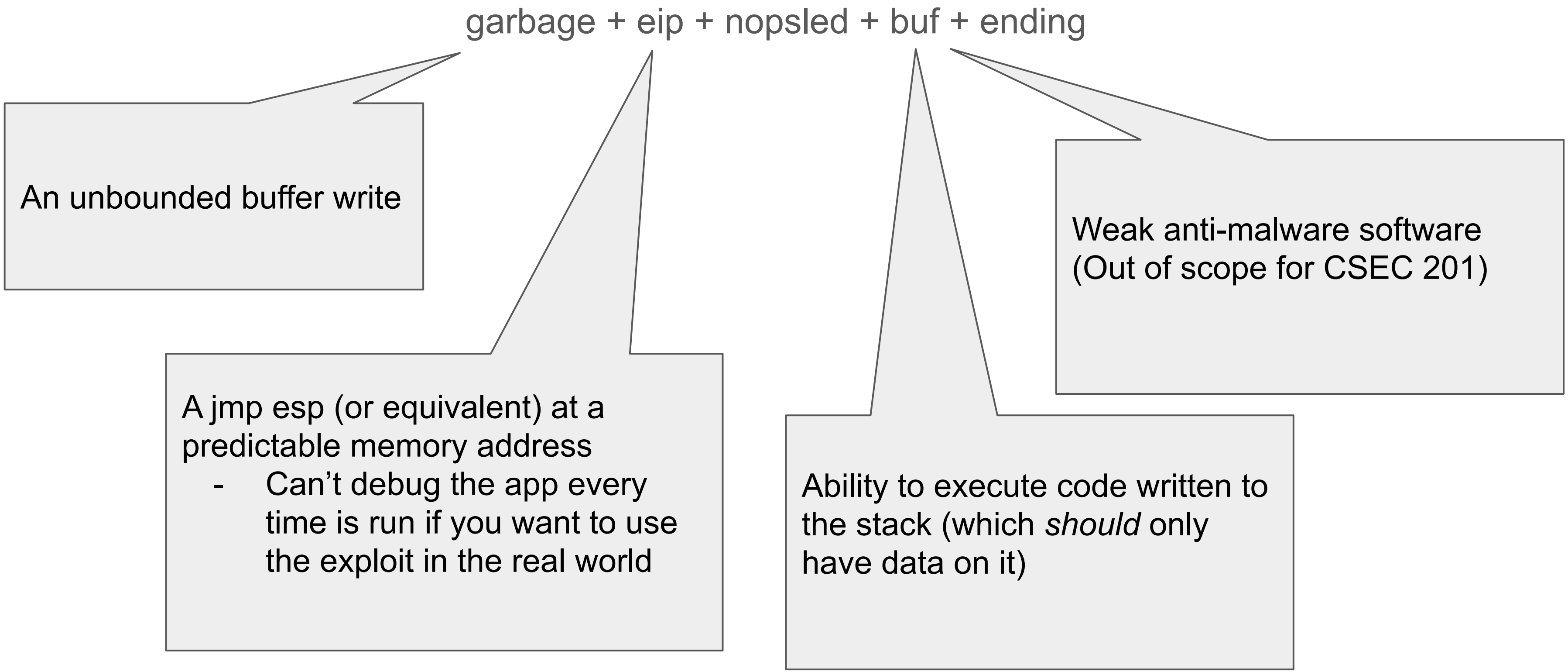
Mitigating Memory Corruption Exploits

CSEC 201
Week 15

Review of Overflow Structure

```
garbage = (“A”* StackSize).encode()        #Junk input, fills up local stack frame
eip = “\x78\x56\x34\x12”                   #Address of jmp esp (or equivalent)
nopsled = “\x90” * sledsize                #Wiggle room
buf = <shellcode generated by msfvenom>    #malware, often a stager
ending =”\r\n”.encode()                    #Ends server-side socket read
badstring = garbage + eip + nopsled + buf + ending
sock.send(badstring)
```

Overflow Preconditions



garbage + eip + nopsled + buf + ending

An unbounded buffer write

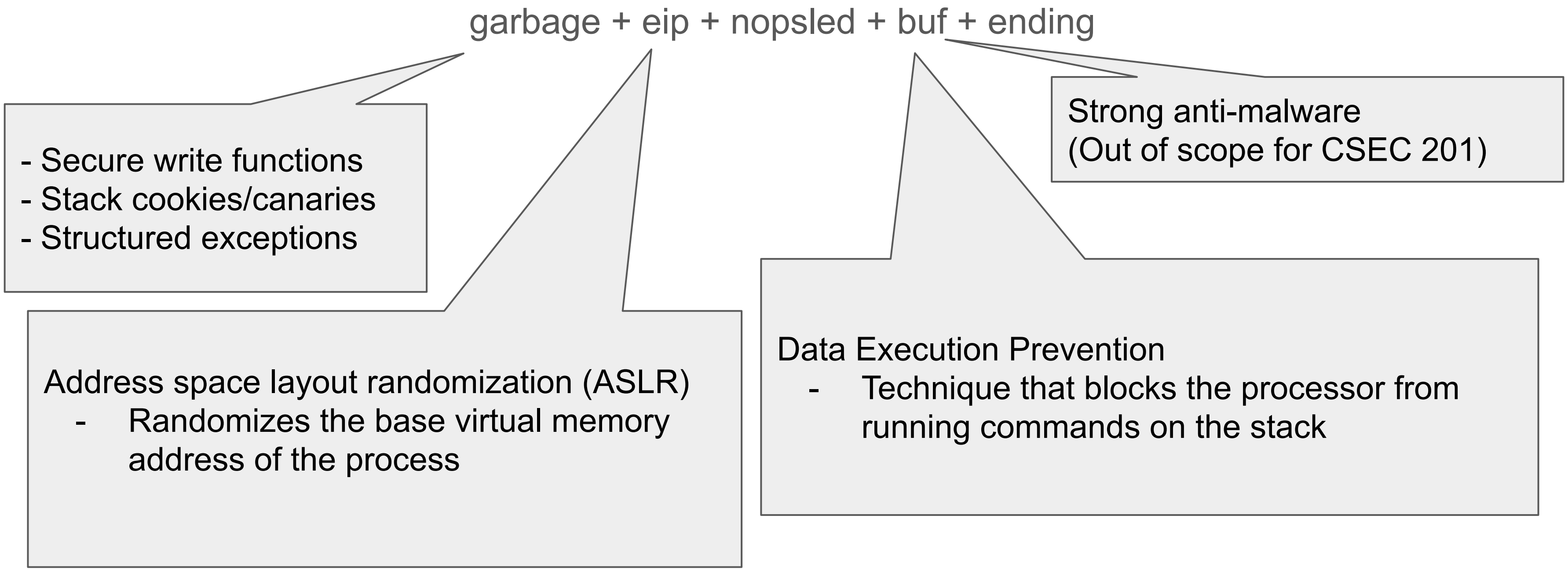
A jmp esp (or equivalent) at a predictable memory address

- Can’t debug the app every time is run if you want to use the exploit in the real world

Ability to execute code written to the stack (which *should* only have data on it)

Weak anti-malware software (Out of scope for CSEC 201)

Eliminating Preconditions



garbage + eip + nopsled + buf + ending

- Secure write functions
- Stack cookies/canaries
- Structured exceptions

Address space layout randomization (ASLR)

- Randomizes the base virtual memory address of the process

Data Execution Prevention

- Technique that blocks the processor from running commands on the stack

Strong anti-malware
(Out of scope for CSEC 201)

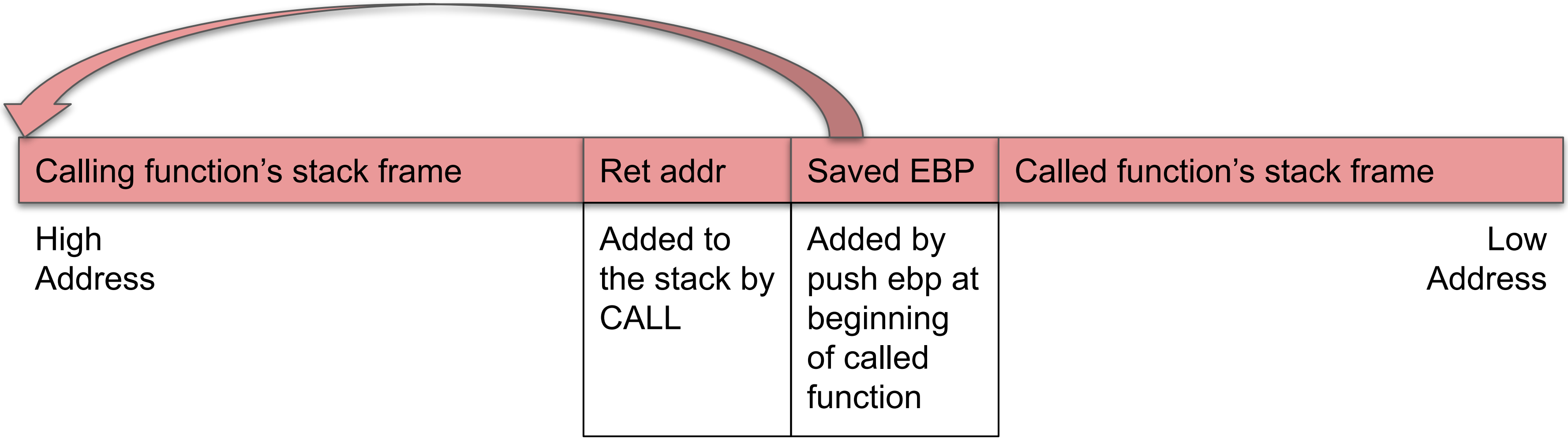
Secure Write Functions [1]

- strcpy(dest, src)
  - Copies the entirety of src buffer into dest
  - **Unsafe**, since the src buffer can be longer than dest buffer
  - Logic holds for scanf, gets, sprintf (for some argument lists), etc.
- strncpy(dest, src, len)
  - Copies len-many characters from src buffer into dest buffer
  - Intended use: strncpy(dest, src, sizeof(dest))
  - Better than strcpy, but still considered **unsafe** since len can be longer than dest
  - If len is reached before end of src, dest will also not be null terminated (Buffer overreads)
  - Encourages the anti-pattern: strncpy(dest, src, strlen(src))
    - If len > strlen(src), strncpy will pad with 0, a cause of errors [src in notes]
  - Logic also holds for sprintf, fgets, sprintf (for some argument lists), etc.

Secure Write Functions [2]

- “<function>_s” family of functions (strncpy_s, scanf_s, etc.)
    - Visual Studio specific
    - strncpy_s(dest, dest_len, src, src_len)
        - Copies the smaller of dest_len and src_len from src into dest.
        - Addresses strncpy anti-pattern by requiring both buffer lengths
            - Nothing stopping: strncpy_s(dest, strlen(src), src, strlen(src))
    - scanf_s(format-spec, buffer, len)
        - Reads len-many characters from stdin into the buffer
        - Intended use: scanf_s(format-spec, buffer, sizeof(buffer))
- Glibc (Linux)
    - Refuses to add memory-safe functions, puts onus on developers to use functions securely
    - Argument - even Microsoft versions don’t completely remove developer responsibility
    - Cisco created a library *safelibc*, which receives/received very little use

Stack Cookies / Canaries [1]



Calling function’s stack frame

Ret addr

Saved EBP

Called function’s stack frame

High
Address

Added to the stack by CALL

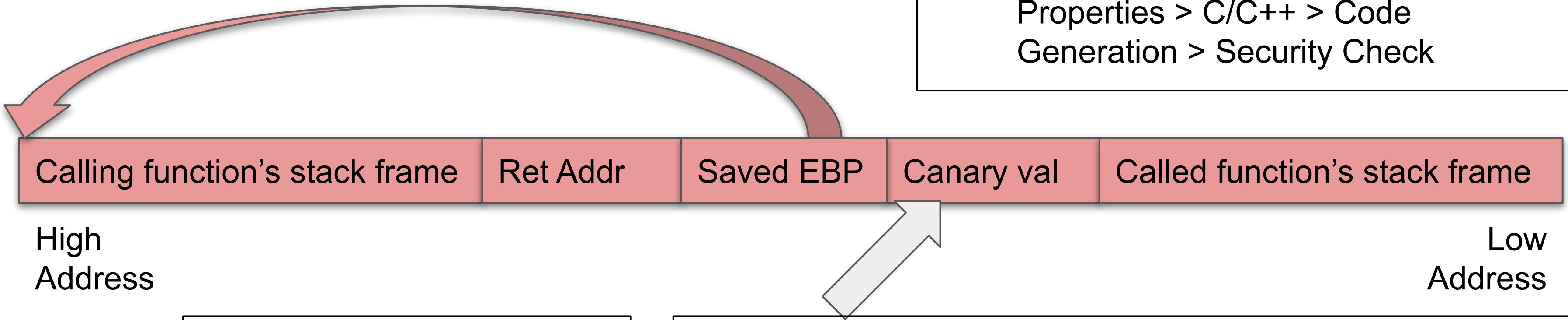
Added by push ebp at beginning of called function

Low
Address

Stack Cookies / Canaries [2]

Referred to as GuardStack in Visual Studio

- Compile flag: /GS
- Project Properties > Configuration Properties > C/C++ > Code Generation > Security Check



Random constant value pushed at beginning of called function

Ex:

```
Funct2:
        Push ebp
        Push 1234
```

Check at end of function to see if value changed

Ex:

```
…
mov esp, ebp        ; clear local stack
pop ebx             ; pop canary into ebx
cmp ebx,1234        ; Check val on stack against constant
jne overflowerror   ; Overflow happened if canary changed
pop ebp             ; restore calling function's stack frame
ret                 ; pop saved address into eip
```

Stack Cookies / Canaries [3]

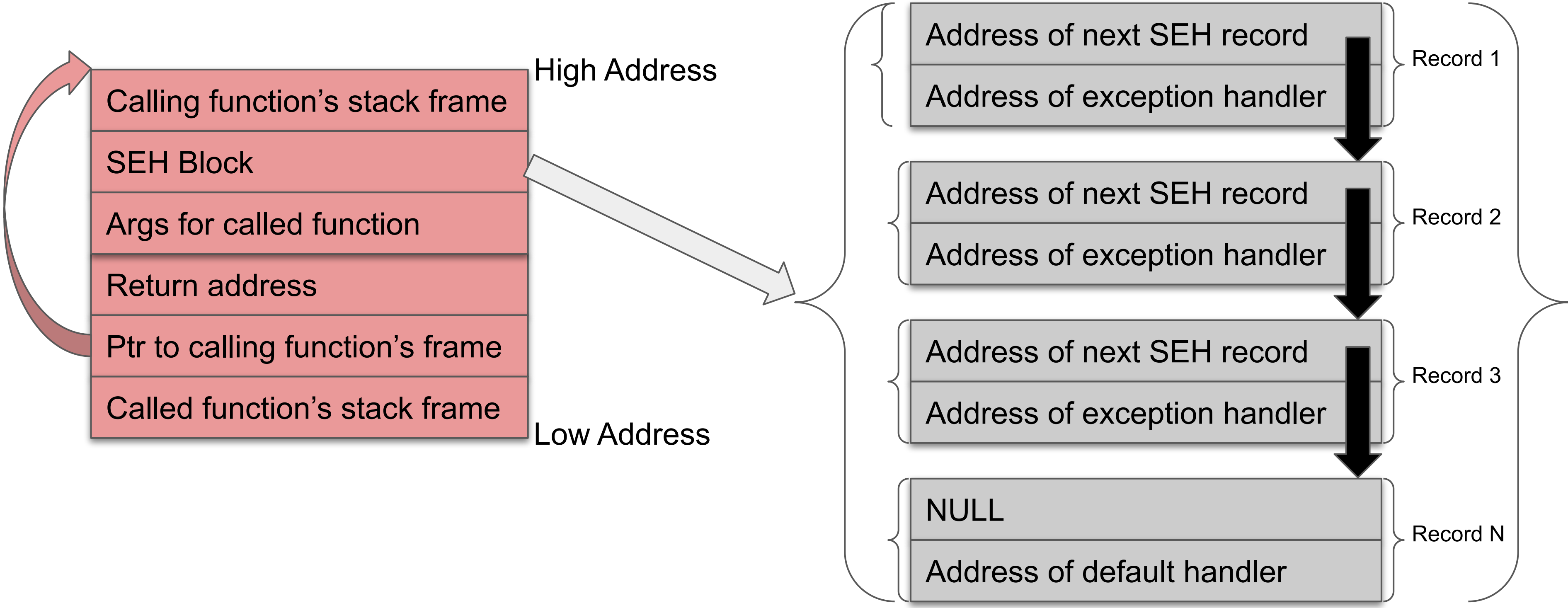
- Different kinds of canaries
  - Null canary - 0x00000000
    - Many string operation will terminate once they hit the null-byte, stopping overreads and some overflows
  - Terminator canary - 0x00000aff
  - Random canary - 0x00<random int>
  - XOR canary - like a random canary, but the value is intended to be XOR’d against a non-static value to produce a result that is difficult to pre-calculate
    - Often the EBP
- Can be bypassed (except XOR canary)
  - Canary type needs to be known (can be reverse engineered via debuggers)
  - The location of the canary on the stack can be read

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

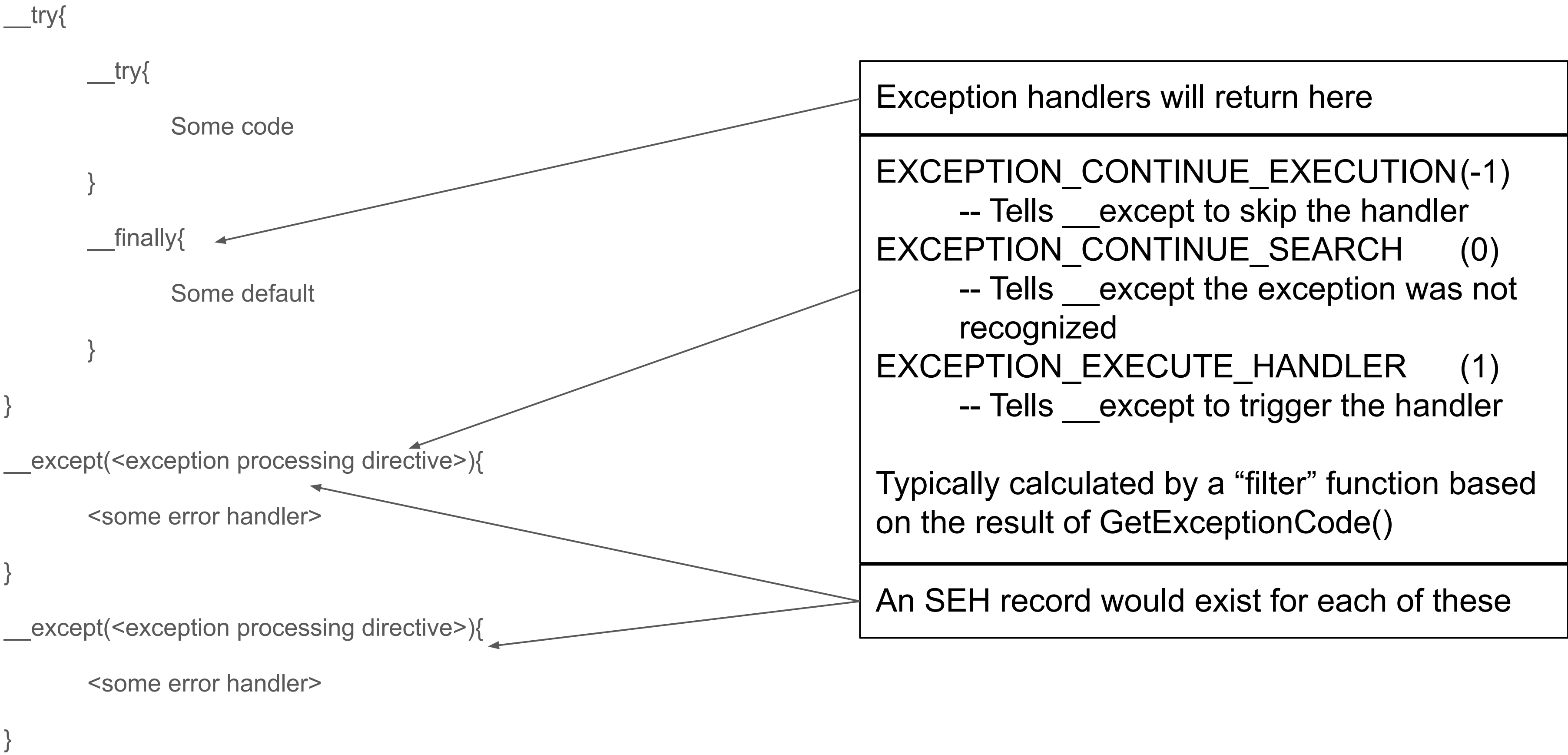
Structured Exception Handling [1]

- A Windows-specific add-on
  - But not just to C, pattern holds for other Windows languages (VB, C#, etc)
- Two mechanisms- *try-except* and *try-finally*
  - try-except -> “Exception Handlers”
  - try-finally -> “Termination Handlers”
  - From a development perspective, behaves like exception handling in Python / Java
- If used, Visual Studio compile command must include /EHa or /Ehsc flags
- Adds an SEH block to the stack whenever a function is called

Structured Exception Handling [2]


Calling function’s stack frame
SEH Block
Args for called function
Return address
Ptr to calling function’s frame
Called function’s stack frame
High Address
Low Address
Address of next SEH record
Address of exception handler
Record 1
Address of next SEH record
Address of exception handler
Record 2
Address of next SEH record
Address of exception handler
Record 3
NULL
Address of default handler
Record N

Structured Exception Handling [3]



```
__try{

        __try{

                Some code

        }

        __finally{

                Some default

        }

}

__except(<exception processing directive>){

        <some error handler>

}

__except(<exception processing directive>){

        <some error handler>

}
```

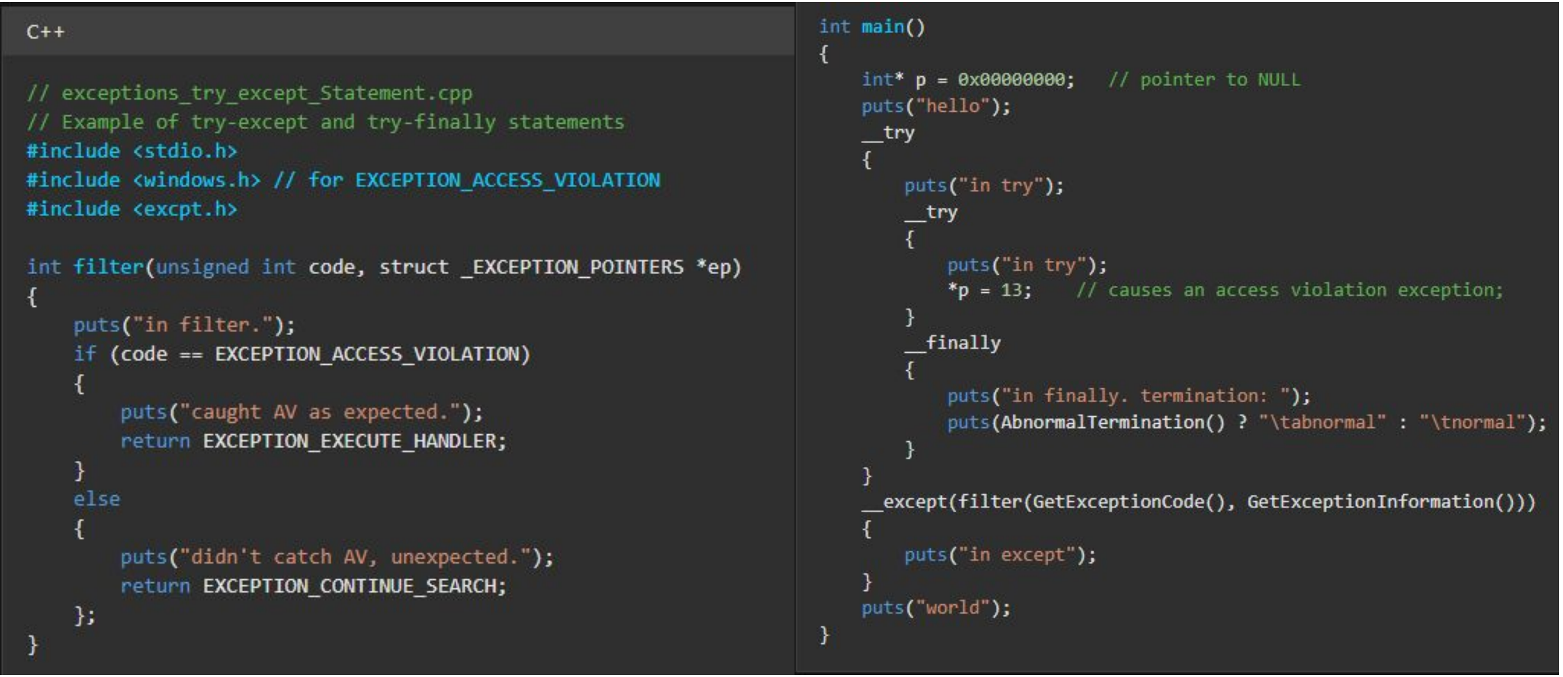
Exception handlers will return here

EXCEPTION_CONTINUE_EXECUTION(-1)
-- Tells __except to skip the handler
EXCEPTION_CONTINUE_SEARCH (0)
-- Tells __except the exception was not recognized
EXCEPTION_EXECUTE_HANDLER (1)
-- Tells __except to trigger the handler

Typically calculated by a “filter” function based on the result of GetExceptionCode()

An SEH record would exist for each of these

Structured Exception Handling [4]

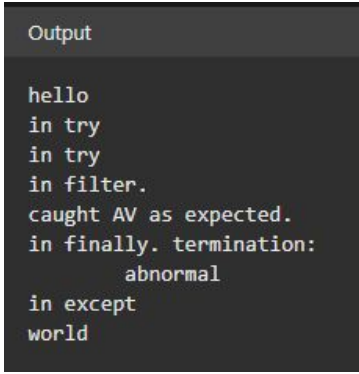


C++

```cpp
// exceptions_try_except_Statement.cpp
// Example of try-except and try-finally statements
#include <stdio.h>
#include <windows.h> // for EXCEPTION_ACCESS_VIOLATION
#include <excpt.h>

int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    puts("in filter.");
    if (code == EXCEPTION_ACCESS_VIOLATION)
    {
        puts("caught AV as expected.");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        puts("didn't catch AV, unexpected.");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}
```

```cpp
int main()
{
    int* p = 0x00000000;   // pointer to NULL
    puts("hello");
    __try
    {
        puts("in try");
        __try
        {
            puts("in try");
            *p = 13;    // causes an access violation exception;
        }
        __finally
        {
            puts("in finally. termination: ");
            puts(AbnormalTermination() ? "\tabnormal" : "\tnormal");
        }
    }
    __except(filter(GetExceptionCode(), GetExceptionInformation()))
    {
        puts("in except");
    }
    puts("world");
}
```



Output

```
hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
        abnormal
in except
world
```

https://docs.microsoft.com/en-us/cpp/cpp/try-except-statement?view=msvc-170

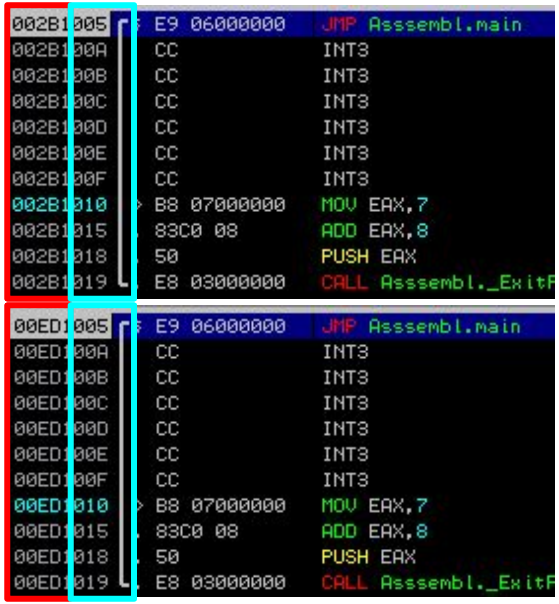
Structured Exception Handling [5]

- Incomplete list of exception codes…
  - EXCEPTION_ARRAY_BOUNDS_EXCEEDED
  - EXCEPTION_ACCESS_VIOLATION
  - EXCEPTION_STACK_CHECK
  - EXCEPTION_STACK_OVERFLOW
- SEH can be bypassed
  - Basic SEH often includes commands that can *facilitate* exploit development
  - Involves overwriting the SEH Block on the stack and replacing exception handler addresses
- SEH has been hardened in SEHOP and SAFESEH
  - SEHOP - Structured Exception Handling Overwrite Protection
    - Validates the record chain in the SEH Block when __except fires to ensure exception handler addresses have not been replaced
  - SAFESEH - Moves SEH Blocks to memory locations outside the program stack
    - All DLLs loaded by the application must be compiled with SAFESEH for it to work
  - There are bypasses for these too, of course

https://docs.microsoft.com/en-us/windows/win32/Debug/getexceptioncode

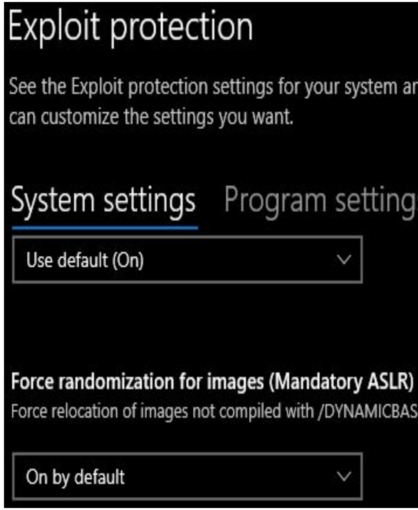
Address Space Layout Randomization (ASLR)

- Varies program’s virtual memory address space
  - Windows ***may*** change image base over time
- Makes exploit development harder by making it more difficult to predict addresses for jmp esp (or equiv)
- Windows supports mandatory ASLR on top of compiled version
- Compiler flag: /DYNAMICBASE
- Project Properties > Configuration Properties > Linker > Advanced > Randomized Base Address



```
002B1005  E9 06000000   JMP Asssembl.main
002B100A  CC            INT3
002B100B  CC            INT3
002B100C  CC            INT3
002B100D  CC            INT3
002B100E  CC            INT3
002B100F  CC            INT3
002B1010  B8 07000000   MOV EAX,7
002B1015  83C0 08       ADD EAX,8
002B1018  50            PUSH EAX
002B1019  E8 03000000   CALL Asssembl._ExitF
```

```
00ED1005  E9 06000000   JMP Asssembl.main
00ED100A  CC            INT3
00ED100B  CC            INT3
00ED100C  CC            INT3
00ED100D  CC            INT3
00ED100E  CC            INT3
00ED100F  CC            INT3
00ED1010  B8 07000000   MOV EAX,7
00ED1015  83C0 08       ADD EAX,8
00ED1018  50            PUSH EAX
00ED1019  E8 03000000   CALL Asssembl._ExitF
```



Exploit protection

See the Exploit protection settings for your system ar can customize the settings you want.

Use default (On)

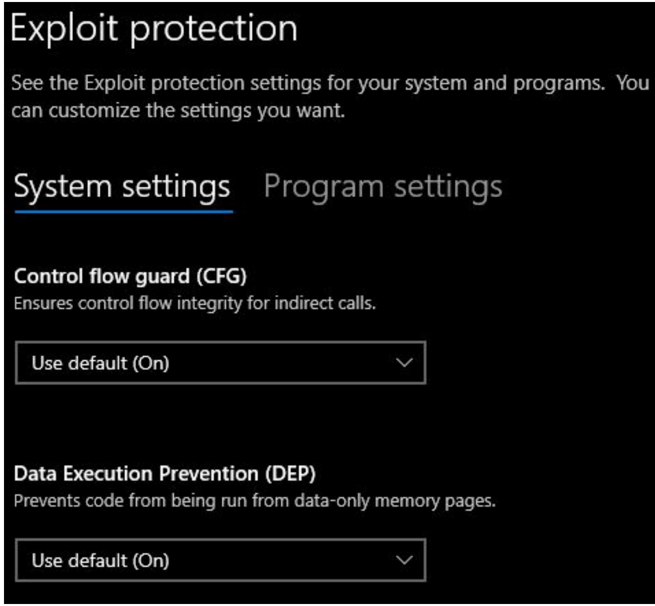
Force randomization for images (Mandatory ASLR)

Force relocation of images not compiled with /DYNAMICBAS

On by default

Data Execution Prevention

- Marks portions of memory used for data as non-executable
    - Virtual memory is marked with an access control constant, indicating permissions:
        - Ex: PAGE_EXECUTE_READ, PAGE_READONLY, etc
    - Stack / Heap marked PAGE_READWRITE
- A stack / heap address landing in EIP throws STATUS_ACCESS_VIOLATION exception
- Compiler flag: /NXCOMPAT
- Project Properties > Configuration Properties > Linker > Advanced > Data Execution Prevention (DEP)
- Windows supports mandatory DEP
- Can be bypassed (of course)



Exploit protection

See the Exploit protection settings for your system and programs. You can customize the settings you want.

Control flow guard (CFG)
Ensures control flow integrity for indirect calls.

Use default (On)

Data Execution Prevention (DEP)
Prevents code from being run from data-only memory pages.

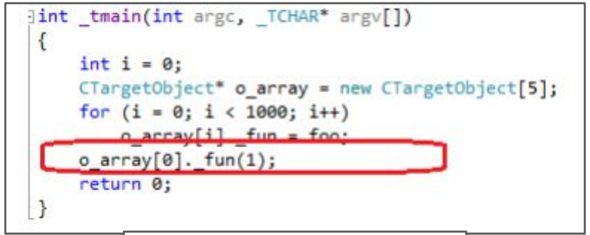
Use default (On)

https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

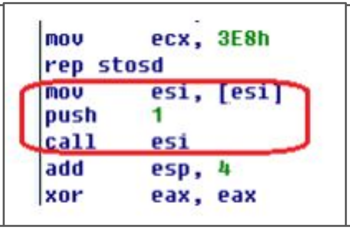
https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection-constants

Control Flow Guard (CFG) [1]

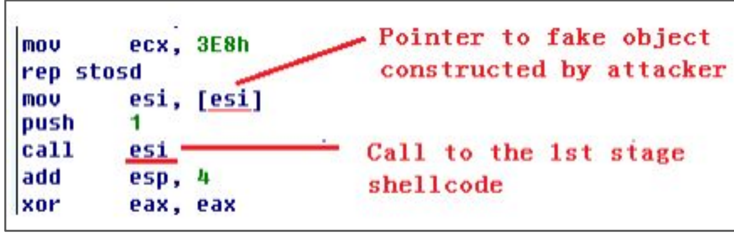
- Platform feature (like DEP / [SAFE]SEH[OP] / ASLR)
- Compiler flag: /guard:cf
- Project Properties > Configuration Properties > Linker > Advanced > Randomized Base Address
- Intended to secure indirect function calls
  - Follow the pattern:
    mov regA, [regB]
    call regA
  - If the value of regB is changed, call will jump to a different location
  - Note - address of the function being called is not decided until runtime



```
int _tmain(int argc, _TCHAR* argv[])
{
    int i = 0;
    CTargetObject* o_array = new CTargetObject[5];
    for (i = 0; i < 1000; i++)
        o_array[i]._fun = foo;
    o_array[0]._fun(1);
    return 0;
}
```



```
mov     ecx, 3E8h
rep stosd
mov     esi, [esi]
push    1
call    esi
add     esp, 4
xor     eax, eax
```



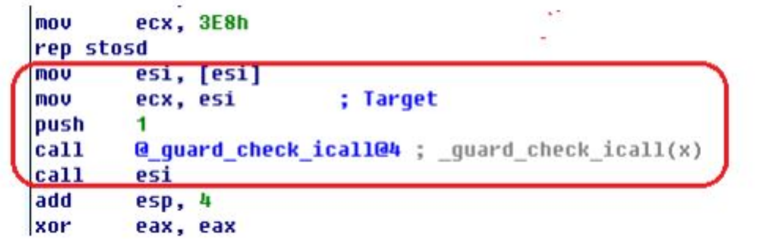
```
mov     ecx, 3E8h
rep stosd
mov     esi, [esi]
push    1
call    esi
add     esp, 4
xor     eax, eax
```

Pointer to fake object constructed by attacker

Call to the 1st stage shellcode

https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

Control Flow Guard (CFG) [2]



```
mov     ecx, 3E8h
rep stosd
mov     esi, [esi]
mov     ecx, esi          ; Target
push    1
call    @_guard_check_icall@4 ; _guard_check_icall(x)
call    esi
add     esp, 4
xor     eax, eax
```

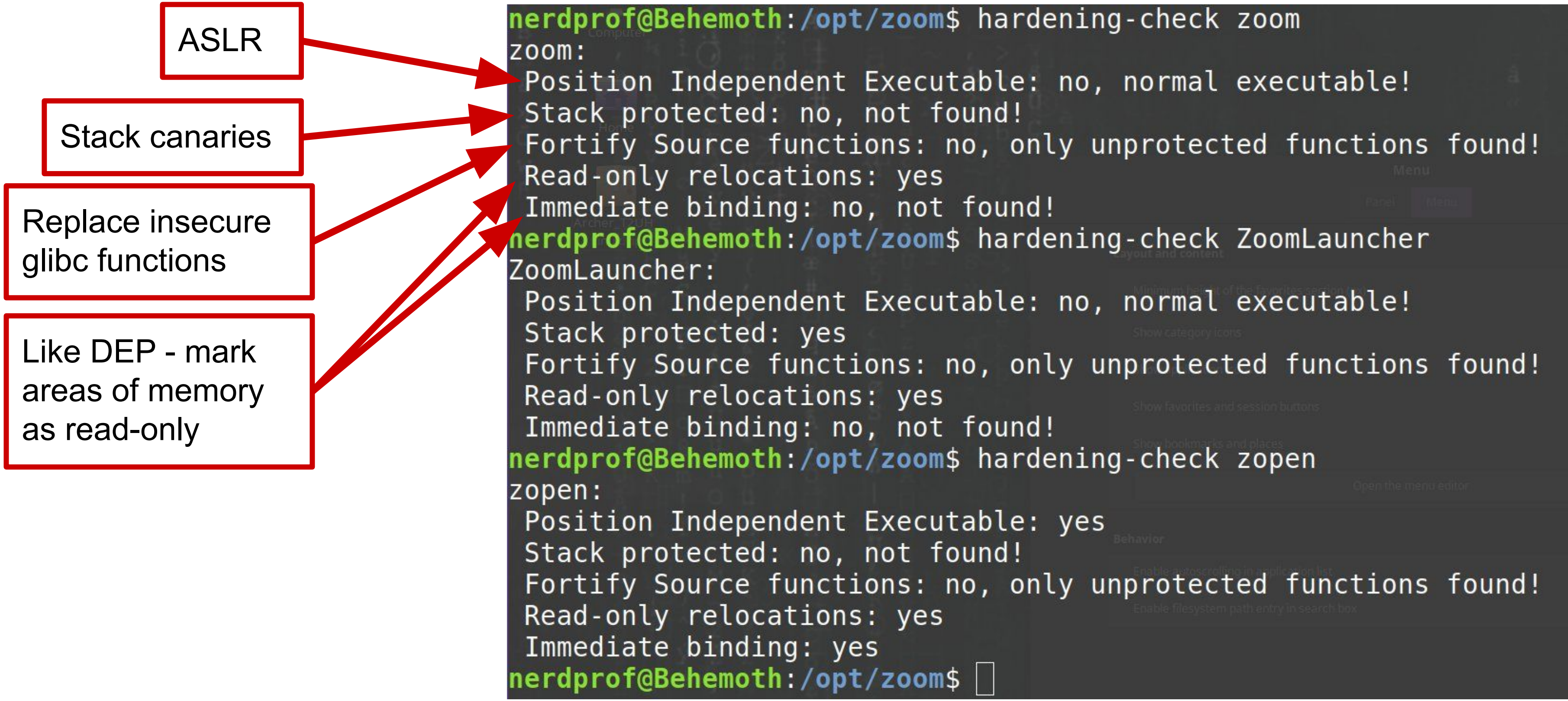
- Compiler computes a “bitmap” (CFGBitmap)
  - Based on starting addresses of all functions
  - Calculated at runtime (Because of ASLR)
  - Every 8 bytes of process memory corresponds to 1 bit in the CFG Bitmap
  - If there is a function starting address in a group of 8 bytes, set the corresponding bit to 1, 0 otherwise
- Compiler adds a call to a guard function before indirect call
  - In version of Windows w/o CFG, this does nothing
- Guard function looks up address to call in CFGBitmap
  - If corresponding bit is 1, call is (likely) valid
    - There must be a starting function call within 7 bytes of address of function call, so attacker’s ability to jump is limited
  - If corresponding bit is 0, call is invalid

https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

Linux Stack Protections - Linux

- Insecure Functions
  - -D_FORTIFY_SOURCE=2 will replace some unsafe functions with safer counterparts
- Stack Canaries
  - On by default in gcc (-fno-stack-protector disables)
- Data Execution Prevention
  - Iffy - some older Linux applications *require* DEP be disabled
  - Decision is made by the linker
    - ‘-z execstack’ indicates that binary requires executable stack
    - ‘-z noexecstack’ indicates that binary does not require executable stack (default behavior)
- Address Space Layout Randomization
  - Referred to as “Position independent executable” (-pie or -fpie)
  - Default behavior is to have PIE enabled

Checking Linux Binaries (Screenshot from 4/2020)



ASLR

Stack canaries

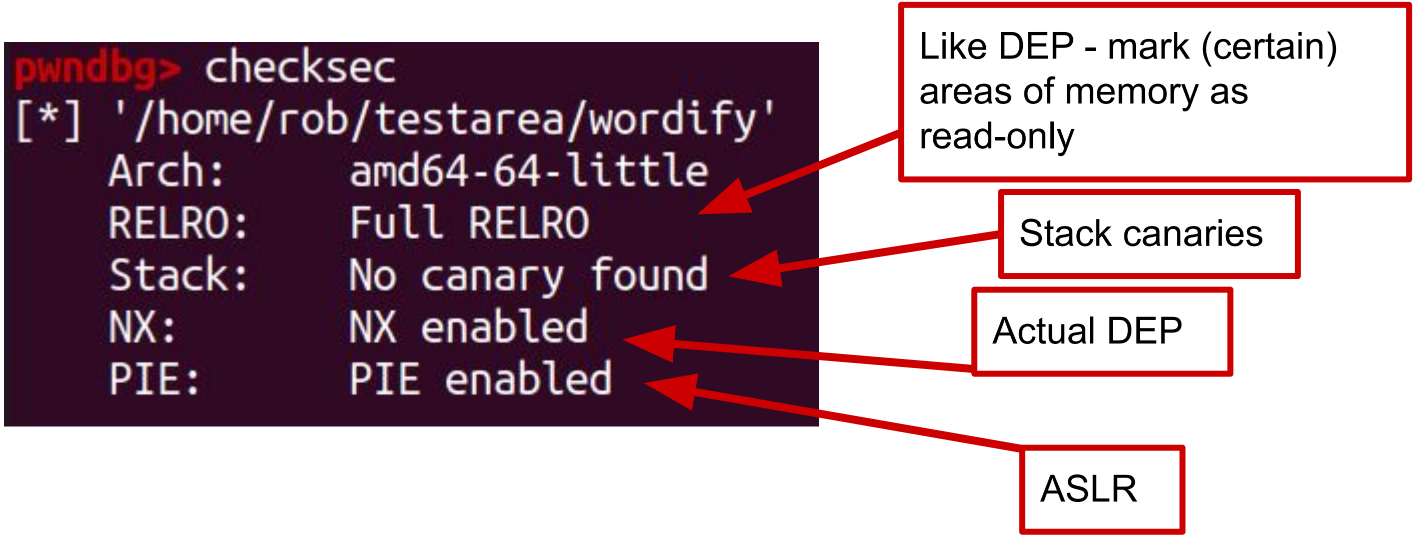
Replace insecure glibc functions

Like DEP - mark areas of memory as read-only

```
nerdprof@Behemoth:/opt/zoom$ hardening-check zoom
zoom:
 Position Independent Executable: no, normal executable!
 Stack protected: no, not found!
 Fortify Source functions: no, only unprotected functions found!
 Read-only relocations: yes
 Immediate binding: no, not found!
nerdprof@Behemoth:/opt/zoom$ hardening-check ZoomLauncher
ZoomLauncher:
 Position Independent Executable: no, normal executable!
 Stack protected: yes
 Fortify Source functions: no, only unprotected functions found!
 Read-only relocations: yes
 Immediate binding: no, not found!
nerdprof@Behemoth:/opt/zoom$ hardening-check zopen
zopen:
 Position Independent Executable: yes
 Stack protected: no, not found!
 Fortify Source functions: no, only unprotected functions found!
 Read-only relocations: yes
 Immediate binding: yes
nerdprof@Behemoth:/opt/zoom$ 
```

Checking Linux Binaries

- https://github.com/pwndbg/pwndbg
  - Extension for GDB (install and then run gdb)
  - Requires pwntools Python3 module (pip install pwntools)... *not documented*



```
pwndbg> checksec
[*] '/home/rob/testarea/wordify'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

Like DEP - mark (certain) areas of memory as read-only

Stack canaries

Actual DEP

ASLR

Where to go after this?

- More advanced exploit development
  - Heap Sprays
  - SEH Bypasses
  - DEP Bypasses
  - ASLR Bypasses
- Investigating how to build these security controls into software development lifecycles
- Bug bounty hunting!
  - Always ensure that you follow the rules of bug bounty programs