CWL
CyberWarFare Labs

CERTIFIED EXPLOIT DEVELOPMENT PROFESSIONAL (CEDP)

Labs

- Linux
  - MicroHttpServer - CVE-2023-50965
    - Vanilla Stack Overflow
    - Stack Overflow + NX bypass (ret2libc)
    - Stack Overflow + NX bypass (rop chain)
  - Custom Binary
    - Stack Overflow + Format String BUG
      - Canary, NX, PIE, ASLR

Labs

- Windows (Win32)
  - General Device Manager
    - SEH Overflow - no DEP
  - Easy File Share
    - SEH Overflow + DEP Bypass + ASLR bypass
      - Non-aslr-enabled module

Lab Setup

Intel x86 Insights

CPU Registers

- Small, high speed storage locations within the CPU
- Stores data temporarily and controls CPU operations
  - Handling interrupts, memory operations
- Essential for storing and manipulating data and executing instructions

General Purpose Registers

32-Bit	31 ... 16	15 ... 8	7 ... 0	16-Bit
EAX		AH	AL	AX
EBX		BH	BL	BX
ECX		CH	CL	CX
EDX		DH	DL	DX
ESP		SP		
EBP		BP		
ESI		SI		
EDI		DI		

General Purpose Registers

- There are eight General purpose registers
  - eax -> Accumulator
    - Generally used in arithmetic and logical operations. Also, most of the time it stores function return value.
  - ebx -> Base Register
    - Commonly holds base address of certain memory locations such as base address of the array.
    - Also helps calculate effective addresses for data access

General Purpose Registers

- ..
  - ecx -> Counter
    - Generally used as a counter, for instance counting number of iteration is loop, counting length of the string.
  - edx -> Data register
    - Often use in conjunction with accumulator to store or handle 64-bit values in certain operations
      - edx:eax

General Purpose Registers

- ..
  - esp -> Stack Pointer
    - Always points at the top of the stack.
  - ebp -> Stack Base Pointer
    - It’s a stack base pointer for current function stack frame. Also, use in accessing local variables and parameters.

General Purpose Registers

- ..
  - esi -> source index
    - Often used as the pointer to the source address when copying a block of data.
  - edi -> destination index
    - Often used as the pointer to the destination address when copying a block of data.

Special Purpose Registers

- eip -> Instruction Pointer
  - It serves as a program counter, indicating the memory address of the next instruction to be executed by the CPU.

Basic x86 Instructions set

- Data Movement Instructions
  - Mov, lea, push, pop, xchg
    - mov eax, ebx -> register to register
    - mov eax, 0x10 -> immediate to register
    - mov [eax], 0x10 -> immediate to memory
    - mov eax, [eax+0x10] -> memory to register

Basic x86 Instructions set

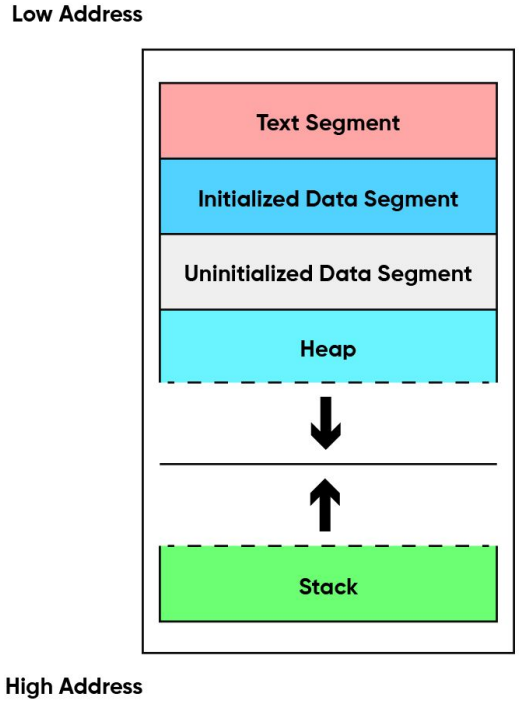
- Arithmetic Instructions
  - add, sub, mul, imul, div, idiv, inc, dec, neg, cmp
    - add eax, ebx -> adding 2 registers
    - add eax, 0x10 -> adding immediate with register value
    - mov eax, [ecx] -> adding memory value with register

Basic x86 Instructions set

- Logical Instructions
  - and, or, xor
- Control Transfer Instructions
  - jz, jnz, jl, jle, jmp, call, loop, ret
- Special Instructions
  - int

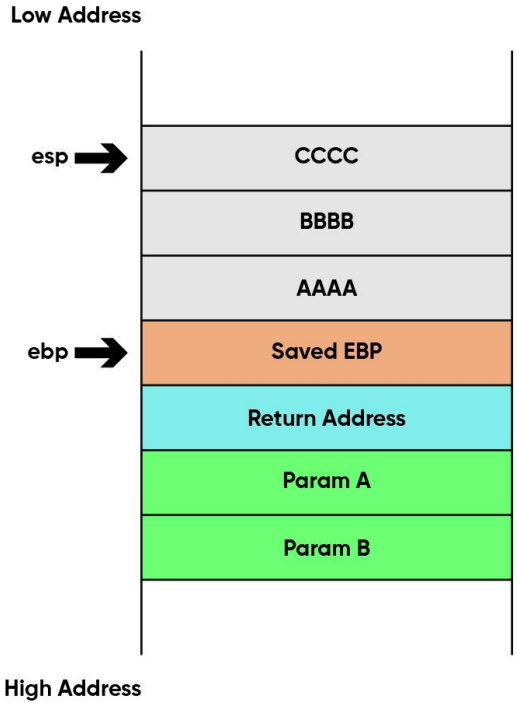
Memory Layout

- Typical memory Layout consist of
  - Stack
  - Heap
  - Uninitialized Data Segment
  - Initialized Data Segment
  - Text/Code Segment



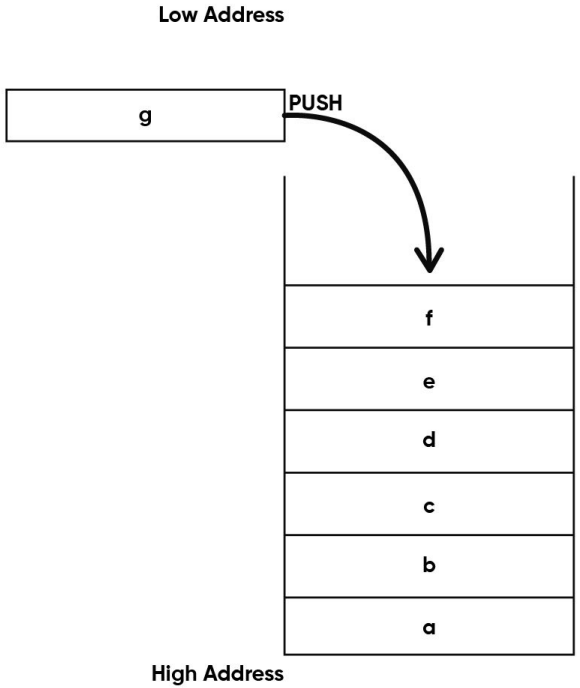
STACK

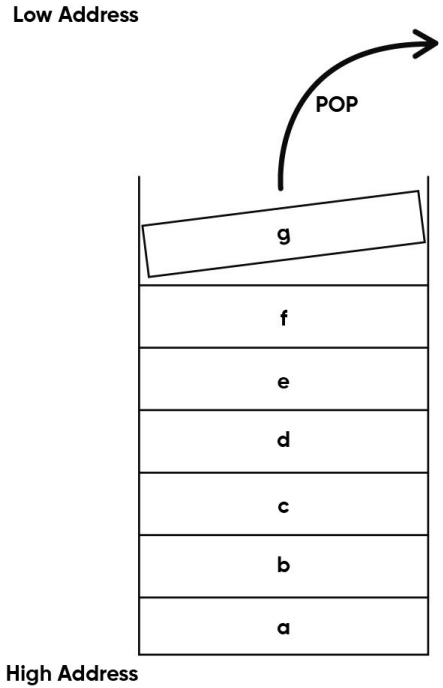
- Block of memory that holds temporary data
  - Operates in LIFO (Last In, First Out) principal
- Grows and shrinks dynamically during program execution
  - Grows towards the lower address (higher -> lower)
- Each function call creates the stack frame, containing parameters, local variables and return address



STACK Operations

- PUSH
  - Adds data onto the stack
- POP
  - Removes data from the stack



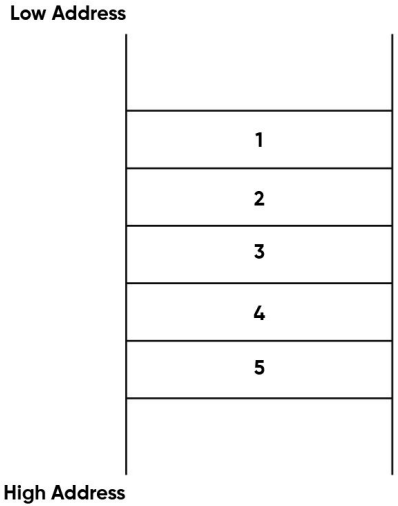


Calling Conventions

- Defines how functions should be called in the program such as:
  - how parameters are passed
    - X86 architecture follows right-to-left parameter passing scheme
  - Handling return address
  - Managing stack space
- cdecl is default calling convention for c and c++ for x86 architecture

Parameters -x86

- Parameters are passed from right-to-left into the stack
  - int abc(1,2,3,4,5)



Function Prologue

- Setup the stack frame
  - Saves old base pointer & set new base pointer
- Allocate space for local variables

```
00F21040    55              push ebp
00F21041    8BEC            mov ebp,esp
00F21043    83EC 18         sub esp,18
00F21046    A1 0030F200     mov eax,dword ptr ds:[<___security_cookie>]
00F2104B    33C5            xor eax,ebp
00F2104D    8945 FC         mov dword ptr ss:[ebp-4],eax
```

Function epilogue

- Restores the stack frame
  - Clean up the stack
- Returns to the caller

```
00F21098    33CD            xor ecx,ebp
00F2109A    33C0            xor eax,eax
00F2109C    E8 04000000     call <stack.@__security_check_cookie@4>
00F210A1    8BE5            mov esp,ebp
00F210A3    5D              pop ebp
00F210A4    C3              ret
```

Demo

Debuggers, Disassemblers & Debugging

Debugger

- Tool used for examining the running program
  - Allows to analyze & troubleshoot the program
  - With features like:
    - Breakpoints
    - Visibility on variables, registers, stack etc.
    - Controlling the flow of execution
    - Memory dumps
    - Registers

Debugger

- Common debuggers
  - GDB
  - X64dbg
  - Immunity debugger
  - windbg

Disassembler

- Tool that converts the machine code instructions into human readable form (assembly)
- Helps in analyzing the compiled binary code
  - Generally, provides the blueprint for the program
- Common disassemblers include IDA Pro, Ghidra, Binary Ninja, Hopper

Debugging

- Process of analyzing the binary making use of both disassembler & debugger
  - aids in identifying, understanding & fixing the problems or bugs in the software

- This process includes:
  - Manual code inspection, dynamic analysis with both debugger & disassemblers, also, automated testing

Introduction to Stack Overflow

Stack Overflow

- A flaw in software that occurs when more data is written to a buffer on the stack than it can hold,
  - resulting in the overwriting of adjacent memory, including other variables and the return address.

- If exploited correctly and all required conditions are met
  - attacker can overwrite the EIP (Instruction Pointer) register
    - potentially redirecting program execution to malicious code.

Stack Overflow

- If overflow doesn't meet all the required conditions for control flow hijacking
  - it often results in a program crash
    - leading to a Denial of Service (DoS).

Stack Overflow Condition

- Stack overflow occurs when certain condition meets
  - Unchecked Buffer Size
  - Buffer Copy without Checking Size of Input
  - Insufficient Bound Checking

Unchecked Buffer Size

- Reads the user input into a fixed-size buffer
  - doesn't check if input exceeds the the buffer size

```
// Unchecked buffer size
void read_input() {
    char buffer[10];
    gets(buffer); // does not check the length of input
}
```

Insufficient bound checking

- Copies data to buffer but doesn't check if the buffer can hold the data being copied

```
// Insufficient Bound Checking
void copy_data(int *source, int length) {
    int buffer[10];
    for (int i = 0; i < length; i++) {
        buffer[i] = source[i]; // No bounds checking on buffer
    }
}
```

Buffer Copy without Checking Size of Input

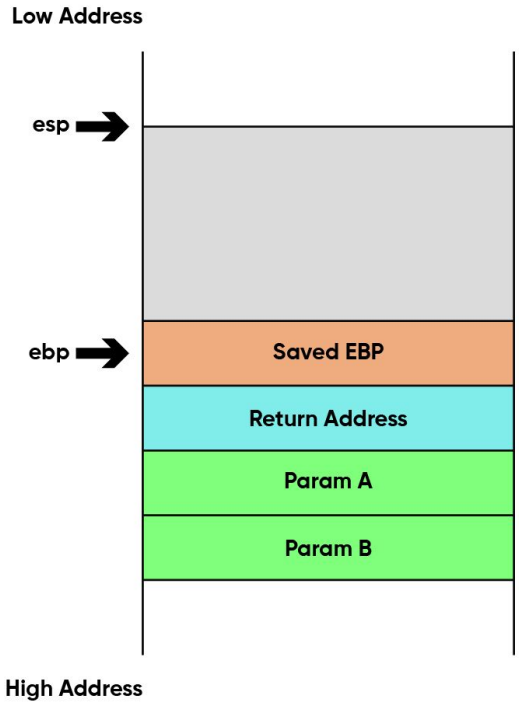
- Copies user input buffer without validating the input length

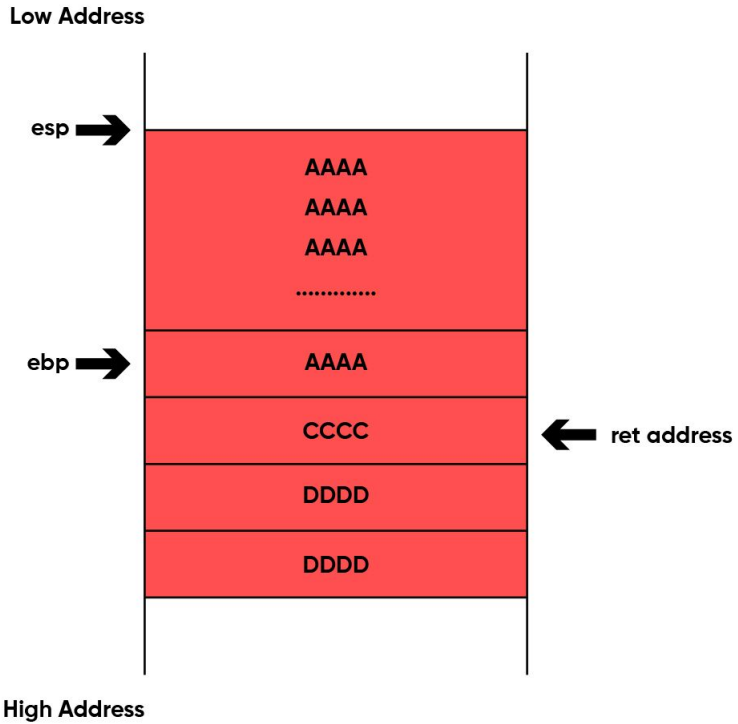
```c
// Buffer Copy without Checking Size of Input
void copy_user_input(char *user_input) {
    char buffer[20];
    strcpy(buffer, user_input); // does not check the length of user_input
}
```

Stack Overflow

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char** argv) {
    char buffer[20];
    memcpy(buffer, argv[1], strlen(argv[1]));
    printf("[+] Printing buffer: %s \n", buffer);
}
```





LAB

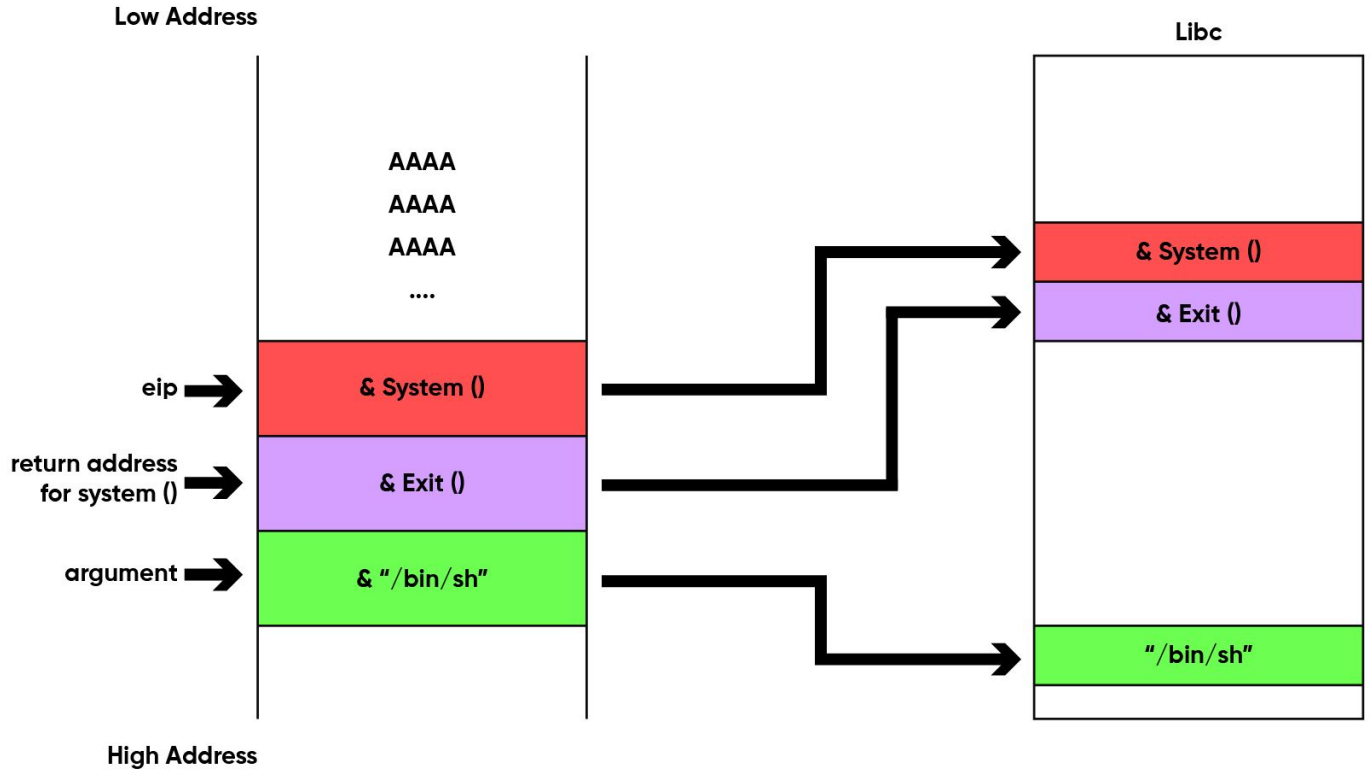
NX

- NX is short for Non-executable
  - segregates region of memory either as data or executable (W^X)
  - cpu won't execute any code or instructions resides in non-executable region

- Some important region of the memory that are marked as non-executable are stack and heap
  - This feature prevents buffer overflow attack to some extent

ret2libc

- It’s a technique that reuse existing code from executable region (libc functions)
  - Certain functions like system(), execv() are used
  - Bypass NX protection
- Instead of replacing return address with shellcode location located at stack, it’ll be overwritten by the address of function like system, execv

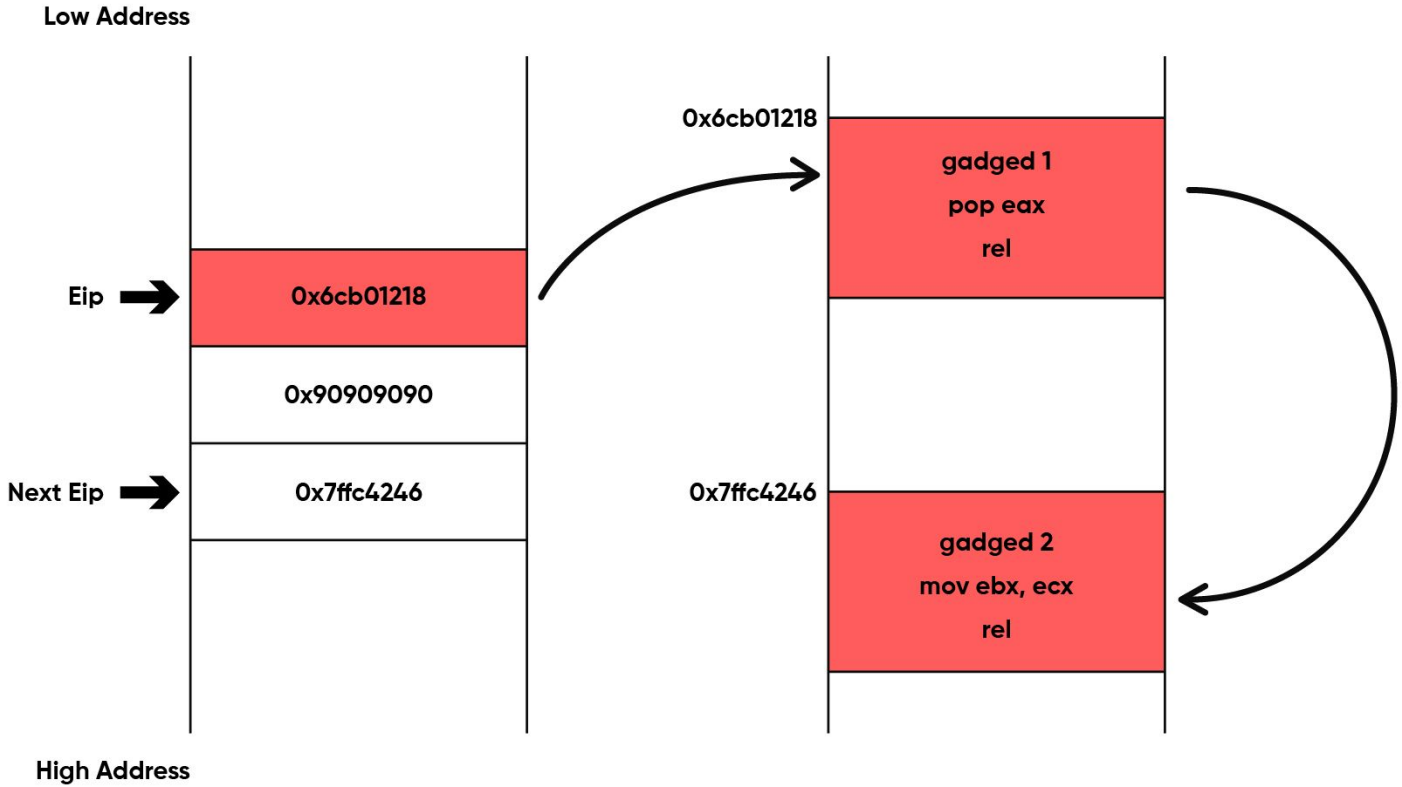


LAB

Return-Oriented Programming (ROP)

- Similar to ret2libc in terms of reusing executable code.
- Sequence of small instructions ending with a “ret” instruction are used.
  - The selected sequence of instructions is called “gadgets.”
  - Chaining these gadgets is what they call a ROP chain.

- Concept is very simple, but finding and designing gadgets could be tricky.
  - Gadgets can be selected from any modules or shared libraries however the location of gadget should be in a executable region



Making Stack Executable Again

- NX protection can be defeated and protected memory region can be made re-executable again including stack
  - mmap()
    - Creates a new region of virtual memory in process address space
    - New region can be created as executable
  - mprotect()
    - Change the memory protection of specific chunk of memory

ROP & mprotect()

- Even if we control the execution, if nx is enabled we can't directly execute our shellcode from stack
- We can craft a ROP chain that calls the mprotect() targeting stack memory address

mprotect() - system call

- Mprotect function definition
  - int mprotect(void *addr, size_t len, int prot);
- Accepts 3 parameters
  - address
  - Size
  - prot
- Syscall number 0x7d

Syscall

- Requests service from the kernel
- For syscall parameters has to be passed in a specific registers:

x86 (32-bit)

Compiled from Linux 4.14.0 headers.

NR	syscall name	references	%eax	arg0 (%ebx)	arg1 (%ecx)	arg2 (%edx)	arg3 (%esi)	arg4 (%edi)	arg5 (%ebp)
0	restart_syscall	man/ cs/	0x00	-	-	-	-	-	-
1	exit	man/ cs/	0x01	int error_code	-	-	-	-	-
2	fork	man/ cs/	0x02	-	-	-	-	-	-
3	read	man/ cs/	0x03	unsigned int fd	char *buf	size_t count	-	-	-
4	write	man/ cs/	0x04	unsigned int fd	const char *buf	size_t count	-	-	-

X86 (32-bit) syscall table [1]

mprotect() - system call - ROPing

- Requires 3 parameters, however need to take care of return address
- According to syscall table, we need to find some gadgets that fulfills following criteria:
  - %eax -> syscall number
  - %ebx -> stack address
  - %ecx -> len
  - %edx -> edx

LAB

Format String

- Special text contains special token known as format specifier
  - In c, inbuilt function like printf, fprintf etc.

- Developers use this to build dynamic and stable output output
  - Based on the specifier various type of data are formatted and inserted into the string

Format Specifier

- Defines type of the variable that'll be feeded as input into the format string
  - Starts with "%" sign
- For instance,
  - %d -> integer
  - %s -> string
  - %f -> float
  - %x -> hexadecimal integer
  - %p -> pointer address

Example

```
int main() {
    char team[20] = "cyberwarfarelabs";
    printf("hello from %s\n", team);
    return 0;
}
```

Format String Vulnerability

- Occurs when user input is improperly treated as a format string by printf family functions (e.g., printf(), sprintf(), fprintf()).
  - Leads to arbitrary memory read/write
    - Allowing to read sensitive information or modify memory contents
      - printf(buffer)

Format String Vulnerability

- Arbitrary Read
  - %p -> print pointer address
  - %2$p -> positional argument to be referenced that are passed into printf
- Arbitrary Write
  - %n -> write 4 byte
  - %hn -> write 2 byte
  - %hhn → write 1 byte
  - %num$n

Mitigations

- NX bit
- Canary
- ASLR / PIE
- FORTIFY_SOURCE

NX Bit

- Makes memory region either writable or executable (W^X)
  - cpu won't execute any code or instructions resides in non-executable region
- Prevents execution in some memory region
  - Stack
  - Heap
- This feature prevents buffer overflow attack to some extent

NX Bit

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start      End        Offset     Perm Path
0x08048000 0x08049000 0x00000000 r-- /home/cped-lin/webinar/lab/overflow/validator-nx
0x08049000 0x0804a000 0x00001000 r-x /home/cped-lin/webinar/lab/overflow/validator-nx
0x0804a000 0x0804b000 0x00002000 r-- /home/cped-lin/webinar/lab/overflow/validator-nx
0x0804b000 0x0804c000 0x00002000 rw- /home/cped-lin/webinar/lab/overflow/validator-nx
0x0804c000 0x0806e000 0x00000000 rw- [heap]
0xf7dcb000 0xf7de4000 0x00000000 r-- /usr/lib32/libc-2.31.so
0xf7de4000 0xf7f3d000 0x00019000 r-x /usr/lib32/libc-2.31.so
0xf7f3d000 0xf7fb1000 0x00172000 r-- /usr/lib32/libc-2.31.so
0xf7fb1000 0xf7fb2000 0x001e6000 --- /usr/lib32/libc-2.31.so
0xf7fb2000 0xf7fb4000 0x001e6000 r-- /usr/lib32/libc-2.31.so
0xf7fb4000 0xf7fb5000 0x001e8000 rw- /usr/lib32/libc-2.31.so
0xf7fb5000 0xf7fb8000 0x00000000 rw-
0xf7fc9000 0xf7fcb000 0x00000000 rw-
0xf7fcb000 0xf7fcf000 0x00000000 r-- [vvar]
0xf7fcf000 0xf7fd1000 0x00000000 r-x [vdso]
0xf7fd1000 0xf7fd2000 0x00000000 r-- /usr/lib32/ld-2.31.so
0xf7fd2000 0xf7ff0000 0x00001000 r-x /usr/lib32/ld-2.31.so
0xf7ff0000 0xf7ffb000 0x0001f000 r-- /usr/lib32/ld-2.31.so
0xf7ffc000 0xf7ffd000 0x0002a000 r-- /usr/lib32/ld-2.31.so
0xf7ffd000 0xf7ffe000 0x0002b000 rw- /usr/lib32/ld-2.31.so
0xfffdd000 0xffffe000 0x00000000 rw- [stack]
```

Canary

- Random value that stores before return address in stack
- Random value gets pushed into the stack at function prologue
- Detects the stack smashing
    - While returning from the function
        - Canary value gets checked if overwritten
            - Program terminates and throw message:
                - “Stack smashing detected”

Canary

```
gef➤  checksec
[+] checksec for '/home/cped-lin/webinar/lab/overflow/validator-canary'
Canary                        : ✓
NX                            : ✘
PIE                           : ✘
Fortify                       : ✘
RelRO                         : ✘
gef➤
```

```
gef➤  canary
[+] The canary of process 4298 is at 0xffffda0b, value is 0x64b8b900
gef➤
```

Canary

```
0x080492df <+0>:     endbr32
0x080492e3 <+4>:     push   ebp
0x080492e4 <+5>:     mov    ebp,esp
0x080492e6 <+7>:     push   ebx
0x080492e7 <+8>:     sub    esp,0x94
0x080492ed <+14>:    call   0x80491d0 <__x86.get_pc_thunk.bx>
0x080492f2 <+19>:    add    ebx,0x20aa
0x080492f8 <+25>:    mov    eax,DWORD PTR [ebp+0x8]
0x080492fb <+28>:    mov    DWORD PTR [ebp-0x8c],eax
0x08049301 <+34>:    mov    eax,gs:0x14
0x08049307 <+40>:    mov    DWORD PTR [ebp-0xc],eax
0x0804930a <+43>:    xor    eax,eax
```

Canary

```
0x0804949b <+444>:    mov    eax,DWORD PTR [ebp-0xc]
0x0804949e <+447>:    xor    eax,DWORD PTR gs:0x14
0x080494a5 <+454>:    je     0x80494ac <check_candidate+461>
0x080494a7 <+456>:    call   0x8049610 <__stack_chk_fail_local>
0x080494ac <+461>:    mov    ebx,DWORD PTR [ebp-0x4]
0x080494af <+464>:    leave
0x080494b0 <+465>:    ret
```

ASLR (Address Space Layout Randomization)

- ASLR randomizes the memory address layout
  - stack, heap, shared libraries
- Makes difficult to find the accurate memory address
  - Prevents from controlling the flow of the execution
- Position Independent Executable (PIE) randomizes the binary memory base address

ASLR (Address Space Layout Randomization)

```
gef➤  checksec
[+] checksec for '/home/cped-lin/webinar/lab/overflow/validator-pie'
Canary                            : ✘
NX                                : ✘
PIE                               : ✓
Fortify                           : ✘
RelRO                             : ✘
gef➤
```

ASLR (Address Space Layout Randomization)

```
gef➤  info proc map
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
        0x5656c000 0x5656d000     0x1000        0x0 /home/cped-lin/webinar/lab/overflow/validator-pie
        0x5656d000 0x5656e000     0x1000     0x1000 /home/cped-lin/webinar/lab/overflow/validator-pie
        0x5656e000 0x5656f000     0x1000     0x2000 /home/cped-lin/webinar/lab/overflow/validator-pie
        0x5656f000 0x56570000     0x1000     0x2000 /home/cped-lin/webinar/lab/overflow/validator-pie
        0xf7d86000 0xf7d9f000    0x19000        0x0 /usr/lib32/libc-2.31.so
        0xf7d9f000 0xf7ef8000   0x159000    0x19000 /usr/lib32/libc-2.31.so
        0xf7ef8000 0xf7f6c000    0x74000   0x172000 /usr/lib32/libc-2.31.so
        0xf7f6c000 0xf7f6d000     0x1000   0x1e6000 /usr/lib32/libc-2.31.so
        0xf7f6d000 0xf7f6f000     0x2000   0x1e6000 /usr/lib32/libc-2.31.so
        0xf7f6f000 0xf7f70000     0x1000   0x1e8000 /usr/lib32/libc-2.31.so
        0xf7f8c000 0xf7f8d000     0x1000        0x0 /usr/lib32/ld-2.31.so
        0xf7f8d000 0xf7fab000    0x1e000     0x1000 /usr/lib32/ld-2.31.so
        0xf7fab000 0xf7fb6000     0xb000    0x1f000 /usr/lib32/ld-2.31.so
        0xf7fb7000 0xf7fb8000     0x1000    0x2a000 /usr/lib32/ld-2.31.so
```

ASLR (Address Space Layout Randomization)

```
gef➤  info proc map
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
        0x5664b000 0x5664c000     0x1000        0x0 /home/cped-lin/webinar/lab/overflow/validator-pie
        0x5664c000 0x5664d000     0x1000     0x1000 /home/cped-lin/webinar/lab/overflow/validator-pie
        0x5664d000 0x5664e000     0x1000     0x2000 /home/cped-lin/webinar/lab/overflow/validator-pie
        0x5664e000 0x5664f000     0x1000     0x2000 /home/cped-lin/webinar/lab/overflow/validator-pie
        0xf7d5c000 0xf7d75000    0x19000        0x0 /usr/lib32/libc-2.31.so
        0xf7d75000 0xf7ece000   0x159000    0x19000 /usr/lib32/libc-2.31.so
        0xf7ece000 0xf7f42000    0x74000   0x172000 /usr/lib32/libc-2.31.so
        0xf7f42000 0xf7f43000     0x1000   0x1e6000 /usr/lib32/libc-2.31.so
        0xf7f43000 0xf7f45000     0x2000   0x1e6000 /usr/lib32/libc-2.31.so
        0xf7f45000 0xf7f46000     0x1000   0x1e8000 /usr/lib32/libc-2.31.so
        0xf7f62000 0xf7f63000     0x1000        0x0 /usr/lib32/ld-2.31.so
        0xf7f63000 0xf7f81000    0x1e000     0x1000 /usr/lib32/ld-2.31.so
        0xf7f81000 0xf7f8c000     0xb000    0x1f000 /usr/lib32/ld-2.31.so
        0xf7f8d000 0xf7f8e000     0x1000    0x2a000 /usr/lib32/ld-2.31.so
```

FORTIFY_SOURCE

- Compile-time security feature in the GNU C Library (glibc)
- Provides runtime protection for detecting buffer overflow
- Certain buffer manipulation related functions are protected with additional wrapper function:
  - strcpy, gets, memcpy, memmove, etc. [2]
- Wrapper function ends with _chk.

FORTIFY_SOURCE

```
cped-lin@ubuntu ~/w/l/overflow> gcc -m32 -fno-stack-protector -D_FORTIFY_SOURCE=2 -no-pie -z execstack -Wl,-z,norelro -O2 main.c -o validator-fortify
main.c: In function ‘main’:
main.c:70:5: warning: ignoring return value of ‘fgets’, declared with attribute warn_unused_result [-Wunused-result]
   70 |     fgets(buf, MAX_BUFFER, stdin);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
In file included from /usr/include/string.h:495,
                 from main.c:3:
In function ‘strncpy’,
    inlined from ‘check_candidate’ at main.c:51:9:
/usr/include/bits/string_fortified.h:106:10: warning: ‘__builtin___strncpy_chk’ specified bound depends on the length of the source argument [-Wstringop
-overflow=]
  106 |   return __builtin___strncpy_chk (__dest, __src, __len, __bos (__dest));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.c: In function ‘check_candidate’:
main.c:51:9: note: length computed here
   51 |         strncpy(lower_candidate, candidates[i], strlen(candidates[i]));
      |         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

FORTIFY_SOURCE

```
gef➤  checksec
[+] checksec for '/home/cped-lin/webinar/lab/overflow/validator-fortify'
Canary                        : ✘
NX                            : ✘
PIE                           : ✘
Fortify                       : ✓
RelRO                         : ✘
```

FORTIFY_SOURCE

```
0xffffd6a0│+0x0000: 0xffffd6d6  →  0x00000000    ← $esp
0xffffd6a4│+0x0004: 0xffffd76c  →  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\n"
0xffffd6a8│+0x0008: 0x00000025 ("%"?)
0xffffd6ac│+0x000c: 0x0000001e
0xffffd6b0│+0x0010: 0xf7fc9110  →  0xf7dcb000  →  0x464c457f
0xffffd6b4│+0x0014: 0xf7fdbe36  →   add esp, 0x20
0xffffd6b8│+0x0018: 0xf7e454df  →  <_IO_default_xsputn+000f> add ebx, 0x16eb21
0xffffd6bc│+0x001c: 0xffffd6d6  →  0x00000000
──────────────────────────────────────────────────────────────── code:x86:
    0x804942a <check_candidate+009a> mov    DWORD PTR [esp+0x18], eax
    0x804942e <check_candidate+009e> mov    edi, eax
    0x8049430 <check_candidate+00a0> push   eax
 →  0x8049431 <check_candidate+00a1> call   0x8049120 <__strncpy_chk@plt>
   ↳   0x8049120 <__strncpy_chk@plt+0000> endbr32
       0x8049124 <__strncpy_chk@plt+0004> jmp    DWORD PTR ds:0x804b454
       0x804912a <__strncpy_chk@plt+000a> nop    WORD PTR [eax+eax*1+0x0]
       0x8049130 <__strcpy_chk@plt+0000> endbr32
       0x8049134 <__strcpy_chk@plt+0004> jmp    DWORD PTR ds:0x804b458
       0x804913a <__strcpy_chk@plt+000a> nop    WORD PTR [eax+eax*1+0x0]
──────────────────────────────────────────────────────────── arguments (guesse
__strncpy_chk@plt (
   [sp + 0x0] = 0xffffd6d6 → 0x00000000,
   [sp + 0x4] = 0xffffd76c → "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\n",
   [sp + 0x8] = 0x00000025,
   [sp + 0xc] = 0x0000001e
)
```

FORTIFY_SOURCE

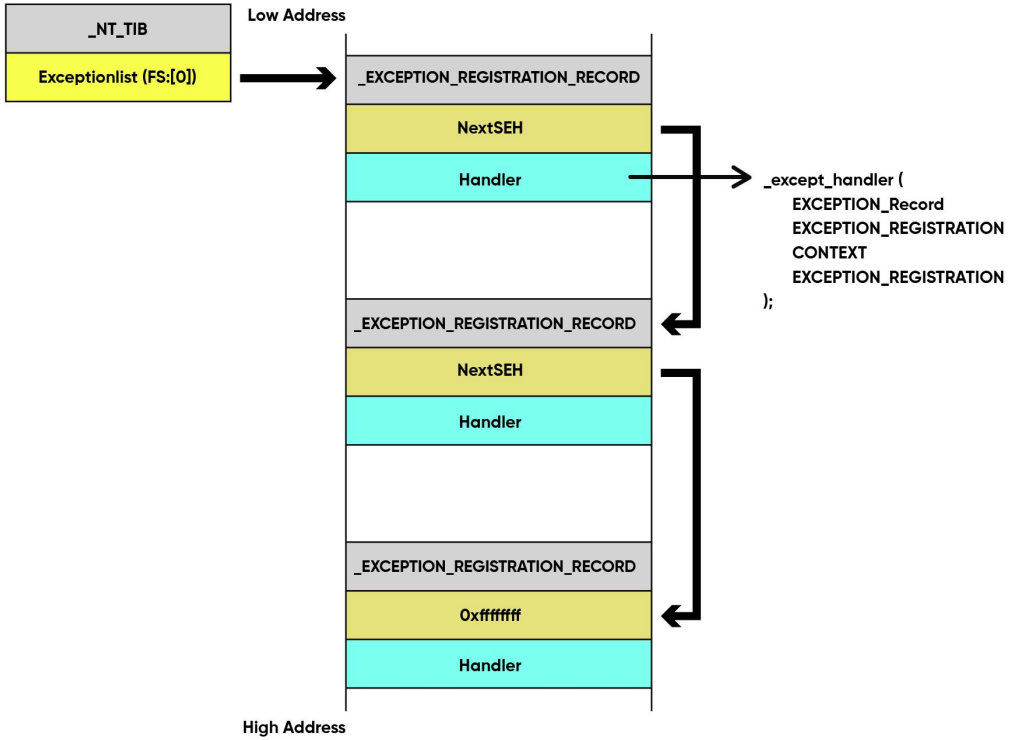
```
cped-lin@ubuntu ~/w/l/overflow> ./validator-fortify
[+] Are you a selected candidate?
[+] Enter your name: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** buffer overflow detected ***: terminated
fish: Job 1, './validator-fortify' terminated by signal SIGABRT (Abort)
cped-lin@ubuntu ~/w/l/overflow [SIGABRT]>
```

LAB

Windows - Win32 Exploit Development

Win32 Structured Exception Handling (SEH)

- Windows OS mechanism for handling hardware or software faults
- Per-thread basis
  - Each thread has its own exception handler callback
- It also provides the extensions to the Microsoft visual c++ compiler
  - Allowing developer to handle faults in their program
  - __try & __except keywords are used to guard the code



Structured Exception Handling (SEH)

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

Structured Exception Handling (SEH)

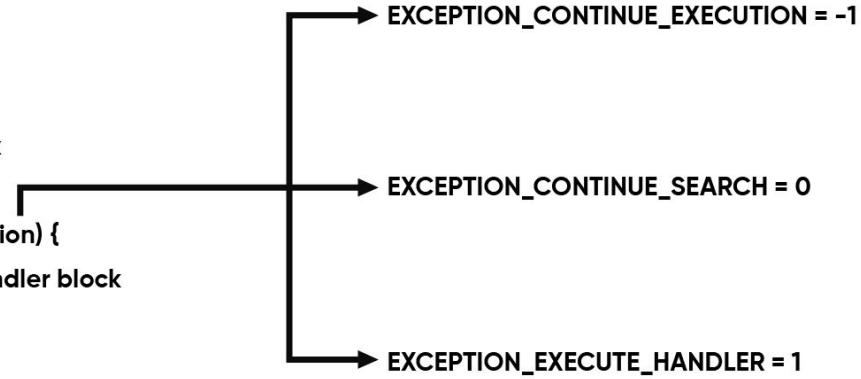
```
EXCEPTION_DISPOSITION
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    oid EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
);
```

Structured Exception Handling (SEH)

```
typedef struct _EXCEPTION_RECORD {
        DWORD ExceptionCode;
        DWORD ExceptionFlags;
        struct _EXCEPTION_RECORD *ExceptionRecord;
        PVOID ExceptionAddress;
        DWORD NumberParameters;
        DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

__try/__except

```
__try{
        //guarded block
        //of code
}__except (filter-expression) {
        //exception-handler block
}
```

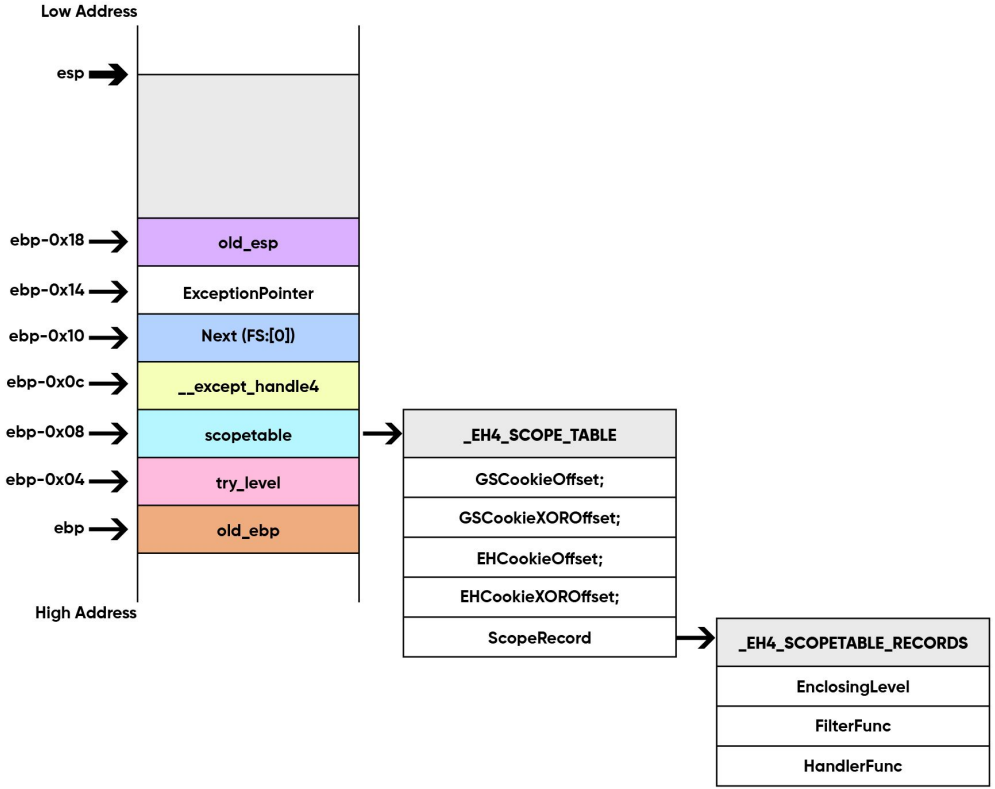


__try/__except

```
int exception_handler(long exception_code, _EXCEPTION_POINTERS *ep ) {
    printf("Exception Code: 0x%x \n", exception_code);
    printf("Exception Address: 0x%x", ep->ExceptionRecord->ExceptionAddress);
    return EXCEPTION_CONTINUE_SEARCH;
}

int main() {
    printf("Attach to Debugger \n");
    system("pause");
    __try {
        void* p = 0x00000000;
        char str[10] = "mydata";
        memcpy(p, str, strlen(str));
    }
    __except (exception_handler(GetExceptionCode(), GetExceptionInformation())) {
        printf("Exception Called \n");
    }

}
```

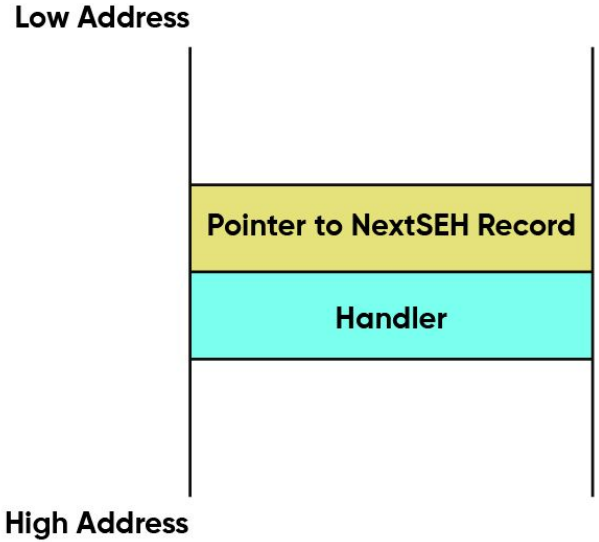


SEH - DEMO

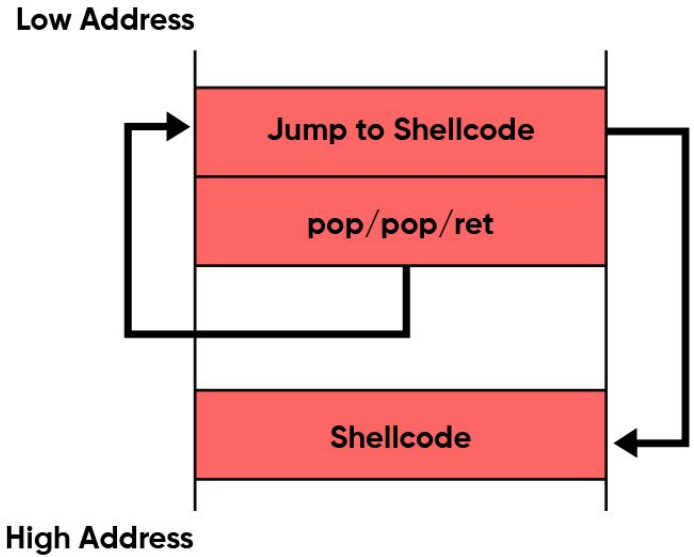
SEH Overflow

- Overflowing the buffer past its limit to reach SEH records
  - placing controlled address in the handler
- Next time when exception occurs
  - code from the controlled address will get executed

SEH Overflow



SEH Overflow – Exploitation



Bad Characters

- Bytes that could be misinterpreted by the application leading to
  - Truncation during buffer copy
  - Byte modification
  - crashing shellcode
- Common bad characters:
  - 0x00, 0x0A, 0x20, 0x2c, 0x2b, 0x2f, 0x5c

Identifying Bad Characters

- Creating buffer containing all possible byte values from 0x00 to 0xff
- Sending the buffer to the program
- Analyzing the buffer in memory vs buffer in disk, Generally looking for
  - Truncation
  - Alteration

Identifying Bad Characters

Disk →	0x0a	0x0c	0x43	0x5F
In Memory →	0x1b	0x0c	0x43	0x60

Conclusion

References

1. https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit
2. https://cwe.mitre.org/data/definitions/121.html
3. https://limbioliong.wordpress.com/2022/01/09/understanding-windows-structured-exception-handling-part-1/
4. https://www.gnu.org/software/libc/manual/html_node/Source-Fortification.html
5. https://github.com/starnight/MicroHttpServer/issues/5
6. https://github.com/advisories/GHSA-p7xp-hqxr-fpq2
7. https://www.exploit-db.com/exploits/44522

Image References

1. https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86-32_bit