The Evolution of Microsoft's Exploitation Mitigations

Agenda

What are mitigations?

Blocking the transfer of control

Blocking the malicious code itself

What is a vulnerability?

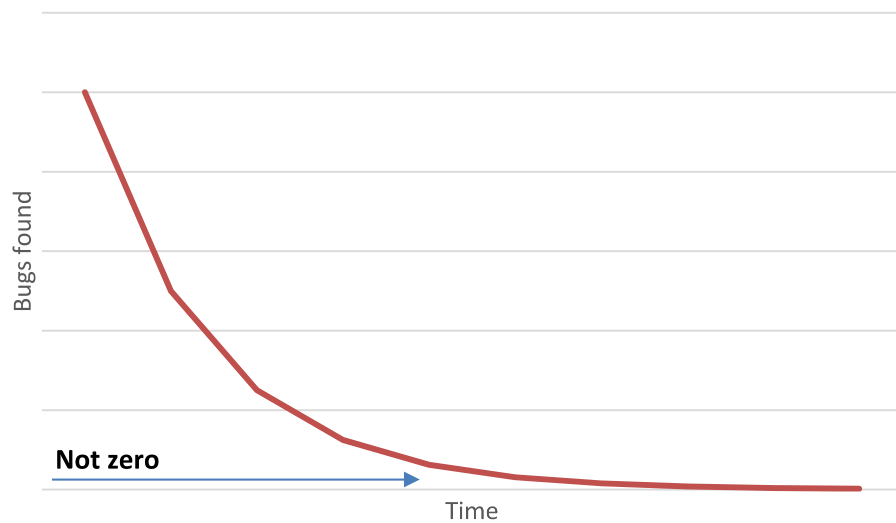
- A software defect that allows an attacker to do something they shouldn’t
- For this presentation, we consider only memory corruption vulnerabilities
  - i.e. buffer overflows
- Used in real life to install viruses

Exploiting a vulnerability

- Step 0: Find vulnerability
- Step 1: Exploit the vulnerability to transfer control to malicious code
  - The processor goes where it’s pointed. In this step the exploit points the processor to malicious code.
- Step 2: Execute the malicious code.
  - In this step the bad stuff actually happens

Why not just fix all the bugs?

- Have you ever written bug-free code?
- Finding the last bug is really really hard
- Incremental cost to fix each bug



What are mitigations?

- Address Steps 1 and 2

- Countermeasures to exploitation techniques
  - *Prevent*
  - *Reduce reliability*

- Generic protection for known & unknown vulnerabilities

Arms Race

1. Implement a mitigation
2. Someone finds a way to bypass it
    - Sometimes only partially
3. Goto 1

Assumptions

- C/C++
- Non-exotic architecture, e.g. x86/x64, ARM, PowerPC

Block the Transfer of Control

Buffer Overflow: Under the Hood

- It's all numbers
- Programs compiled to machine code
  - Very low-level numeric instructions to the CPU

L.Address	Bytes	Mnemonic
00007c00	(2) 33C0	***xor ax, ax***
00007c02	(2) 8ED0	mov ss, ax
00007c04	(3) BC007C	mov sp, 0x7c00
00007c07	(2) 8EC0	mov es, ax
00007c09	(2) 8ED8	mov ds, ax
00007c0b	(3) BE007C	mov si, 0x7c00
00007c0e	(3) BF0006	mov di, 0x0600
00007c11	(3) B90002	mov cx, 0x0200
00007c14	(1) FC	cld
00007c15	(2) F3A4	rep movsb byte ptr es:[di], byte ptr ds:[si]
00007c17	(1) 50	push ax
00007c18	(3) 681C06	push 0x061c
00007c1b	(1) CB	retf
00007c1c	(1) FB	sti
00007c1d	(3) B90400	mov cx, 0x0004
00007c20	(3) BDBE07	mov bp, 0x07be
00007c23	(4) 807E0000	cmp byte ptr ss:[bp], 0x00
00007c27	(2) 7C0B	jl .+11 (0x00007c34)
00007c29	(4) 0F851001	jnz .+272 (0x00007d3d)
00007c2d	(3) 83C510	add bp, 0x0010
00007c30	(2) E2F1	loop .-15 (0x00007c23)
00007c32	(2) CD18	int 0x18
00007c34	(3) 885600	mov byte ptr ss:[bp], dl
00007c37	(1) 55	push bp
00007c38	(4) C6461105	mov byte ptr ss:[bp+17], 0x05
00007c3c	(4) C6461000	mov byte ptr ss:[bp+16], 0x00
00007c40	(2) B441	mov ah, 0x41

Buffer Overflow: Memory

- When you call a function, the CPU needs:
  - To tell the function its parameters
  - To leave room for the function's variables
  - To remember where it came from, so you can go back when the function returns
    - This is a memory address – a number

Local variable 1
Local variable 2
Frame pointer
Return address
Function parameters

Memory Layout, Oversimplified

- Stack:
  - Highly structured memory, not very flexible, fast
  - Used for:
    - System operations like function calls
    - Local variables
  - One big chunk of memory per program
- Heap:
  - Unstructured, dynamic, flexible
  - Used for malloc, new, etc.
- Program memory: the code itself

Memory Management

- At the beginning and end of every function, the compiler inserts standard code
  - Called “prologue” and “epilogue”
  - Sets up and cleans up the stack for the function
- The heap has its own memory manager
  - More on this at the end if we have time
- The OS handles initializing program memory

Buffer Overflow: Strings

- Strings are really an array of characters
  - And characters are really numbers (“A” = 65)
- In C, strings have predefined lengths
  - Called “char *” instead of String
  - Each character can be accessed individually

A String in Memory
MyString[0]
MyString[1]
MyString[2]
Etc.

Buffer Overflow: Now, we hack!

```
Char* removeEnd(char* inStr, int length)
{
        char result[255];
        for (int i = 0; i < length-1; i++)
                result[i] = inStr[i];
        return result;
}
```

- What happens if length > 255?

Memory
inStr
length
Result[0] = inStr[0] = "R"
Result[1] = inStr[0] = "e"
Result[2] = inStr[0] = "d"
Etc.
Result[254]
Frame pointer
Return address

Buffer Overflow: Now, we hack!

result[255] = input[255];

Stack
inStr
length
Result[0] = Input[0]
Result[1] = Input[1]
Result[2] = Input[2]
Etc.
Result[254] = Input[254]
Return address Input[255]

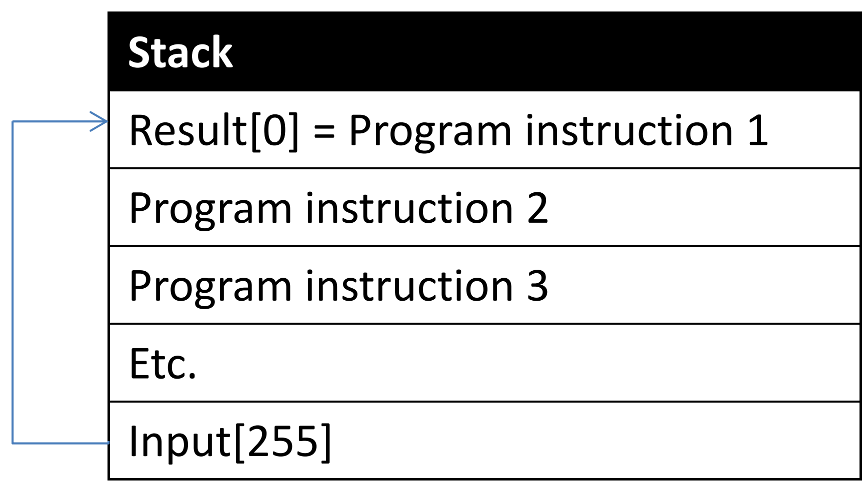
<- Input[255] goes here!

Buffer Overflow: Code Runs

- The function ends
  - Time to return to where we were
  - Where we were?
    - Input[255] has overwritten the original location!
    - Input[255] will be interpreted as a memory address!
    - And we'll start executing whatever is there
- How could a bad guy use this?

Buffer Overflow: Virus Running

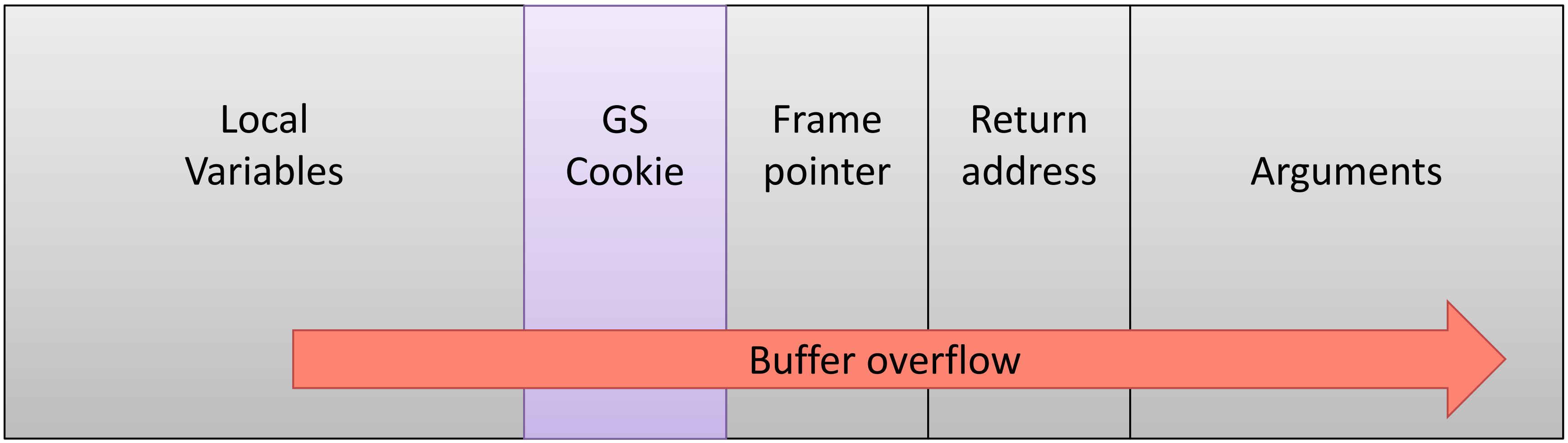
- What if we constructed Input[255] to point to a program?
  - We could store the program in Input[0], input[1]…



Mitigation: Stack Cookie

- /GS adds to the program initialization, prologue and epilogue:
  - At run time, get a pseudo-random number
  - In prologue, in between locals and frame pointer, store a copy of this pseudo-random number ("cookie")
  - In epilogue, just before returning, check the cookie to make sure it's the same number

Mitigation: Stack Cookie (/gs)



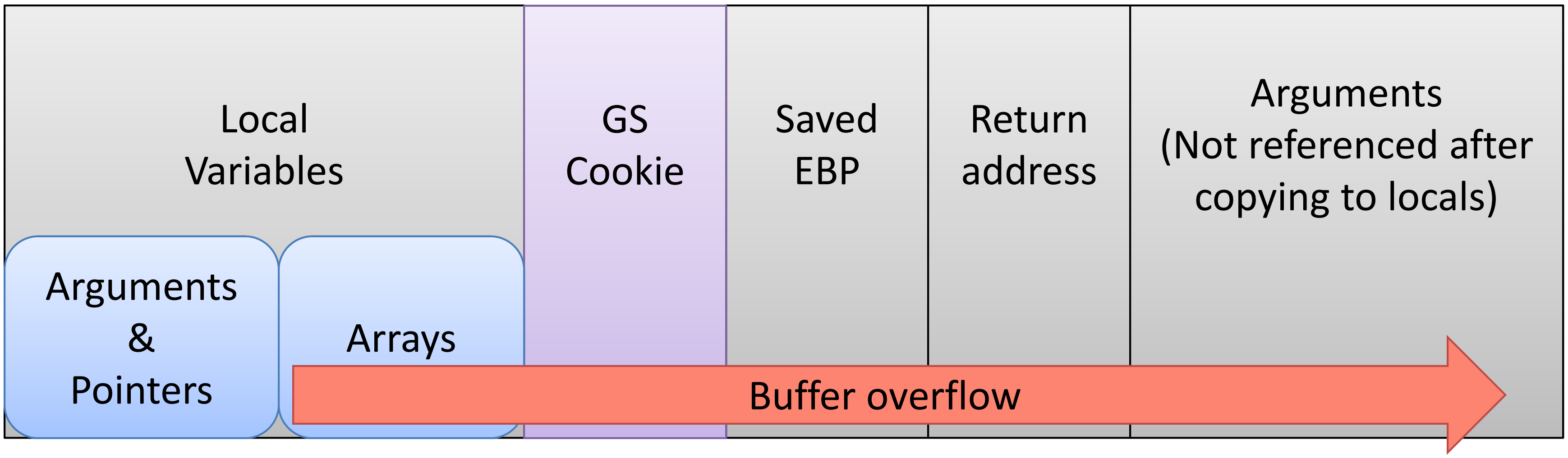
- The attacker must overwrite the GS cookie on the way to the return address
- If the GS cookie doesn't match our pre-calculated value, an exploit has occurred

Exploit: Overwrite variables

```
void vulnerable(char *in, char *out) {
   char buf[256];
   strcpy(buf, in);    // overflow!
   strcpy(out, buf);   // out is corrupt
   return;             // GS cookie checked
}
```

- Cookie is only checked at function return
- Corrupt arguments or locals may be used before return
  - In this example we just did a strcpy, but we might have done something more interesting like send data to the internet
- Attacker could overwrite cookie or other memory[2,8]

Mitigation: /GS improvements

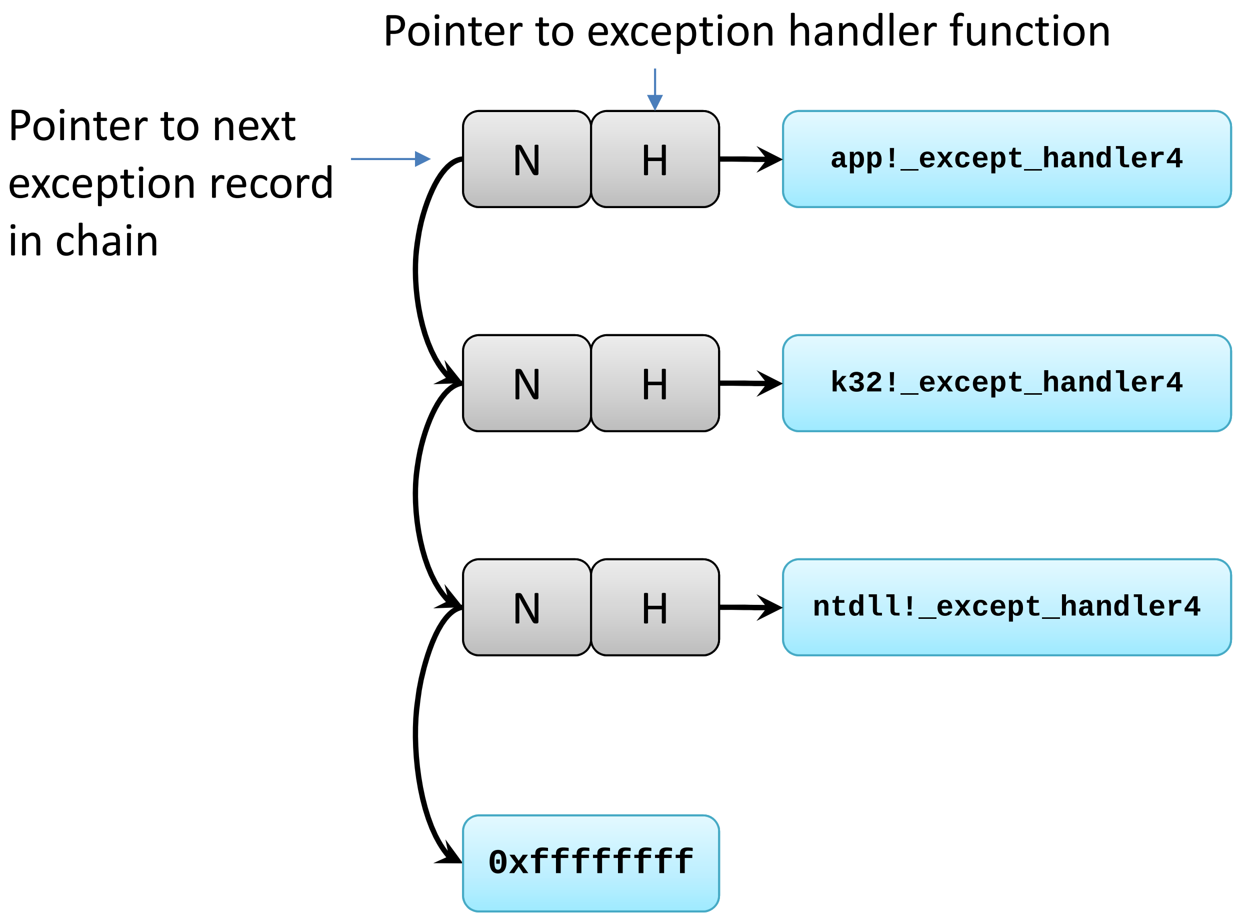


- Safe copies of arguments made as locals
- Arrays positioned directly adjacent to GS cookie
- Corruption of dangerous locals and arguments is less likely

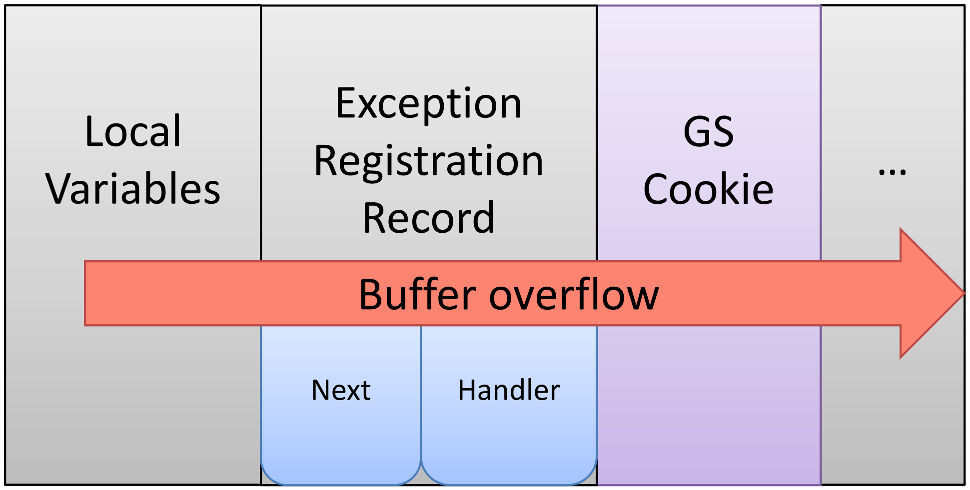
Exceptions Quick Intro

- Exception = an error
  - Exceptions have types e.g. TimeoutException
- Chainable:
  - Class A handles TimeoutExceptions
  - Class B inherits Class A and also handles TimeoutExceptions
  - When an instance of Class B throws a TimeoutException:
    - Class B's Exception Handler gets called
    - Then Class A's Exception Handler gets called too
- Lets you build very dynamic error handling
  - And also lets you bypass /gs…

Exceptions Memory Structure



Exploit: SEH Overwrite

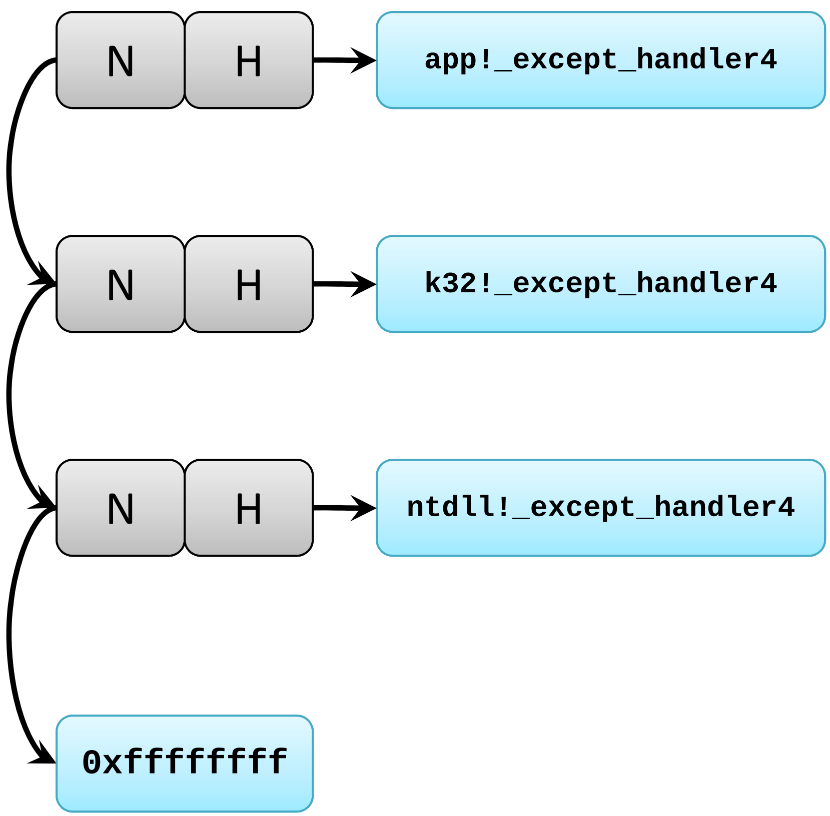



```
void vulnerable(char *ptr){
    char buf[128];
    try {
        strcpy(buf, ptr);
        … exception …
    } except(…) { }
}
```

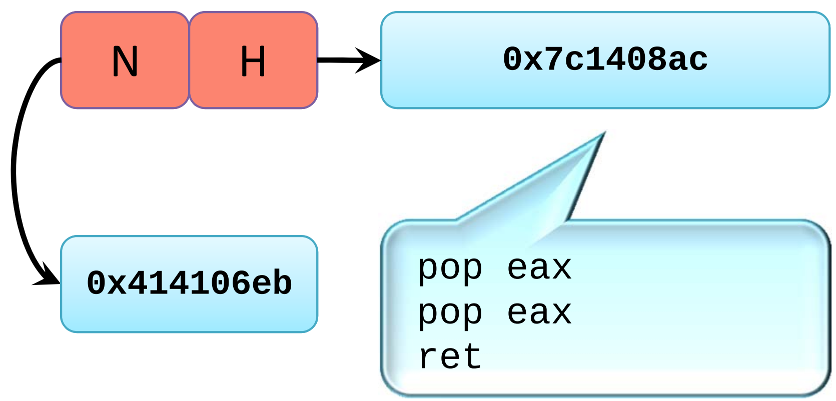
1. Overwrite an exception handler using the vulnerability being exploited
2. Trigger an exception some other way
   - Pretty easy to do

Exploit: SEH Overwrite

Normal SEH Chain



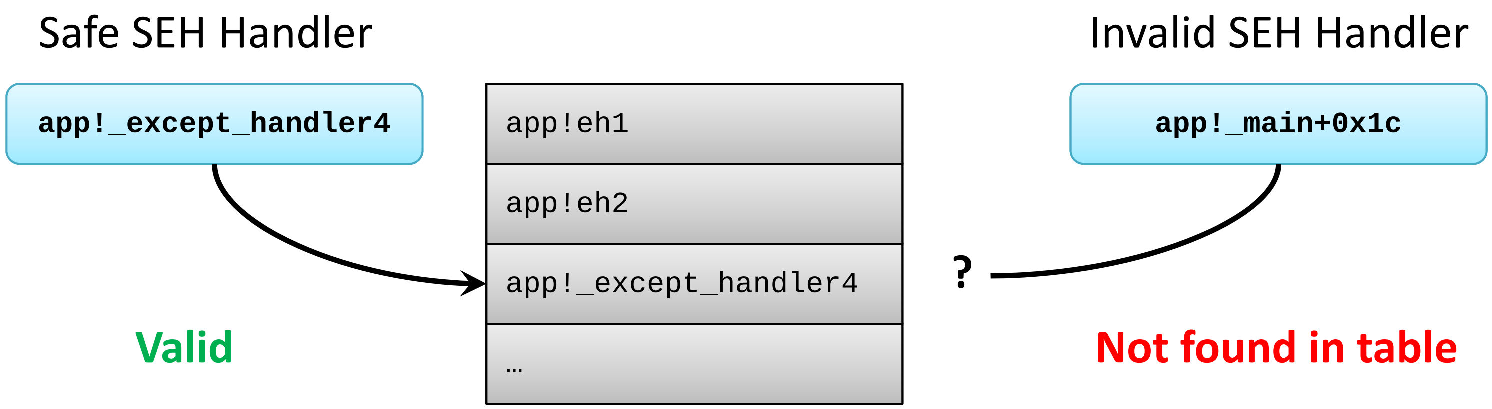
Corrupt SEH Chain



An exception will cause `0x7c1408ac` to be called as an exception handler as:

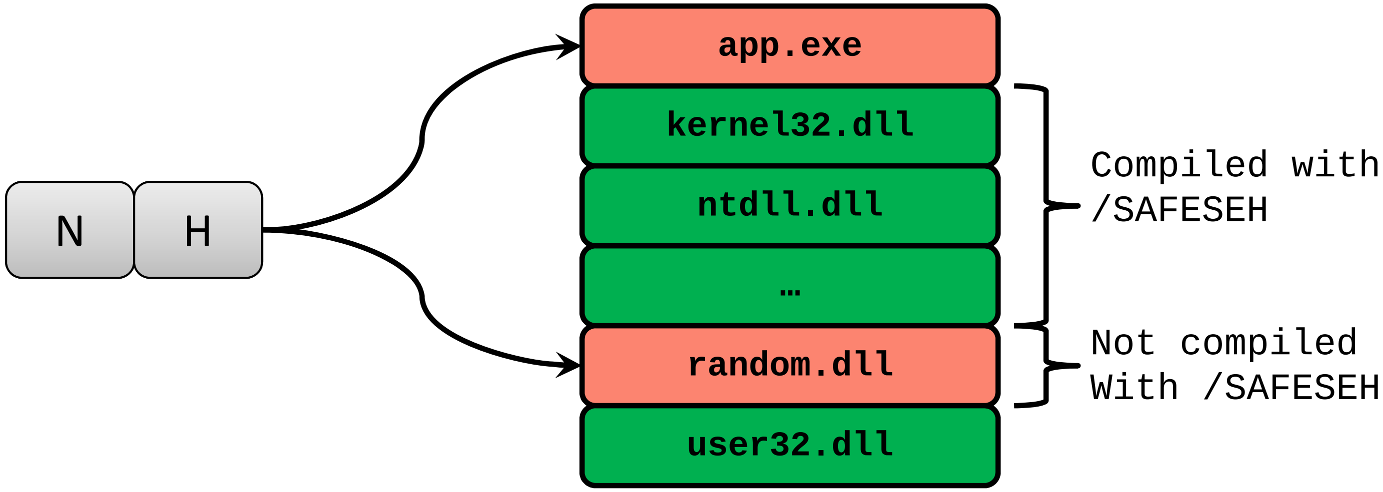
```
EXCEPTION_DISPOSITION Handler(
   PEXCEPTION_RECORD Exception,
   PVOID EstablisherFrame,
   PCONTEXT ContextRecord,
   PVOID DispatcherContext);
```

Mitigation: SafeSEH



- VS2003 linker change (/SAFESEH) [9]
- Binaries are linked with a table of safe exception handlers
  - Stored in program memory – not corruptible by an attacker
- Exception dispatcher checks if handlers are safe before calling

Exploit: Modules without SafeSEH

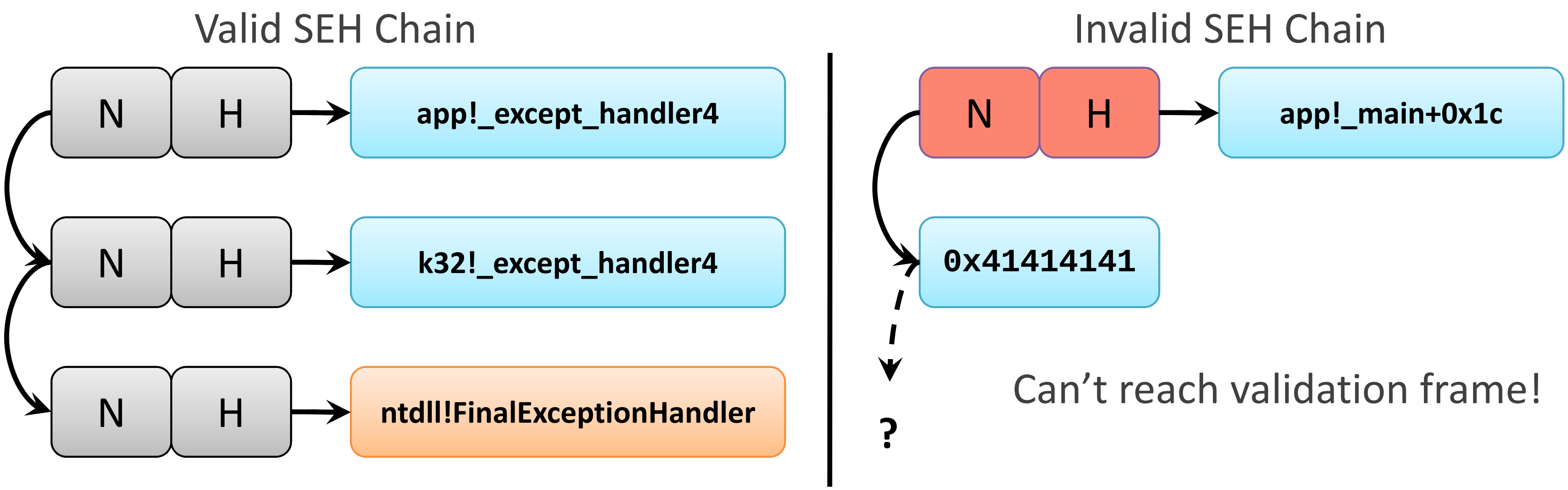


- SafeSEH is most effective if all binaries in a process have been linked with it
- `Handler` can be pointed into a binary that does not have a safe exception handler table

Sentinels

- Sentinel: a fake value in a linked list whose only role is to be recognizable
- We insert the sentinel at the end of a list
- We also keep a copy of it
- When following the list, we compare every record to the sentinel
  - When we get to the sentinel, we know we reached the end of the list
  - If we never get to the sentinel, we know the list was tampered with

Mitigation: SEHOP



- Dynamic protection for SEH overwrites in Srv08/Vista SP1 [4]
  - No compile/link time hints required
- Symbolic *validation frame* inserted as final entry in chain
- Corrupt `Next` pointers prevent traversal to validation frame

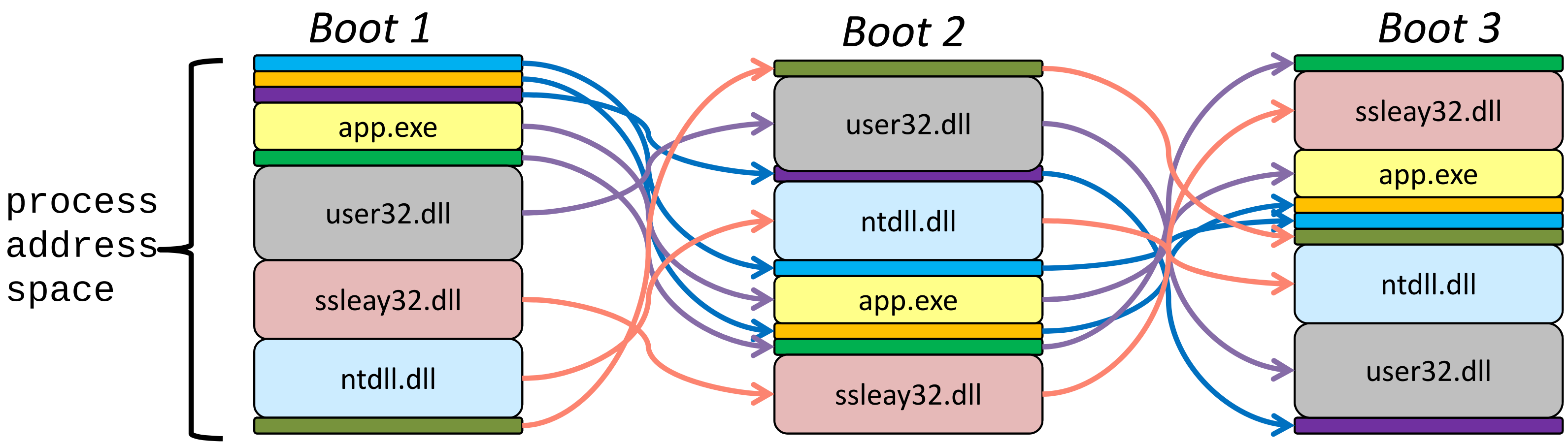
Recap: GS, SafeSEH, and SEHOP

- Attacker cannot…
  - Overwrite the frame pointer or the return address
  - Overwrite arguments & non-buffer local variables
- They can overwrite SEH…
  - But SafeSEH/SEHOP prevent it from being called
- These primarily protect stack overflows
  - We'll talk about the heap if we have time.

More about Memory

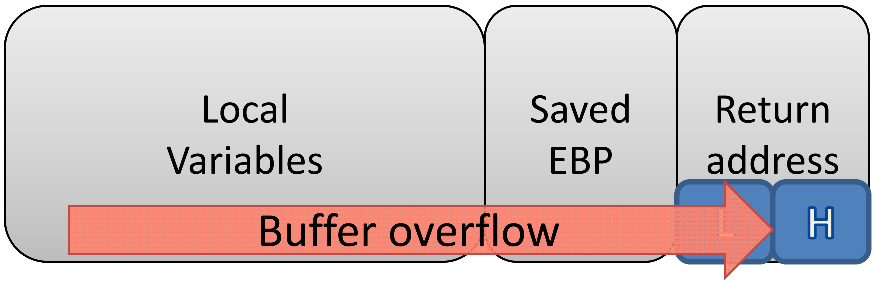
- Memory used to be very predictable
- A program would always load into the same position in memory
  - Performance, simplicity
- An attacker needs to know where in memory the malicious code is
- In Windows XP, this was simple: it was always in the same place

Mitigation: ASLR



- *Address Space Layout Randomization* (ASLR) [12]
  - Randomize where applications are placed in memory
  - Introduced in Vista/Server 2008, 8 bits of entropy
  - Images must be linked with `/DYNAMICBASE`

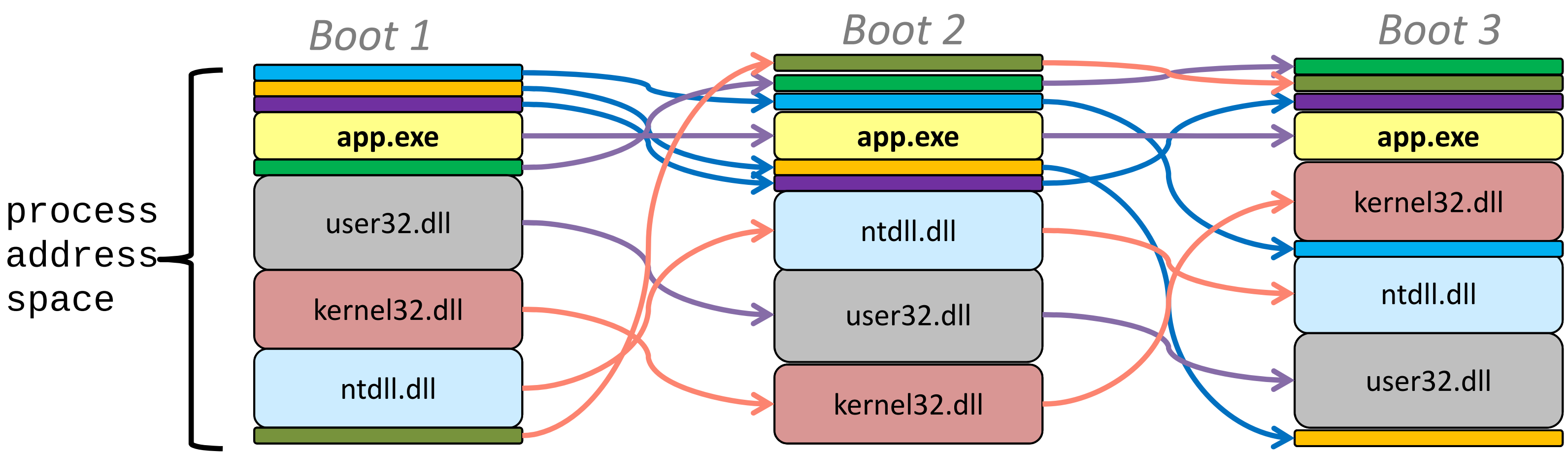
Exploit: Partial overwrite




```
memcpy(
   dest,       ← Stack buf
   src,        ← Controlled
   length);    ← Controlled
```

- Only the high-order two bytes are randomized in image mappings
  - The application moves around. Things within the application don't.
  - This works because the attacker has to hard-code the memory location – can't use relative locations. Except…
- Low-order two bytes can be overwritten to return into another location within a mapping
  - Overwriting `0x1446`**`047c`** with `0x1446`**`1846`**
- Only works with specific vulnerabilities that allow partial overwrites

Exploit: non-reloc/fixed executables



- Not all binaries are compiled with relocation information
  - Executables often don't have relocations (`/FIXED:YES`)
  - .NET IL-only assemblies in IE [13]
- ASLR is most effective if *all* regions are randomized

Exploit: Brute force

- DLLs are generally randomized once per-boot
  - Some attacks can be tried repeatedly
- Brute forcing addresses less likely on Windows
  - No “forking” daemons in Windows
  - Vista service restart policy limits number of times a service can crash and automatically restart

Exploit: Information disclosure

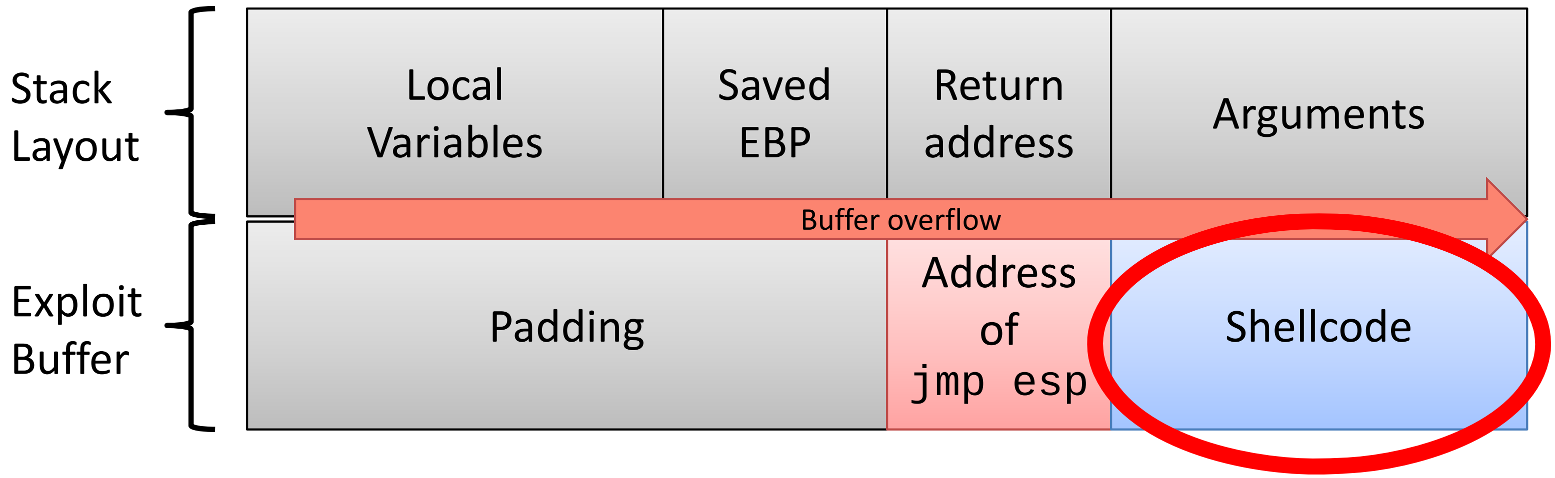
- Software bugs may leak address space information
  - Requires a second, lower severity vulnerability
- Can be used to construct reliable return addresses

Block the Malicious Code Itself

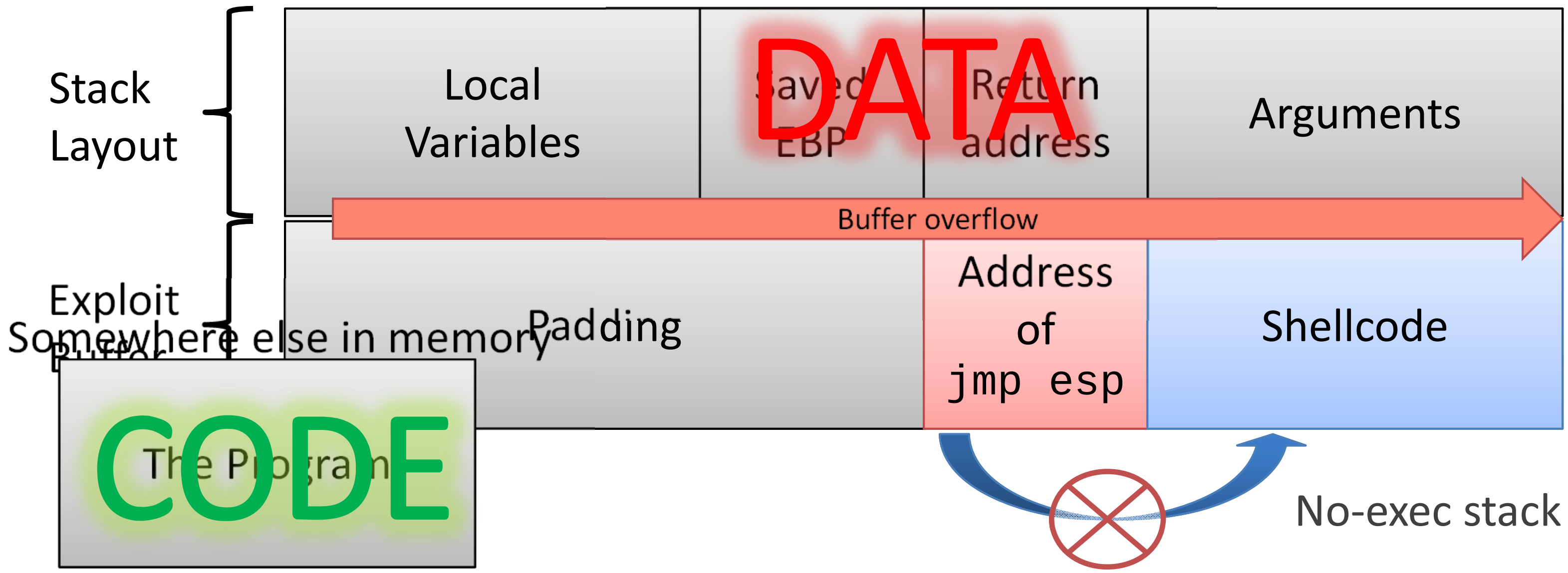
Problems if you're a bad guy

- ✓ Found a vulnerability
- ✓ Wrote an exploit
- ✓ Bypassed the other mitigations
- ? Where do I put my malicious code?
  - I'm already sending the user a malicious webpage
  - I'll store the code as text in that page!
    - That'll get it loaded in memory where I can jump to it

Exploit: Execute data as code

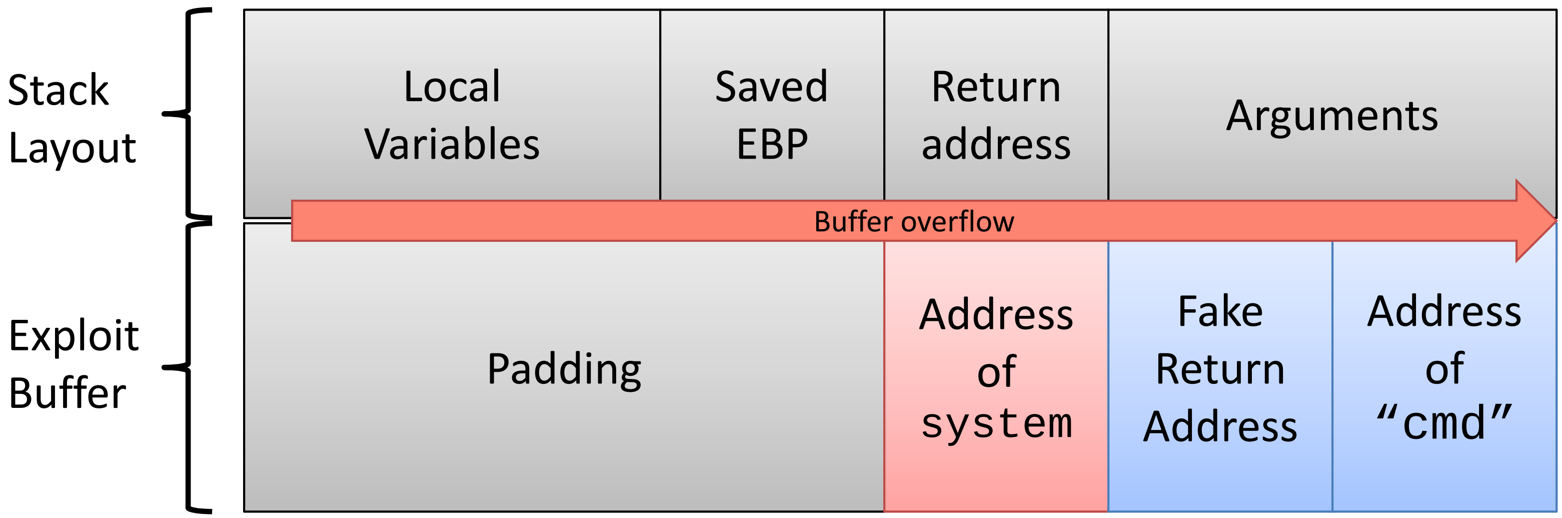


Mitigation: Hardware DEP (NX)



- Hardware-enforced DEP allows memory regions to be non-executable
  - Leverages NX features of modern processors
- Shellcode stored in these regions cannot be executed

Exploit: Ret2libc



- NX pages can prevent arbitrary code execution
- However, executable code in loaded modules can be abused[11]
  - Return into a library function with a fake call frame

Simplified Example

- Put the literal text “bash rm –rf” on the stack using your buffer overflow
- Set the return address to point to “exec”
- Result: the system API “exec” runs with the parameter “bash rm –rf”
  - “exec” simply runs the command line specified: bash rm –rf
    - This erases all files in the home directory
  - No attacker-provided code was executed!

Exploit: Disable DEP for a process

- There is an API `VirtualProtect` to change how a piece of memory is marked as code vs. data
  - Required for interpreters, compilers, etc.

- Abusing `VirtualProtect` requires the ability to use NULL bytes
  - Often impossible (string-related overflows)

- Windows has an API to disable NX for a process
  - `NtSetInformationProcess` [info class 0x22]

- Exploit can use ret2libc to return into this function and easily disable NX[3]

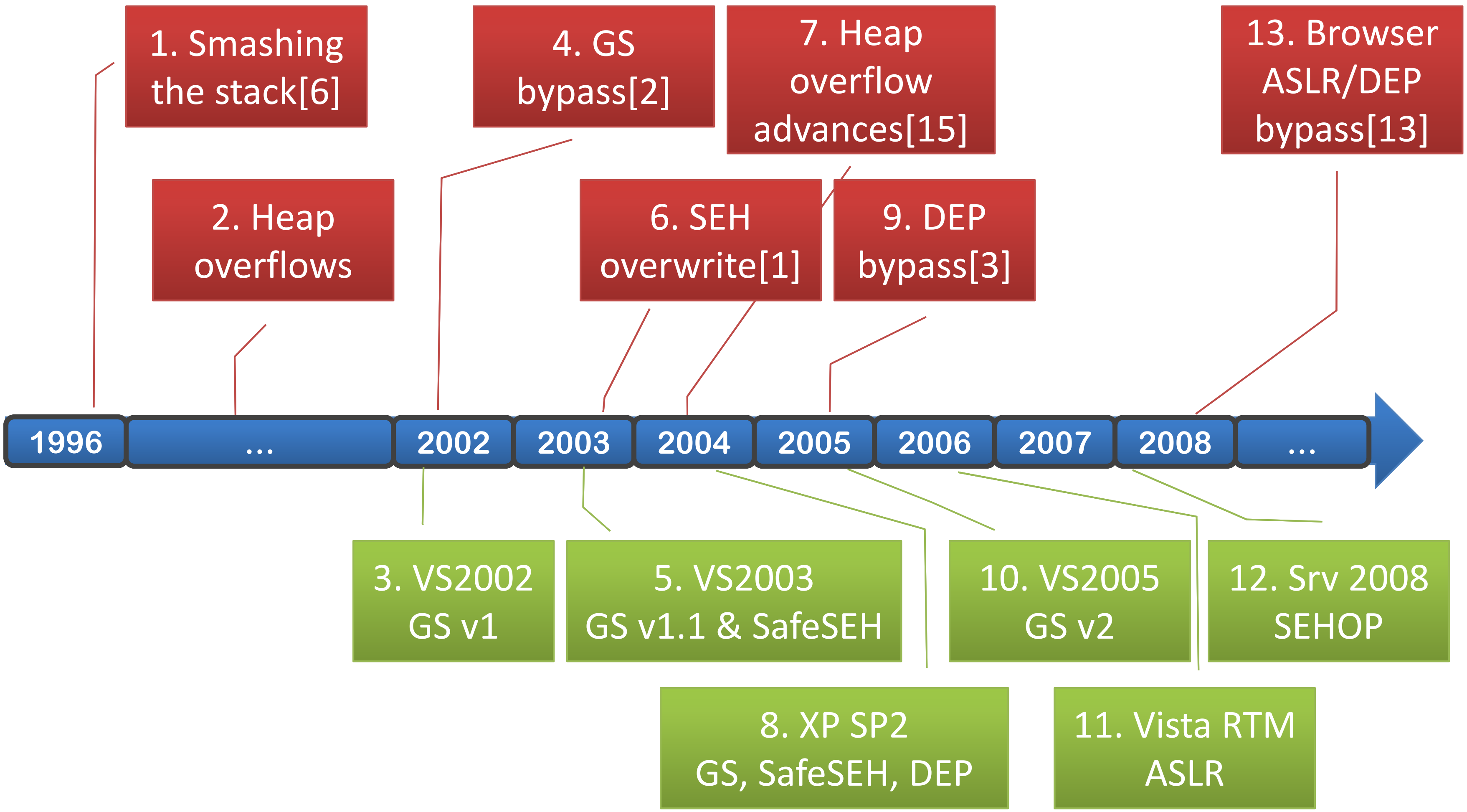
Mitigation: Permanent flag

- Boot flag can force all applications to run with NX enabled (AlwaysOn)`[10]`

- Processes can prevent future updates to execute flags
  - `NtSetInformationProcess`[22] with flag 0x8

- Does not mitigate return into `VirtualProtect`

Exploitation & Mitigation Chronology



References

[1] Litchfield, David. *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 20003 Server.* http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf.

[2] Ren, Chris et al. *Microsoft Compiler Flaw Technical Note*. http://www.cigital.com/news/index.php?pg=art&artid=70.

[3] skape, Skywing. *Bypassing Windows Hardware-enforced DEP*. http://www.uninformed.org/?v=2&a=4&t=sumry.

[4] skape. *Preventing the Exploitation of SEH Overwrites*. http://www.uninformed.org/?v=5&a=2&t=sumry.

[5] skape. *Reducing the Effective Entropy of GS Cookies*. http://www.uninformed.org/?v=7&a=2&t=sumry.

[6] Aleph1. *Smashing the Stack for Fun and Profit*. http://www.phrack.org/issues.html?issue=49&id=14#article.

[7] Microsoft. */GS Compiler Switch*. http://msdn2.microsoft.com/en-us/library/8dbf701c(VS.80).aspx.

[8] Whitehouse, Ollie. *Analysis of GS Protections in Microsoft Windows Vista.* http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf.

[9] Microsoft. */SAFESEH Compiler Switch*. http://msdn2.microsoft.com/en-us/library/9a89h429(VS.80).aspx.

[10] Microsoft. *A detailed description of DEP.* http://support.microsoft.com/kb/875352.

[11] Wikipedia. *Return-to-libc attack*. http://en.wikipedia.org/wiki/Return-to-libc_attack.

[12] Wikipedia. *Address Space Layout Randomization (ASLR)*. http://en.wikipedia.org/wiki/ASLR.

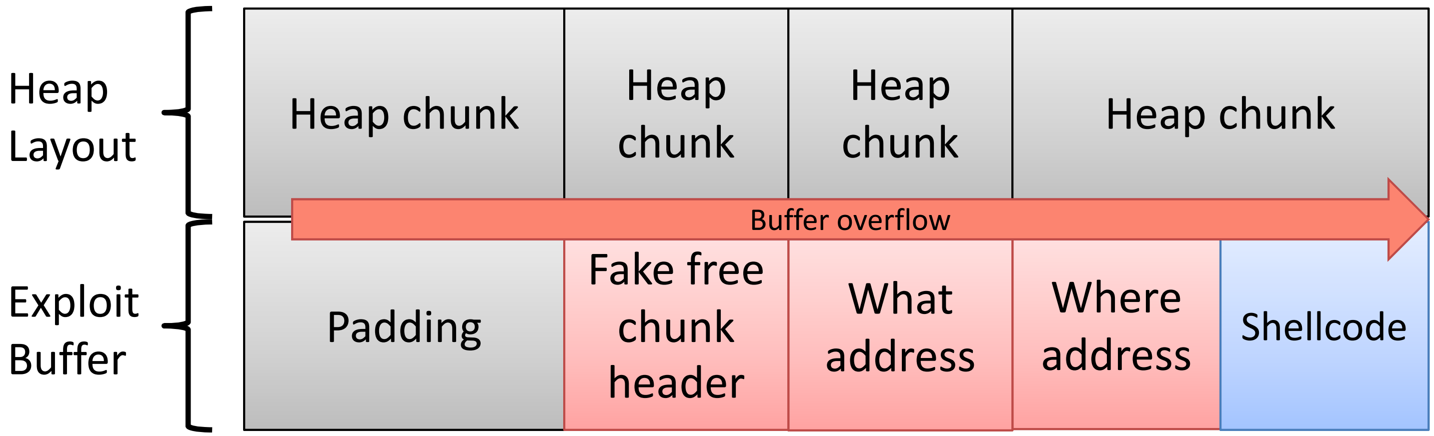
[13] Mark Dowd and Alex Sotirov. *Impressing girls with Vista memory protection bypasses*. http://taossa.com/index.php/2008/08/07/impressing-girls-with-vista-memory-protection-bypasses/.

[14] Johnson, Richard. *Windows Vista Exploitation Countermeasures.* http://rjohnson.uninformed.org/Presentations/200703%20EuSecWest%20-%20Windows%20Vista%20Exploitation%20Countermeasures/rjohnson%20-%20Windows%20Vista%20Exploitation%20Countermeasures.ppt

[15] Matt Conover and Oded Horovitz. *Windows Heap Exploitation*. http://ivanlef0u.free.fr/repo/windoz/heap/XPSP2%20Heap%20Exploitation.ppt

Extras

Exploit: Heap metadata overwrite



- Interesting things happen on the heap:
  - Heap coalesce i.e. defrag
  - Lookaside list allocation: The memory manager keeps a short list of fixed-size blocks to perform rapid allocations
- Corrupt the heap metadata and…

Mitigation: Heap hardening

- Safe unlinking during heap coalesce
  - List entry integrity verified prior to coalesce
- Heap cookies
  - 8-bit cookie verified on allocation from free list
- Heap chunk header encryption
  - Header fields are XOR'd with a random value

Exploit: Re-protect memory via ret2libc

Address of `VirtualProtect`	**Address of** `jmp esp`	**Address of** shellcode	**Size of** shellcode	**RWX**	**Writable address**	shellcode

Return from vulnerable function

Entry to `VirtualProtect`

Return from `VirtualProtect`

- Windows makes extensive use of `stdcall`
  - Caller pushes arguments
  - Callee pops arguments with `retn`

- Allows multiple functions to be changed with ret2libc

Exploit: Heap Spray/NOP Sled

- Attacker can’t predict where in memory the malicious payload will be
- Exploit:
  - Fill the entire memory space with “NOP” instructions
    - NOP = No Operation = Do nothing
  - Place malicious payload at end
  - Jump anywhere. Eventually you wind up at the payload.
- Can take a while to fill 8 GB of RAM
- Typically requires script

Mitigation: Heap Spray Protection

- Pre-allocate blocks throughout memory and fill them with exit instructions
- Makes it impossible to construct a continuous NOP sled