Stack

- An array with the property that the last object placed on the stack will be the first object removed, called LIFO.
- Stack grows from high memory to low memory

Top of The Stack:

- Stack Pointer (SP) is a register that points to the top of the stack (lowest address)
- SP is implementation dependent. It may point to the last address on the stack, or the next free available address after the stack

Frame Pointer/Local Base Pointer

- Frame Pointer/Local Base (FP/LB) pointer points to a fixed location within a frame
- Local variable distance from FP do not change with PUSHes and POPs
- Called EBP on Intel CPUs.

Procedure:

- Prologue
    1) Save previous EBP so that it can be restored at procedure exit
    2) Copy ESP into EBP to create the new EBP. (Move the EBP down)
    3) Advance ESP to reserve space for local variables

- Epilogue:
    1) Clean Up stack on procedure exit

- Arguments to a function or pushed on the stack from right to left
- call instruction pushes instruction pointer (IP) onto the stack, called return address (RET)
- Memory can only be addressed in multiples of the word size. For example, in a 32-bit system, the word size is 4 bytes. So a 5 byte buffer needs 8 bytes (2 words of memory), and a 10 byte buffer needs 12 bytes of memory (3 words)

How To Generate Assembly Output:

- Use the -s argument in your gcc command

Buffer Overflow:

- Putting more data in a buffer than it can take
- Can be used to change the return address of a function, changing the flow of execution of a program

Shell Code:

- If you can modify the return address and flow of execution, then the program you would want to execute is a shell. Because with a shell, you can issue other commands.
- Place the code you are trying to execute in the buffer you are overflowing, and overwrite the return address so it points back into the buffer
- Null bytes in shellcode are considered the end of the string. Must get rid of null bytes by substituting equivalent assembly instructions that do not contain null bytes

Lecture 6: Control Flow Hijacking 2

Problems With Injecting Shellcode In Buffer

- Position of the buffer in memory is unknown.
- How do you determine what return address to put?
- Return address (position of buffer) is determined by load_elf_binary() when a new program is loaded, and it is typically at a high address
- load_elf_binary() loads environment variables at the bottom of the stack. Which affects the locations at the top of the stack
- The return address (buffer position) depends on what gets loaded at the bottom of the stack
-

Solution For Unknown Buffer Location

- Use nop padding.
- Leave a few bytes in the buffer as padding
- Return address can jump into nop sled. Executes from low addy to high addy, and eventually reaches the execve system call

Channeling Vulnerabilities

- Arise when control and data are mixed into one channel

Situation	Data Channel	Control Channel	Security
Format Strings	Output string	Format parameters	Disclose or write to memory
malloc buffers	malloc data	Heap metadata info	Control hijack/write to memory
Stack	Stack data	Return address	Control hijack

Lecture On 10/19/21

Dynamic Linking

- When a program gets loaded into memory, it's not just main function that executes. There's a bunch of other code from libraries that needed to be loaded too. Those libraries don't have to be compiled together with our program, otherwise it makes our program bloated. Dynamic linking separates our program and libraries.
- Include header in c does this.
- Whenever you make a call to print fat runtime, control gets transferred to PLT entry of printf. Because we don't know the location of printf. The dynamic linker will figure out where printf is and load it into memory. Dynamic Linker overwrites the entry for printf with the location of printf, and subsequent calls to printf do not require the linker cause we filled the GOT (global offset table)
- Compiler uses dynamic linking by default
- Procedure linking table and global offset table are at fixed locations

Exploiting The Linking Process

GOT Hijacking

- Since GOT at fixed location, we can assume we know exactly which address corresponds to printf
- If we find a format string vulnerability, then we can overwrite an entry in the GOT with the attacker's address
- So if you call printf in the future, you will transfer control to the attacker

Quiz: What defenses can defeat the GOT Hijacking attack?

Read Vulnerability:

- Read arbitrary memory and disclose values
- Figure out which address is storing what
- Attacker can try inferring how it's randomized

Side Channels:

- Advanced and indirect method
- ???

ASLR Notes

- ASLR makes common exploits fail by crashing the process with a high probability
- Moving a process entry point to random locations

Lecture 8: Control Defense and ROP

Ret2eax (Stack Juggling)

- Strcpy returns address of buf. That return address is stored in eax
- A subsequent call *eax would redirect control to buf
- Instead of returning to shellcode, we call directly to the shellcode
- Assumes no DEP

Other Non-randomized Sections

- Dynamically linked libraries are loaded at runtime. Called lazy binding
- Two important data structures
    1) Global Offset Table
    2) Procedure Linkage Table
- These data structures allow the dynamic linking of libraries

Dynamic Linking:

- When program sees printf, go to PLT (procedure linkage table)
- Then from plt, go to GOT. If GOT does not have address, the linker will fill the GOT with the correct address to printf
- Location of plt and GOT are in fixed locations by default

Got Hijacking

- You can use format string to overwrite a GOT entry

Quiz: What defenses can defeat GOT Hijacking?

- Maybe ASLR. But ASLR of GOT is not enabled by default

ROP (Returned oriented programming)

- A gadget is a sequence of instructions found already in our program
- A gadget is any instruction sequence ending with ret