Stack Based Attacks in Linux (an intro)

Saint Louis Linux Users Group (STLLUG)

Bryce L. Meyer

April 20th, 2023

Overview

Note: most slides are extracted from a course I developed and teach at Franciscan University of Steubenville…it uses the Grey Hat Hacking textbook, and a lab we built called the Danger lab

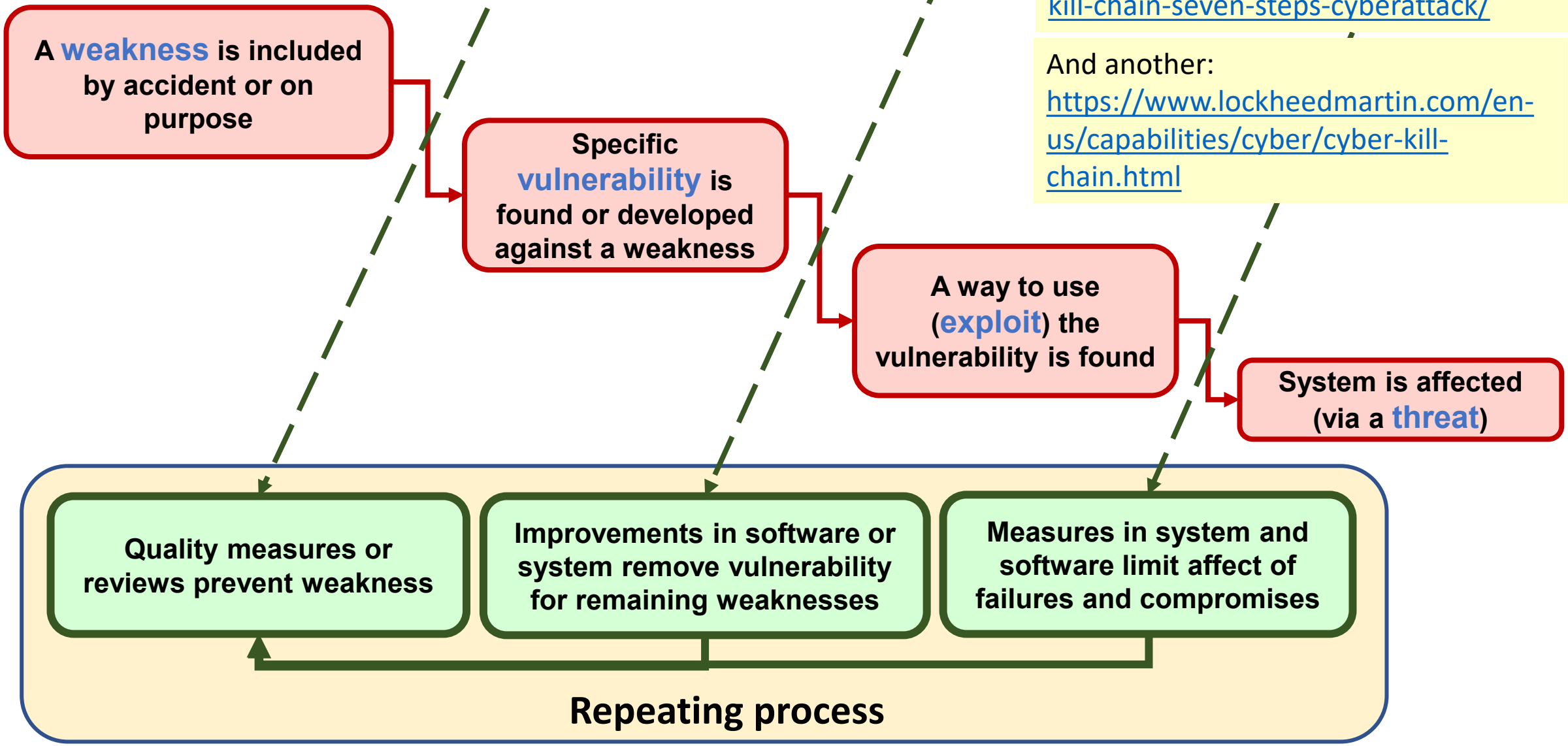
- From Weakness to Exploit: What makes a hole…
- Programs in Memory 101, where they go, what happens at runtime
- Aleph One's Famous Paper and Stack Overflow Basics
  - Simple Stack Overflow
  - What is shellcode?
  - NOP Sleds
- The Arms Race: Linux overflow preventers, and work arounds
  - 64-bit v. 32-bit
  - Stack Canaries
  - NX Bit and ROP
  - Address Space Randomization
- Do they still exist?

Several example code segments herein come from the book Grey Hat Hacking, 6ed:..get it!!!
Gray Hat Hacking: The Ethical Hacker's Handbook, Sixth Edition, 6th Edition [Book] (oreilly.com)
https://www.oreilly.com/library/view/gray-hat-hacking/9781264268955/

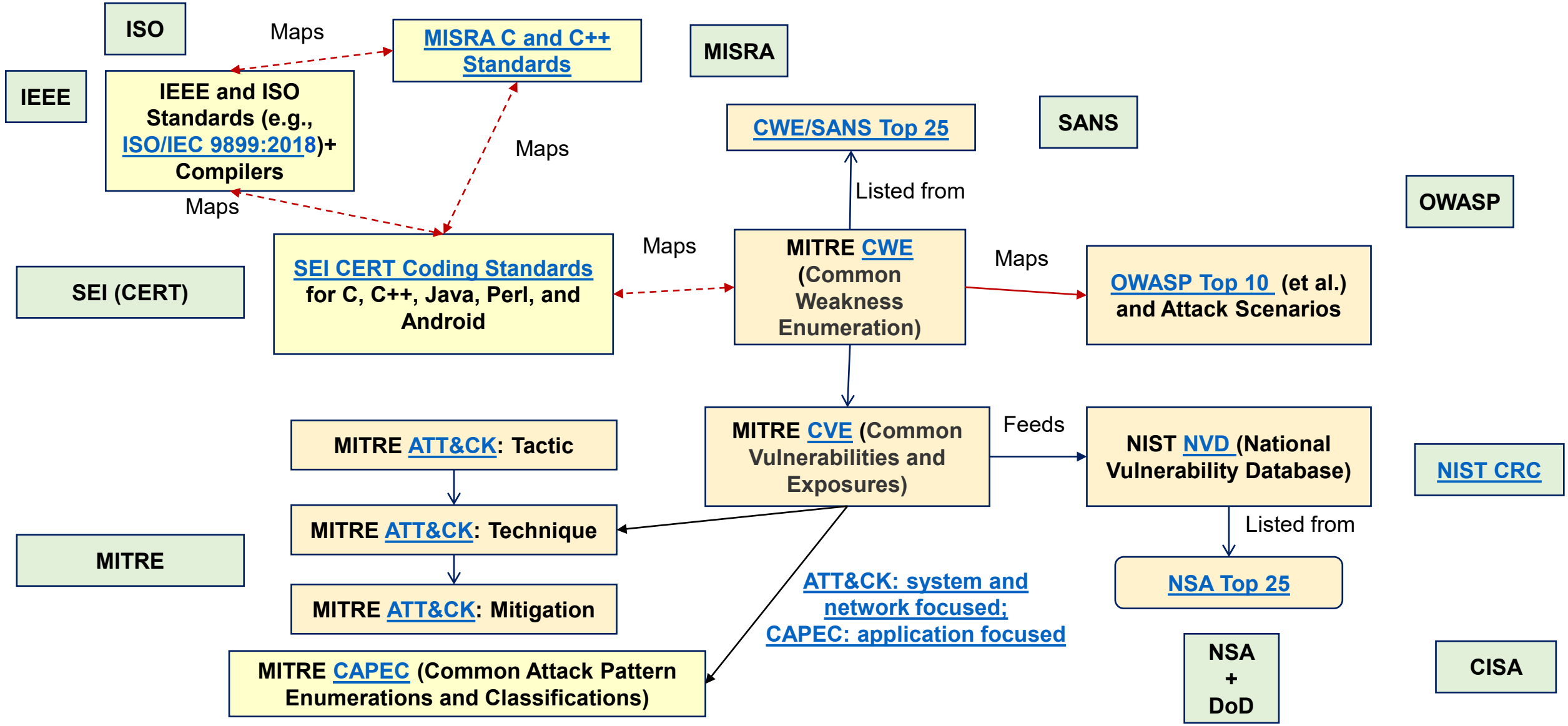
Chain of Danger (Meyer)

See another method:
https://www.eccouncil.org/cybersecurity-exchange/threat-intelligence/cyber-kill-chain-seven-steps-cyberattack/

And another:
https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html



Where can I find threats?



Weaknesses of concern here

- CWE - CWE-1218: Memory Buffer Errors (4.10) (mitre.org)
- CWE - CWE-787: Out-of-bounds Write (4.10) (mitre.org)
- CWE - CWE-121: Stack-based Buffer Overflow (4.10) (mitre.org)
- CWE - CWE-788: Access of Memory Location After End of Buffer (4.10) (mitre.org)

Coding that Leads to Vulnerabilities (A few)

- MEM35-C. Allocate sufficient memory for an object - SEI CERT C Coding Standard - Confluence (cmu.edu)
- STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator - SEI CERT C Coding Standard - Confluence (cmu.edu)
- ARR30-C. Do not form or use out-of-bounds pointers or array subscripts - SEI CERT C Coding Standard - Confluence (cmu.edu)

SEI CERT Coding Standards - CERT Secure Coding - Confluence (cmu.edu)
https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

Tools

Tools you will need:

- gcc or similar compiler
- gdb w/GEF or similar debugger: need these to map out program memory
- w/ Kali (make sure you installed these):
  - python3 Python interpreter
  - Pwntools: toolbox of tools to aid in overflow hacks
  - Ropper: Make ROP shellcode
  - One_gadget: find gadgets for ROPs
  - vmmap (in GEF): maps memory use

pwntools: CTF Framework and Exploit Development Library

- CTF Framework: Capture the Flag (CTF) set of tools and libraires to enable red teaming competitions
- pwntools: a CTF framework and exploit development library collection of many tools and methods
  - Pwnlib has the tools you can invoke and use
  - Disassemblers, assemblers, ELF resolvers, exploit examples and primitives, to make paylods, etc.
- Popular with script-kiddies
- Lots of python code

https://pwntools.readthedocs.io/en/latest/tubes.html

https://github.com/Gallopsled/pwntools-tutorial#readme

https://docs.pwntools.com/en/latest/

Text p. 64-66

Compiler Warnings to prevent overflows (in this case gcc)

Compilers often catch security issues in warnings…

It is critical in real life that YOU DO NOT TURN THEM OFF for deployed code…

Static Analysis and Linting SW should catch the same issues too.

Source Code from book ***Gray Hat Hacking***…a good text for this stuff!!!...I compiled it on Kali :0)

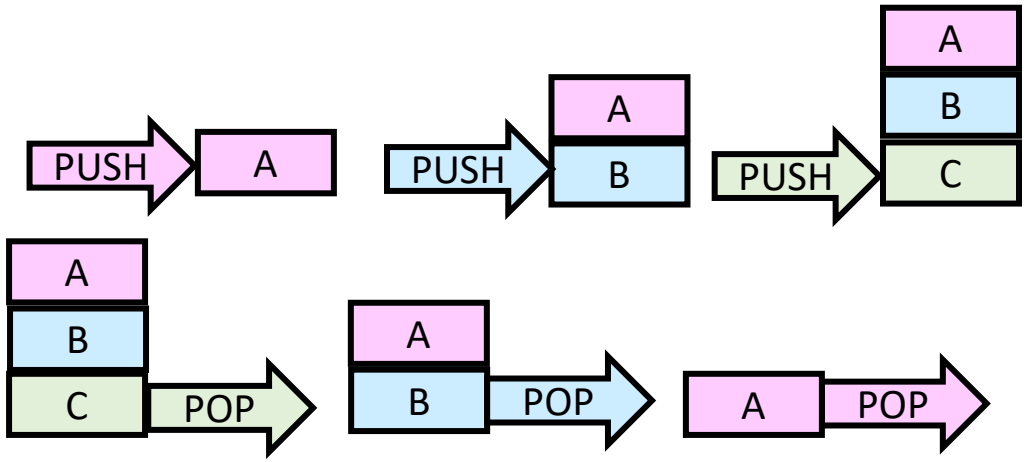
```
$ gcc vuln.c -o vuln
vuln.c: In function ‘auth’:
vuln.c:24:5: warning: ‘read’ writing 512 bytes into a region of size 64 overflows the destination [-Wstringop-overflow=]
   24 |     read(connfd, buf, 512);
      |     ^~~~~~~~~~~~~~~~~~~~~~
vuln.c:23:10: note: destination object ‘buf’ of size 64
   23 |     char buf[BUFLEN];
      |          ^~~
In file included from vuln.c:5:
/usr/include/unistd.h:371:16: note: in a call to function ‘read’ declared with attribute ‘access (write_only, 2, 3)’
  371 | extern ssize_t read (int __fd, void *__buf, size_t __nbytes) __wur
      |                ^~~~
vuln.c:24:5: warning: ‘read’ writing 512 bytes into a region of size 64 overflows the destination [-Wstringop-overflow=]
   24 |     read(connfd, buf, 512);
      |     ^~~~~~~~~~~~~~~~~~~~~~
vuln.c:23:10: note: destination object ‘buf’ of size 64
   23 |     char buf[BUFLEN];
      |          ^~~
/usr/include/unistd.h:371:16: note: in a call to function ‘read’ declared with attribute ‘access (write_only, 2, 3)’
  371 | extern ssize_t read (int __fd, void *__buf, size_t __nbytes) __wur
      |                ^~
```

Programs run in virtual memory

- Processes (programs) are loaded into memory in sections.
- **.text:** Text segment: Contains (binary) executable instructions, usually read-only
- **.data:** Initialized Data Segment: portion of the virtual address space of a program that contains the global variables and static variables initialized by the program. Read only parts and writable parts.
- **.bss:** Uninitialized Data segment (block started by symbol): everything set as 0, or variables without an initial value set in the program
- **Heap:** Dynamic memory allocation area, starts at end of .bss and grows from there into the 'as needed' part, uses malloc, realloc, and free to control it.
- **Stack:** Stores automatic variables, memory values and pointers, Last In First Out, grows to address 0 (i.e. into the 'as needed' part). Memory units in the stack are called frames.
- **Env(ironment)/Arg(uments):**Where system-level variables are stored that control programs, such as path, shell name, hostname, etc.

https://www.geeksforgeeks.org/memory-layout-of-c-program/

Stack Operations



- *Every program has its own stack in virtual memory…*
- *Stack is First In Last Out (FILO=LIFO)*
- **Push:** Add items to stack
- **Pop**: pull items from stack
- **Call**: invoke a section of code stored in memory
- **Walk:** incrementally read (or run) the values in the stack from first in to last in
- **Trace**: debugging result of executing or calling memory, a debugging walk

KEY POINTERS (32-bit, 64-bit=swap E for R, 16-bit=drop the E):

- **ESP:** Extended Stack Pointer: top of the (local) stack address (low address)
- **EBP:** Extended Base Pointer: bottom (high address) of current stack frame (aka FP)
- **EIP: (real EIP) Extended Instruction Pointer: Track current memory being executed (called)**

https://wiki.osdev.org/Stack_Trace

https://www.techtarget.com/whatis/definition/stack-pointer

https://wiki.osdev.org/Stack

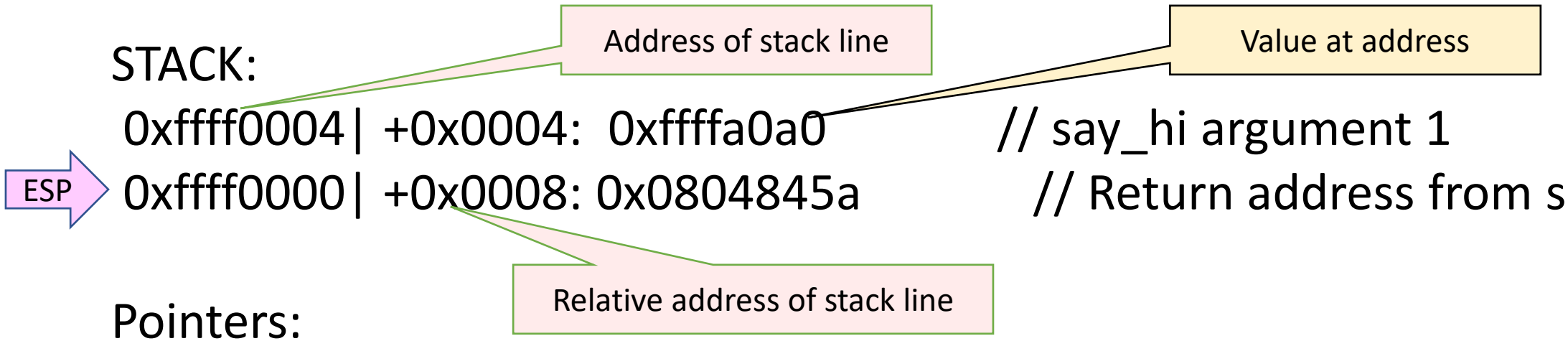
https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/

https://en.wikipedia.org/wiki/Extended_memory

https://en.wikipedia.org/wiki/Function_prologue_and_epilogue

Figuring out Stack Frames

- Track the current EIP, ESP, EBP
- Every line in the stack frame has a memory address in the stack (32-bit: 0xffffffff0 to 0x11111111 usually
- Lines in machine code are not the same space normally, they are .text…



Pointers:

EIP: 804840b (first line of say_hi)

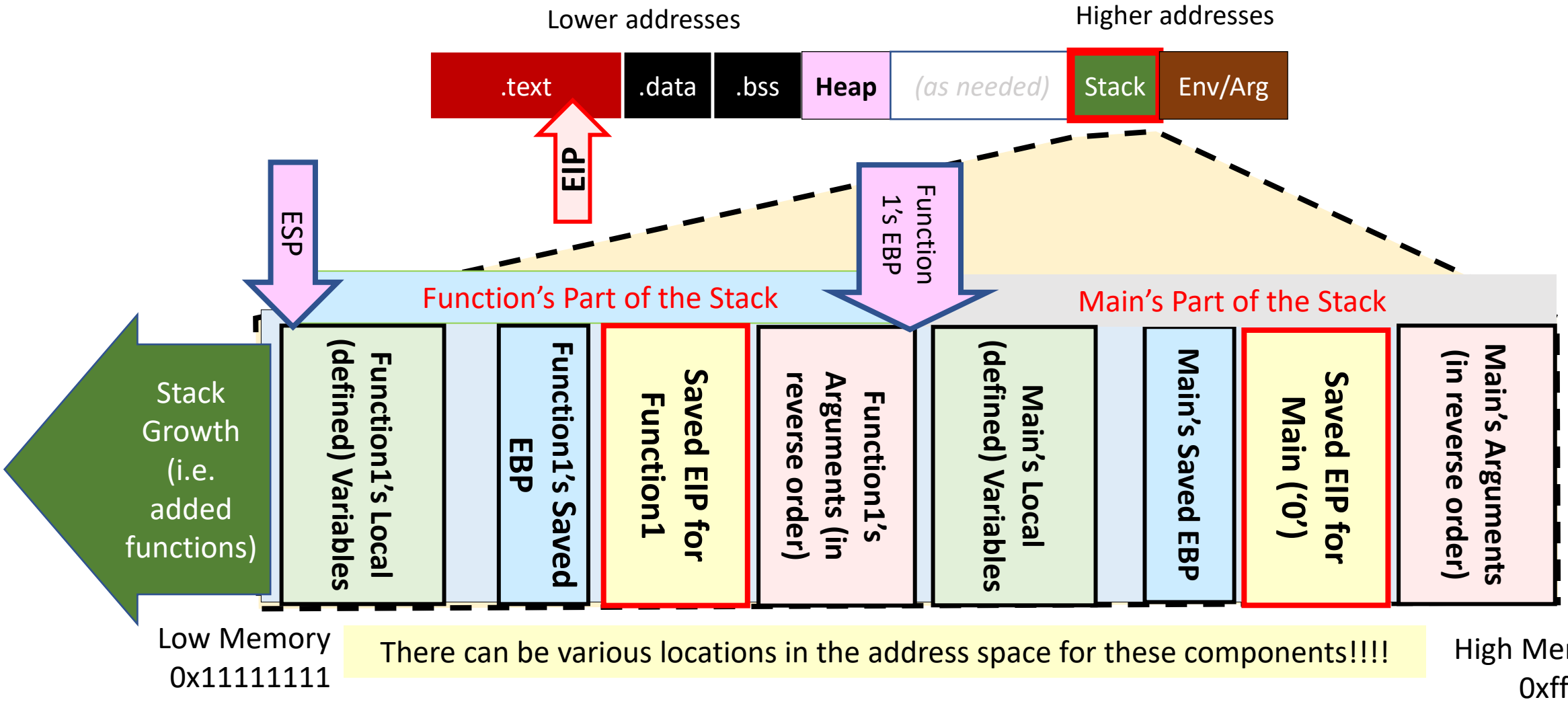
ESP = 0xffff0000

EBP = 0xffff002c (room for say_hi's frame, way above current lines in stack)

Sometimes the stack line is called the stack frame…be wary of context

32-bit stack

.text is where binary instructions go…
The Instruction Pointer moves down the .text list until it hits a ‘return’ instruction which sends it back to its calling function’s binary (usually main, line below call)



Key Compiler Options: gcc

- Many compile options open holes…sometimes driver or 3rd party app makers use them for shortcuts to avoid warnings, or use them in test and leave them in for deployed code on accident…
- -w : ignore warnings….. (bad idea!!!!)

It is bad to use No's for these defaults (i.e. –fno-) :

- -fstack-protector(-all): add stack protection guards (i.e. canaries..more later)
- -fpie: Position Independent Executable
- This disables NX bit implementation:
- *-z execstack*

https://man7.org/linux/man-pages/man1/gcc.1.html

What happens when I compile and run a program with a function

- When a program runs, it walks the machine code instruction lines until it encounters a call to a function.
- At the call, program loads the stack like so (higher value addresses to lower addresses):

1. Stores function arguments in reverse order (i.e. 3,2,1)
2. Stores the instruction pointer for the line after the call
3. Stores the location of the base of the stack
4. Adds a segment of memory as allocated for each variable, strings, numbers, etc.

What happens when I compile and run a program with a function (2)

- When the program fills the data for the variables as it runs, it uses the allocated memory in the stack, from lower addresses to higher addresses, so if there is more input to the variable then allocated, it overwrites into higher addresses
- When the function hits return instruction, it uses the stored instruction pointer to continue into the calling function past the call.
- If the stored instruction pointer is overwritten, it tries to use the data that is now in that spot….usually this causes a segmentation error

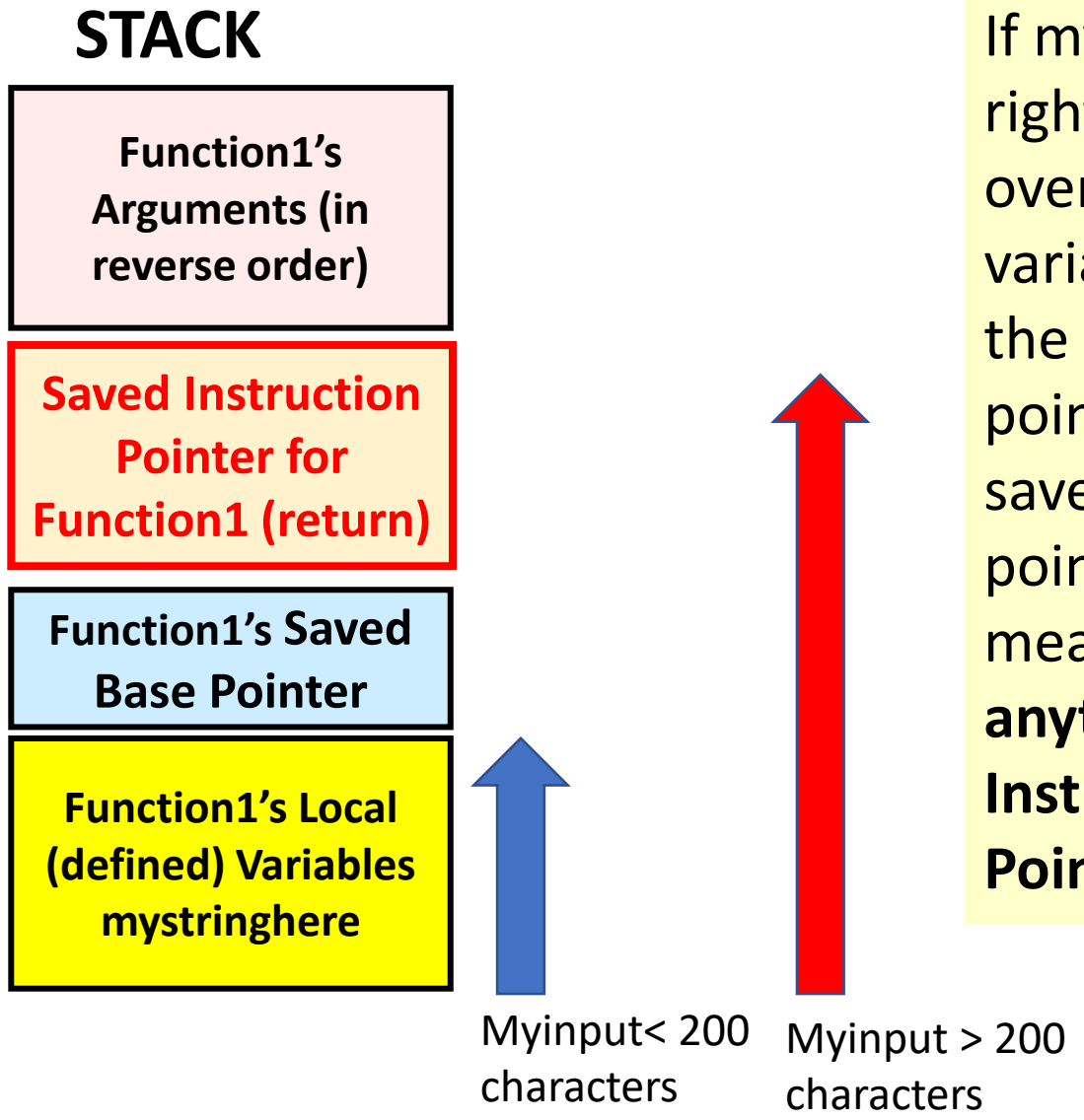
How does a stack overflow work?

- A variable, array, etc. is assigned a size:E.g.

```
char mystringhere [200];
```

- Another part of the program allows more than the size to go into the variable" like:

```
char myinput [600];
strcpy (mystringhere, myinput);
```



If myinput is the right size, it overflows the variable area, past the saved base pointer, into the saved instruction pointer … which means **I can put anything as Saved Instruction Pointer!!!!**

BUFFER OVERFLOW EXAMPLE: Stack Buffer Overflow

Note: if 64-bit os, you need to compile as 32-bitfor these examples.. and TURN OFF THE STACK PROTECTOR…
$ gcc –m32 –fnostack-protector –o <PROGRAM>

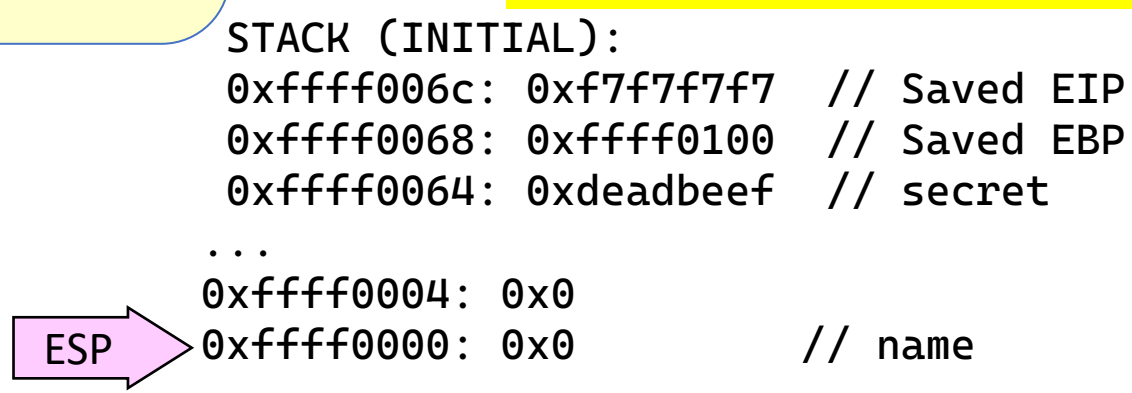
hacked.c:

```
#include <stdio.h>

int main() {
   int secret = 0xdeadbeef;
   char name[100] = {0};
   read(0, name, 0x100);
   if (secret == 0x1337) {
      puts("Wow! Here's a secret.");
   } else {
      puts("I guess you're not cool
enough to see my secret");
   }
}
```

100 decimal vs 0x100 (Hexadecimal =256 decimal)
So, main can read ***name*** more than 100 bytes…OOPS!

BIG NOTE: IT IS NEVER THIS EASY….THIS EXAMPLE ASSUMES A VERY OLD OS…



https://ctf101.org/binary-exploitation/buffer-overflow/

This is an example of this weakness:
CWE - CWE-131: Incorrect Calculation of Buffer Size (4.10) (mitre.org)
With dozens of CVE vulnerabilities….

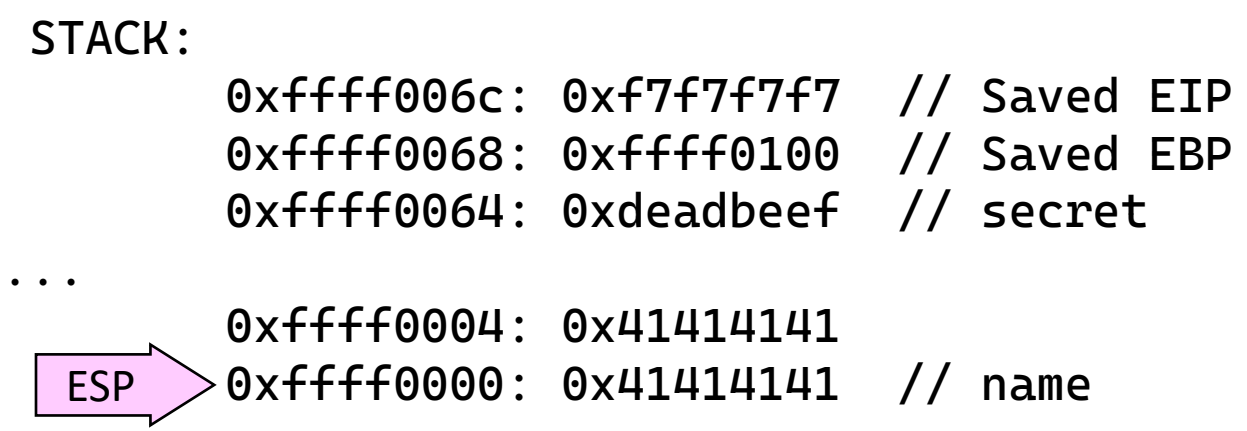
Stack Buffer Overflow (2)

0x100 (hex) = 256 decimal fyi…

```
#include <stdio.h>

int main() {
   int secret = 0xdeadbeef;
   char name[100] = {0};
   read(0, name, 0x100);
   if (secret == 0x1337) {
      puts("Wow! Here's a secret.");
   } else {
      puts("I guess you're not cool
enough to see my secret");
   }
}
```

Set name be 100 ‘A’’s (fyi, character A=0x41)
So
name=“AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA”
./hacked.c (with all those A’s)



ALL OK SO FAR….

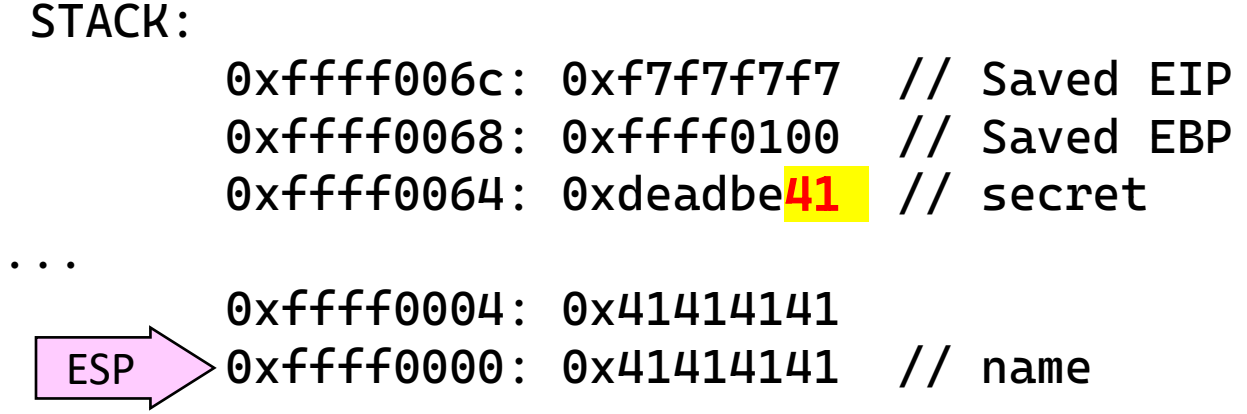
https://ctf101.org/binary-exploitation/buffer-overflow/

Stack Buffer Overflow (3)

```
#include <stdio.h>

int main() {
   int secret = 0xdeadbeef;
   char name[100] = {0};
   read(0, name, 0x100);
   if (secret == 0x1337) {
      puts("Wow! Here's a secret.");
   } else {
      puts("I guess you're not cool
enough to see my secret");
   }
}
```

Set name be 101 'A''s (fyi, character A=0x41)
So
name="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA"



https://ctf101.org/binary-exploitation/buffer-overflow/

Stack Buffer Overflow (4)

```
#include <stdio.h>

int main() {
    int secret = 0xdeadbeef;
    char name[100] = {0};
    read(0, name, 0x100);
    if (secret == 0x1337) {
        puts("Wow! Here's a secret.");
    } else {
        puts("I guess you're not cool
enough to see my secret");
    }
}
```

Set name be 102 'A''s (fyi, character A=0x41)
So
name="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA**AA**"

```
STACK:
          0xffff006c: 0xf7f7f7f7  // Saved EIP
          0xffff0068: 0xffff0100  // Saved EBP
          0xffff0064: 0xdead4141  // secret
...
          0xffff0004: 0x41414141
ESP       0xffff0000: 0x41414141  // name
```

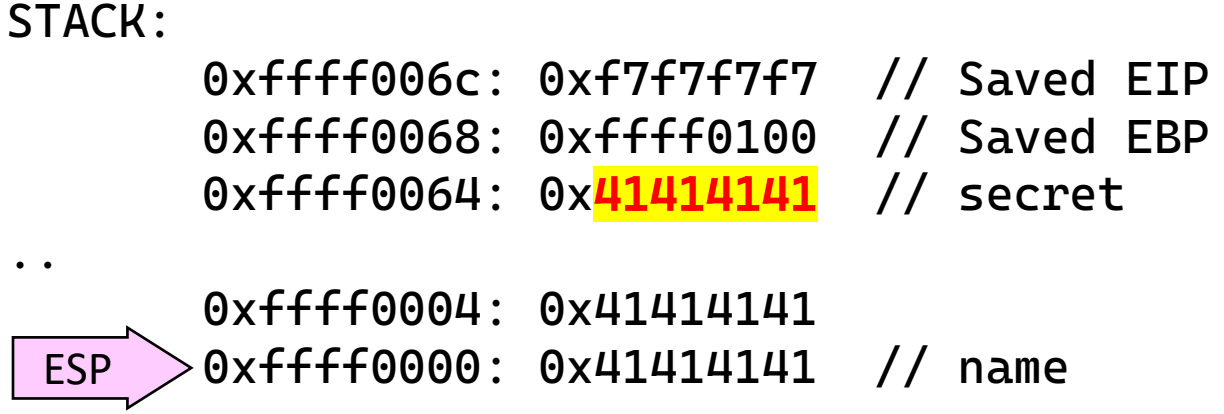
https://ctf101.org/binary-exploitation/buffer-overflow/

Stack Buffer Overflow (5)

```
#include <stdio.h>

int main() {
   int secret = 0xdeadbeef;
   char name[100] = {0};
   read(0, name, 0x100);
   if (secret == 0x1337) {
      puts("Wow! Here's a secret.");
   } else {
      puts("I guess you're not cool
enough to see my secret");
   }
}
```

Set name be 104 'A''s (fyi, character A=0x41)
So input for
name="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"



https://ctf101.org/binary-exploitation/buffer-overflow/

Stack Buffer Overflow (6)

```
#include <stdio.h>

int main() {
   int secret = 0xdeadbeef;
   char name[100] = {0};
   read(0, name, 0x100);
   if (secret == 0x1337) {
      puts("Wow! Here's a secret.");
   } else {
      puts("I guess you're not cool
enough to see my secret");
   }
}
```

Set name be 108 'A''s (fyi, character A=0x41)
So input for
name="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
A"

```
      STACK:
                0xffff006c: 0xf7f7f7f7  // Saved EIP
                0xffff0068: 0x41414141  // Saved EBP
                0xffff0064: 0x41414141  // secret
     ...
                0xffff0004: 0x41414141
    ESP ->      0xffff0000: 0x41414141  // name
```

AT THIS POINT YOU MIGHT GET A SEGMENTATION ERROR!

https://ctf101.org/binary-exploitation/buffer-overflow/

Stack Buffer Overflow (7)

```
#include <stdio.h>

int main() {
   int secret = 0xdeadbeef;
   char name[100] = {0};
   read(0, name, 0x100);
   if (secret == 0x1337) {
      puts("Wow! Here's a secret.");
   } else {
      puts("I guess you're not cool enough to see my secret");
   }
}
```

Set name be 112 'A''s (fyi, character A=0x41)
So input for
name="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA"

```
STACK:
          0xffff006c: 0x41414141   // Saved EIP
          0xffff0068: 0x41414141   // Saved EBP
          0xffff0064: 0x41414141   // secret
  ...
          0xffff0004: 0x41414141
ESP ->    0xffff0000: 0x41414141   // name
```

AT THIS POINT YOU GET A SEGMENTATION ERROR!

If I substitute the last 4 A's (0x41414141) with an address in memory, when the function gets to the end (return in assembly) it will use the substitute in the Stored EIP's place to jump to the substituted address!!!

https://ctf101.org/binary-exploitation/buffer-overflow/

Aleph One's famous smashing the stack paper and shellcode

- Key 1996 paper that shows exactly how to overflow the stack using C code, and then how to insert shellcode to gain a prompt to execute arbitrary code, including using NOPs
- **Shellcode:** machine language code, targeted to a particular processor and OS, to perform a task…usually to gain a root or admin level command shell, to execute whatever (arbitrary) code

From paper, p.18: char shellcode[] =

"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"

"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"

"\x80\xe8\xdc\xff\xff\xff/bin/sh"

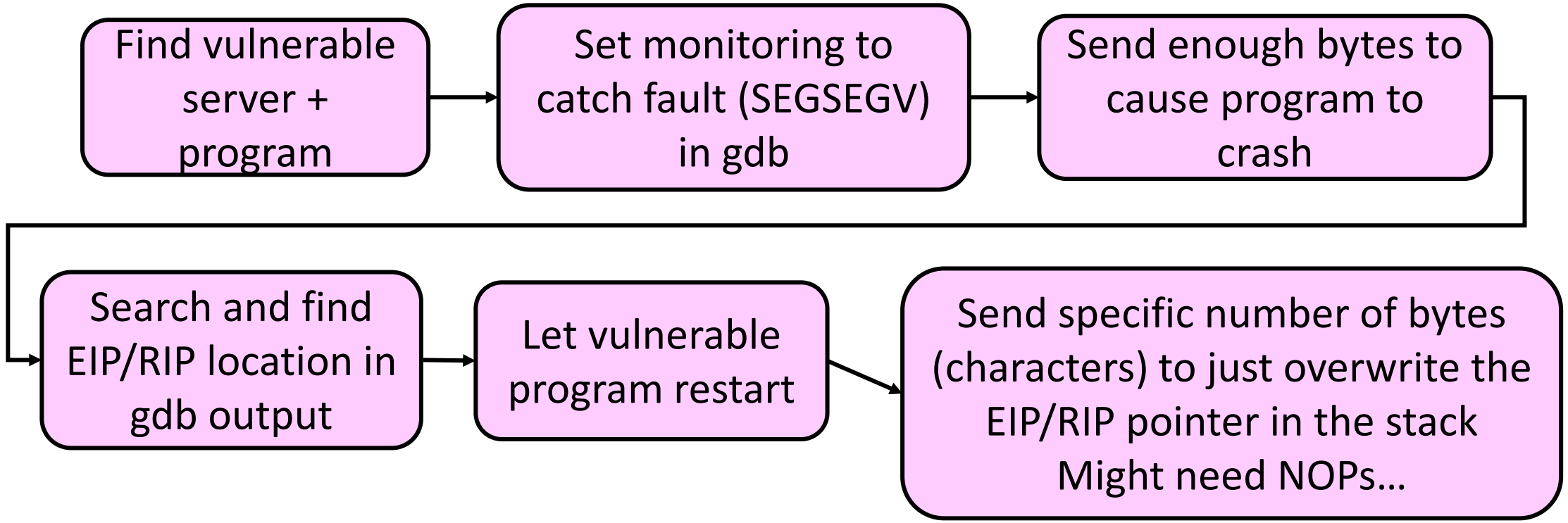
https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

Corrected paper: https://www.eecs.umich.edu/courses/eecs588/static/stack_smashing.pdf

https://www.exploit-db.com/papers/13162

Basic Stack Overflow Hack (Simple)

- You will need GDB… https://sourceware.org/gdb/
- Need to figure out where the stored instruction pointer is stored in the stack, and how to overwrite it without killing the program



Sliding using a NOP sled

- NOP is short for no operation…an operation 0x90 in machine code that does nothing…can be anything that does nothing…originally used to help timing/waits
- If stick shellcode into the stack, I HOPE it goes where I think it does…however, if I pad the area in front of the shellcode with NOPs, the EIP/RIP will just move until it gets to the right spot
- NOP Sled/NOP Ramp is used to steer execution to shellcode or to pad the stack using a jump location to hit the instruction pointer without killing the program…usually due to the fact killing the program alerts the target!

https://www.cs.swarthmore.edu/~chaganti/cs88/f22/lecs/CS88-F22-05-Software-Security-Attacks-Pre-Class.pdf

https://en.wikipedia.org/wiki/NOP_slide

Why all the memory stuff? Shellcode.

Once I get a shell, I can execute arbitrary code…

Shellcode: Payload (usually in binary for a targeted OS/processor) used to execute an exploit for a vulnerability. Shellcode is usually inserted into running processes so that it is executed at the same privilege level as the targeted process. Types:

- **Local:** Confined to hacker directly accessed machine, usually to a single process (initially) and works within that process
- **Remote:** shellcode inserted via network onto a machine on the target network (used to establish C2, see later slides)
- **Download and Execute:** Shellcode hiding in a piece of software downloaded as part of malware
- **Staged:** hacker loads only a loader shellcode with a pointer to the rest of the shellcode. Includes types: Egg Hunt and Omelet

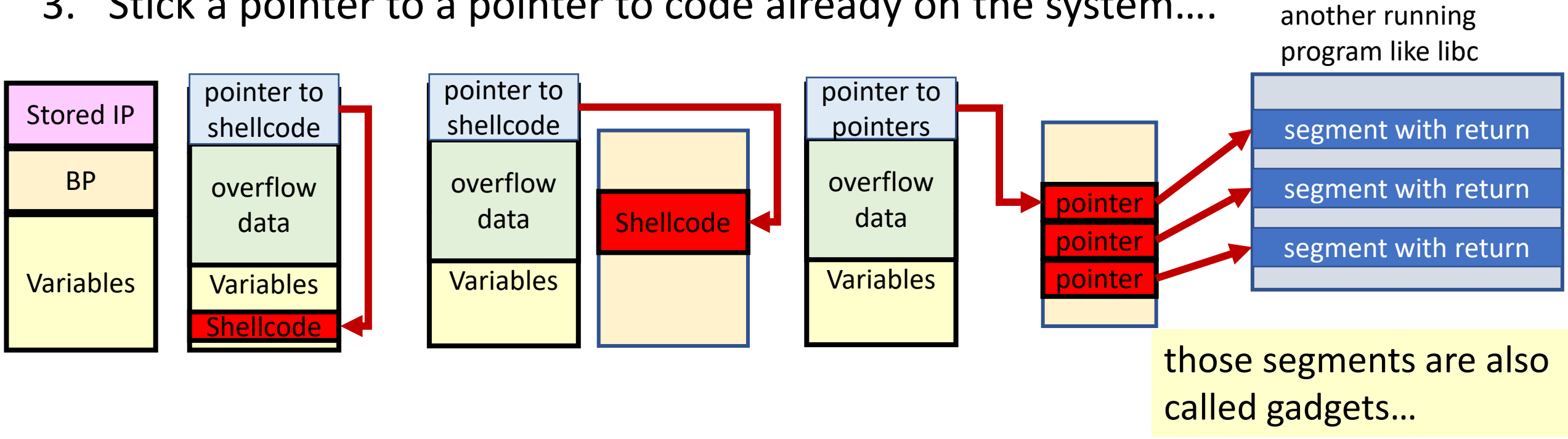
https://en.wikipedia.org/wiki/Shellcode

Shellcode Planting

There are a few options to route a program to the shellcode after an overflow:

- Using an address put into the overwritten stored instruction pointer:

1. Stick the shellcode on the stack somewhere then stick that address into the Stored Instruction Pointer. (command line shellcode uses this)
2. Stick the shellcode somewhere else in memory then point to it
3. Stick a pointer to a pointer to code already on the system….



those segments are also called gadgets…

32-bit exploits vs 64-bit exploits

- Many features in 64-bit systems add security
- **NX-bit,** which marks memory as non-executable
- **64-bit has added registers in assembly, and arguments are passed in those defined registers**!
- Other features:
  - **Address Space Layout Randomization (ASLR)**
  - **Stack Smashing Protector (SSP)/Stack Canaries**

From: “Introduction to x64 Assembly” Intel

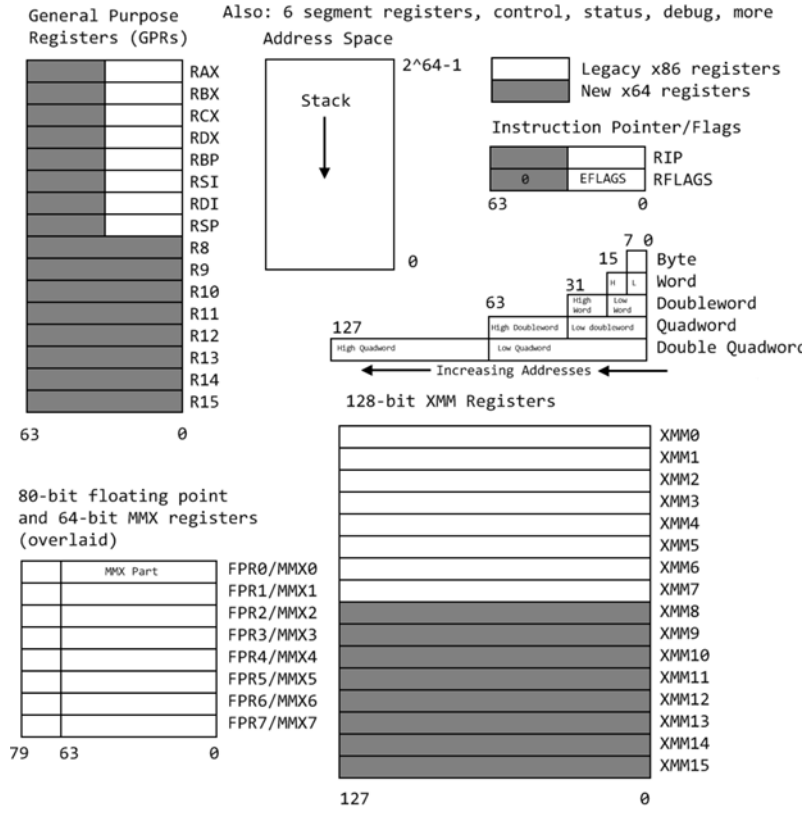


Figure 1 – General Architecture

http://6.s081.scripts.mit.edu/sp18/x86-64-architecture-guide.html

https://www.intel.com/content/dam/develop/external/us/en/documents/introduction-to-x64-assembly-181178.pdf

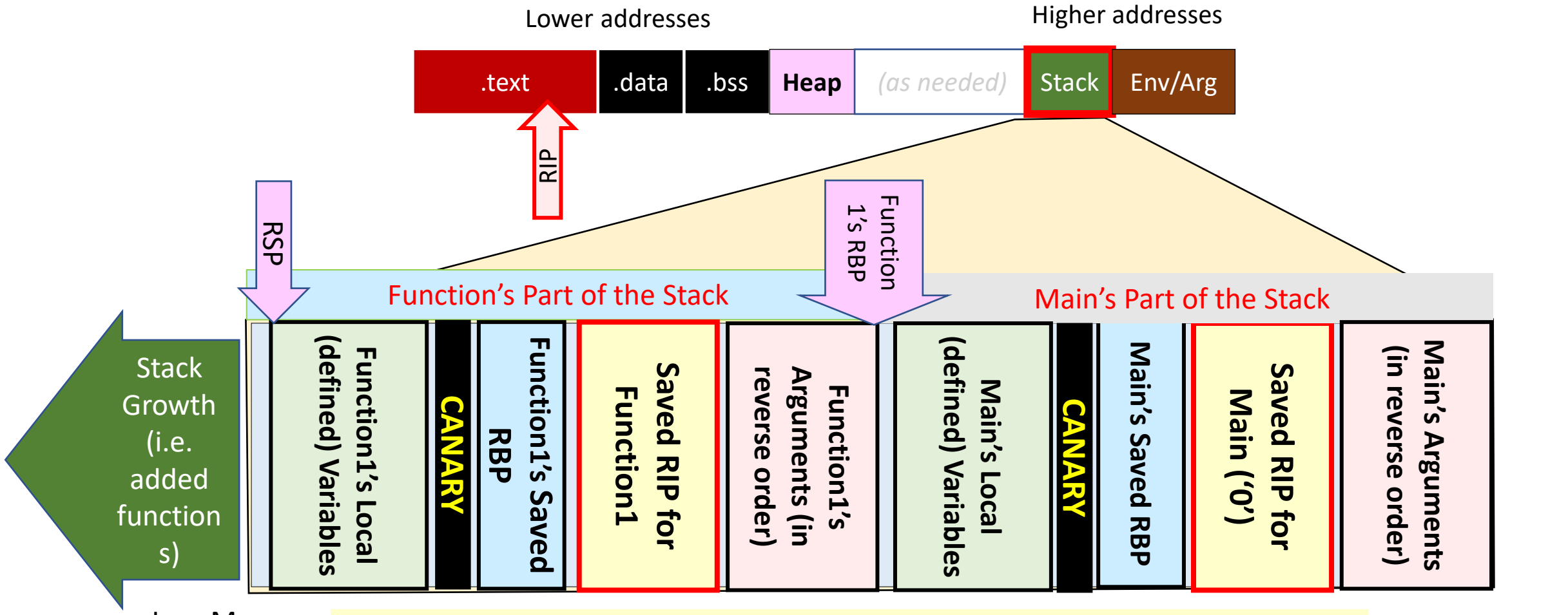
https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/x64-architecture

https://wiki.osdev.org/Stack_Smashing_Protector https://en.wikipedia.org/wiki/X86-64#Virtual_address_space_details

https://software.intel.com/en-us/articles/introduction-to-x64-assembly

https://security.stackexchange.com/questions/169291/x32-vs-x64-reverse-engineering-and-exploit-development

64-bit stack



The order can change, and there can be various locations in the address space for these components!!!!
CAN HAVE NX-BIT MARKING IN SOME AREAS

64-bit Register example … running GHHv6 code

registers

$rax : 0x0
$rbx : 0x007fffffffdef8 → 0x007fffffffe248 → "/home/brycekalel1/GHHv6/ch11/vuln"
$rcx : 0x55
$rdx : 0x14
$rsp : 0x007fffffffdd68 → "faabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabra[...]"
$rbp : 0x6261616562616164 ("daabeaab"?)
$rsi : 0x00000000402010 → "Ultr4S3cr3tP4ssw0rd!"
$rdi : 0x007fffffffdcf0 → "aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama[...]"
$rip : 0x00000000401301 → <auth+171> ret
$r8 : 0x0
$r9 : 0x007ffff7dc9740 → 0x007ffff7dc9740 → [loop detected]
$r10 : 0x007ffff7de6388 → 0x0010001a00001303
$r11 : 0x007ffff7f23490 → 0x41c45a7e01fa8348
$r12 : 0x0
$r13 : 0x007fffffffdf08 → 0x007fffffffe26a → "COLORFGBG=15;0"
$r14 : 0x00000000403e00 → 0x00000000401220 → <__do_global_dtors_aux+0> endbr64
$r15 : 0x007ffff7ffd020 → 0x007ffff7ffe2e0 → 0x0000000000000000
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00

64-bit Stack example…running GHHv6 code

Only 48 bits of memory addresses are used of 64-bits possible by windows/linux on x86_64 (i.e. x64)

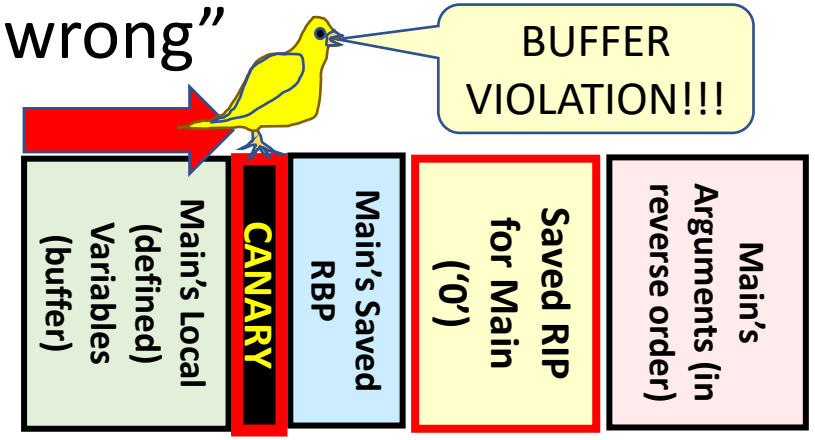
```
stack ——
0x007fffffffdd68|+0x0000: "faabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabra[...]"     ← $rsp
0x007fffffffdd70|+0x0008: "haabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabta[...]"
0x007fffffffdd78|+0x0010: "jaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabva[...]"
0x007fffffffdd80|+0x0018: "laabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxa[...]"
0x007fffffffdd88|+0x0020: "naaboaabpaabqaabraabsaabtaabuaabvaabwaabxaabyaab"
0x007fffffffdd90|+0x0028: "paabqaabraabsaabtaabuaabvaabwaabxaabyaab"
0x007fffffffdd98|+0x0030: "raabsaabtaabuaabvaabwaabxaabyaab"
0x007fffffffdda0|+0x0038: "taabuaabvaabwaabxaabyaab"
```

Virtual memory address (absolute)

Relative memory address In stack frame

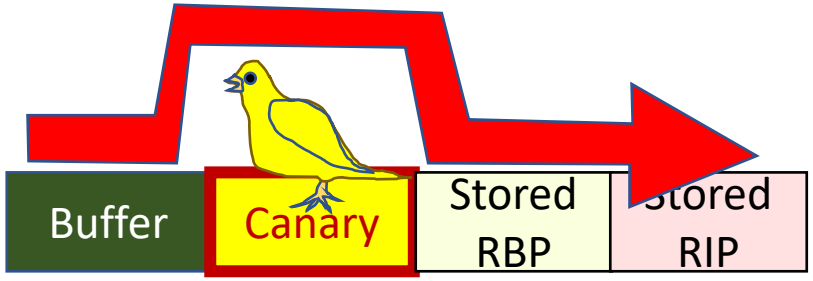
Stack Canaries 101

- **Stack Canaries** are " known values that are placed **between a buffer and control data on the stack** to monitor buffer overflows" via wiki. Canaries are usually a character or field. Canaries, if altered, trigger a handling routine to prevent overflow in secure code
- **Terminator canaries:** uses string terminators such as NULL, LF, CF, FF.
- **Random canaries**: randomly generated codes to hide canaries from attackers
- **Random XOR canaries**: "random canaries that are XOR-scrambled using all or part of the control data. In this way, once the canary or the control data is clobbered, the canary value is wrong"



https://en.wikipedia.org/wiki/Buffer_overflow_protection#A_canary_example
https://en.wikipedia.org/wiki/Buffer_overflow_protection
https://unix.stackexchange.com/questions/453749/what-sets-fs0x28-stack-canary
https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/
https://ctf101.org/binary-exploitation/stack-canaries/

Defeating Stack Canaries



- Essentially, sneak in shellcode execution without tripping the canary values in the stack that stop return
- Trick: Find the canary values then use just enough bytes before the canary gets hit. Store canary value then reinject it, before swapping out Stored IP.

1. Every thread for the same program has the same canary, so find the canary location, i.e. how many bytes can we write before hitting canary location.
2. Find: Iterate values till we get smash error, though intercept error. Iteration can be using strings of characters (e.g., set breakpoint that sets RSI before canary triggers)
3. Use pattern search to look for $rsi value…the offset here this is where the canary is located relative to the string buffer
4. Make exploit do a try that adds $rsi offset + canary value+ padding + ROP chain in exploit

SI/ESI/RSI: *Source index* for string operations

https://stackoverflow.com/questions/47047386/how-to-pass-memory-address-of-a-location-in-stack-from-assembly

https://reverseengineering.stackexchange.com/questions/28059/where-stack-canary-is-located

https://github.com/GrayHatHacking/GHHv6/blob/main/ch11/exploit2.py

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

https://ctf101.org/binary-exploitation/stack-canaries/

RIP= Relative Instruction Pointer
SFP= Saved Frame Pointer=saved RBP

Text p.228-230

Stack Canary Payload Code (example)

payload = b"A"*72 //all the ‘a’s

payload += leak_bytes(payload, "Canary") //canary bypass

payload += p64(0xBADC0FFEE0DDF00D) #SFP //more stuff to get to ROP pointers in stored RIP

payload += (shellcode or pointers to ROP)

Stack Smashing Protector (SSP) (a type of Canary)

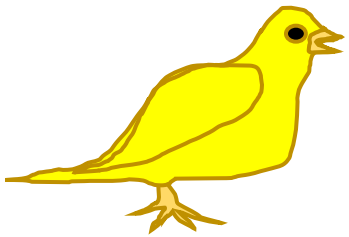
- **Stack Smashing Protector (SSP):** Compiler option to detect stack overrun
- Uses libssp library https://github.com/gcc-mirror/gcc/tree/master/libssp

```
void foo(const char* str)
{
        char buffer[16];
        strcpy(buffer, str);
}
```

gcc **-fstack-protector** *program_name*

```
extern uintptr_t __stack_chk_guard;
noreturn void __stack_chk_fail(void);
void foo(const char* str)
{
        uintptr_t canary = __stack_chk_guard;
        char buffer[16];
        strcpy(buffer, str);
        if ( (canary = canary ^ __stack_chk_guard) != 0 )
                __stack_chk_fail();
}
```

Examples from:
https://wiki.osdev.org/Stack_Smashing_Protector



If stack is overrun (and hits canary), runtime error = “stack smashing detected”

Non-eXecutable (NX) Stack

- Part of memory marked by the NX bit (SW or HW) or similar method to be non-executable to prevent remote code execution
- Called **"Data Execution Prevention" (DEP)** in Windows
- **ExecShield** is one implementation in RedHat Linux
- Concept is to prevent exploits from using code in various blocks in memory
- Work around: ROP!

https://en.wikipedia.org/wiki/Exec_Shield

https://en.wikipedia.org/wiki/Executable_space_protection

https://www.exploit-db.com/docs/english/16030-non-executable-stack-arm-exploitation.pdf

https://en.wikipedia.org/wiki/NX_bit

ROP (Return Oriented Programming)

- **Basic Concept:** Return-oriented programming uses **control of the call stack** to **indirectly execute** machine instructions or groups of machine instructions **(gadgets)** immediately **prior to the return instruction (ret)** in subroutines (functions) within the existing program code (loosely via Wiki)
  - **Some gadgets use JMP or CALL**
- **Return-into-library technique** uses libc code already in memory in lieu of custom shellcode https://ctf101.org/binary-exploitation/return-oriented-programming/
- **Borrowed code chunks:** attack that uses chunks of library functions, instead of entire functions themselves, to exploit buffer overrun https://www.ired.team/offensive-security/code-injection-process-injection/binary-exploitation/rop-chaining-return-oriented-programming
- **ROP Chain:** Sequence of ROP exploits to execute code

https://github.com/JonathanSalwan/ROPgadget https://pypi.org/project/ROPGadget/

ROP Example using libc: https://www.youtube.com/watch?v=cZKV_LZOPug

https://en.wikipedia.org/wiki/Return_statement https://www.cs.cmu.edu/~rdriley/487/labs/lab03.html

https://docs.oracle.com/cd/E19455-01/806-3773/instructionset-67/index.html

https://montcs.bloomu.edu/Information/LowLevel/Assembly/assembly-tutorial.html

https://resources.infosecinstitute.com/topic/return-oriented-programming-rop-attacks/

https://en.wikipedia.org/wiki/Return-oriented_programming

ROP vs JOP vs SROP

- **ROP(Return Oriented Programming):** Executes code ALREADY PRESENT in executable memory (at gadgets)
  - Works in the presence of memory protections!
- **JOP (Jump Oriented Programming):** Inserts a pointer into executable memory to code that is outside the stream of execution…older hack method
- **SROP: Sigreturn Orientated programming (SROP):** exploit that allows an attacker to control the entire state of the CPU and allows an attacker to execute code in presence of security measures such as non-executable memory and code signing.

https://en.wikipedia.org/wiki/Sigreturn-oriented_programming

https://en.wikipedia.org/wiki/Return-oriented_programming https://man7.org/linux/man-pages/man2/sigreturn.2.html

https://security.stackexchange.com/questions/201196/concept-of-jump-oriented-programming-jop

https://developer.arm.com/documentation/102433/0100/Jump-oriented-programming

https://resources.infosecinstitute.com/topic/return-oriented-programming-rop-attacks/

Ropper (.py, and Kali command)

- Display information about binary files in different file formats, search for gadgets to build ROP(*Return Oriented Programming)* chains
- In short, disassembles an executable and can be used to search for gadgets to find a spot to remote execute code in (normal) executable libraries
- Note: install the Capstone engine for disassembly, along with pyvex first

https://www.ibm.com/docs/en/zos/2.3.0?topic=functions-mprotect-set-protection-memory-mapping

http://www.capstone-engine.org/

https://scoding.de/ropper/

https://gitlab.com/kalilinux/packages/ropper

https://www.kali.org/tools/ropper/

https://github.com/sashs/Ropper

https://github.com/angr/pyvex

One_gadget and execve

- Gadgets are spots that can be used to insert code and open a shell to remotely execute code, they end in 'RETN'.
- One_gadget is a tool to jump execution to a particular spot, especially for Remote Code Execution (RCE). i.e. execve("/bin/sh", NULL, NULL);
  - Execve = execute program in shell
- Running One_Gadet readies the system to open the shell by finding spots in the code (gadets) in libc.

EXAMPLES: https://dzone.com/articles/a-ctf-example-shows-you-the-easy-and-powerful-one

https://github.com/david942j/one_gadget https://pypi.org/project/one-gadget/

https://www.ibm.com/docs/en/i/7.3?topic=functions-scanf-read-data

https://www.unix.com/man-page/osx/1/vmmap/

https://linux.die.net/man/2/execve

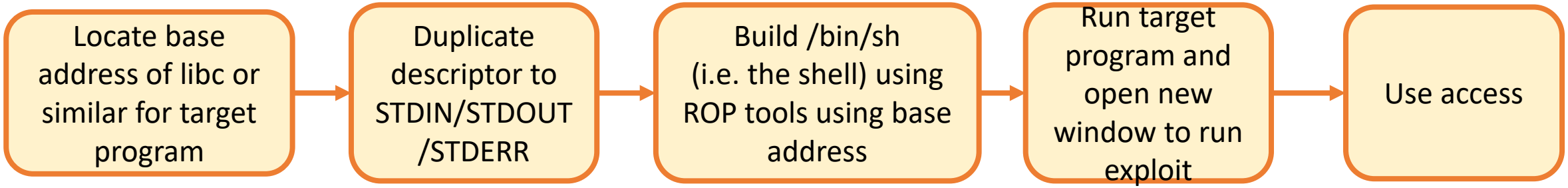
Ropper redux

Ropper looks for gadgets in a target OS or program instance, in this case the VM kernel version of Linux (vmlinux an ELF executable which it disassembles)

```
└─$ ropper --file ~/GHHv6/ch12/vmlinux --console
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
(vmlinux/ELF/x86_64)>
(vmlinux/ELF/x86_64)> search pop rdi
[INFO] Searching for gadgets: pop rdi
[INFO] File: ~/GHHv6/ch12/vmlinux
0xffffffff810baf61: pop rdi; adc byte ptr [rax - 0x75], cl; cmp byte ptr [rbp - 0x75], cl; sbb byte ptr [rax + 0x39], cl; ret;
0xffffffff818812f7: pop rdi; adc byte ptr [rdx + 0x64541568], dl; ret;
(LOTS MORE LINES)
0xffffffff818af975: pop rdi; adc dword ptr [rbp + 0x62], esp; mov bl, 0x6b; adc dl, ch; call qword ptr [rbp + 0x64f73a70]; ret;
(LOTS MORE LINES)
0xffffffff811ad2ec: pop rdi; ret;
(lots more lines)
(vmlinux/ELF/x86_64)> search (whatever gadget you want)
(vmlinux/ELF/x86_64)> quit
```

Pointer to add to ROP chain; Add together to make shellcode (Exploit)

Bypassing non-executable (NX) stack with ROP



https://www.kali.org/tools/

https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/VMPages.html

https://en.wikipedia.org/wiki/Glibc

https://github.com/GrayHatHacking/GHHv6/blob/main/ch11/exploit1.py

https://docs.pwntools.com/en/stable/rop/rop.html

https://linux.die.net/man/8/execstack

https://github.com/Wenzel/linux-sysinternals

ROP Example using libc: https://www.youtube.com/watch?v=cZKV_LZOPug

Text p. 221 to 225

Bypass NX with ROP Example

```
$ gcc vuln.c -o vuln_nx|readelf -l vuln_nx|grep -A1 GNU_STACK
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
vuln.c: In function ‘auth’:
vuln.c:24:5: warning: ‘read’ writing 512 bytes into a region of size 64 overflows the
destination [-Wstringop-overflow=]
   24 |     read(connfd, buf, 512);
      |     ^~~~~~~~~~~~~~~~~~~~~~
(LOTS OF WARNINGS)
┌──(brycekalel1㉿kali1)-[~/GHHv6/ch11]
└─$ gcc -z execstack vuln.c -o vuln_nx && readelf -l vuln_nx|grep -A1 GNU_STACK
vuln.c: In function ‘auth’:
vuln.c:24:5: warning: ‘read’ writing 512 bytes into a region of size 64 overflows the
destination [-Wstringop-overflow=]
 (lots of warnings)
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RWE    0x10
```

- ***-z execstack***: disables non-executable stack protection…therefore allowing stack elements to be shellcode
- In top compile ELF is set to Read and Write (RW) vs. RW and Execute in bottom (RWE)

run using GHHv6 code…

Bypass NX with ROP Example

```
└─$ gdb ./vuln -q -ex "set follow-fork-mode child" -ex "r"
GEF for linux ready, type `gef' to start, `gef config' to configure
90 commands loaded and 5 functions added for GDB 12.1 in 0.00ms using Python engine 3.11
Reading symbols from ./vuln...
(No debugging symbols found in ./vuln)
Starting program: ~/GHHv6/ch11/vuln
[*] Failed to find objfile or not a valid file format: [Errno 2] No such file or directory: 'system-supplied DSO at 0x7ffff7fc9000'
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Listening on 127.0.0.1:4446
^C
Program received signal SIGINT, Interrupt.
0x00007ffff7ed6460 in __libc_accept (fd=0x3, addr=..., len=0x7fffffffddac) at ../sysdeps/unix/sysv/linux/accept.c:26
26      ../sysdeps/unix/sysv/linux/accept.c: No such file or directory
```

- Running vuln like so puts it in a thread in background, on local port **4446** (127.0.0.1 = local to box)
- Runs until interrupted by ^C or killed.
- Setting vuln up so it can be hacked by another terminal window…

run using GHHv6 code…

Bypass NX with ROP Example

.
[Legend: Modified register | Code | Heap | Stack | String]
———————— registers ——
$rax : 0xffffffffffffe00
$rbx : 0x007fffffffdef8 → 0x007fffffffe248 → "/home/brycekalel1/GHHv6/ch11/vuln"
$rsp : 0x007fffffffdd68 → 0x000000004014dc → <**main+474**> mov DWORD PTR [rbp-0x8], eax
$rbp : 0x007fffffffdde0 → 0x0000000000000001
$rsi : 0x007fffffffddb0 → 0x0000000000000000
$rdi : 0x3
$rip : 0x007ffff7ed6460 → 0x5877fffff0003d48 ("H="?)
....
———————————— stack ——
0x007fffffffdd68|+0x0000: 0x000000004014dc → <main+474> mov DWORD PTR [rbp-0x8], eax ← $rsp
0x007fffffffdd70|+0x0008: 0x0000000000000000
….

run using GHHv6 code…

Bypass NX with ROP Example

```
gef➤  vmmap libc
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x007ffff7dcc000 0x007ffff7df2000 0x00000000000000 r-- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7df2000 0x007ffff7f47000 0x00000000026000 r-x /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7f47000 0x007ffff7f9a000 0x0000000017b000 r-- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7f9a000 0x007ffff7f9e000 0x000000001ce000 r-- /usr/lib/x86_64-linux-gnu/libc.so.6
0x007ffff7f9e000 0x007ffff7fa0000 0x000000001d2000 rw- /usr/lib/x86_64-linux-gnu/libc.so.6
```

- This is where I will be grabbing my ROP gadgets

run using GHHv6 code…

ASLR (Address Space Layout Randomization)

- ASLR :”randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries” via Wiki.
- Hackers need to find the positions of all areas they want to attack or use. Makes a ROP attack harder by moving the gadgets….
- In short, the allocated virtual space is randomly used for parts of the program

https://www.tomsguide.com/us/aslr-definition,news-18456.html

https://www.cisa.gov/uscert/ncas/current-activity/2017/11/20/Windows-ASLR-Vulnerability

https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/

https://ctf101.org/binary-exploitation/address-space-layout-randomization/

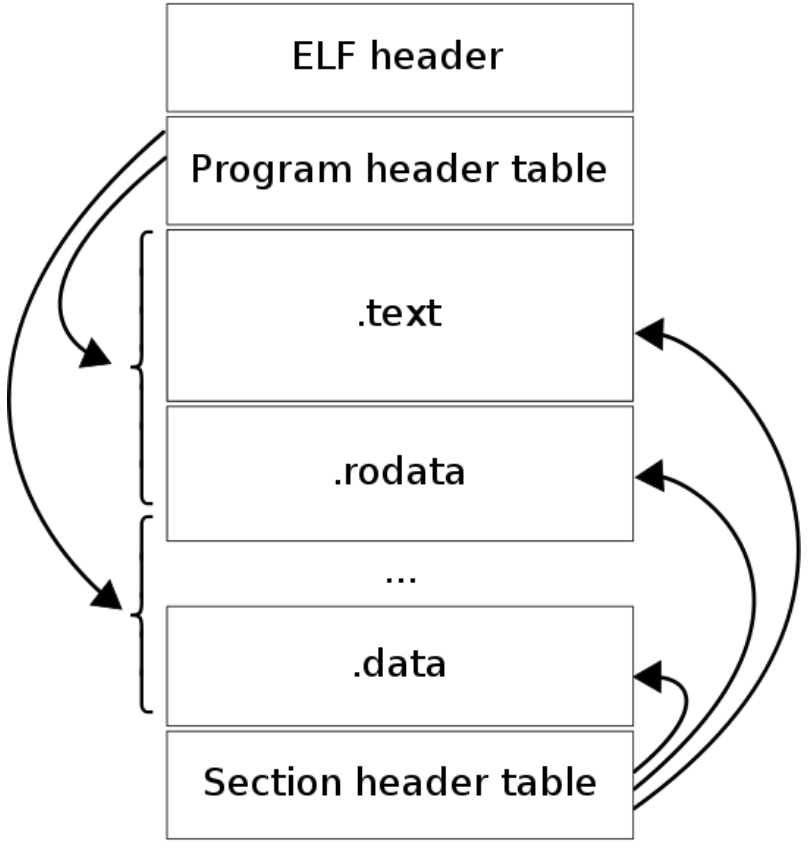
https://learn.microsoft.com/en-us/cpp/build/reference/dynamicbase-use-address-space-layout-randomization?view=msvc-170

https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization

https://en.wikipedia.org/wiki/Address_space_layout_randomization

ELF files Review

- ELF: *Executable and Linkable Format* : Standardized defined format of a binary, ready to run, program file (or part of an object file). ELF File=formatted using ELF.
- Has header and file data. Static (self contained) and Dynamic (needs external data to run) types.
- ELF Header: Starts in binary with “7f 45 4c 46”. Contains all the information to execute the file in runtime memory, including bit type (32 bit, 64 bit), OS, processor type, Entry Points and offsets, and string table (see Class 3 on memory)
- File Data:
  - Program Headers or Segments: maps binary to virtual memory space
  - Section Headers: Defines sections in the executable file
  - Data: the file data



https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
https://man.archlinux.org/man/elf.5.en
https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/

Getting GOT and PLT 101

- **GOT: Global Offset Table:** memory used to enable an ELF (formatted) program's executables and shared libraries to run, independent of the memory address where the program's code or data is loaded at runtime….i.e. random memory assignment to make code harder to hack.
  - Maps symbols (human-readable notes, atoms) to their corresponding absolute memory addresses
  - Addresses to libc function
  - Position Independent Code (PIC)
  - Position Independent Executables (PIE)
- **PLT: Procedure Linkage Table:** links dynamic objects in programs to absolute locations in memory. Read only section of ELF made at compile time.

https://www.man7.org/linux/man-pages/man7/libc.7.html

https://en.wikipedia.org/wiki/Position-independent_code

https://en.wikipedia.org/wiki/Memory_address

https://en.wikipedia.org/wiki/Symbol_(programming)

https://en.wikipedia.org/wiki/Global_Offset_Table

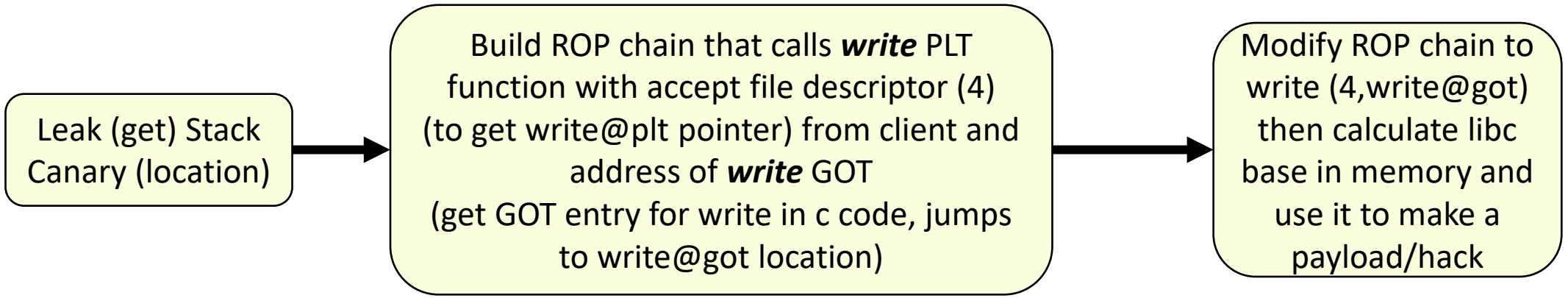
https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-1235.html

https://www.intel.com/content/www/us/en/docs/programmable/683620/current/procedure-linkage-table.html

https://www.redhat.com/en/blog/position-independent-executables-pie

ASLR bypass (with Information Leak)

- Combination of ROP Chain, Bypass of NX , and Defeat Stack Canary exploits



https://www.man7.org/linux/man-pages/man2/mmap.2.html

https://www.fortinet.com/blog/threat-research/tutorial-of-arm-stack-overflow-exploit-defeating-aslr-with-ret2plt

https://github.com/GrayHatHacking/GHHv6/blob/main/ch11/exploit3-v2.py

https://github.com/GrayHatHacking/GHHv6/blob/main/ch11/exploit3.py

See also Grey Hat Hacking p.228-230

PIE (Position Independent Executables)/ PIC (Position-Independent Code)

- Code that operates properly regardless of where it is in memory
- "PIE binary and all of its dependencies are loaded into random locations within virtual memory **each time the application is executed**." (via Redhat) to prevent exploits
- Addresses in code complied as PIE are RELATIVE versus absolute in dynamic compiled code
- Helps stop ROP attacks…hard to figure out where to return and put code
- gcc -fstack-protector…
- **In short, every time I run code, I have to re-find the gadgets**….

https://inst.eecs.berkeley.edu/~cs164/fa11/ia32-refs/ia32-chapter-seven.pdf

https://en.wikipedia.org/wiki/Position-independent_code

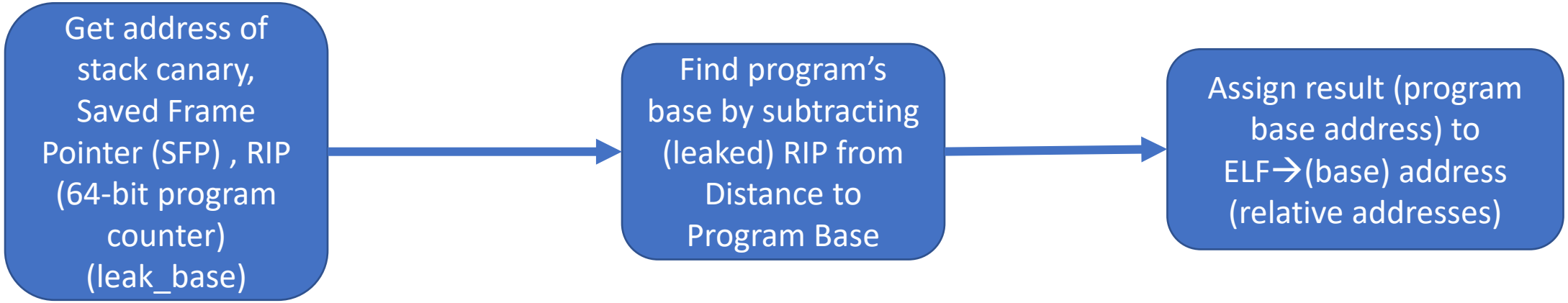
https://www.redhat.com/en/blog/position-independent-executables-pie

https://stackoverflow.com/questions/2463150/what-is-the-fpie-option-for-position-independent-executables-in-gcc-and-ld

PIE bypass with Information Leak

- PIE bypass works by finding the RELATIVE values of key pointers, then exploit the calculated locations given initial base values

•Via Wiki x86:
•BP/EBP/RBP: Stack base pointer for holding the address of the current stack frame.
•IP/EIP/RIP: Instruction pointer. Holds the program counter, the address of next instruction



https://stackoverflow.com/questions/45112182/why-is-saved-frame-pointer-present-in-a-stack-frame
https://stackoverflow.com/questions/27429026/understanding-how-eip-rip-register-works
https://github.com/GrayHatHacking/GHHv6/blob/main/ch11/exploit3.py
https://github.com/GrayHatHacking/GHHv6/blob/main/ch11/exploit4.py
https://github.com/ir0nstone/pwn-notes/blob/master/types/stack/pie/pie-bypass.md

see also Grey Hat Hacking p.230-232

Stack Overflow from User--Ring 3...in 1 slide

PWNTOOLS

Vulnerable program is written, compiled

Compiler warnings ignored, program is fielded

Vulnerability that allows buffer overflow discovered, likely a CWE/CVE

Buffer overflow uses NOPs and crashes with Segmentation Fault

Buffer overflows used to locate stored instruction pointer, an offset from buffer

Buffer + NOPs to full offset + shellcode inserted to redirect instruction pointer

WORKS ?

YES

METASPLOIT

NO

Escalate permissions using **$** sudo, then add bot to Command-and-Control Network and use it

NOTE:>>Hacking kernel gets us ROOT (**#**)

NX Bits? Replace Shellcode with pointers to gadgets (a chain) in existing executable code like libc (Return Oriented Programming, ROP)

ROPPER

Stack Canary alerts on attempt to overwrite pointers?
Locate Stack Canary, copy its offset location and value, then add the value at the right spot in NOPs before ROP Chain

PWNTOOLS

Start Over

NO

WORKS ?

YES

Addresses of stack elements or gadgets is randomized (Address Space Layout Randomization)? Use ELF file key parts to locate stack elements and gadgets then fix offsets and ROP Chain pointers

Create buffer overflow, add fixed NOP offsets with canary inserted, then fixed ROP chain

Conclusion, do they still exist? Yes

- It is an arms race to stop exploits in OS's…some have gotten much harder to exploit!
- Just an intro to what happens if you don't take care when making code and deploy it with bad coding practices…or vulnerabilities
- Programs use the stack to store allocated variables, and if variables are not checked or controlled the overflow takes advantage of how code uses memory
- Most common hacks: overflow, credential stealing, XSS, others
- OS's use processor features to try to prevent overflows: Non-executable memory, stack canaries and also to prevent jumping code to get shells: address-space layout randomization, position independent execution

Basic Preso Data

Saint Louis Linux Users Group (STLLUG)

Thursday, April 20th, 2023
- from 6:30PM till 9:00PM Central Daylight Time

Saint Louis MO - STL Linux Users Group (sluug.org)
TOPIC: Stack Based Attacks in Linux

Presenter: Bryce Meyer

Stack overflows in linux: tools, methods, and vulnerabilities
Stack based attacks against 32bit and 64bit linux has become and arms race between methods to exploit software missteps and operating system and x64 methods to stop them.
Will cover:
•how holes get in,
•compiler switches,
•an intro to return oriented programming,
•shellcode, and
•mitigation like stack canaries.

Intel Processor Registers (x86 Architecture)

- Store and keep track of segments in a process temporarily
- Can be a reference to a real chip location or a virtual location on a virtual processor (or mostly: in a virtual memory location at runtime)
- General Registers: to manipulate data, first part of a memory address
- Segment Registers: holds first part of memory address, pointers to code, stack, and extra data segments
- Offset Registers: to hold an offset from a segment register: top of the stack frame, bottom of the stack frame, destination data offsets (pointers)
- Special Registers: used only by the CPU
- BETWEEN VIRTUAL MEMORY AND PHYSICAL MEMORY IS A MAP

https://en.wikipedia.org/wiki/X86_virtualization

https://en.wikipedia.org/wiki/X86

See also table 2-4 in text

Threat Databases and Uses

Source	Time Frame	Tools	Notes
• **SEI CERT Coding Standards for C, C++, Java, Perl, and Android** • **IEEE standards (e.g., ISO/IEC 9899:2011)+**	BEFORE USE: • During coding in development environment • Part of DevSecOps via static analysis	• Clang, other modern development environments (Eclipse, Visual), and static analysis tools • Manual review	• Standards from IEEE/IEC/ISO informed secure coding practices and examples in the SEI CERT Coding Standards; secure coding should part of an overall code quality process • Many map to CWEs
• **MITRE CWE** • **OWASP Top 10** • **CWE/SANS Top 25**	BEFORE USE: • During coding in development environment • Part of DevSecOps via static analysis **IN UPDATE/PATCHING**	• Static analysis tools • Manual review	• Weaknesses can lead to vulnerabilities (CWEs can be mapped to CVEs if a vulnerability is found for a weakness) • Top lists are the most likely weaknesses for exploit
• **MITRE CVE, which feeds ATT&CK Techniques** • **MITRE CAPEC** • **NIST NVD** • **NSA Top 25**	Before use in field: • To patch in-use software and systems • As checks as part of DevSecOps process • When upgrades and revisions are required to fix known vulnerabilities in fielded code and systems.	• Static and dynamic analysis tools • Manual review • Custom comparison: release to vulnerability • ATT&CK focuses on system level, CAPEC on application and software level	• Vulnerabilities can be exploited by various hacking tools • NVD maps CVEs with known exploits; the worst vulnerabilities become NSA Top 25 risks • CVEs can have exploitation techniques listed in ATT&CK and in CAPEC

Arguments in Registers: 64-bit uses registers to pass arguments into a function….

- RDI gets arg 1, RSI gets arg 2, RDX gets arg 3, RCX gets arg4, R8 gets arg 5, R9 gets arg 6
- Arguments 7 and above are pushed on to the stack.
- (in 32-bit the args are pushed onto the stack in reverse order)

32-bit:

```
Function:
pushl   %ebp
movl    %esp, %ebp

Main:
movl    (%eax), %eax
pushl   %edx
pushl   %eax
call    greeting
```

64-bit:

```
Function:
pushq   %rbp
movq    %rsp, %rbp

Main:
movq    (%rax), %rax
movq    %rdx, %rsi
movq    %rax, %rdi
call    greeting
```

https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/linux-x64-calling-convention-stack-frame