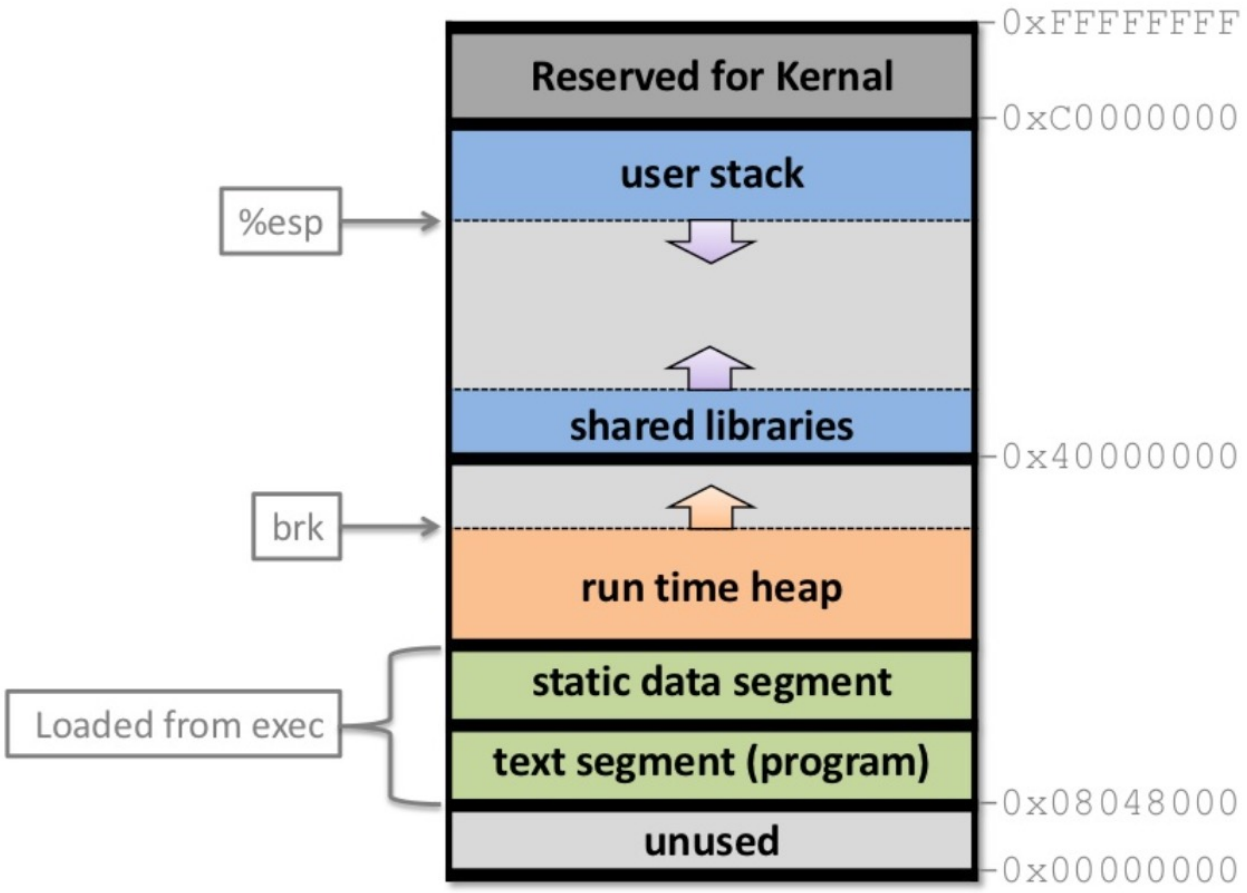
```
#include <stdio.h>

int read_req(void) {
  char buf[128];
  int i;
  gets(buf);
  i = atoi(buf);
  return i;
}

int main(int ac, char **av) {
  int x = read_req();
  printf("x=%d\n", x);
}
```



# Linux Memory Layout



# Linux Memory Layout



https://duo.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0
https://wiki.ubuntu.com/Security/Features

# ASLR – Linux

- **Stack:** Userspace stack mapping set up by the kernel during exec(2) should be sufficiently randomized. Stack randomization is performed by the randomize_stack_top() function.
- **Heap:** The heap location returned by the brk(2) system call when a program is first executed should be randomized. Heap randomization is performed by the arch_randomize_brk() function.
- **Libs and mmap:** After NX was introduced, static library mapping led to the popularity of ret-to-libc and more generic ret-to-lib attacks. The location of libraries and other mmap'ed regions should be randomized.
- **Exec:** Even if you have randomized the mapping of all the shared libraries that an executable uses, you still need to randomize the location of the executable itself when it is mapped into the address space. Otherwise, the executable mapping can be used as a source for ROP gadgets.
- **Linker:** On most Linux systems, the ld.so dynamic linker provided by glibc can self-relocate itself, so its mapping is randomized. This is not the case for all linkers.
- **VDSO:** The VDSO (Virtual Dynamically-linked Shared Object) is an executable mapping of a virtual shared library provided by the kernel for syscall transitions. However, most Android devices run on the ARM architecture, which does not use a VDSO.

https://stackoverflow.com/questions/35932518/aslr-entropy-bits-for-stack-on-linux

# Entropy for Stack ASLR

- We need to know the address space the stack can be placed in
- Address space `0x00000000` - `0xbfffffff` for user memory
- Split user address space into three ranges:

| Range | 2 MSBs |
|---|---|
| `00000000 - 3fffffff` | `00` |
| `40000000 - 7fffffff` | `01` |
| `80000000 - bfffffff` | `10` |

- First range reserved for the heap; only two possible combinations for the two most significant bits remain; one bit of entropy is lost
- Pages aligned to 4096-byte boundaries; 12 bits of entropy are lost
- For 32-bit addresses, the stack entropy is 32 - 12 - 1 = 19
- For 64-bit addresses 30 bits of entropy can be achieved

# Entropy for Stack ASLR

```
static unsigned long randomize_stack_top(unsigned long
stack_top)
{
   unsigned long random_variable = 0;
   if ((current->flags & PF_RANDOMIZE) &&
       !(current->personality & ADDR_NO_RANDOMIZE))
      {
        random_variable = (unsigned long) get_random_int();
        random_variable =& STACK_RND_MASK; //0x3FFFFF on x86_64
        random_variable <<= PAGE_SHIFT; //12 on x86_64
      }
#ifdef CONFIG_STACK_GROWSUP
   return PAGE_ALIGN(stack_top) + random_variable;
#else
   return PAGE_ALIGN(stack_top) - random_variable;
#endif
}
```

- How much randomness? 22 bits (0x3FFFFF gives you 22 bits)
- 8 more bits of randomness come from sub-page randomization

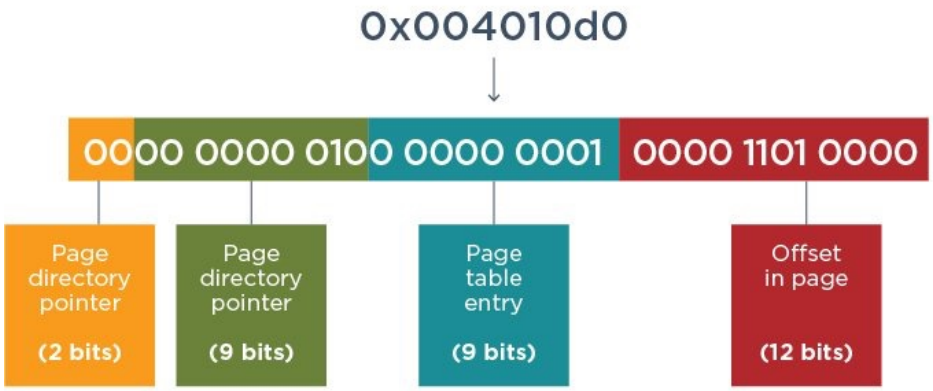
https://msdn.microsoft.com/en-us/library/bb430720.aspx

# ASLR – Windows

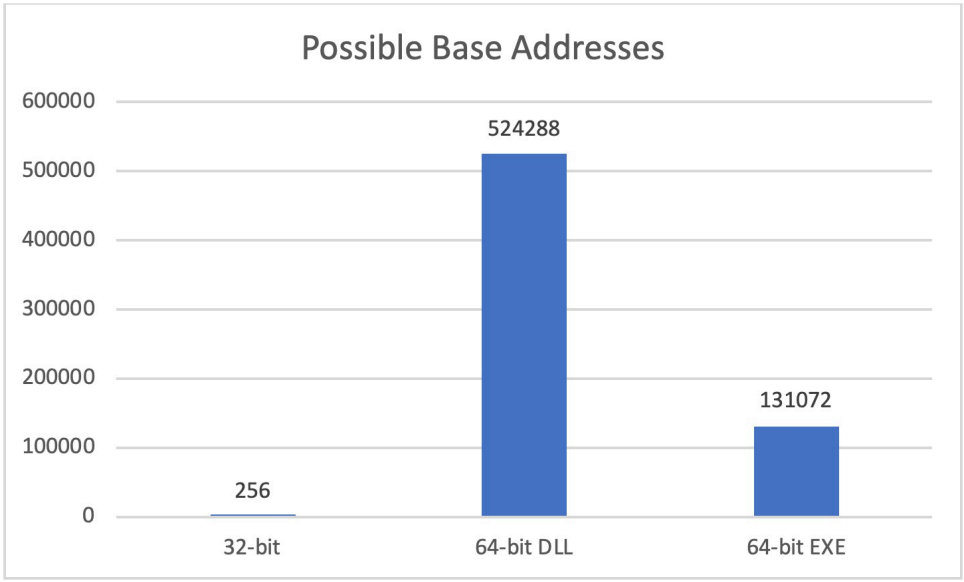
- Moves executable images into random locations when a system boots
- For a component to support ASLR, all components that it loads must also support ASLR
  - For example, if A.exe consumes B.dll and C.dll, all three must support ASLR
- By default, Windows Vista and later will randomize system DLLs and EXEs, but DLLs and EXEs created by third parties must opt in to support ASLR using the /DYNAMICBASE linker option
- ASLR also randomizes heap and stack memory

- http://www.zdnet.com/article/microsoft-says-aslr-behavior-in-windows-10-is-a-feature-not-a-bug/
- https://insights.sei.cmu.edu/cert/2014/02/differences-between-aslr-on-windows-and-linux.html

# ASLR – Windows




- Windows only attempts to randomize 8 bits of a 32-bit address
  - Bits 16 through 23
  - An attacker can potentially guess the base address of an EXE in 256 guesses
- For 64-bit binary, Windows is able to randomize 17-19 bits of the address
  - Depending on whether it is a DLL or EXE)
  - Recompiling a 32-bit program to 64-bit makes ASLR more effective




Source: https://www.fireeye.com/blog/threat-research/2020/03/six-facts-about-address-space-layout-randomization-on-windows.html

https://copperhead.co/blog/2015/05/11/aslr-android-zygote

# ALSR – Android

- Early Android versions only had stack randomization due to lack of kernel support for ASLR on ARM
- The 4.0 release introduced mmap randomization. The kernel also gained support for ARM exec and brk randomization but Android still lacked userspace support
- The 4.1 release introduced support for full ASLR by enabling heap (brk) randomization and adding linker support for self-relocation and Position Independent Executables (PIE)
- Lollipop is the latest step forwards, as non-PIE executable support was dropped and all processes now have full ASLR
- The vDSO has been randomized since it was introduced on 64-bit ARM and from the beginning of x86 Android. It does not yet exist on 32-bit ARM

https://www.bleepingcomputer.com/news/security/openbsd-will-get-unique-kernels-on-each-reboot-do-you-hear-that-linux-windows/

# OpenBSD – KARL

- The OpenBSD kernel had used a predefined order to link and load internal files inside the kernel binary, resulting in the same kernel for all users (as do all other OSs)
- KARL (Kernel Address Randomized Link) relinks internal kernel files in a random order so that a unique kernel blob is generated every time
  - The kernel is linked such that the startup assembly code is kept in the same place, followed by randomly-sized gapping, followed by all the other .o files randomly re-organized
  - Hence, distances between functions and variables are entirely new
  - An info leak of a pointer will not disclose other pointers or objects

# Jump Oriented Programming

Weird machines

# Recap: Return-Oriented Programming

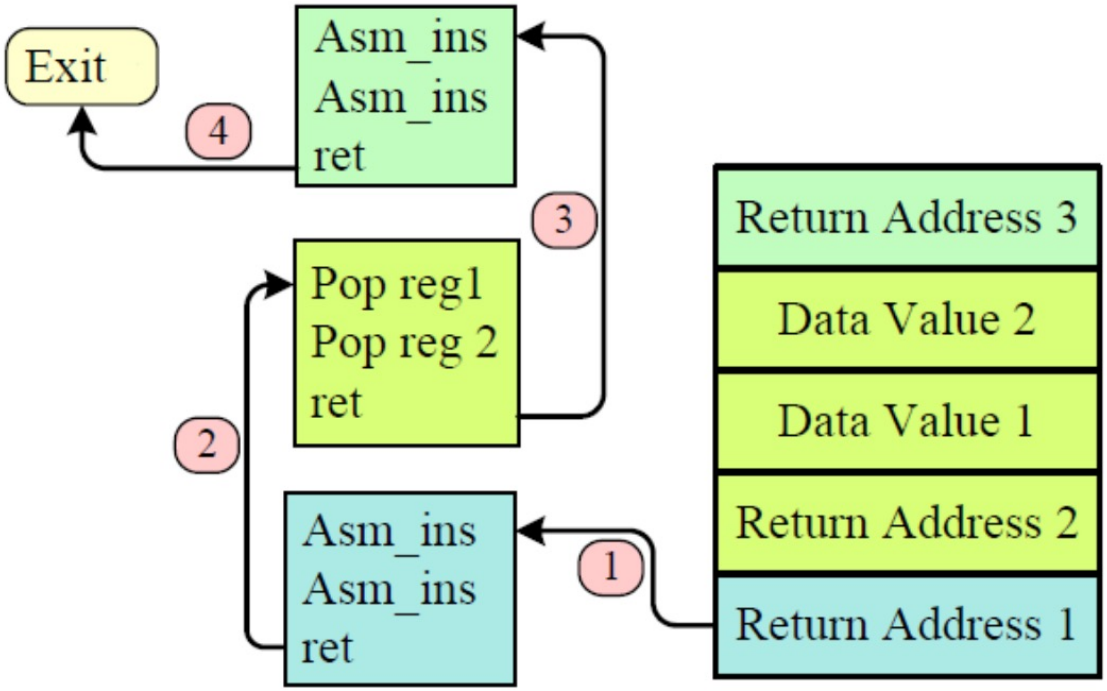
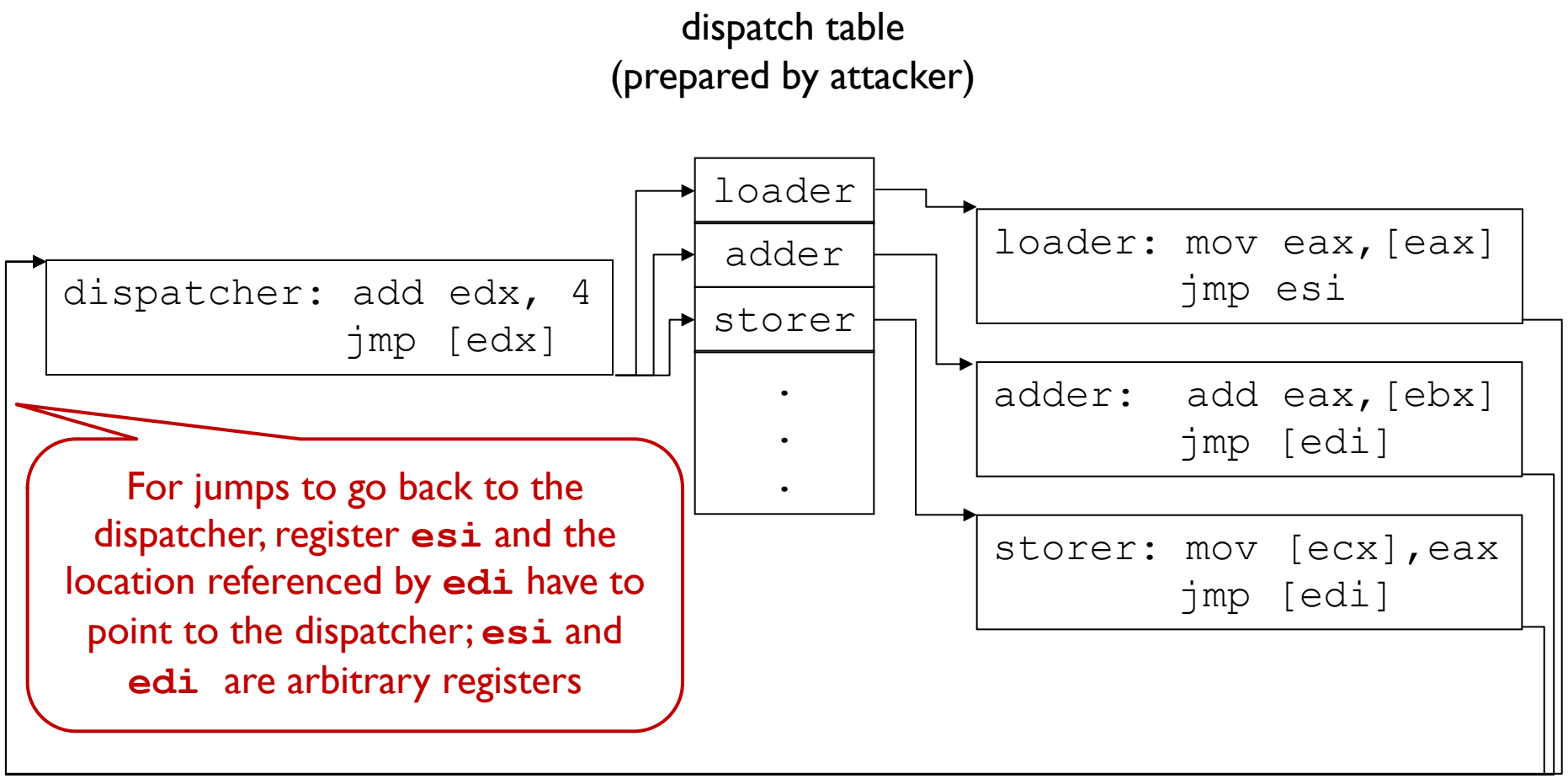



Image: Dwarf Frankenstein is Still in Your Memory: Tiny Code Reuse Attacks
10.22042/ISECURE.2017.0.0.4

# Jump-Oriented Programming (JOP)

- ROP: stack pointer ESP serves as "program counter" for gadgets; in JOP, a dispatch table holds gadget addresses and data
- The "program counter" (*pc*) can be any register that points into the dispatch table
- Control flow is managed by a dispatcher gadget that executes the sequence of gadgets
  - At each invocation, the dispatcher advances the *pc* and launches the associated gadget
- Functional gadgets must return control to dispatcher

# Jump-Oriented Programming (JOP)



# Dispatcher Gadget

- *pc* can be a memory address or a register that represents a pointer into the dispatcher table
- Any gadget that carries out the following algorithm is a dispatcher candidate

```
pc ← f(pc);
goto *pc;
```

- The way the *pc* is advanced affects the organization of the dispatch table
  - Array if *pc* is advanced by a constant value, e.g., `f(pc)= pc+4`
  - Linked list if memory is dereferenced, e.g., `f(pc)= *(pc-18)`

# Dispatch Table

- The dispatch table is the program of a weird machine
  - A weird program is a sequence of gadget addresses
- The attacker has to get this table on the target machine at a known memory location
  - DEP would mark the dispatch table as non-executable
- For the CPU, the "weird instructions" are addresses but not machine instructions
  - Marking the entries in the dispatch table as non-executable does not impede the attack

# Functional Gadgets

- A functional gadget is a block of useful instructions that have no unintended side effects on the *pc*, ending with a jump to the result of an expression
- Such an expression may be a
  - register, e.g., **`jmp edx`**
  - register dereference, e.g., **`jmp [edx]`**
  - complex dereference expression, e.g., **`jmp [edx+esi*4-1]`**
- On execution, the expression must evaluate to the address of the dispatcher
  - or to another gadget that leads to the dispatcher

# Turing-Complete Gadget Library

- Loading data: any gadget that loads from and advances a pointer is a candidate
- Memory access: gadgets that take a memory address and read or write a byte/word at that location
- Arithmetic and logic: gadgets with suitable opcodes (add, sub, and, or, …) working on CPU registers
- Branching: for unconditional branch, modify the *pc*; for conditional branch, adjust *pc* based on the result of a previous computation
- System calls: can also be made from JOP programs

# Preparing for a JOP Attack

- JOP requires control of EIP and all memory locations or registers used to run the dispatcher gadget
- Can be achieved by first running an initializer gadget
  - Fills the relevant registers either by arithmetic and logic or by loading values from memory
  - Once this is done, the initializer jumps to the dispatcher, and the jump-oriented program can begin
- The functional gadgets are the machine instructions of a weird machine