19. Code Reuse Attacks

CS 4352 Operating Systems

Countermeasure I: W⊕X

- Non-executable stack was proposed to prevent stack buffer overflow attack
    - Solar Designer (Alexander Peslyak) proposed it in 1997 for Linux
    - It means injected code on the stack cannot be executed as instructions
    - How about heap or global data area? still executable
- PaX team generalized this idea for Linux in 2000
    - Data pages are marked as writable but not executable (so stack/heap/data is not executable)
    - Code pages are marked as executable but not writable
    - If the MMU of a processor has support for the NX (non-executable) bit, PaX will use it; otherwise, PaX can emulate it in software
        - If the MMU lacks a per-page executable bit (e.g., relatively old x86 chips), emulation will incur a significant performance penalty
        - In 2004, Microsoft introduced their DEP (Data Execution Protection)
        - In 2006, Apple eventually used in their MacOS
- End of story?
    - You wish

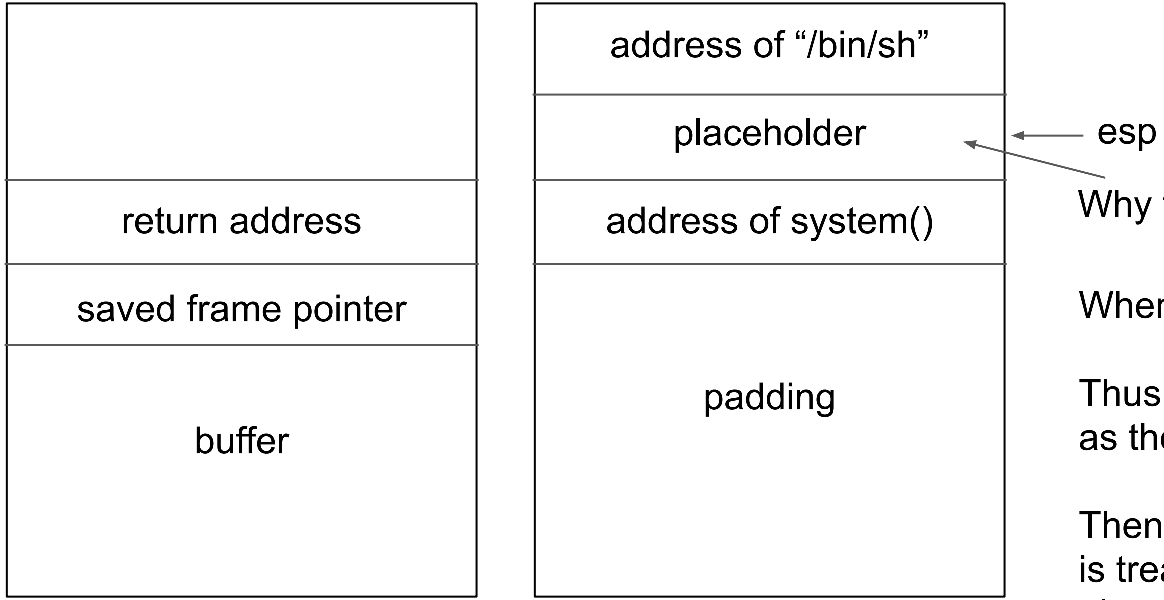
Well, Let’s Try Another Angle

- What if we can reuse some legitimate code in the system to gain control instead of providing our own shellcode which resides in some data page
    - Since legitimate code should be in some executable page(s), W⊕X is not a problem for us
    - The problem is how to reuse existing code?
        - Return-into-libc attack
            - Take advantage of existing C library functions
        - Return-oriented programming (ROP)
            - Chain small code snippets (gadgets) together to carry out malicious operations
            - Large code bases like C library can provide enough gadgets for Turing-completeness
        - Jump-oriented programming (JOP)
            - Leverage indirect jumps as well for chaining (generalization of ROP)
- The previously mentioned attacks belong to **code injection attacks**, and now we are going to learn **code reuse attacks**

Return-into-libc Attack

- Like classical buffer overflow attacks, this kind of attack also changes the return address, but the control is transferred to a C library function rather than to a shellcode
    - For example, if we can overwrite the return address with the address of system(3), when the current function returns →
        - The control will go to the beginning of the system(3)
        - If we can somehow provide the system(3) with “/bin/sh”, the execution of system(3) will spawn a shell
            - system(“/bin/sh”) is a very popular option in return-into-libc attack
- Why libc? → It is loaded into every Unix/Linux program and encapsulates the system call APIs by which a program can access kernel services such as forking child processes and communicating over network sockets
    - Actually, this kind of attack was originally suggested by Solar Designer

Example Overview




```
mov   %ebp, %esp
pop   %ebp
ret
```

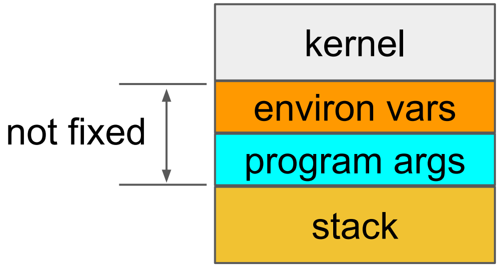
Why this placeholder?

When system() is entered, sp points there

Thus, system() will think the placeholder as the return address pushed by its caller

Then, the argument that system() expects is treated as the one that is above the placeholder

Where Is “/bin/sh”?



- You may want to put this string in the buffer
    - You need to know the exact address where you put
    - Main issue -- the null character that terminates the string (‘/’, ‘b’, ‘i’, ‘n’, ‘/’, ‘s’, ‘h’, ‘\0’)
        - One way is to put the string at the end of your input
- We can actually put it in an environment variable (what if shellcode?)
    - You may have known environment variables can be set from the shell
        - export MYSHELL=/bin/sh
            - Actually, if you use env command to show all the environment variables, you will find there is a default variable SHELL
                - E.g., on my machine, SHELL is “/bin/bash”
    - You may not know environment variables are located on the stack
        - They are stored before the stack frame of the main() function (higher addresses)
        - C’s getenv(3) standard library function can get an environment variable’s address
            - The name of the program will also be on the stack, so getenv(3) gives slightly different addresses for two programs with different name lengths

Where Are The Functions in libc?

- In most cases, libc is dynamically linked (e.g., by ld-linux.so which is the dynamic linker in Linux)
  - You can calculate the address of any function in libc by the following two steps
    - The address where libc is mapped can be found at the /proc, or by using pmap
      - You can find the start address of libc
        - cat /proc/<pid>/maps or pmap -x <pid>
    - The address where system() locates can be computed by adding the offset to system()
      - The offset to system() within libc can be read from the object file of libc
        - objdump -d libc.so
  - Or, you use gdb to run the program and set a breakpoint at main()
    - You can simply use “p system” to get the address

Hints

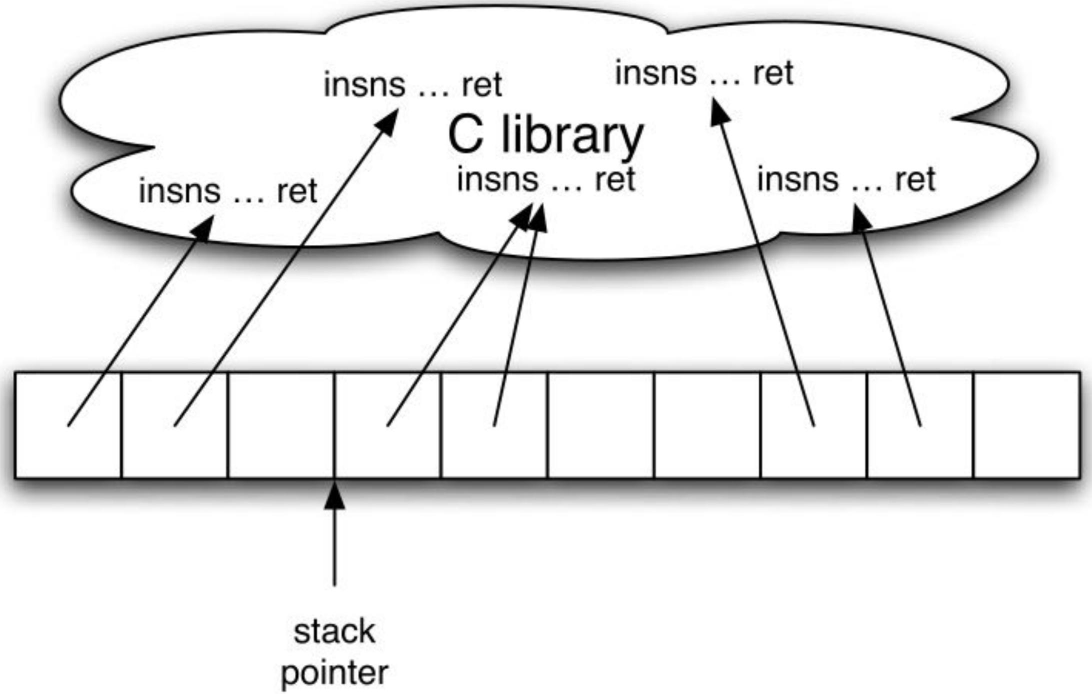
- We create an environment variable “export MYSHELL=/bin/sh”
    - With gdb, we can list the environment variables by probing the “environ” variable
        - “p/x environ” prints where the environment variables are put in the address space
        - “p system” gives the starting address of system()
    - Without gdb? check out getenvaddr.c
- We use system(3) of libc. Where is it?
    - With gdb, we can set a breakpoint “b main”, and “p system”

Bummer!

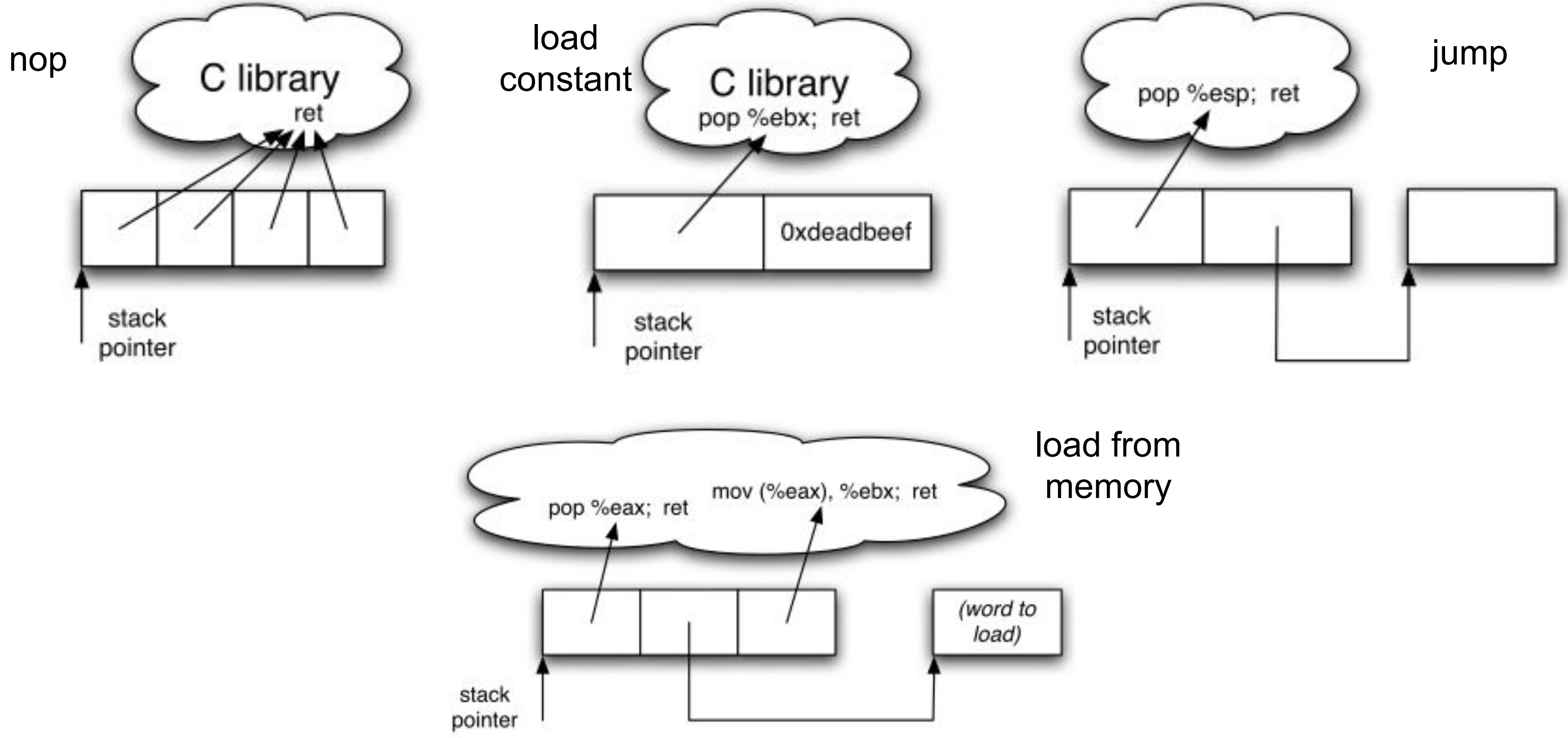
- What if the program is compiled with “-static”?
    - Static linking is performed at compile time by ld
    - ld copies the relocatable object files in the archives that are referenced by the program
    - strcpy(3) is defined in strcpy.o of libc.a
    - Let’s see an example if we still can find our friends like system(3) or exec family members
- Even using dynamic linking, some countermeasures for return-into-libc attack also try to get rid of unused functions
- Moreover, some mitigation techniques try to move libc to very low locations in the address space, so some 0x00’s may appear in the address

Return-Oriented Programming (ROP)

- Any cod   e reused
    - We ju
- ROP ge   te)
  computa
    - Retu   ted in knowing
      this,
        - things out
- ROP's b
    - A ga
    - The   attacker wants

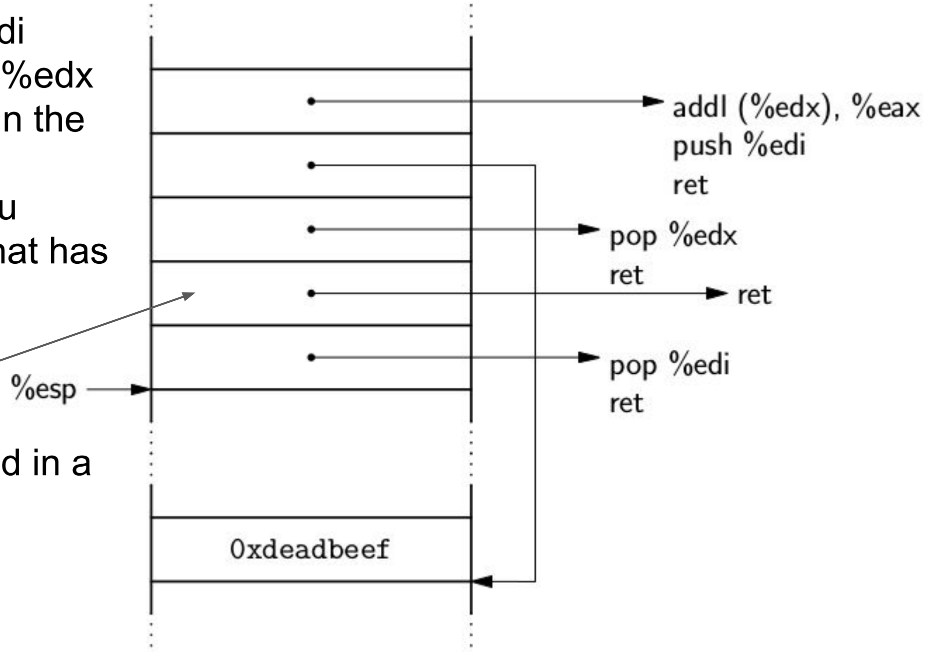


Simple Gadget Examples



Arithmetic Add Example

- We load the address of a ret instruction into %edi
- We load the address of a memory operand into %edx
- We add the contents in %eax and the contents in the memory location
- An annoying "push %edi" instruction follows (you cannot avoid this if this gadget is the only one that has add operation)
    - It causes the top word on the stack to be overwritten by the contents of %edi
- What if this add operation needs to be performed in a loop?



Conditional Jumps

- These are substantially trickier in ROP
  - In x86, you normally use the following steps
    - The cmp instruction or many arithmetic operations set the flags (%eflags)
    - The jcc instruction cause a jump when the flags satisfy certain conditions
  - Because the jcc is expressed as a change in the PC (%eip), the conditional jump instructions are not useful for ROP
- What we need is a conditional change in the stack pointer (three tasks)
  - Perform some operation that sets (or clears) flags of interest
  - Copy the flag of interest from %eflags to a general-purpose register
  - Use the flag of interest to change %esp conditionally by the desired jump amount

Task 1 & 2

- For the first task, we use the carry flag, CF
  - To test whether a value is zero, we can use the neg instruction
    - It clears CF if its operand is zero and sets CF otherwise
  - To test whether two values are equal, we can subtract one from the other and test zero
  - To test whether one value is smaller than another, we can subtract the first from the second
    - The sub instruction sets CF when the first is smaller than the second
- For the second task, we use the add with carry instruction, adc
  - Add with carry computes the sum of its two operands and the carry flag
  - If we take the two operands to be zero, the result is 1 or 0 depending on whether the carry flag is set

Convert 1 or 0 to Displacement

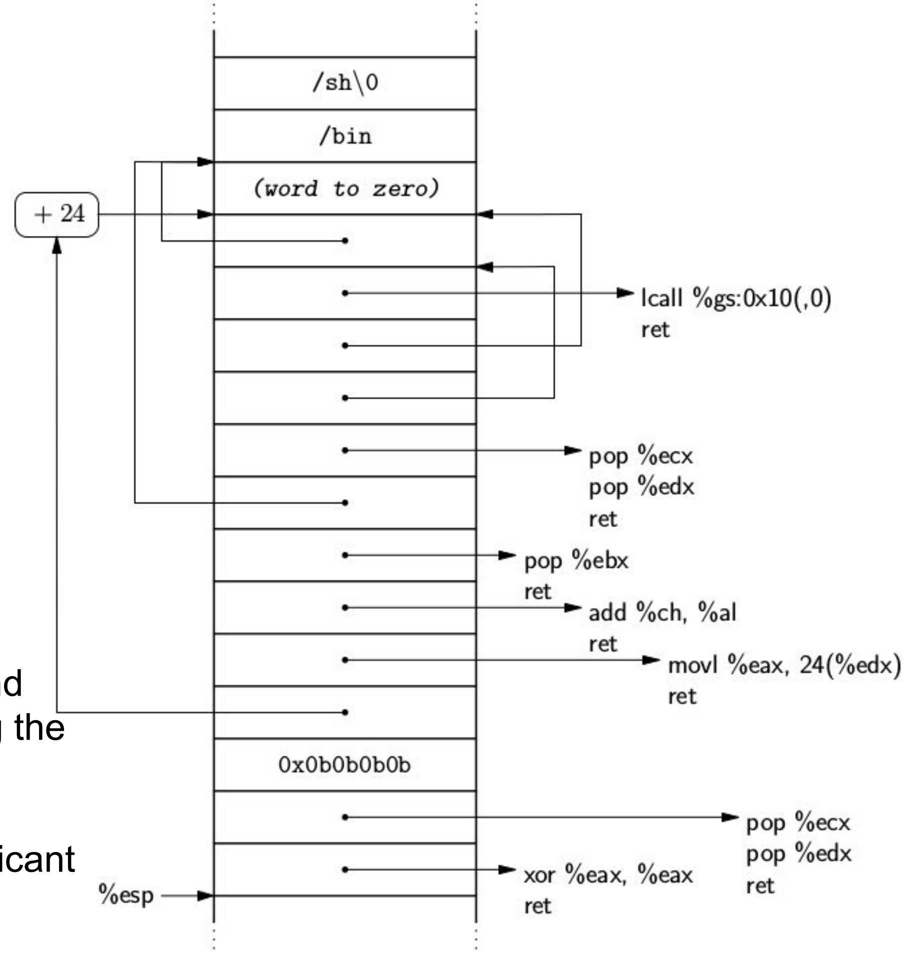
- Recall what our goal is?
  - If a condition holds (e.g., a < b), %esp is added with a proper displacement (x)
  - Otherwise, continues (namely, %esp is added with the displacement 0)
- What is the binary representation of (-1) and 0?
  - It is all-1 pattern and all-0 pattern
- Converting 1 or 0 to appropriate displacement has three steps
  - Apply neg operation on the word containing CF
    - We have all-1’s or all-0’s
  - Take bitwise and operation on the result with x
    - We have either x or 0
  - Add the result to %esp

ROP Shellcode Example

Use the execve system call to run a shell

- Setting the system call number, in %eax, to 0xb
- Setting the path of the program to run, in %ebx, to the string "/bin/sh"
- Setting the argument vector argv, in %ecx, to an array of two pointers, the first points to "/bin/sh" and the second is null
- Setting the environment vector envp, in %edx, to an array of one pointer, which is null

1. Word 1 (from the bottom) sets %eax to zero
2. Words 2, 3, 4 load into %edx the address of the second word in argv (minus 24); and, in preparation for setting the system call number, load into %ecx the all-0b word
3. Word 5 sets the second word in argv to zero
4. Word 6 sets %eax to 0x0b by modifying its least significant byte, %al
5. Words 7, 8 point %ebx at the string "/bin/sh"
6. Words 9, 10, 11 set %ecx to the address of the argv array and %edx to the address of the envp array



Why x86 Is Attractive When Doing ROP?

- Of course, under any architecture, we can find gadgets
  - But, under x86, we can even make gadgets
- x86 ISA has several features that favor gadget making
  - The length of x86 instructions varies from 1-byte to 20-byte
  - The instruction addresses are unaligned
  - The density of x86 instruction encodings is high
    - A random byte stream may be interpreted as a series of valid instructions
- Unintended instruction sequence example
  - Two original instructions inside ecb_crypt(3) of libc
    - ```
      f7 c7 07 00 00 00    test $0x00000007, %edi
      0f 95 45 c3          setnzb -61(%ebp)
      ```
  - If we start one byte later, we can make a gadget
    - ```
      c7 07 00 00 00 0f    mov $0x0f000000, (%edi)
      95                   xchg %ebp, %eax
      45                   inc %ebp
      c3                   ret
      ```

Countermeasure II: ASLR

- Address space layout randomization (ASLR) is a moving target defense technique that tries to add randomness into a process's address space
  - Most of the systems just add random shifts to stack, heap, and shared libs (mmaped areas)
    - Why? → if not PIC (position-independent code), the program will not run (fixed addresses are used in instructions)
      - PIC means it does not require static memory addresses to fulfill its duties
      - The base address of PIC can be randomized
      - PIC relies on GOT as well as PLT, which will be talked in a moment
    - Position-independent executables (PIE) are executables made entirely from PIC
- This countermeasure affects all the attacks we mentioned before
  - Effective to toughen return-into-libc/ROP attacks, since both the start address of libc and the address of the environment variables are changing (we will see clever ways to circumvent it)
  - Effective to prevent classical code injection attacks as well, since it is hard to predict where the shellcode is (even it has a NOP sled)

Call/Jump with Register Indirect Addressing Mode

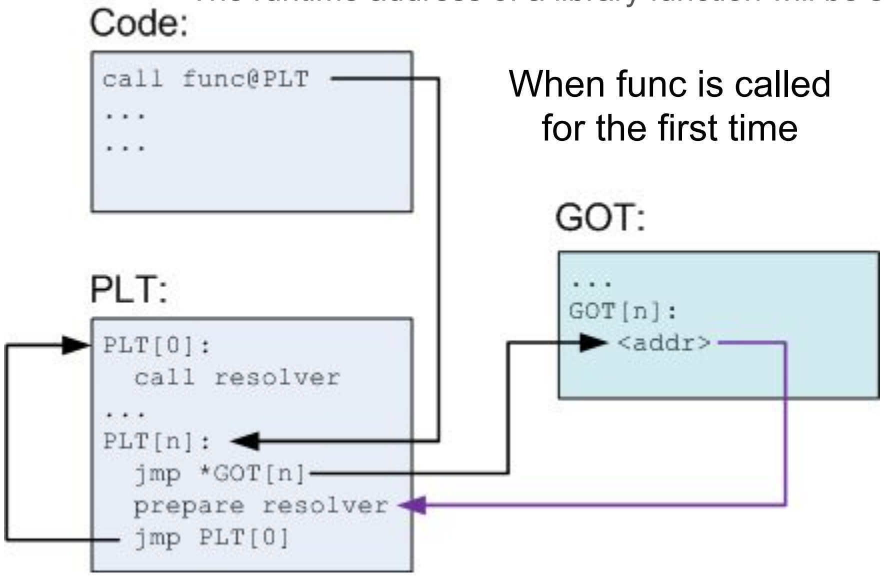
- When returning from a function with buffer overflow vulnerability, some register may still have the start address of the buffer
  - E.g., when doing strcpy, the start address of the buffer is often left behind in %eax register
- ASLR in Linux does not randomize the addresses of non-PIC code and data
  - Otherwise, we need to rewrite the binary to change some of the fixed addresses in instructions
- Where to find call/jump instructions that transfers control to *<reg>?
  - There are many such instructions in libc, but can we leverage them?
    - No! Because their addresses are randomized as well
  - We can find some in .text
    - objdump -d a.out | grep "call.*<reg>"
      - . is for one character match, and .* is for matching multiple characters (wildcard)

Another Bummer!

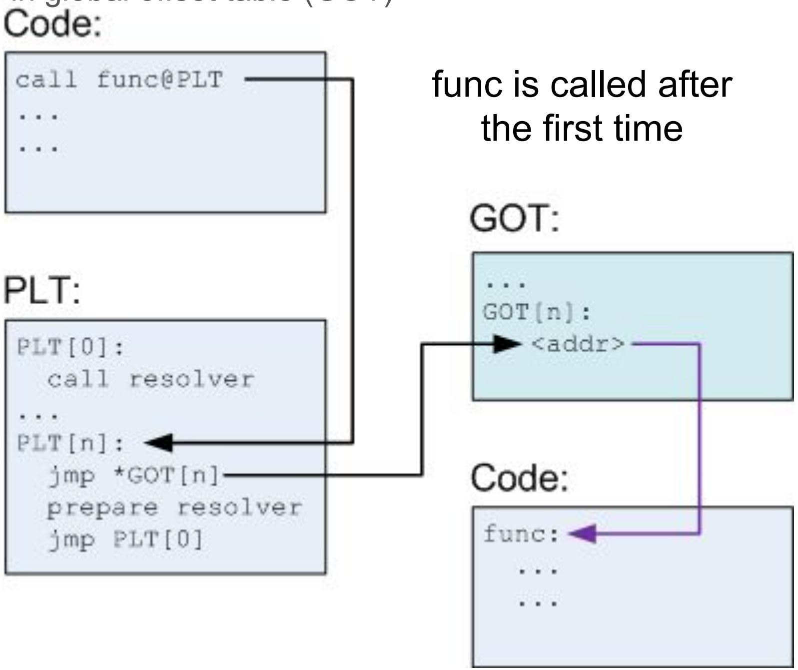
- %eax (also %ecx and %edx) should be saved and restored by the caller according to the ABI convention
  - If the caller does need the contents of %eax after the invocation of the callee, the compiler does not bother generating code to save %eax before the call and restore %eax after the call
- So, %eax may not contain what we need

PLT & GOT

- A dynamic library will be loaded somewhere, which is not fixed
  - How can our program figure out the address of a library function when calling it?
- It relies on a layer of indirection called procedure linkage table (PLT)
  - The runtime address of a library function will be stored in global offset table (GOT)



When func is called for the first time



func is called after the first time

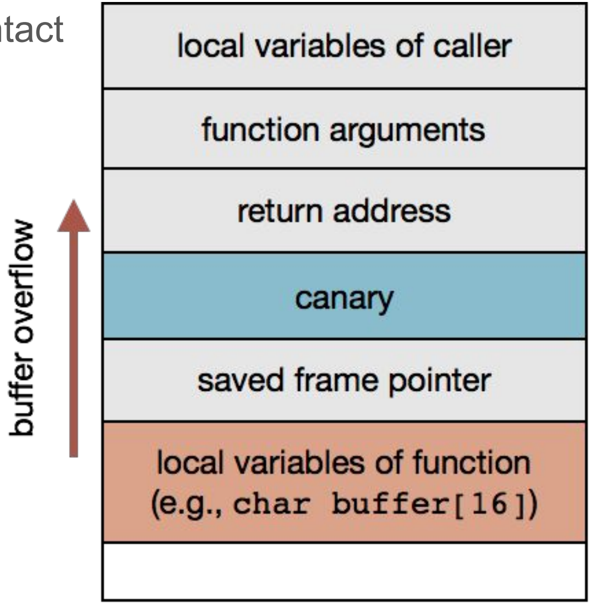
https://nuc13us.wordpress.com/2015/12/25/hack-using-global-offset-table/

PLT/GOT Entry Exploitation

- Of course, we can overwrite some GOT entry to point to some malicious code, which will be invoked when this GOT entry is used
  - Nowadays, **RELRO** countermeasure makes GOT entry overwriting impossible
    - Dynamic linker fills in all the entries and mark GOT as read-only
- If some library functions are used in the program (say, system(3)), they will have corresponding PLT as well as GOT entries
  - We can apply return-into-libc attack by returning into its PLT
- What if you want to use system(3), but it is not called in the program? Let's use ROP to show an attack outline
  - We read the runtime address of some libc function
  - We use the offset between that function and system(3) to compute the address of system(3)
  - We return into the computed address
    - Works with both ASLR (if not PIE) and W⊕X enabled

Countermeasure III: StackGuard

- The idea uses a so-called “canary” value (named after the miner’s canary)
    - When entering a function, it places a canary value (random or terminating) adjacent to the return address on the stack
    - Buffer overflows that reach the return address will necessarily overwrite the canary value
    - Before returning, the function check if the canary value is intact
- Popular compilers have implemented this idea
    - Microsoft Visual Studio: /GS option
        - It is enabled by default
    - GCC: -fstack-protector flag
        - It is enabled by default, based on a random value
    - Why our demos so far work?
        - We use gcc-3.3 (people tried to include it in gcc-3.x at the GCC 2003 Summit, but didn’t work out)
        - gcc-4.1 has it included



C Library Functions

- Common culprits
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - sprintf(char *s, const char *format, …)

  …
- You may try to use their safer buddies
  - strlcpy(char *dest, const char *src, size_t size)
  - strlcat(char *dest, const char *src, size_t size)
  - fgets(char * str, int n, FILE * file)
    - file can be stdin
  - snprintf(char * s, size_t n, const char * format, ...)

  …

Additional References

- “The Advanced Return-into-lib(c) Exploits” by Nergal
- “Return-Oriented Programming: Systems, Languages, and Applications” by Roemer et al.

Next Lecture

We will look at vm

http://www.icanbarelydraw.com/comic/2726