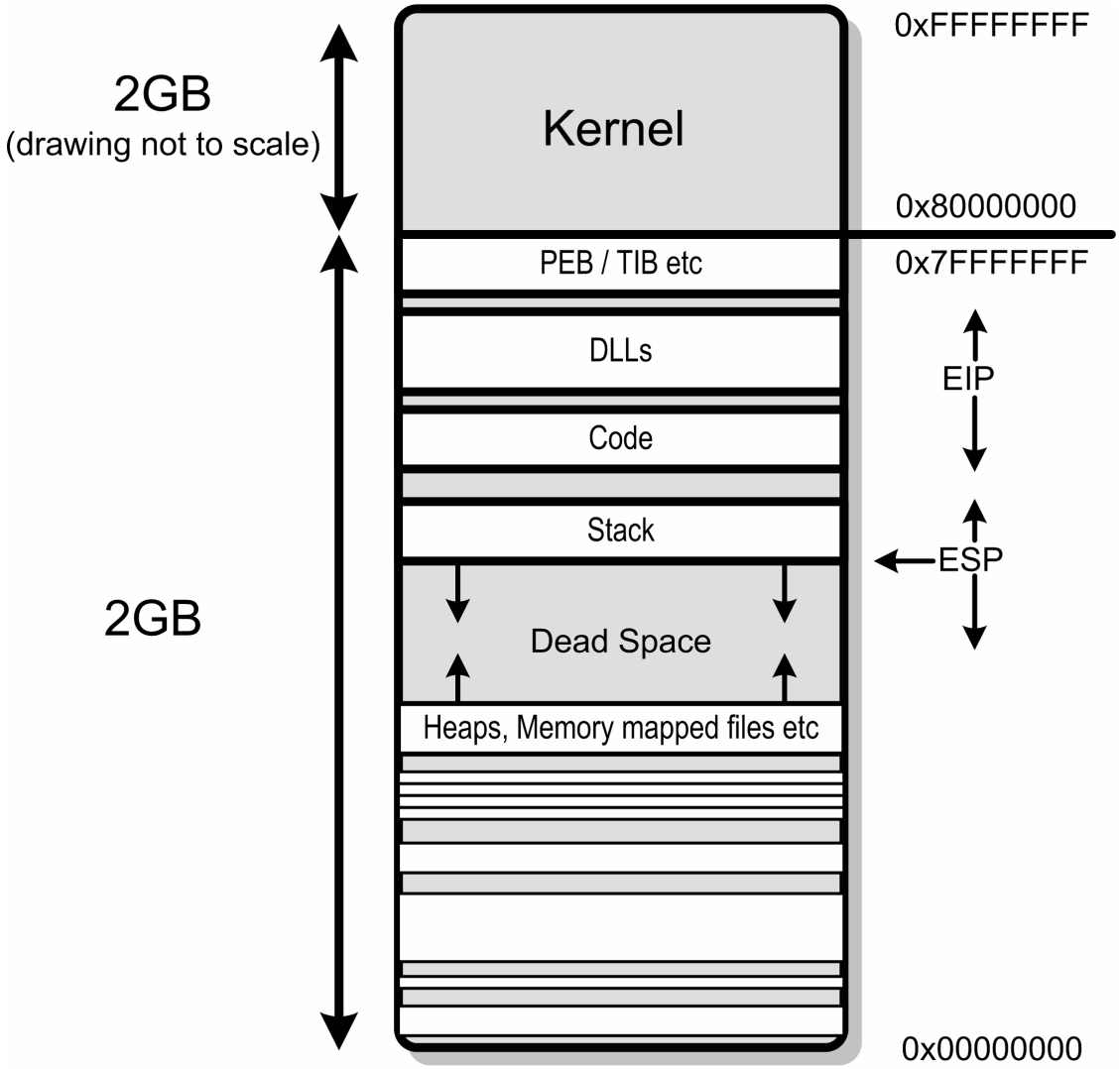
eEye Digital Security®

Beyond NX

An attacker's guide to Windows anti-exploitation technology

Ben Nagy
bnagy@eeye.com

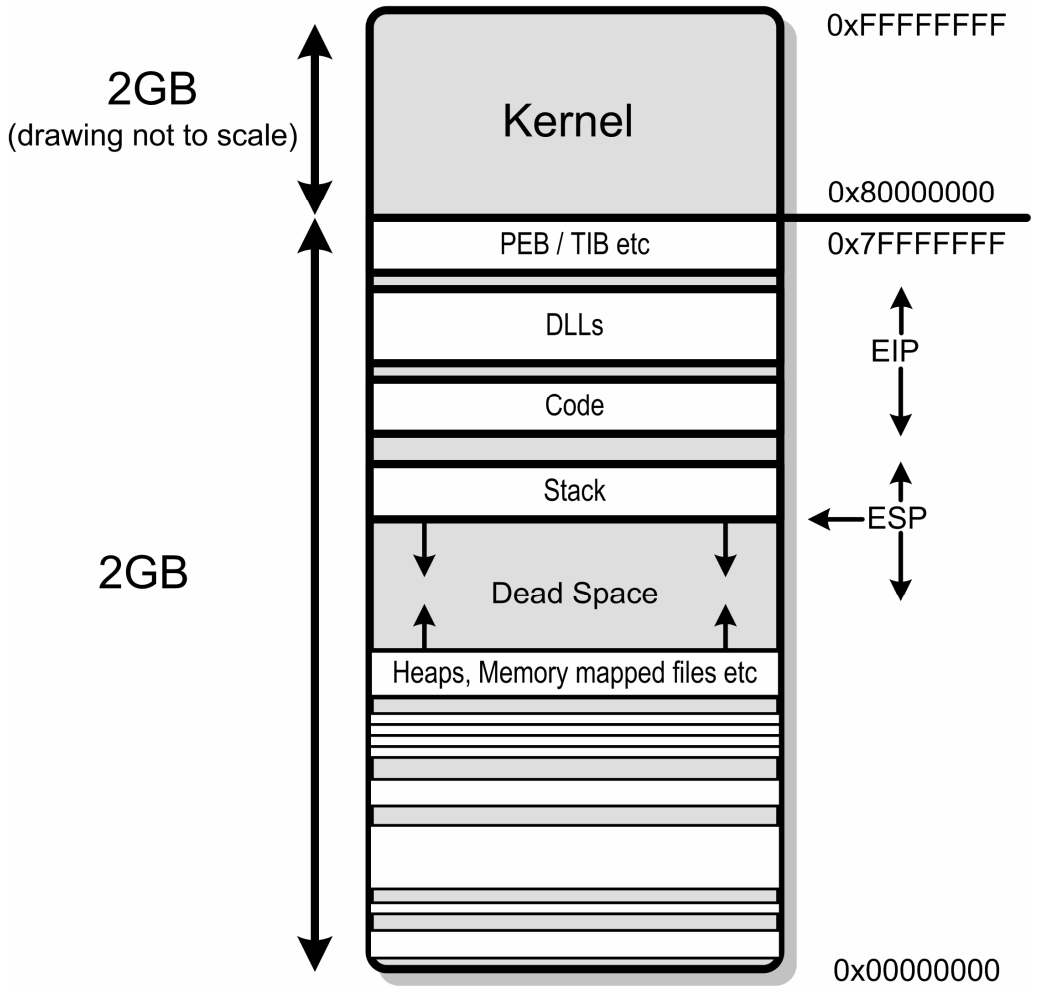
Basics – Windows Process Memory



How functions use the stack

Quick and dirty:

```
(CALL pushes EIP)
sub     esp, 28h
[do stuff]
add     esp, 28h
retn
```

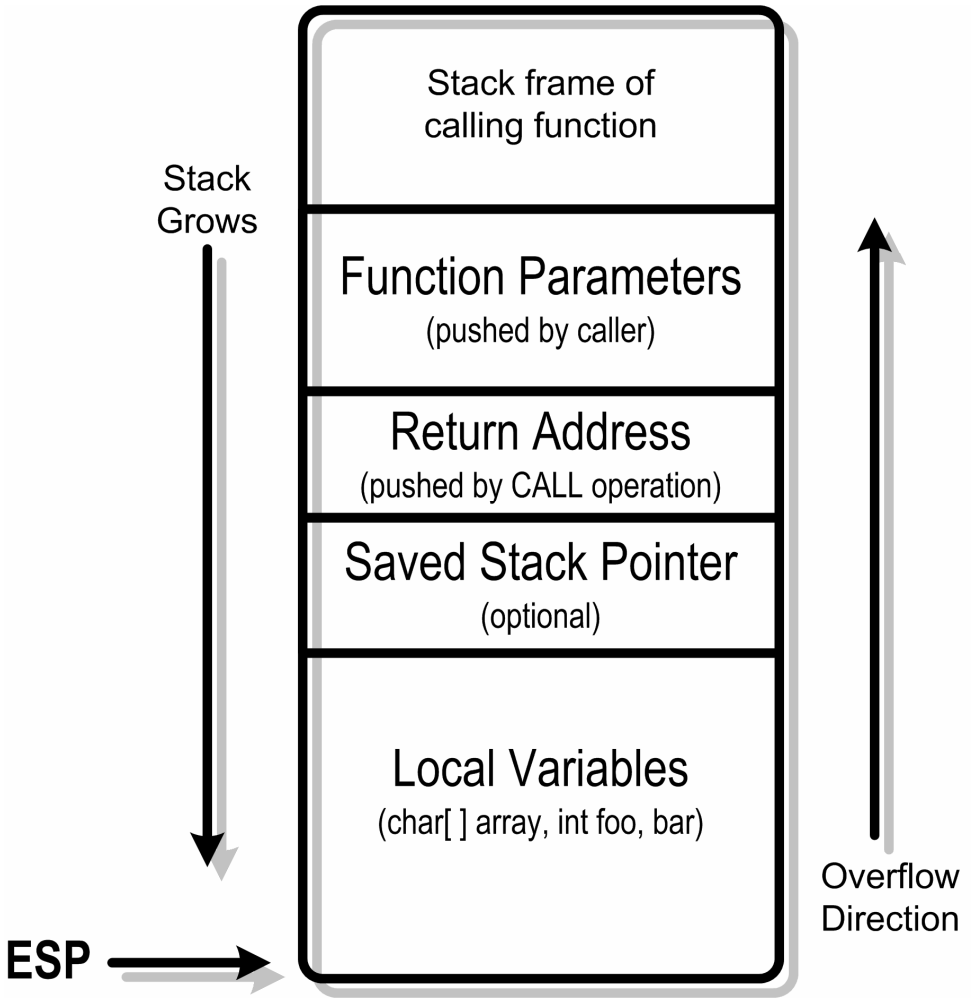


eEye Digital Security®

How functions use the stack

"Normal" Windows:

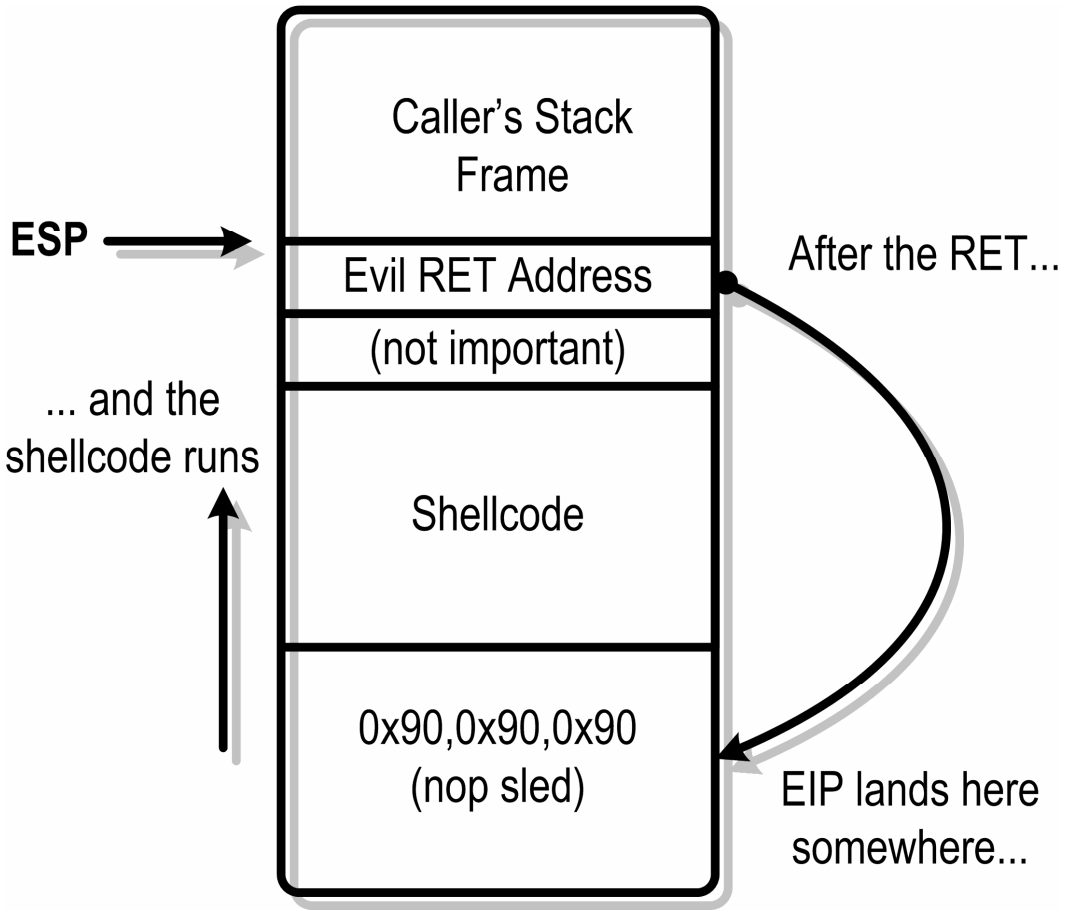
```
(CALL pushes EIP)
push    ebp
mov     ebp, esp
sub     esp, 18h
[do stuff]
add     esp, 18h
pop     ebp
retn    14h
```



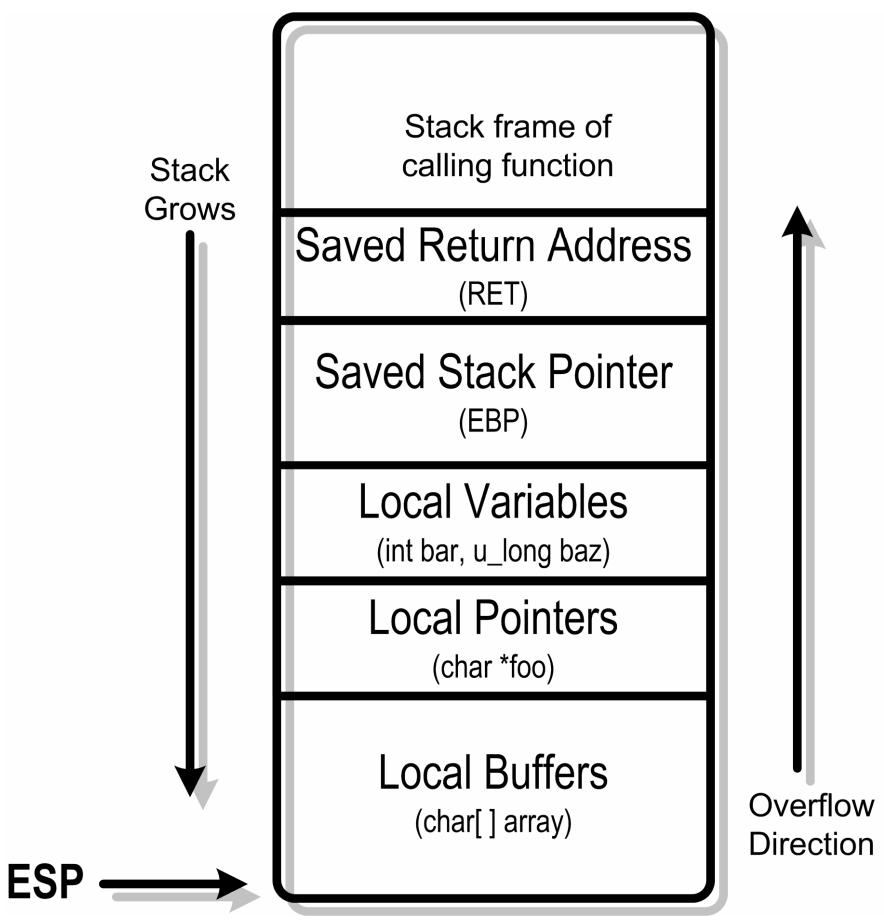
eEye Digital Security®

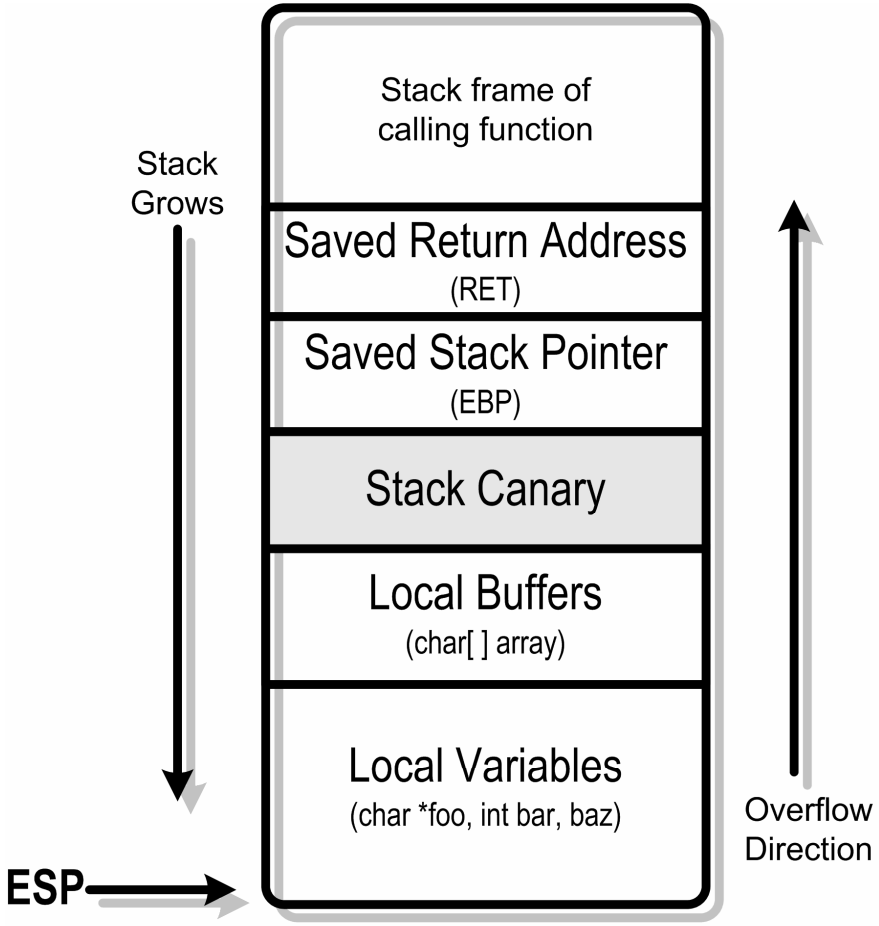
Standard, boring, buffer overflow...


```
(CALL pushes EIP)
push    ebp
mov     ebp, esp
sub     esp, 18h
[overflow happens here]
add     esp, 18h
pop     ebp
retn    14h
```



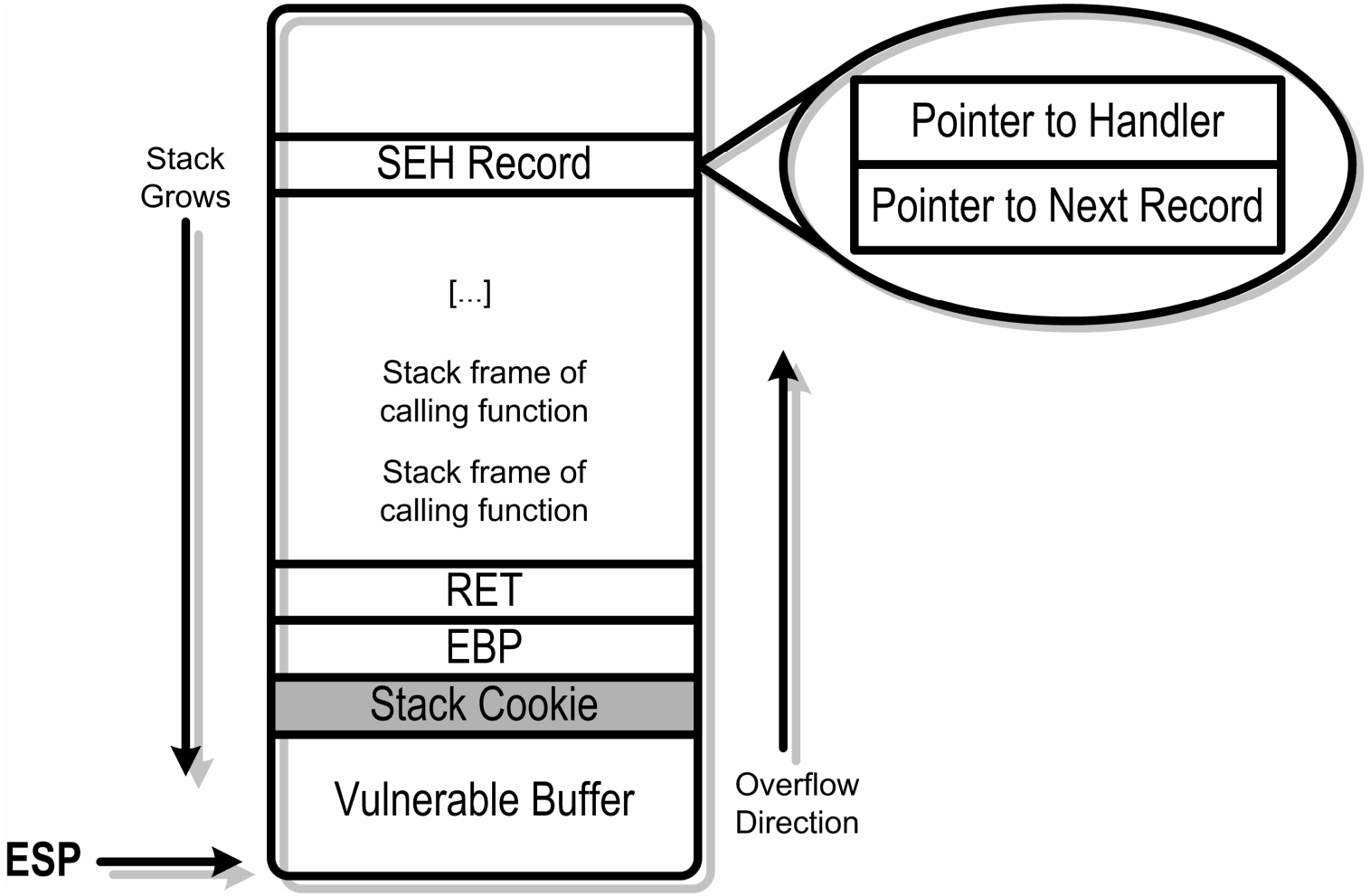
Windows Stack Protection





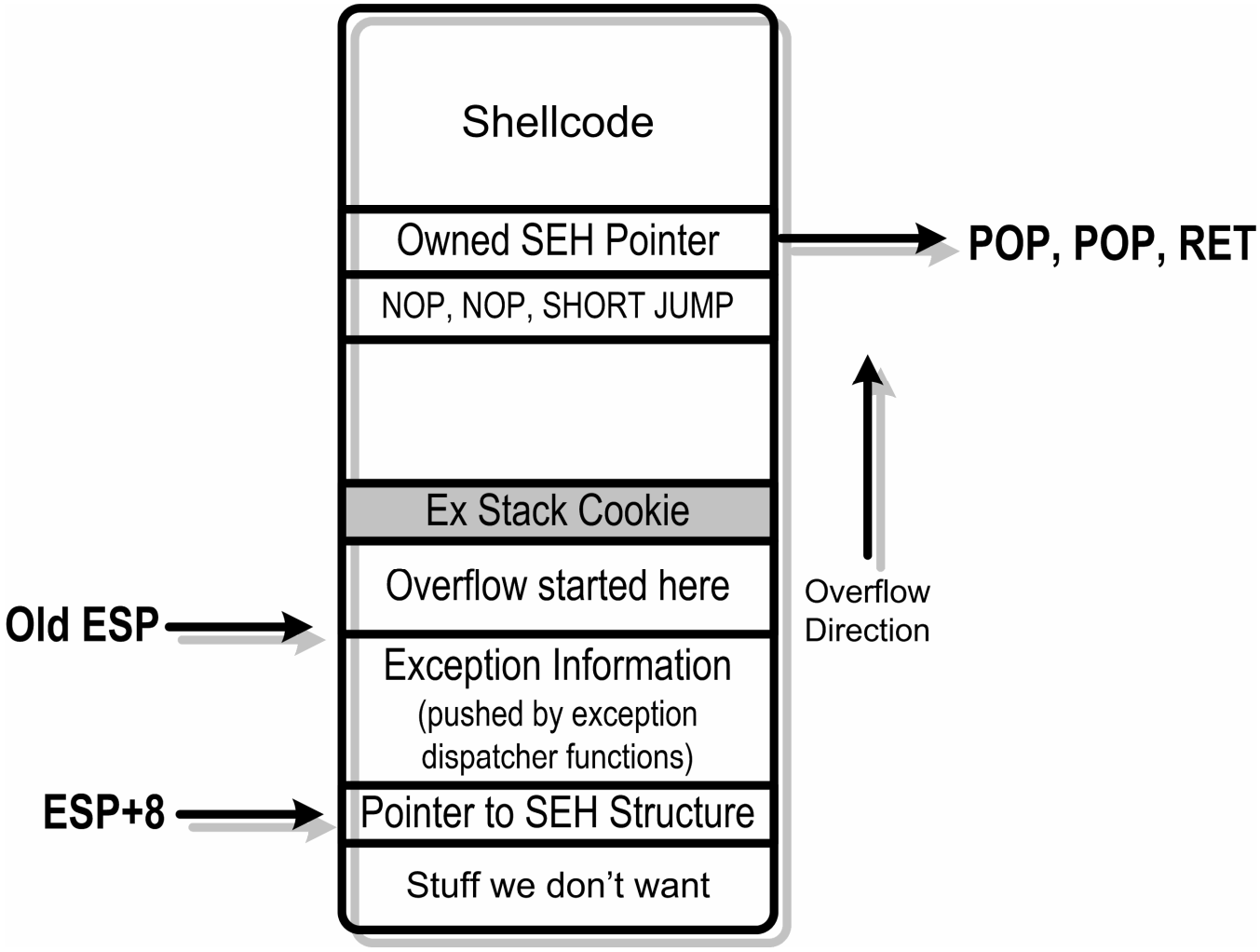
"Ideal Stack Layout"

Beating Windows Stack Protection



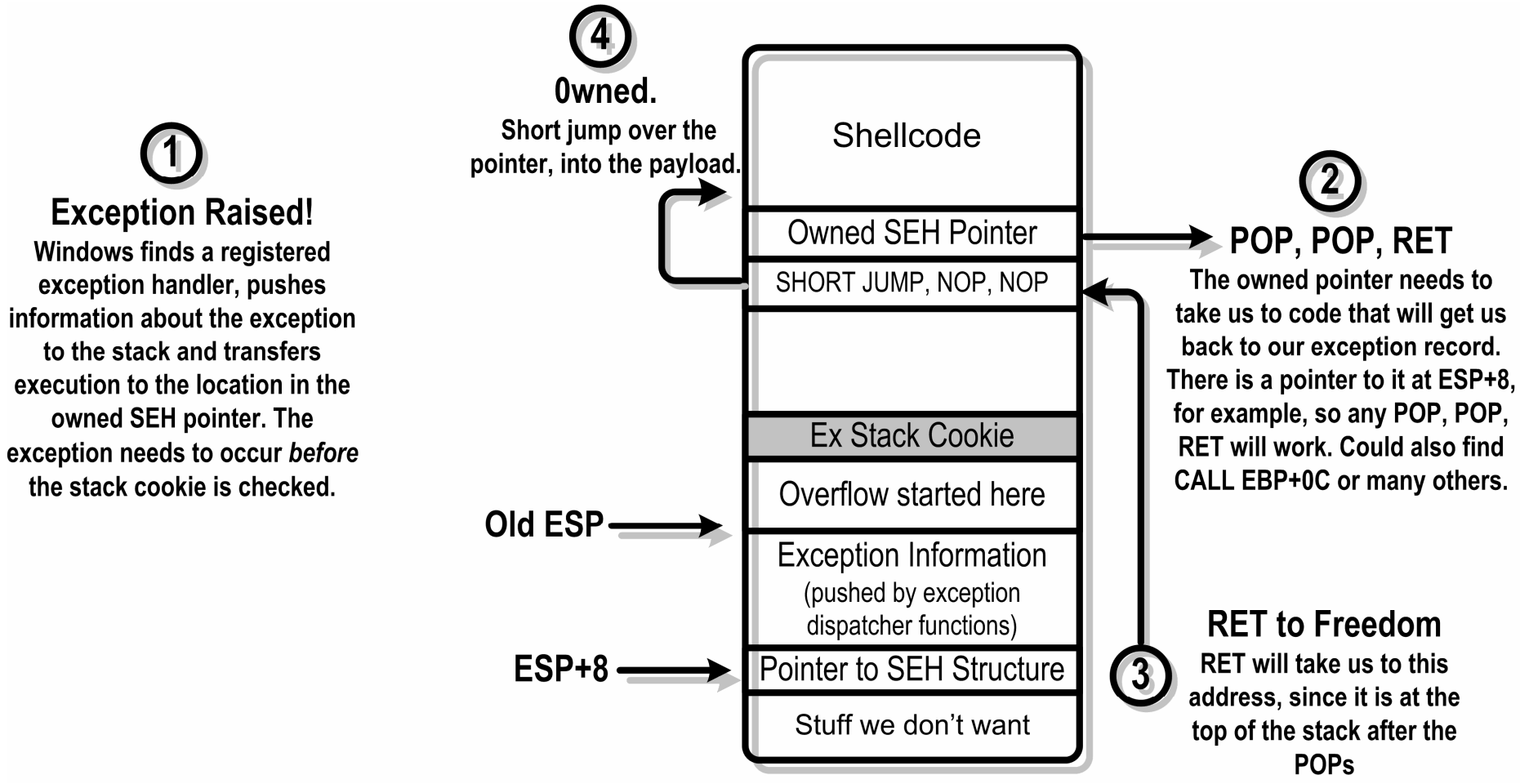
eEye Digital Security®

Beating Windows Stack Protection, Part II



eEye Digital Security®

Beating Windows Stack Protection, Part III



They fixed it. ☹

- New function RtlIsValidHandler() called during “raw” exception handling in NTDLL.DLL.
- New function called __ValidateEH3RN called during Visual C++ runtime library processing of exceptions (specific to VC, haven’t checked others)

RtlIsValidHandler pseudocode


```
if (SEHTable != NULL && SEHCount != 0) {
      if (SEHTable == -1 && SEHCount == -1) {
            // Managed Code but no SEH Registration table
            // or IMAGE_LOAD_CONFIG.DllCharacteristics == 4
            return FALSE;
      }
      if (&handler is registered) {
            return TRUE;
      else
            return FALSE;
      }
}
// otherwise...
if (&handler is on an NX page) {
      if (DEP is turned on) {
            bail(STATUS_ACCESS_VIOLATION);
      else
            return TRUE;
      }
}
if (&handler is on a page mapped MEM_IMAGE) {
// normally only true for executable modules
      if (SEHTable == NULL && SEHCount == 0) {
            return TRUE;
            // probably an old or 3rd party DLL
            // without SEH registrations
      }
      return FALSE // we should have caught this before
             // so something is wrong.
}
// Handler is on a eXecutable page, but not in module space
// Allow it for compatibility.
return TRUE;
```

__ValidateEH3RN, some highlights

__ValidateEH3RN is HUGE. I didn't reverse the whole thing, just enough to make me depressed.

1. Check to ensure scopetable array is not on the stack and that it is 4-byte aligned.
2. Sanity check on the array, made by walking the array from scopetable[0] to scopetable[trylevel].
3. Nested handlers also sanity checked in step 2, above. This means that any existing code being used as a fake scopetable entry needs to have previousTryLevel set to -1 (ie 0xFFFFFFFF preceding the payload address)
4. NtQueryVirtualMemory check on the scopetable against MEM_IMAGE and READONLY.
5. A lot of other code. Probably some kind of check against the lpfnFilter pointer itself

Some good references:

Pietrek, “A Crash Course on the Depths of Win32 Structured Exception Handling”
http://www.microsoft.com/msj/0197/exception/exception.aspx

HDM, Exploit for MS05-039
http://www.metasploit.com/projects/Framework/modules/exploits/ms05_039_pnp.pm

Litchfield, “Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server.”
http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf

Yours Truly, “Generic Anti-Exploitation Technology for Windows”
available at http://www.eeye.com/research/whitepapers

Adding NX

What it does:

Marks memory pages as non-executable at the paging level – which means it requires hardware support.

This is NOT the same as just calling VirtualProtect(), those settings mean nothing to the CPU

So, with an NX stack, we can't use any method that brings us back to a stack based payload.

Let's come back to this later...

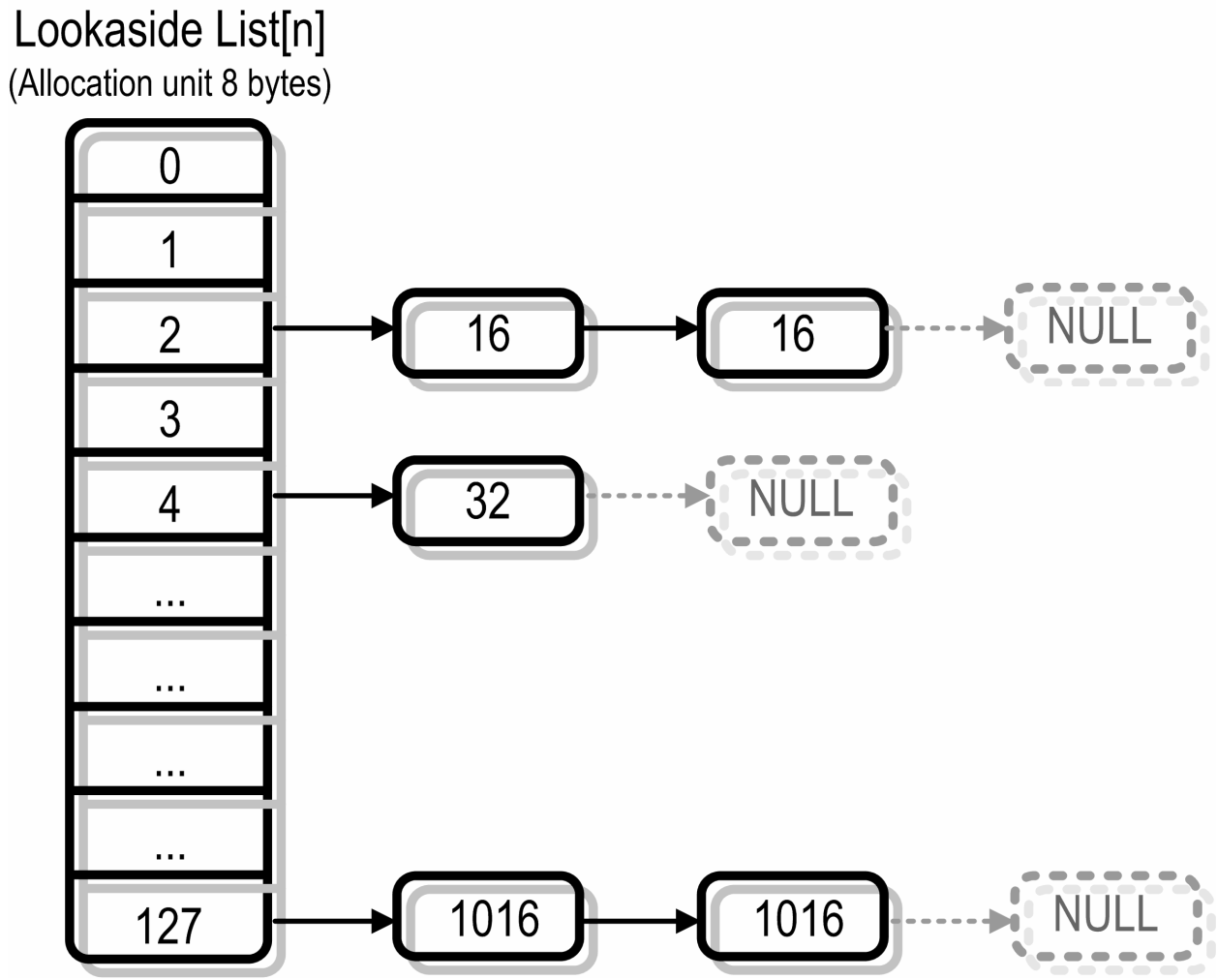
Heaps

Heap overflows are really hard.

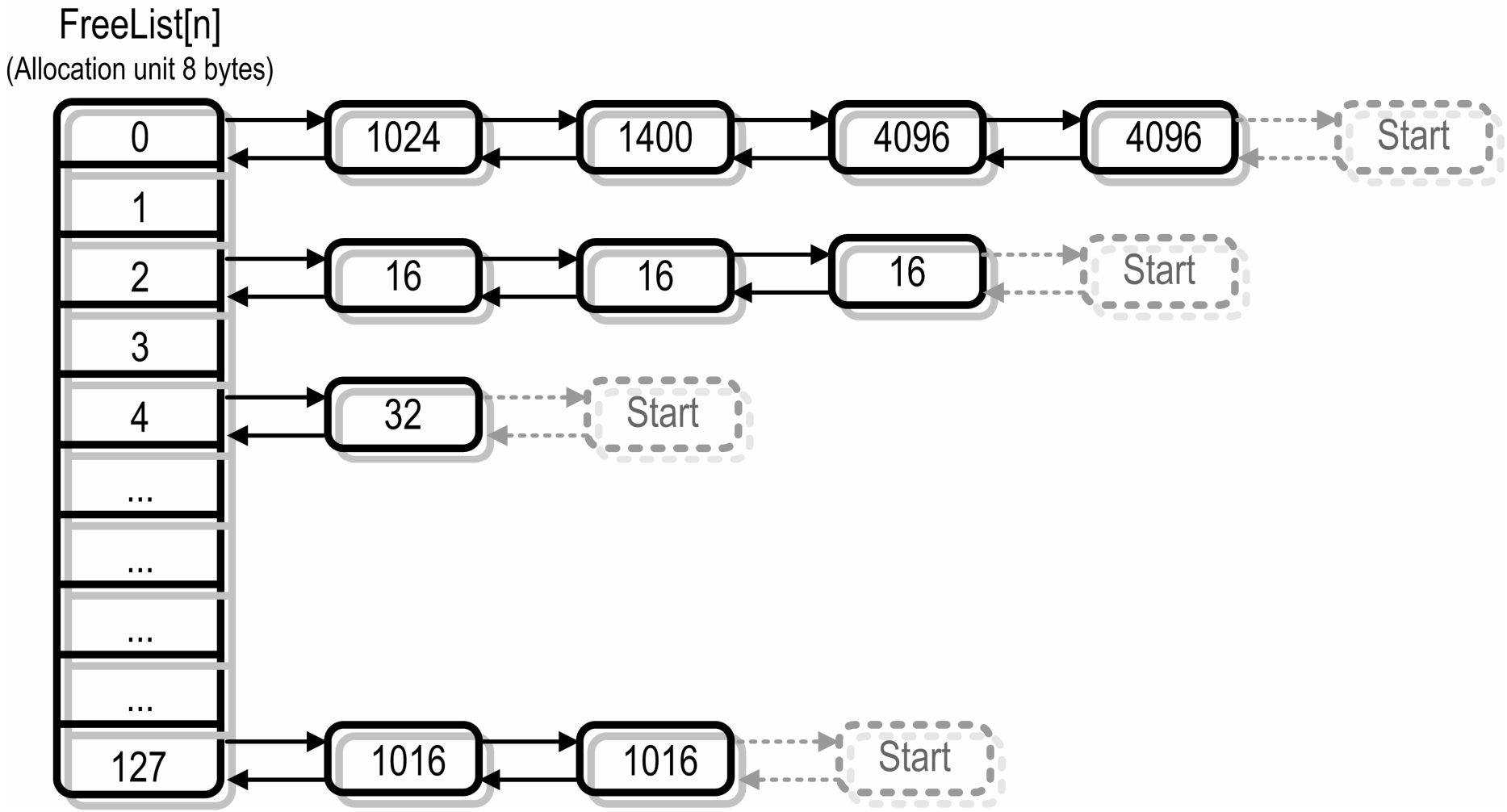
Post XPSP2 they become diabolical.

... but still possible.

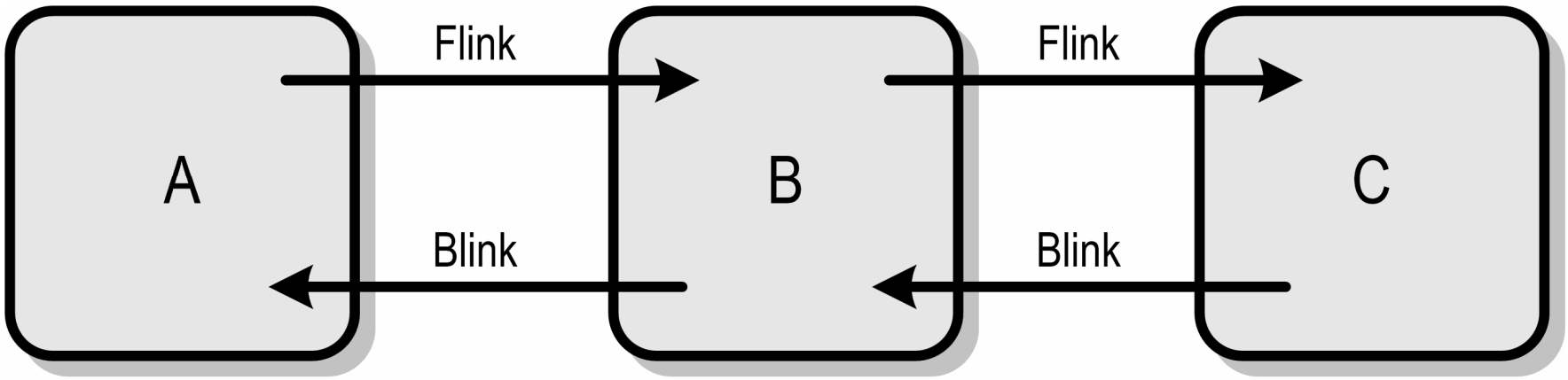
Heap Recap – Lookaside List



Heap Recap - Freelists

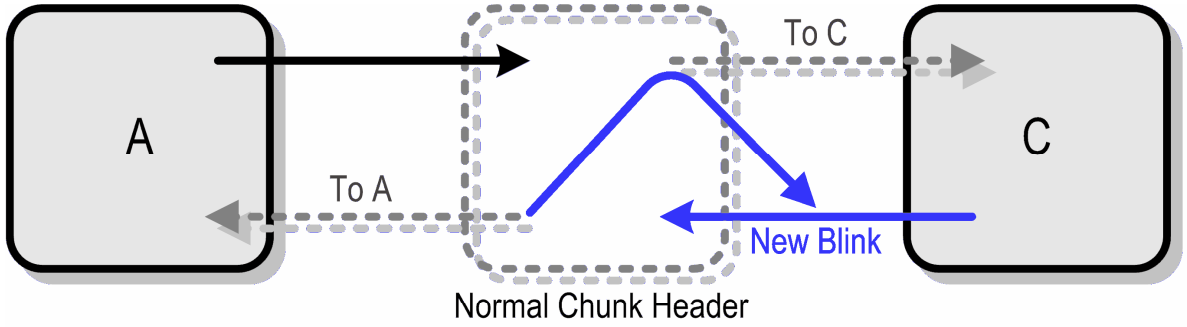


Heap Recap – Doubly Linked Lists



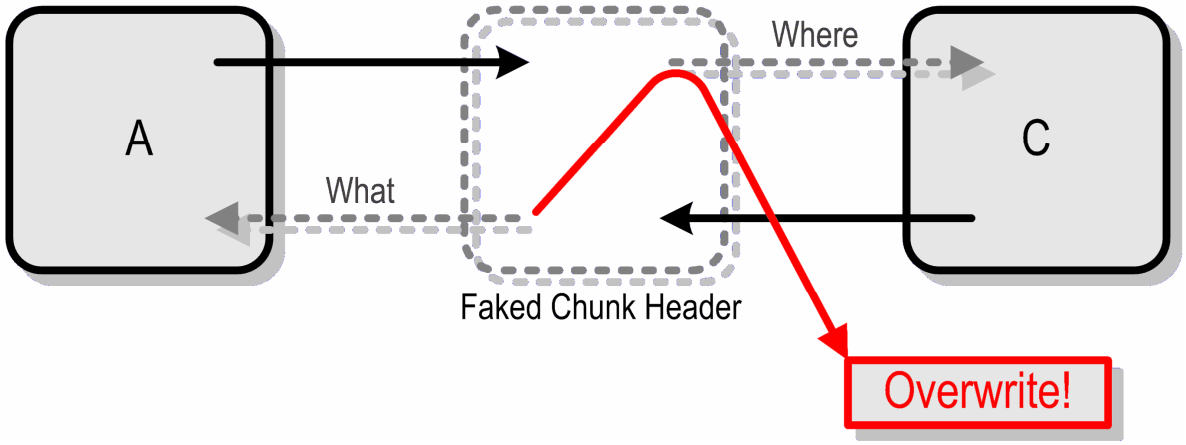
Doubly-Linked Free List

When Unlinking Macros Attack



mov Blink →[Flink]

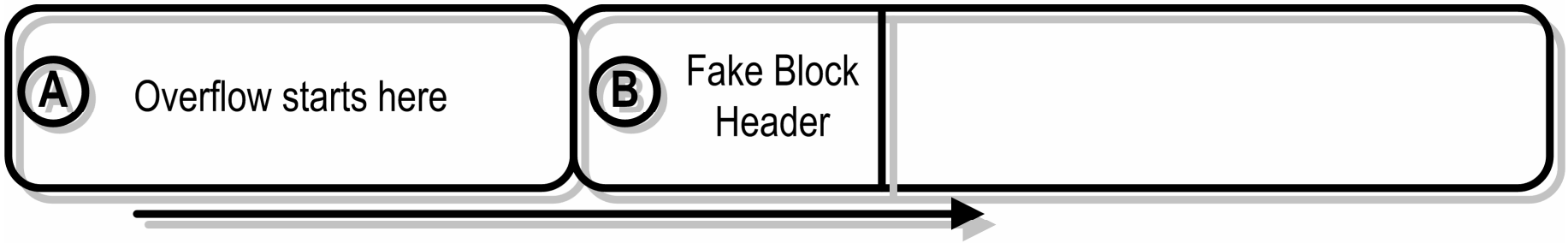
mov Flink →[Blink+4]



4-byte overwrite

Heap Overflows – The 4-byte Overwrite

Overflow past the end of current heap block, create adjacent fake block. Halvar used a fake VirtualAlloc header (fake busy block). Conover/Oded used fake free blocks, and waited for a heap coalesce.



Halvar's 4-byte Overwrite

Fake block contains VirtualAlloc headers. When Block B is freed, the prev and next VirtualAlloc blocks need to get updated, which means a linked list pointer update, which means 4 byte overwrite.

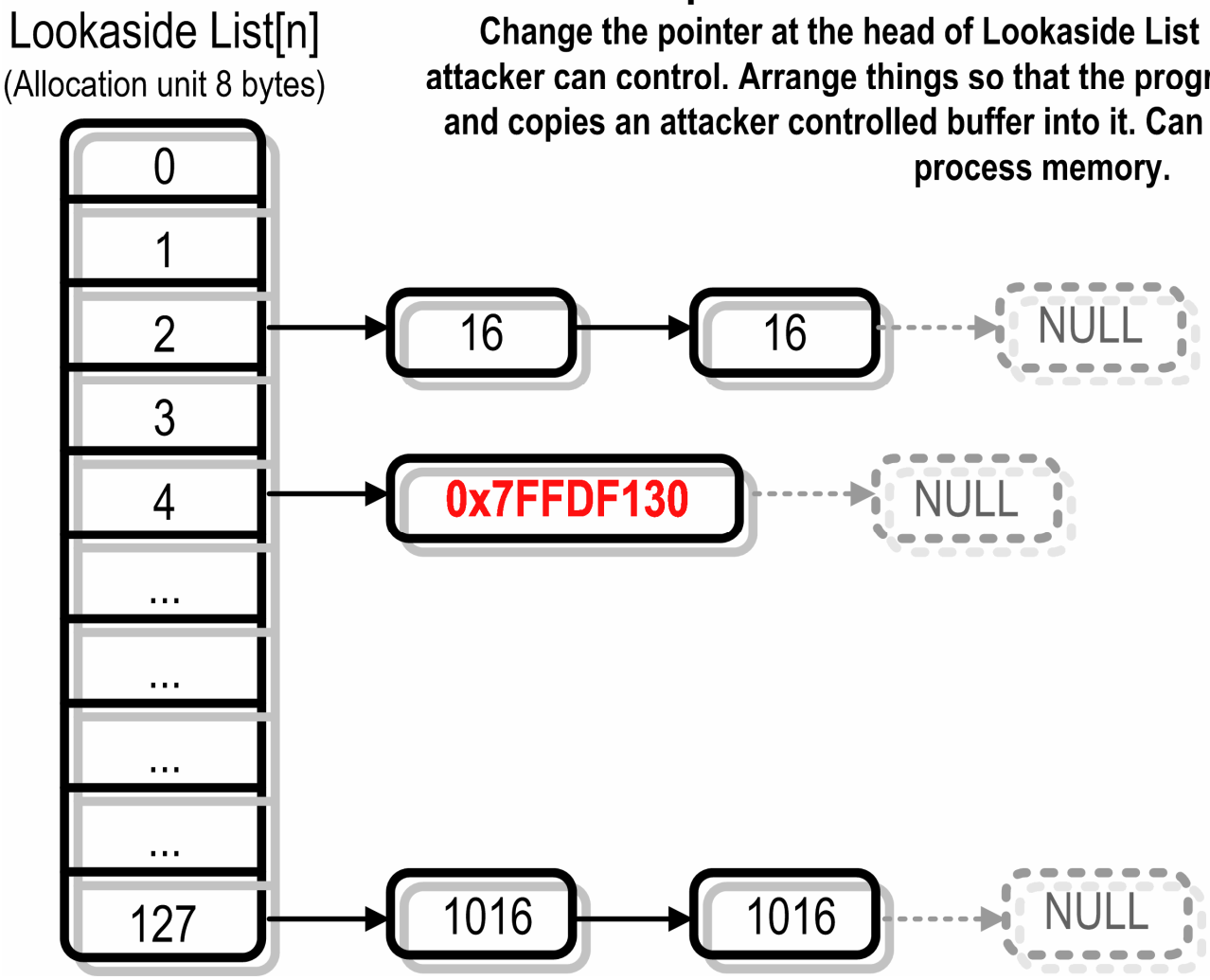
Coalesce on free 4-byte Overwrite

Fake block has a free header. When Block A is freed, RtlFreeHeap sees two adjacent free blocks and wants to coalesce them. Before it does that it needs to remove Block B from its Freelist, which means a linked list pointer update, which means 4 byte overwrite

4-to-n-byte overwrite

Heap Overflows – The 4-to-n-Byte Overwrite

Change the pointer at the head of Lookaside List [n] to point somewhere the attacker can control. Arrange things so that the program allocates a block of size n, and copies an attacker controlled buffer into it. Can overwrite up to 1016 bytes of process memory.



4-byte Overwrite – Then What?

Pre XPSP2 / 2003SP1

1. Replace a pointer with location of shellcode

- UEF, VEH, FastPebLock/Unlock (0x7ffdf020/4)

2. Copy shellcode somewhere stable

- PEB, Heap (many copies), Stack

3. ???

4. Profit!

Heap Protection

8-bit heap header cookie

- Checked on allocate and removal from freelist

Safe Unlinking Check for Doubly Linked Lists

- (B→Flink)→Blink == B && (B→Blink)→Flink == B

PEB Randomisation

Use of RtlEncodePointer for UEF and VEH

Removal of FastPebLockRoutine pointers from PEB

- (Win2k3) only

Attacking Heap Protection

2 Main Attacks

Unsafe Unlinking (Conover)

- Not even going to *try* to explain this.

Chunk on Lookaside (Conover / Anisimov)

- Overflow a chunk which is on a lookaside list
- On the second alloc, malicious Flink is returned
- Up to you how to provoke the copy and control transfer
- Should work for multi-shot vulnerabilities... eventually

Attacking Heap Protection

Some good references:

Halvar Flake, "Third Generation Exploitation“
http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt

David Litchfield, "Windows Heap Overflows"
http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt

**Matt Conover, Oded Horowitz, "Reliable Windows Exploits"
http://cansecwest.com/csw04/csw04-Oded+Connover.ppt**

Alexander Anisimov, "Defeating Windows XP SP2 Heap protection and DEP bypass“
http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf

funnywei & jerry, “Windows Xp Sp2□□□□”
http://www.xfocus.net/articles/200412/762.html

Heap Protection, Summary

4-byte overwrites getting much harder to provoke

- Safe unlinking check
- Heap Cookies

Even if we can provoke them, what pointer to attack?

- No more 1st Vectored Exception Handler (encoded)
- No more Unhandled Exception Filter (encoded)
- No more PebLockRoutine (Win2k3) or...
- PEB Randomised (XPSP2)
- SystemDirectory pointer in kernel32.dll? (Litchfield)

Heap Protection, Summary

Future Outlook is Worse

- Low Fragmentation Heap, 32-bit security cookie

Other approaches are needed...

Heap Spray (not really a heap overflow)

- Perfect example is InternetExploiter (SkyLined)
- Allocates many heap blocks like [nop][nop][...][shellcode]
- Land “somewhere” in the heap

Find “Interesting Things” on the heap

- Critical Section Linked List? (Falliere, Sep 2005)
- Application Specific, GDI objects, class destructors, etc etc

Back to NX

Normally, you would use ret-libc

Problems:

- Can't RET without bouncing via SEH (stack cookie)
- SEH is fixed now.
- PAGE_EXECUTE_READWRITE → 0x00000040
- Bottom of the stack is full of exception rubbish

Possible Solutions

- Overwrite the stack using chunk-on-lookaside, ret-libc (Anisimov)
- faultrep.dll and SystemDirectory pointer in kernel32.dll (Litchfield)
- Get your code into an eXecutable segment (ie a 2 step process)

Summary

Protection Mechanism	Applies	Focus
Stack Cookies	Per App	Detect Attack
Stack Layout Optimisation	Per App	Complicate Exploitation
Heap Cookies	Global	Detect Attack
Safe Unlinking	Global	Detect Attack
PEB Randomisation (XP)	Global	Complicate Exploitation
Remove Pointers in PEB (2K3)	Global	Complicate Exploitation
Pointer Encoding, UEF, VEH	Global	Complicate Exploitation
NX (Hardware DEP)	Configurable	Detect Attack
Safe SEH	Per App	Complicate Exploitation
Generic SEH Improvements	Global	Detect Attack

Summary, II

All protections enabled, no NX memory.

Stack

- I don't know.

Heap

- Tricky...

Other

- Things like dirty reads are still exploitable (eg IE Window() 0day, COM+ Object Instatiation Bug)

Summary, III

All protections enabled, including NX memory.

Stack

- I still don't know.

Heap

- Still tricky, but not much trickier than before.

Other

- Check out the IE Window() 0day – the dirty read is from a mapped shared segment, which is mapped as ... RX!

Questions?