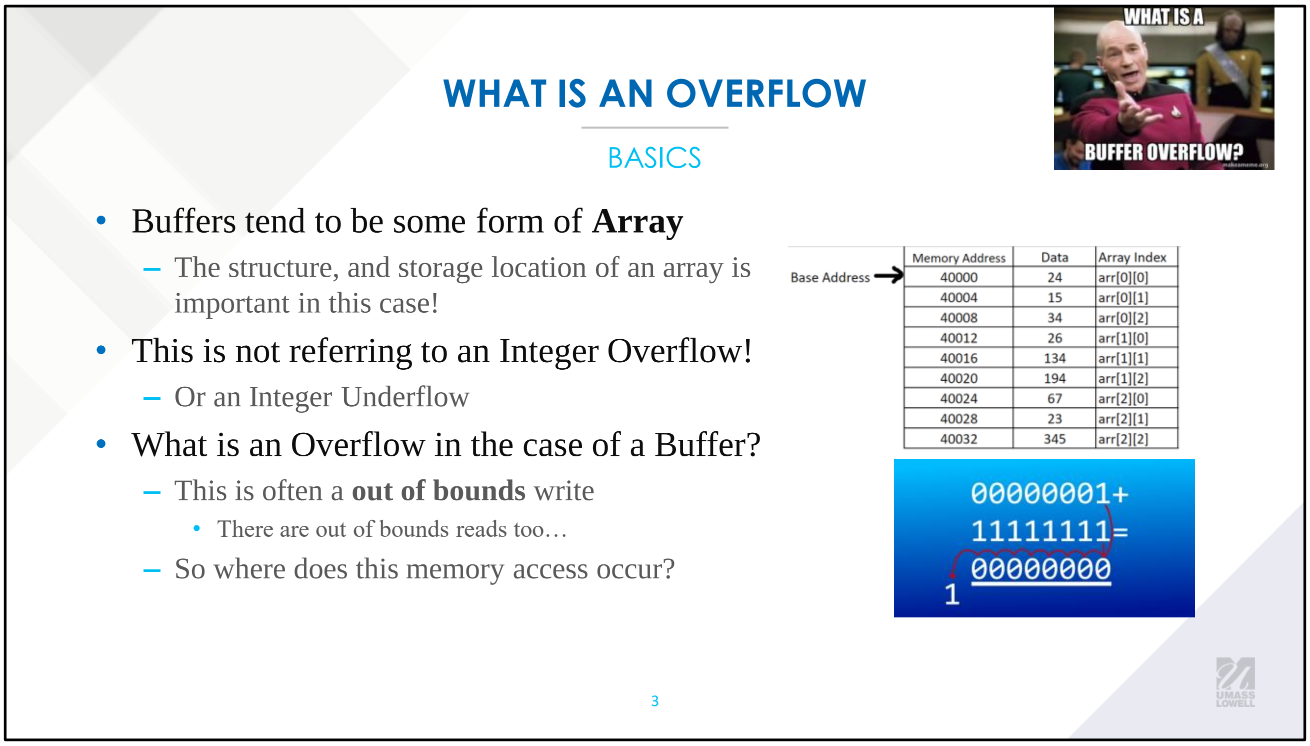
BORING BUFFER OVERFLOWS

CYBER SECURITY!



WHAT IS AN OVERFLOW

BASICS

WHAT IS A
BUFFER OVERFLOW?

- Buffers tend to be some form of **Array**
  - The structure, and storage location of an array is important in this case!
- This is not referring to an Integer Overflow!
  - Or an Integer Underflow
- What is an Overflow in the case of a Buffer?
  - This is often a **out of bounds** write
    - There are out of bounds reads too…
  - So where does this memory access occur?

Base Address →

Memory Address	Data	Array Index
40000	24	arr[0][0]
40004	15	arr[0][1]
40008	34	arr[0][2]
40012	26	arr[1][0]
40016	134	arr[1][1]
40020	194	arr[1][2]
40024	67	arr[2][0]
40028	23	arr[2][1]
40032	345	arr[2][2]

```
 00000001+
 11111111=
 00000000
1
```

3

- To answer this question we first need to know what a buffer is!
  - A buffer in our case is a simple array that stores (buffers) some input.
    - That is they are some contiguous, and discreet set of space in memory that is used to store some form of user input.
    - There are many kinds of buffers (1D, 2D, dynamic, static ect), but they are all internally represented as a 1D segment of memory.
  - Buffers are everywhere, in operating systems, webservers, and videogames. You cannot escape the buffer
    - This is a general statement, you probably can but we are not concerned with the edge cases too much
- Then what is a buffer overflow? this is simply an out of bounds **write**. That is we use some unsafe instruction (Or do it intentionally if you are having fun), to write to an address that is not owned by the array
  - This can be done by a malicious user (https://blog.qualys.com/vulnerabilities-threat-research/2023/10/03/cve-2023-4911-looney-tunables-local-privilege-escalation-in-the-glibcs-ld-so)
    - Dynamic linkers are on all if not most machines, maybe not the gcc dynamic linker, this exploited a buffer overflow with the environment variable (GLIBC_TUNABLES) used to "tune" and configure the systems dynamic linker (without having to recompile it)
      - This dynamic loader is responsible for resolving calls to shared libraries that a program would use when it is running.

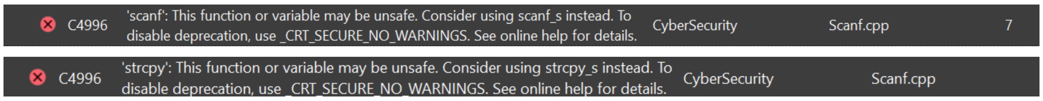
    - This can be done by manipulating the environment and startup of a program (https://www.qualys.com/2022/01/25/cve-2021-4034/pwnkit.txt)
        - Poll kit is common on many Linux machines, exploits overflow on argv array into envp used for discovering an executable file
- We can perform out of bounds reads, and even underflows but we are more focused on the overflows (as they are easier)
    - [Underflow](https://learn.microsoft.com/en-us/cpp/sanitizers/error-stack-buffer-underflow?view=msvc-170)
    - Since we don't often choose where the array reads, underflows are harder.

Ref:
https://owasp.org/www-community/vulnerabilities/Buffer_Overflow
https://cwe.mitre.org/data/definitions/122.html
https://ieeexplore.ieee.org/document/6956567
https://www.cs.swarthmore.edu/~newhall/unixhelp/C_arrays.html

UNSAFE INSTRUCTIONS

When you are the reason for the company safety video

- There are more unsafe instructions then I would know
- We can also use common conventions in an unsafe manner
  - That is accessing memory out of bounds manually



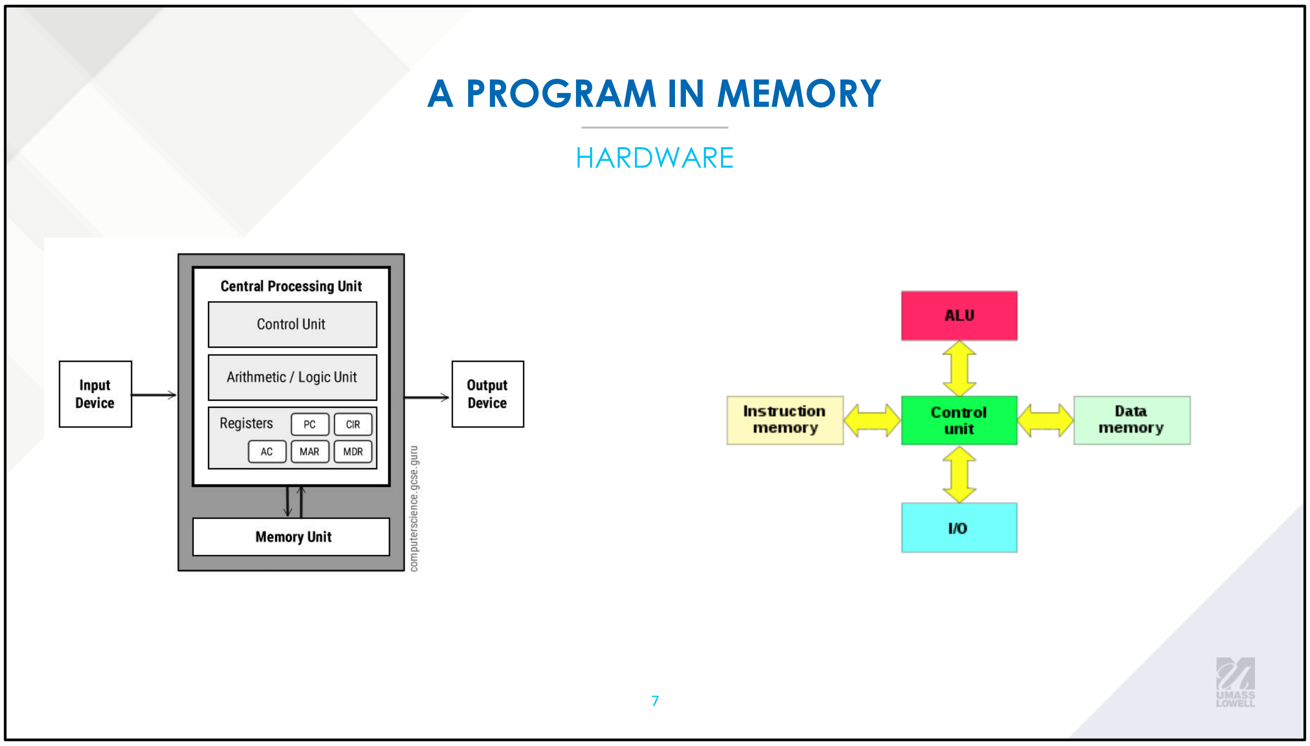
```
C4996  'scanf': This function or variable may be unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.  CyberSecurity  Scanf.cpp  7
C4996  'strcpy': This function or variable may be unsafe. Consider using strcpy_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.  CyberSecurity  Scanf.cpp
```

5

- There are a number of unsafe instructions that exist in many languages, primarily the older languages, but due to human error they can appear anywhere.
  - Undefined behaviors are unsafe, but we can for simplicity (and my sanity) not cover unsafe instructions due to undefined behavior
- The classic example, that you have likely used before are the *scanf* and *strncpy*
  - They perform copies of some input string into a destination string until all elements of the input string have been copied.

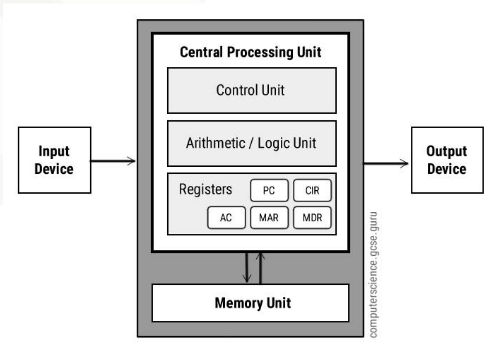
Goto -
https://www.qualys.com/2022/01/25/cve-2021-4034/pwnkit.txt#:~:text=435%20main%20(int%20argc%2C%20char%20*argv%5B%5D)
- guint is just a typedef for unsigned int
- g_strdup duplicates a string!
  - https://docs.gtk.org/glib/func.strdup.html
- g_find_program_in_path finds the absolute path to a file/executable by searching through a users $PATH, just as execvp would
  - https://docs.gtk.org/glib/func.find_program_in_path.html

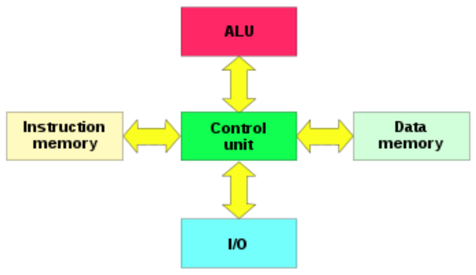
Refs:
https://blog.regehr.org/archives/213 -- Interesting



A PROGRAM IN MEMORY

HARDWARE





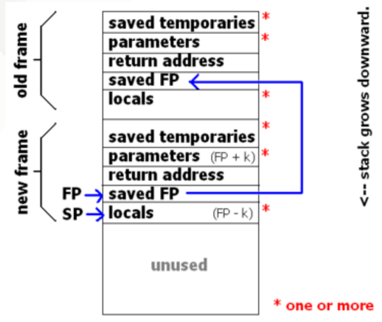
7

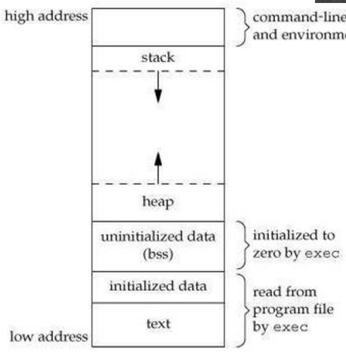
- Depending on the platform you use, the exact memory layout of a given program, and what you can access when running said program may vary
  - This is not only limited to a Windows (PE – Portable Executable) and Linux (ELF –Executable and Linkable Format) files
    - Different systems (Microcontrollers and whatnot) may use their own… unique layout.
  - We commonly use a form of Von Neumann architectures – that is the data of a program is located in the same memory (region) space as the instruction (code) for a given program
  - Some computers, primarily smaller chips such as microcontrollers and FPGAs use the Harvard Architecture. This separates the program data, and instructions (code). They also commonly make the data segments non-executable.
    - This architecture provides unique challenges when it comes to preforming buffer overflows

Refs:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_x86-64.html
https://isaaccomputerscience.org/concepts/sys_arch_architecture

A PROGRAM IN MEMORY

LOADED IN MEMORY

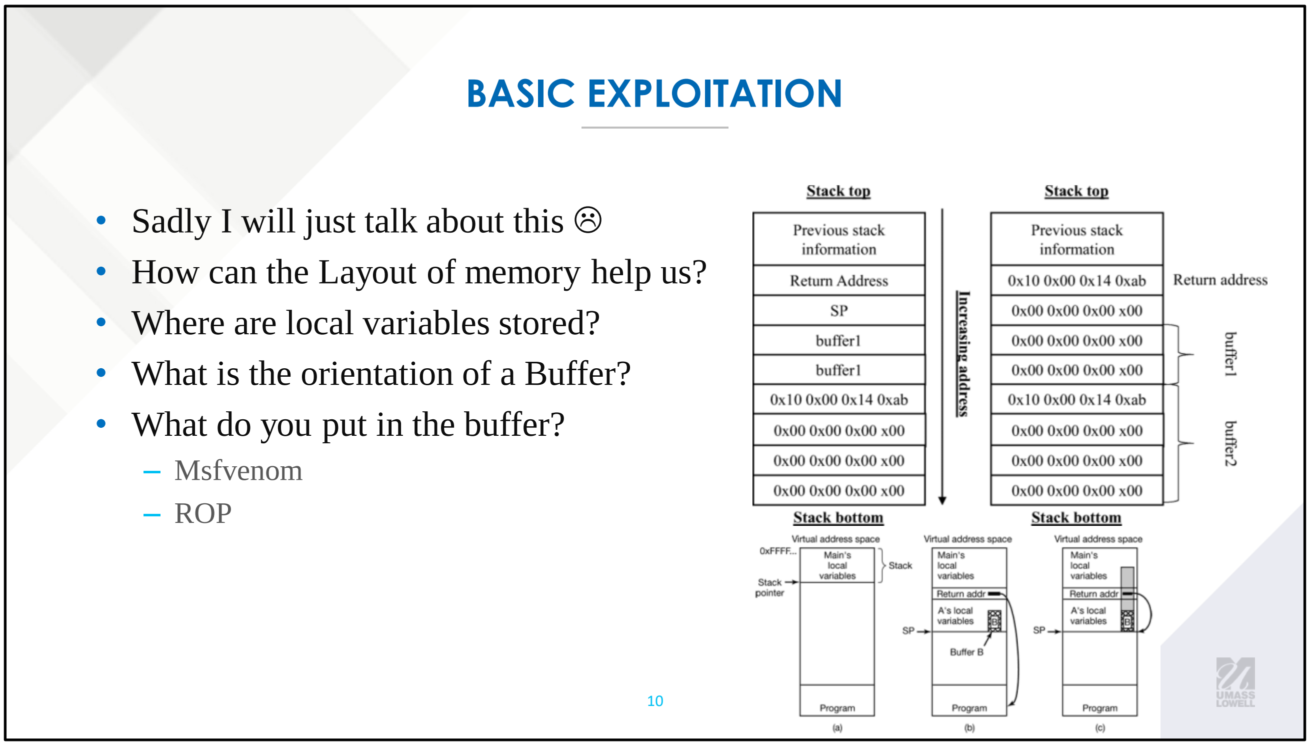



high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

- Generally, the principles of the overflow, and the way a program is loaded into memory is a constant
    - Look at this slide here, **all** processes (that is any running thing on the computer, be that something in user or kernel space) has at least a stack, data, and code segments. We of course we have the heap, and assorted headers or other segments too, but all we are concerned with in this case is the Stack and Heap segments.
        - The stack is where all local data declared (among other things) in the function is stored. So if we write to a buffer, locally declared this is where it will be stored. If we overflow said buffer, we will write to the stack.
        - This means a overflow allows us to arbitrarily write to the stack
- The stack store some important information
    - Each stack frame, that is something created during a function call contains the base pointer of the previous call, and a return address. We aim to overwrite the return address of a function, so when it returns from the current function we can modify the control flow so it does what we want it to do.
        - This can be through directly writing to the stack with executable code and jumping to it, or by chaining useful gadget together to perform some other purpose
- There are also heap overflows, but I will not be focusing on those are I did not prep

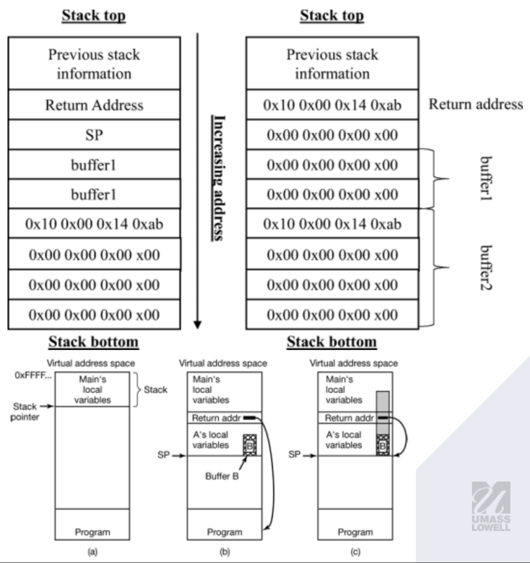
them.

Refs:
https://people.cs.rutgers.edu/~pxk/419/notes/frames.html
https://isaaccomputerscience.org/concepts/prog_sub_stack?examBoard=all&stage=all
https://manybutfinite.com/post/journey-to-the-stack/



BASIC EXPLOITATION

- Sadly I will just talk about this ☹
- How can the Layout of memory help us?
- Where are local variables stored?
- What is the orientation of a Buffer?
- What do you put in the buffer?
  - Msfvenom
  - ROP



10

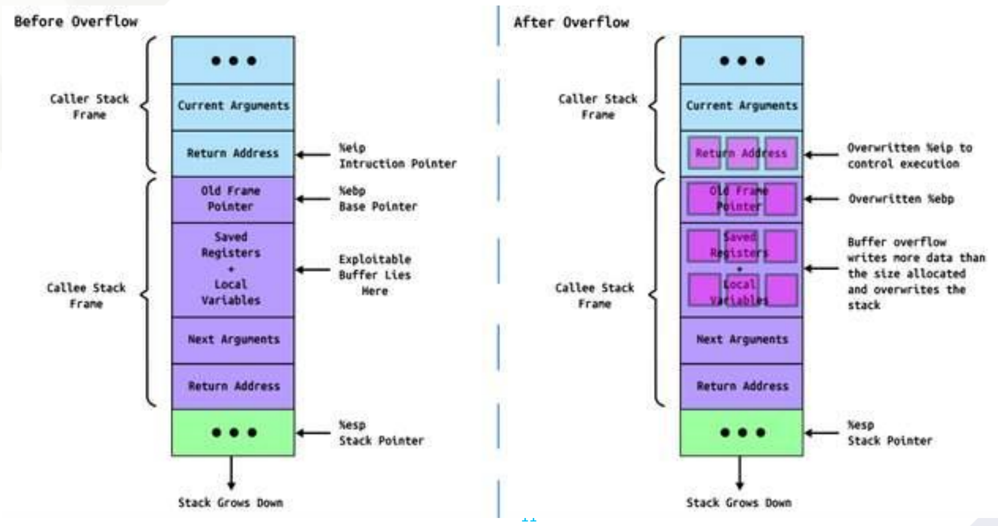
Due to time (Before, during and after this presentation) I cut the "Lets watch me fail at showing a buffer overflow" additionally headache.

- The layout of the program in memory, that is all local variables define in a function will be located in the stack frame of a function.
  - That is we have the
  - RETURN Address
  - Old Base (Frame Pointer) – Don't know why this image I stole from a research paper has SP
- Local variables are stored in the segments after the old Base pointer
- Buffers store from bottom to top (low to high in most cases)
  - That is arr[0] is at the bottom of the allocated memory and arr[n-1] is at the top of the allocated memory. So if we overflow the buffer we are corrupting memory upwards toward the OLD BP and Return Address.
- If the buffer is large enough, we could put the shell code directly into the buffer, and execute it from there (with some additional assumptions)
- Generally, with protections you would modify the return address, and chain useful gadgets together (ROP)
- If we want to pass in shell code, we could use msfvenom to generate some!

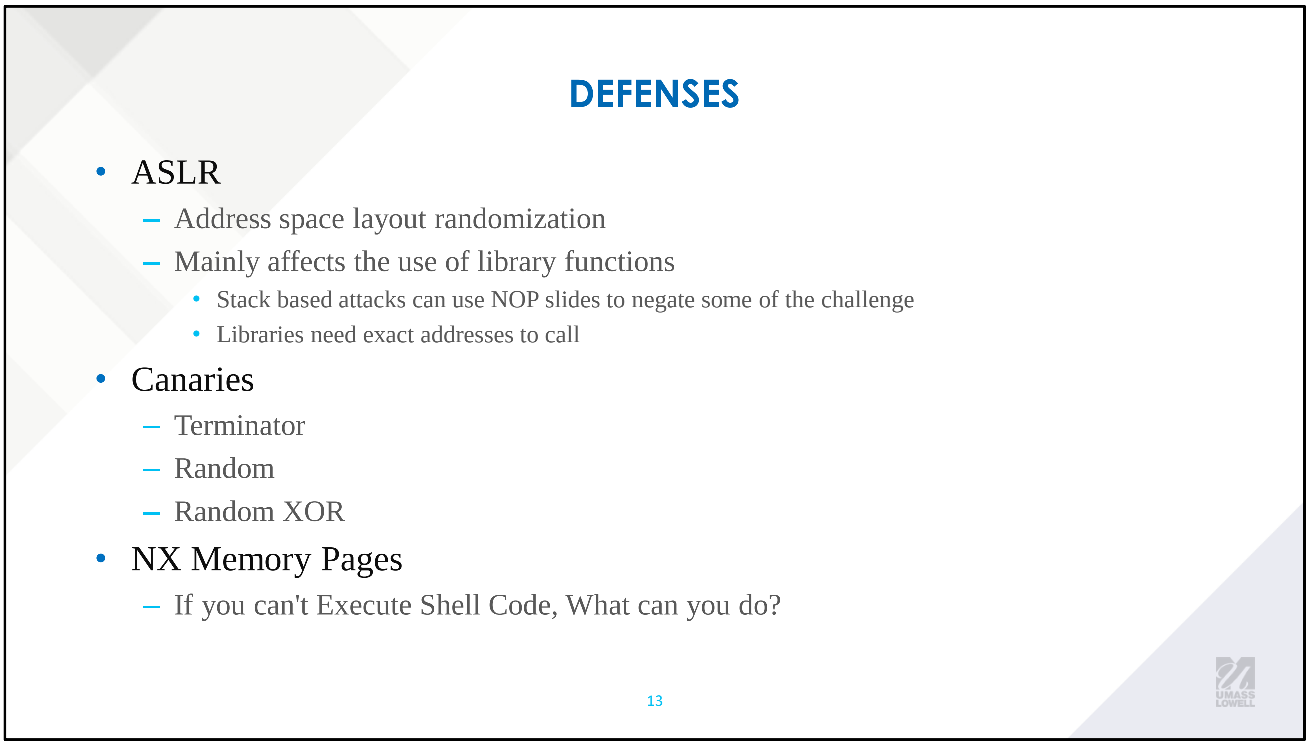
Ref:
MSF Venom: https://www.offsec.com/metasploit-unleashed/msfvenom/

https://www.imperva.com/learn/application-security/buffer-overflow/
https://owasp.org/www-community/vulnerabilities/Buffer_Overflow
https://www.cloudflare.com/learning/security/threats/buffer-overflow/
https://www.malwarebytes.com/blog/threats/buffer-overflow

ANOTHER IMAGE



Awful Image, too big for other slide


DEFENSES

- ASLR
  - Address space layout randomization
  - Mainly affects the use of library functions
    - Stack based attacks can use NOP slides to negate some of the challenge
    - Libraries need exact addresses to call
- Canaries
  - Terminator
  - Random
  - Random XOR
- NX Memory Pages
  - If you can't Execute Shell Code, What can you do?

13

- ASLR
  - You would be surprised at how much a computer does for you (ASM Address offsets, and loads)
  - We are in this case primarily concerned with Dynamic (shared) Libraries the system loads at some “globally accessible” address , and dynamically links programs to for you.
    - Linux does have Position Independent Executable (PIE) too so this is not only for shared libraries (SO files)
  - In the past the system would load these in a constant memory location, ASLR will randomize the locations of not only these Dynamic libraries, but also the other segments of a memory, the stack, heap, text, ect.
    - This makes it impossible for attackers to make assumptions about the system, and each attack needs to be “semi” tailored to the host system.
  - The randomization provided in a 32 bit system is limited, and can easily be brute forced. This is due to the layout of the 32 bit memory address. Windows 32 bit machines only randomize 8 bits, as 14 are off limits, so we can easily iterate through each of the 256 options
    - 64 bit programs are in the hundred thousand range
    - 64 bit DLLS are in the half a million range
- Canaries (Many varieties)
  - Different Types

    - Terminator –null bytes and other terminating characters that would end a string buffer overflow. This may be bypassed
    - Random – Null byte followed by random values inserted into the program at initialization, before returning, the programchecks if this has been changed.
        - Can be read or in some scenarios guessed on bit at a time (fork no exec)
    - Random XOR – Null byte followed by a random value XORed with some control variable (Such as the Base pointer), if either is modified the overflow is detected.

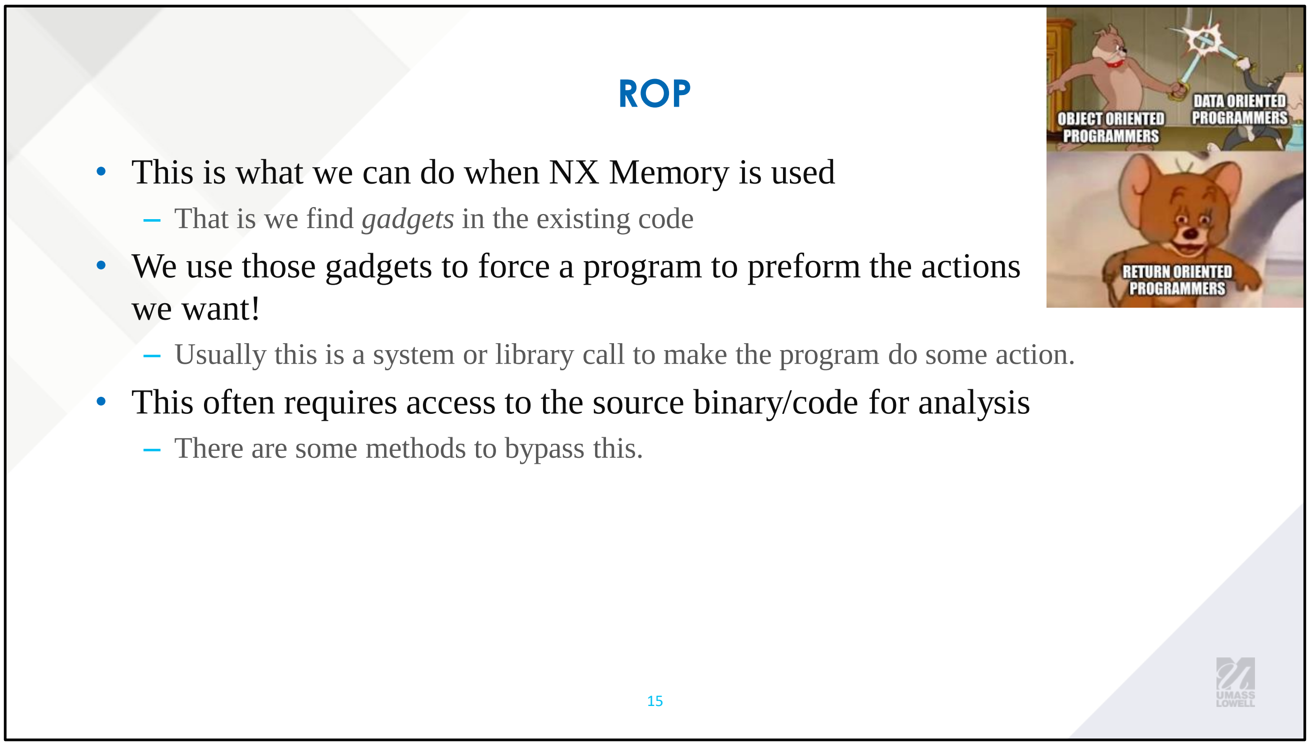
    - **NX Memory Pages**: As it says, if the stack is non-executable, it is hard to execute shell code. SO they use ROP

- NX Memory

Ref:
Neet source on ASLR Facts: https://www.mandiant.com/resources/blog/six-facts-about-address-space-layout-randomization-on-windows#:~:text=ASLR%20mixes%20up%20the%20address,laid%20out%20at%20the%20time.

Canaries:
https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/

NX:
https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention
https://learn.microsoft.com/en-us/windows/win32/win7appqual/dep-nx-protection
https://textbook.cs161.org/memory-safety/mitigations.html – Has alot not sure how good all of It is



ROP

- This is what we can do when NX Memory is used
  - That is we find *gadgets* in the existing code
- We use those gadgets to force a program to preform the actions we want!
  - Usually this is a system or library call to make the program do some action.
- This often requires access to the source binary/code for analysis
  - There are some methods to bypass this.

OBJECT ORIENTED PROGRAMMERS
DATA ORIENTED PROGRAMMERS
RETURN ORIENTED PROGRAMMERS

15

We use this method when the **nonexecutable** memory protections are in place, since we cannot just load the necessary instructions into the stack and execute them to perform arbitrary commands.
- On windows this is Data Execution Prevention (DEP)

This is done by chaining useful gadgets together, these often consist of instructions that load a value into a register and return. We chain them together by placing their return addresses (and any arguments) onto the stack one after another, so they are chained together by the *return* calls in the assembly which Pops off return address, start executing from there!

Ref:
https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/rop-chaining-return-oriented-programming
https://ctf101.org/binary-exploitation/return-oriented-programming/
https://research.nccgroup.com/2023/06/12/defeating-windows-dep-with-a-custom-rop-chain/

BROP

- Blind!
  - No info about target system
- Requires special circumstances

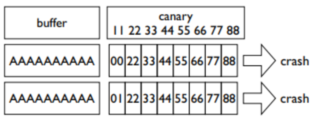


Figure 5. Stack reading. A single byte on the stack is overwritten with guess X. If the service crashes, the wrong value was guessed. Otherwise, the stack is overwritten with the same value and no crash occurs. After at most 256 attempts, the correct value will be guessed. The process is then repeated for subsequent bytes on the stack.

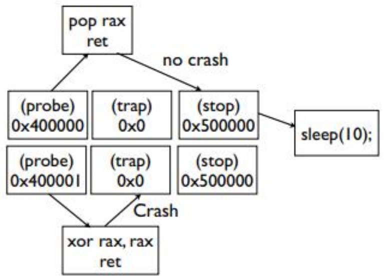


Figure 10. Scanning for pop gadgets. By changing the stack layout, one can fingerprint gadgets that pop words from the stack. For example, if a "trap gadget" is executed rather than popped, the program will crash.

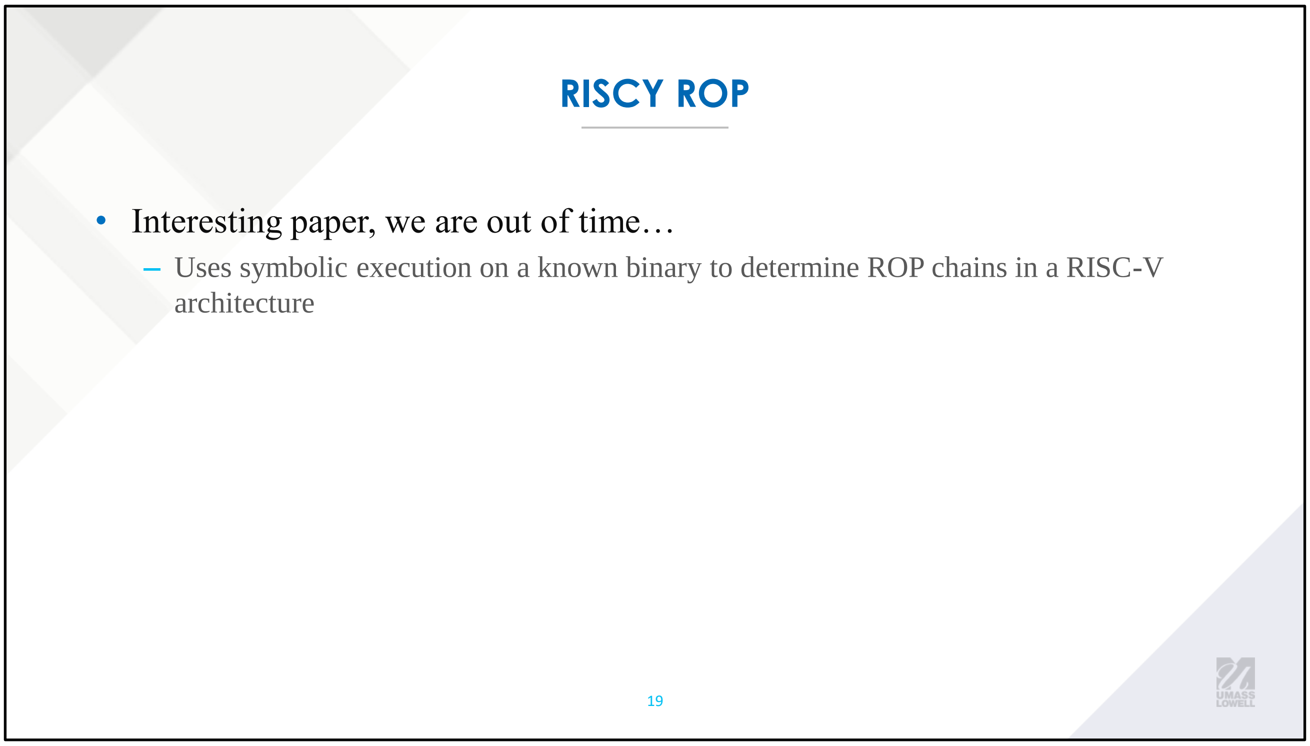
17

Blind Return Oriented programming is ROP done on a (often remote) system without possessing the binary

- This is implemented in the paper "Hacking Blind"
  - It defetes ASLR, NX Memory, and Stack Canaries
- It requires a special set of circumstances
  - Remote program restarts on crash
    - That is it forks from a parent process (Copies parent process characteristics including canaries), but does not exec (this would overwrite the current process's memory, randomizing the canary each time).
    - This allows us to overflow into the canary with a 1 bit difference each time, slowly rewriting the canary!
- The goal of BROP is to find enough gadets (Generally by looking in the **procedure linkage table**) to perform a **write** system call to read out the binary of the program for further analysis
  - They jump into the Text segment of the program and have a stop gadet on the stack if the connection closes they know they know they jumped to an invalid address, if it hangs they know they are in a valid address (The attacker could use a stop gadget that writes to the network instead – it does not always need to loop forever)
  - They vary the layout of the stack to identify gadgets
    - probe, stop, traps (trap, trap, . . .). Will find gadgets that do not pop

the stack like ret or xor rax, rax; ret
- probe, trap, stop, traps. Will find gadgets that pop exactly one stack word like pop rax; ret or pop rdi; ret. Figure 10 shows an illustration of this.
    - Used if we want to just find a large number of pop register gadgets and deduce their actual function by examining the behavior of a system call
- probe, stop, stop, stop, stop, stop, stop, stop, traps. Will find gadgets that pop up to six words (e.g., the BROP gadget).
    - Optimized, easiest to think on
- The find the PLT which can be done in great confidence (If a number of calls 16 bytes apart do not crash the system – return failure code not exit!)
    - Modify the stack with the BROP gadet to locate the signature of Strcmp.

REFs
https://maskray.me/blog/2021-09-19-all-about-procedure-linkage-table



RISCY ROP

- Interesting paper, we are out of time…
  - Uses symbolic execution on a known binary to determine ROP chains in a RISC-V architecture

19

https://dl.acm.org/doi/10.1145/3545948.3545997

CONCLUSION

DONE, THERE IS NONE