Practical Return-Oriented Programming

Dino A. Dai Zovi
Funemployed Security Researcher
ddz@theta44.org / @dinodaizovi
http://www.theta44.org / http://blog.trailofbits.com

Why am I here?

- Show the practical applications of return-oriented programming to exploitation of memory corruption vulnerabilities
  - "Preventing the introduction of malicious code is not enough to prevent the execution of malicious computations"[1]
- Demonstrate that while exploit mitigations make exploitation of many vulnerabilities impossible or more difficult, they do not prevent all exploitation
  - Modern computing needs more isolation and separation between components (privilege reduction, sandboxing, virtualization)
  - The user-separation security model of modern OS is not ideally suited to the single-user system
  - Why do all of my applications have access to read and write all of my data?

1. "The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)", Hovav Shacham (ACM CCS 2007)

Agenda

- Current State of Exploitation
- Return-Oriented Exploitation
- Bypassing Permanent DEP
- Exploiting IE "Aurora" Vulnerability on Windows 7
  - MS10-002 / CVE-2010-0249
- Borrowed Instructions Synthetic Computer (BISC)
- Conclusions

Current State of Exploitation

A Brief History of Memory Corruption

- Morris Worm (November 1988)
  - Exploited a stack buffer overflow in BSD in.fingerd on VAX
  - Payload issued execve(“/bin/sh”, 0, 0) system call directly
- Thomas Lopatic publishes remote stack buffer overflow exploit against NCSA HTTPD for HP-PA (February 1995)
- “Smashing the Stack for Fun and Profit” by Aleph One published in Phrack 49 (August 1996)
- Researchers find and exploit stack buffer overflows in a variety of Unix software throughout the late 90’s
- Many security experts thought (incorrectly) that stack buffer overflows were the only exploitable problem

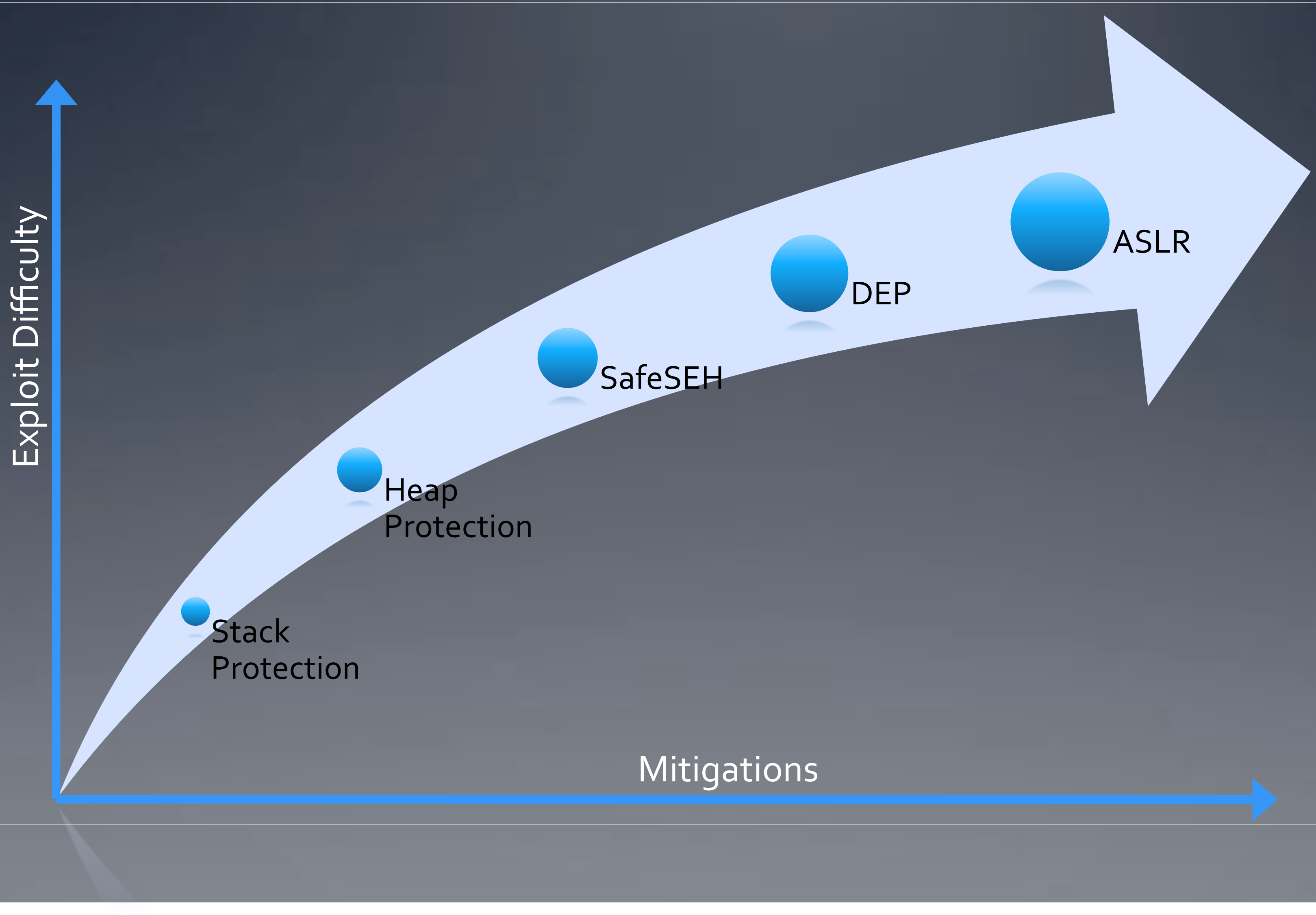
A Brief History of Memory Corruption

- "JPEG COM Marker Processing Vulnerability in Netscape Browsers" by Solar Designer (July 2000)
  - Demonstrates exploitation of heap buffer overflows by overwriting heap free block next/previous linked list pointers
- Apache/IIS Chunked-Encoding Vulnerabilities demonstrate exploitation of integer overflow vulnerabilities
  - Integer overflow => stack of heap memory corruption
- In early 2000's, worm authors took published exploits and unleashed worms that caused widespread damage
  - Exploited stack buffer overflow vulnerabilities in Microsoft operating systems
  - Results in Bill Gates' "Trustworthy Computing" memo
- Microsoft's Secure Development Lifecycle (SDL) combines secure coding, auditing, and exploit mitigation

Exploit Mitigation

- Patching every security vulnerability and writing 100% bug-free code is impossible
  - Exploit mitigations acknowledge this and attempt to make exploitation of remaining vulnerabilities impossible or at least more difficult
- Windows XP SP2 was the first widespread operating system to incorporate exploit mitigations
  - Protected stack metadata (Visual Studio compiler /GS flag)
  - Protected heap metadata (RtlHeap Safe Unlinking)
  - SafeSEH (compile-time exception handler registration)
  - Software, Hardware-enforced Data Execution Prevention (DEP)
- Windows Vista implements Address Space Layout Randomization (ASLR)
  - Invented by and first implemented in PaX project for Linux

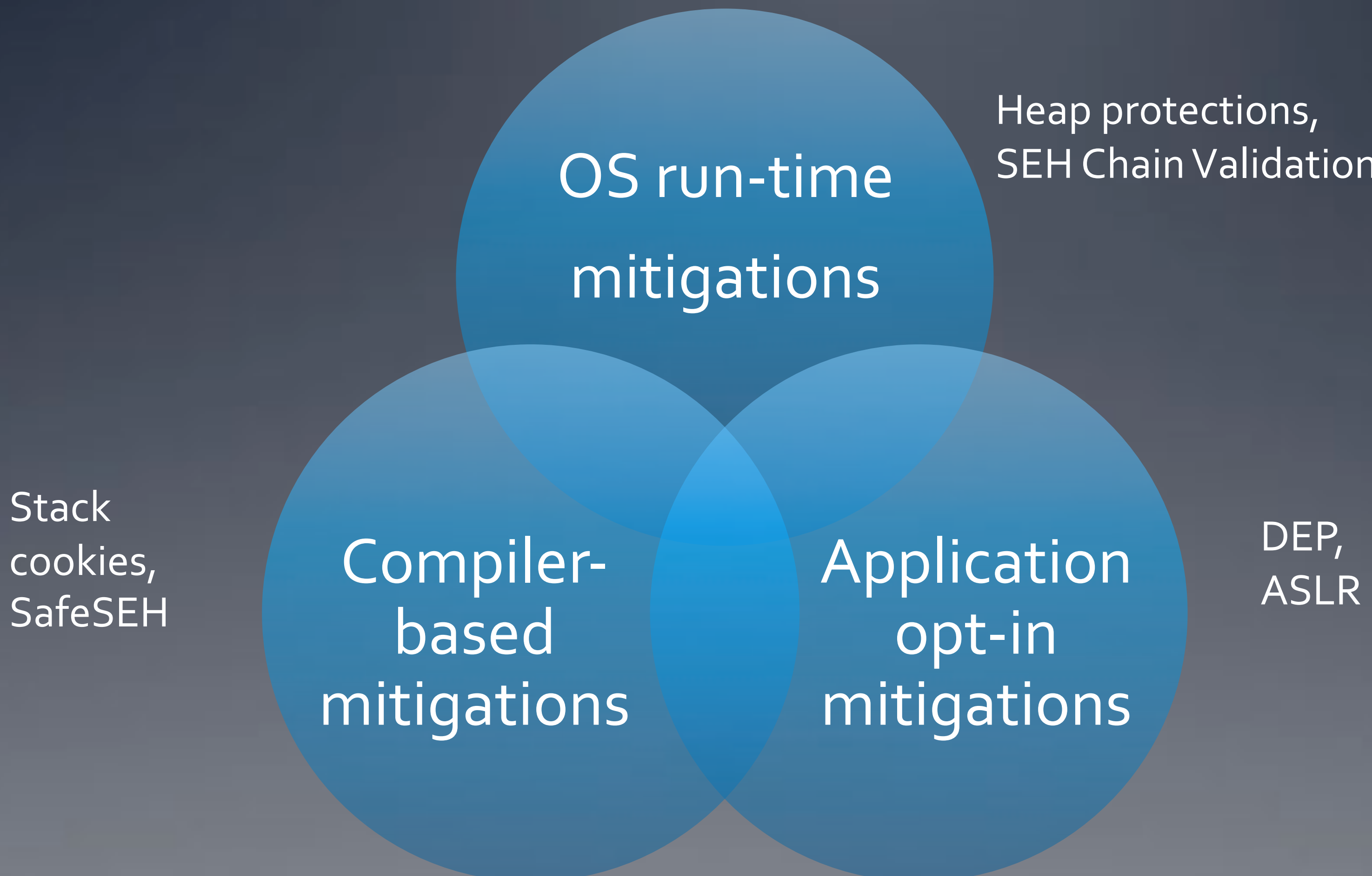
Mitigations Make Exploitation Harder



Exploit techniques Rendered Ineffective

- Stack return address overwrite
- SEH frame overwrite
- Heap free block metadata overwrite
- Application-specific data
- ???

Mitigations requires OS, Compiler, and Application Participation and are additive



What mitigations are active in my app?

- It is difficult for even a knowledgeable user to determine which mitigations are present in their applications
  - Is the application compiled with stack protection?
  - Is the application compiled with SafeSEH?
  - Do all executable modules opt-in to DEP (NXCOMPAT) and ASLR (DYNAMICBASE)?
  - Is the process running with DEP and/or Permanent DEP?
- Internet Explorer 8 on Windows 7 is 100% safe, right?
  - IE8 on Windows 7 uses the complete suite of exploit mitigations
  - … as long as you don't install any 3rd-party plugins or ActiveX controls
- What about Adobe Reader?
  - You don't want to know…

Return-Oriented Exploitation

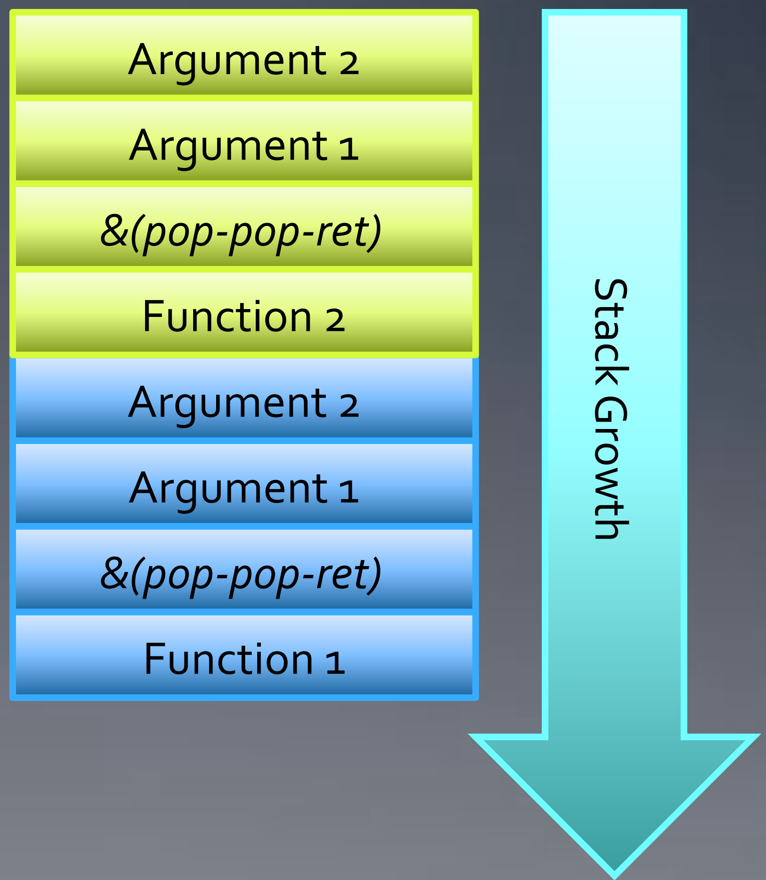
Return-to-libc

- Return-to-libc (ret2libc)
  - An attack against non-executable memory segments (DEP, W^X, etc)
  - Instead of overwriting return address to return into shellcode, return into a loaded library to simulate a function call
  - Data from attacker's controlled buffer on stack are used as the function's arguments
  - i.e. call system(*cmd*)

"Getting around non-executable stack (and fix)", Solar Designer (BUGTRAQ, August 1997)

Return Chaining

- Stack unwinds upward
- Can be used to call multiple functions in succession
- First function must return into code to advance stack pointer over function arguments
  - i.e. pop-pop-ret
  - Assuming cdecl and 2 arguments

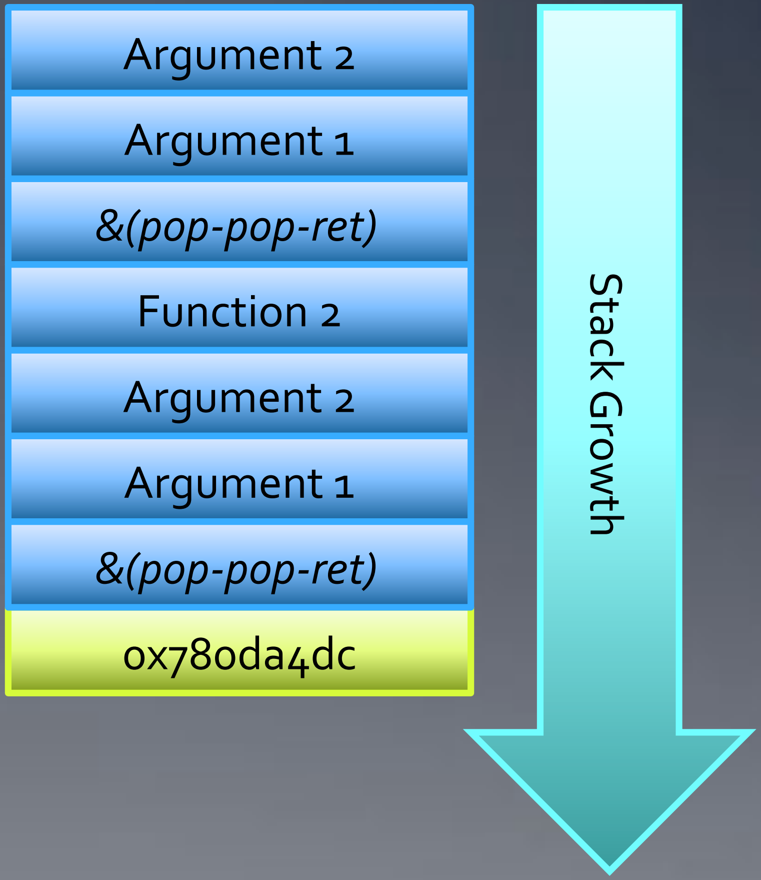


Return Chaining

```
0043a82f:

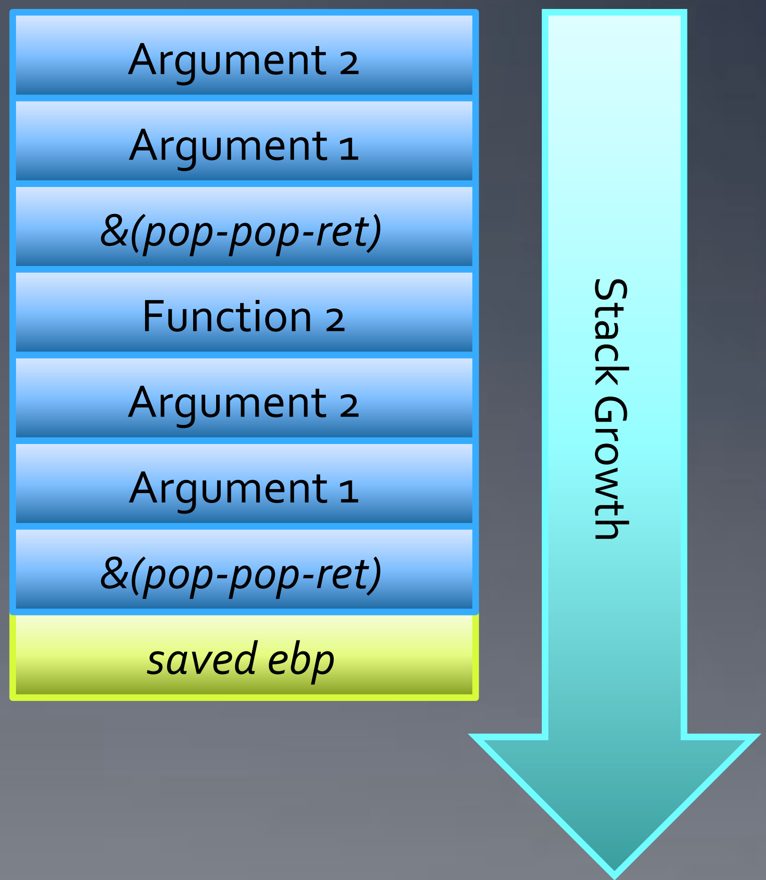
    ret

    …
```



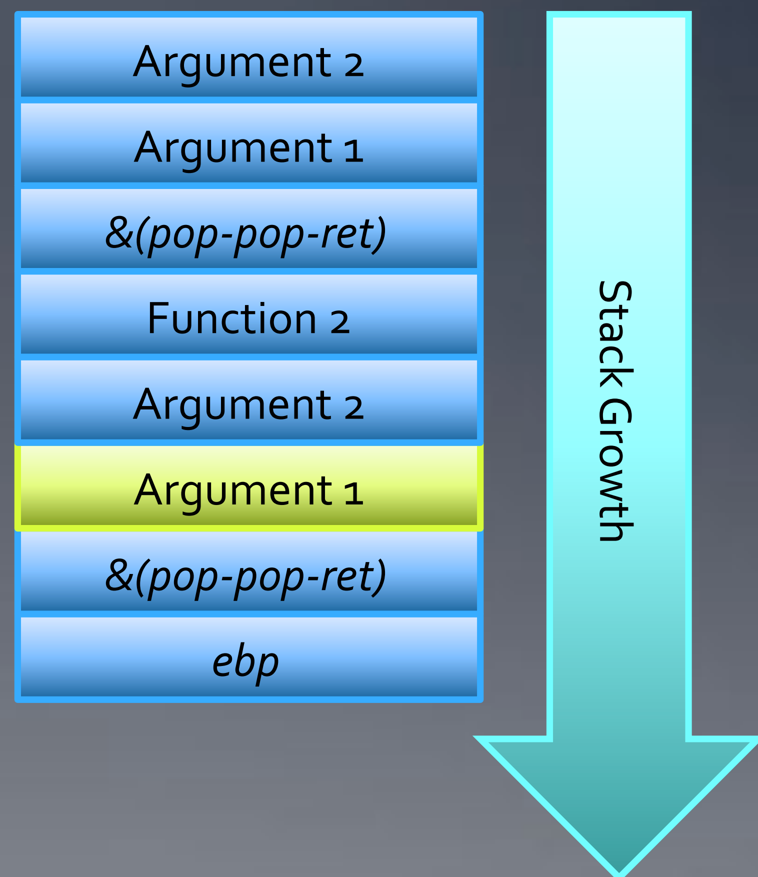
Return Chaining

```
780da4dc:
   push ebp
   mov ebp, esp
   sub esp, 0x100
   …
   mov eax, [ebp+8]
   …
   leave
   ret
```



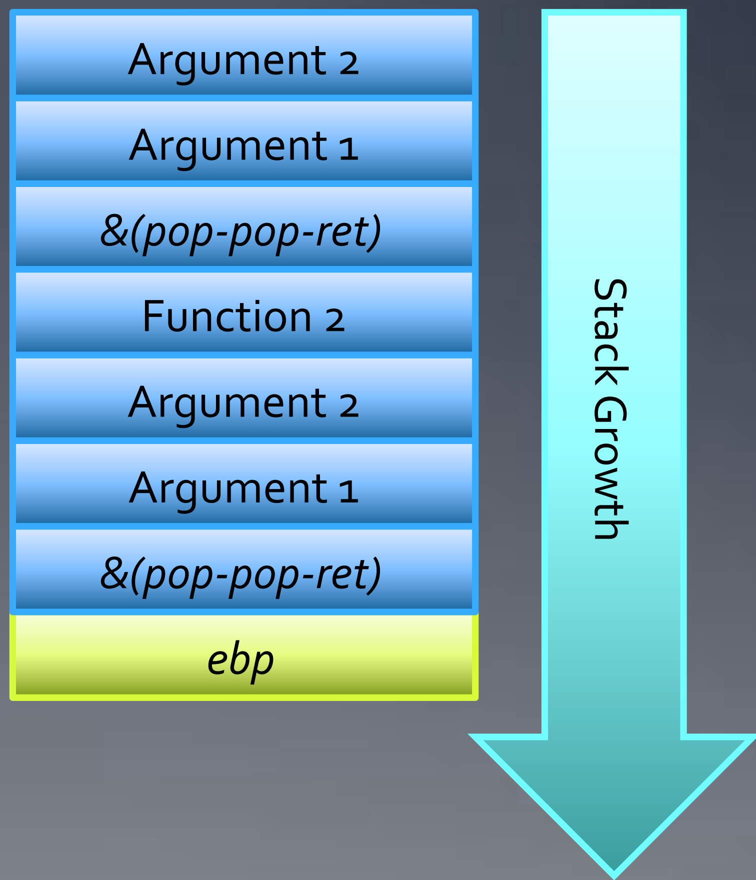
Return Chaining

```
780da4dc:
    push ebp
    mov ebp, esp
    sub esp, 0x100
    …
    mov eax, [ebp+8]
    …
    leave
    ret
```



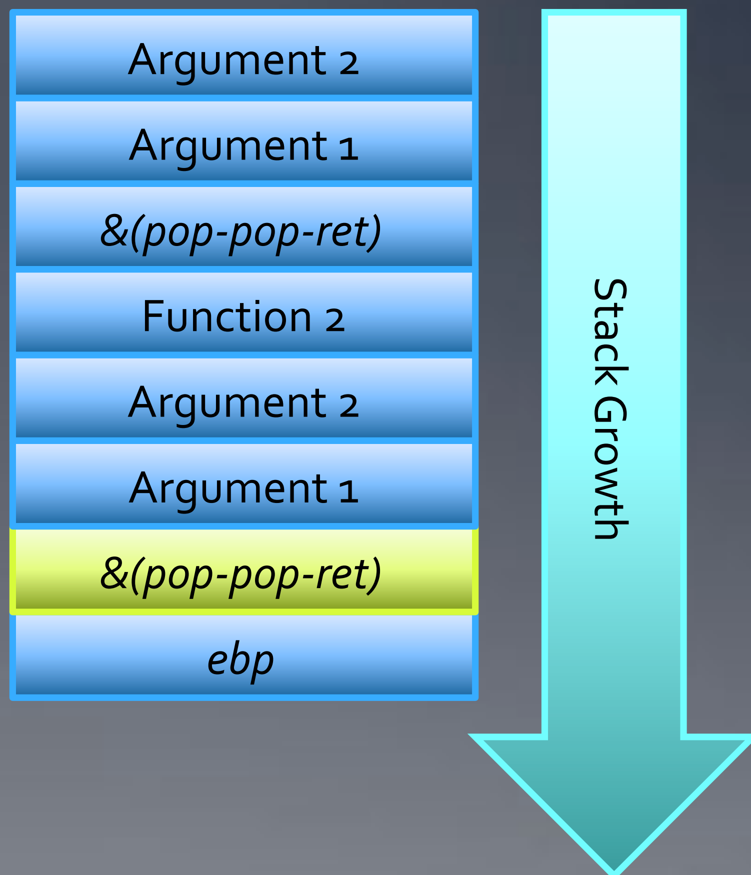
Return Chaining

```
780da4dc:
  push ebp
  mov ebp, esp
  sub esp, 0x100
  …
  mov eax, [ebp+8]
  …
  leave
  ret
```



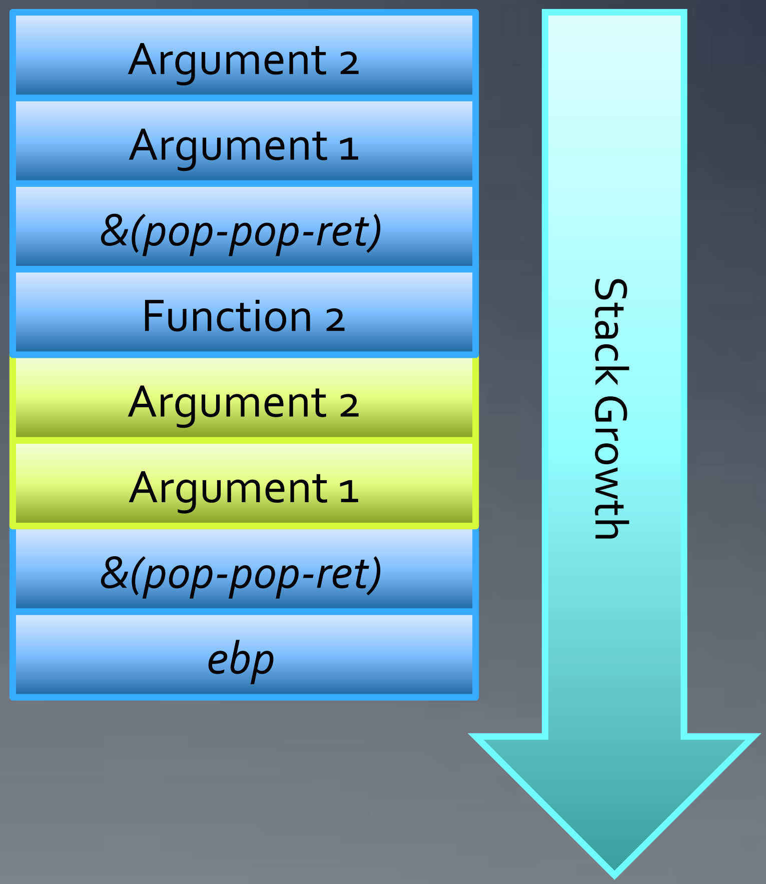
Return Chaining

```
780da4dc:
   push ebp
   mov ebp, esp
   sub esp, 0x100
   …
   mov eax, [ebp+8]
   …
   leave
   ret
```



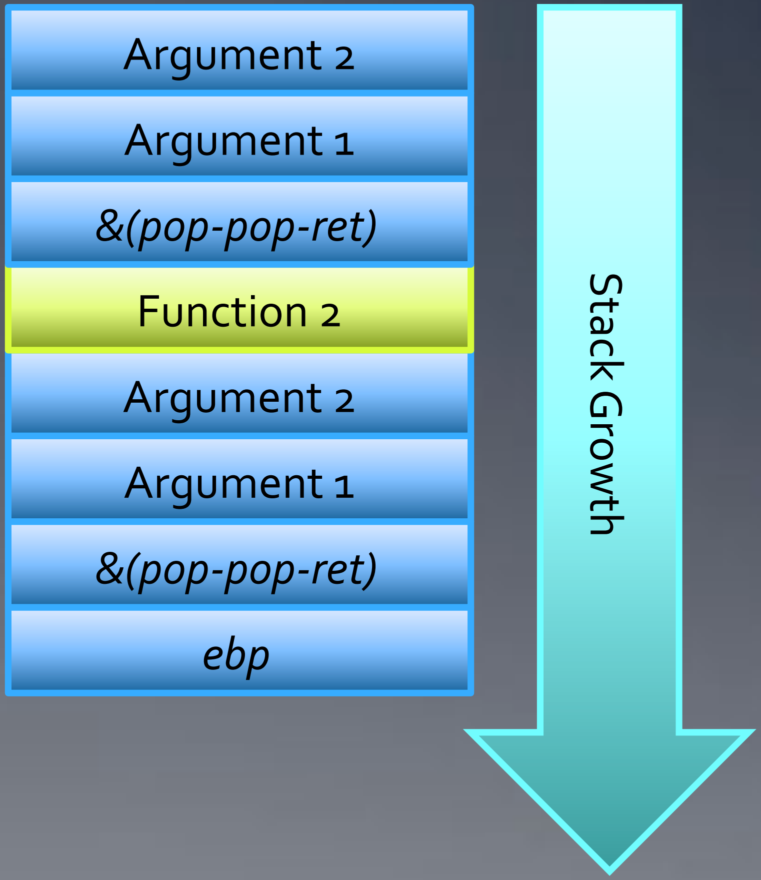
Return Chaining

```
6842e84f:
  pop edi
  pop ebp
  ret
```



Return Chaining

```
6842e84f:
  pop edi
  pop ebp
  ret
```



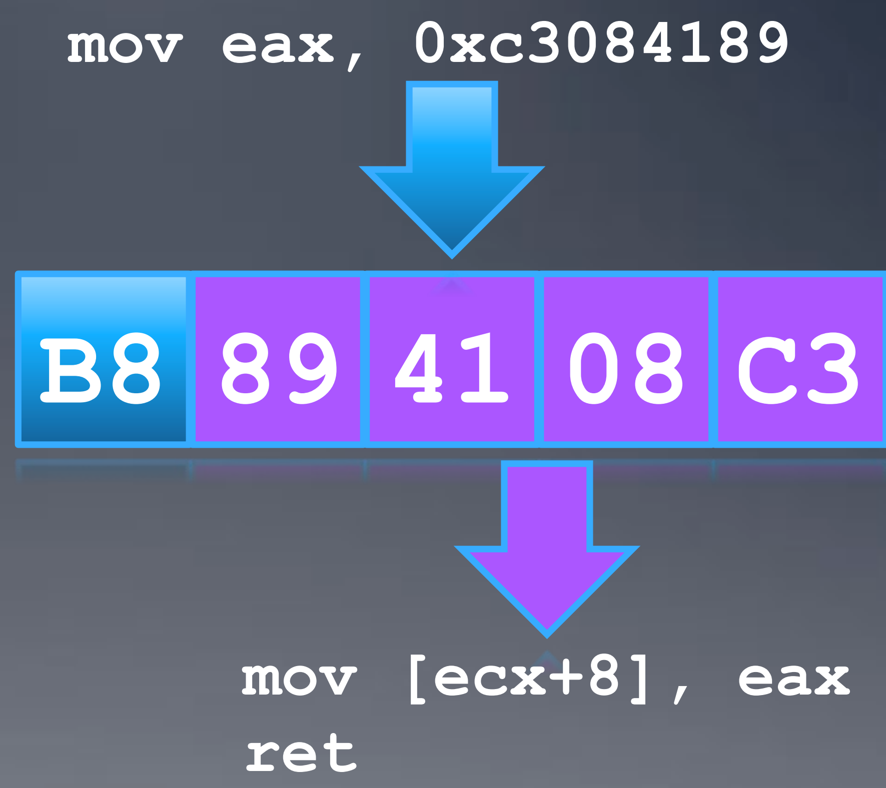
Return-to-Libc

- Return-to-Libc and return chaining are enough to disable DEP on XP SP2 and Vista SP0
  - `NtSetInformationProcess(-1, 34, &2, 4)`[1]
  - WriteProcessMemory() Self-Patch Technique[2]
- XP SP3, Vista SP1, and Windows 7 responded with "Permanent DEP"
  - SetProcessDEPPolicy(PROCESS_DEP_ENABLE)
  - This requires attackers to "up their game"

1. "Bypassing Windows Hardware-Enforced Data Execution Prevention", skape and Skywing (Uninformed Journal, October 2005)
2. "Exploitation With WriteProcessMemory()", Spencer Pratt (Full-Disclosure, 3/30/2010)

Return-oriented Programming

- Instead of returning to functions, return to instruction sequences followed by a return instruction
- Can return into middle of existing instructions to simulate different instructions
- All we need are useable byte sequences anywhere in executable memory pages



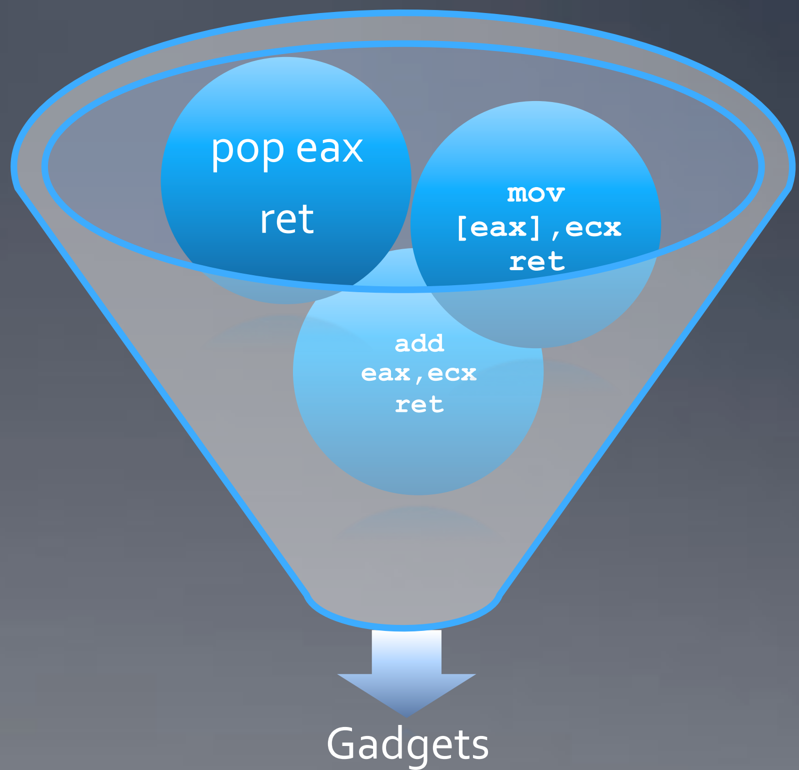
"The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)", Hovav Shacham (ACM CCS 2007)

Return-Oriented Programming

is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from .text segments

Return-Oriented Programming

- Various instruction sequences can be combined to form *gadgets*
- Gadgets perform higher-level actions
  - Write specific 32-bit value to specific memory location
  - Add/sub/and/or/xor value at memory location with immediate value
  - Call function in shared library



Example Gadget

```
pop eax
ret
```

+

```
pop ecx
ret
```

+

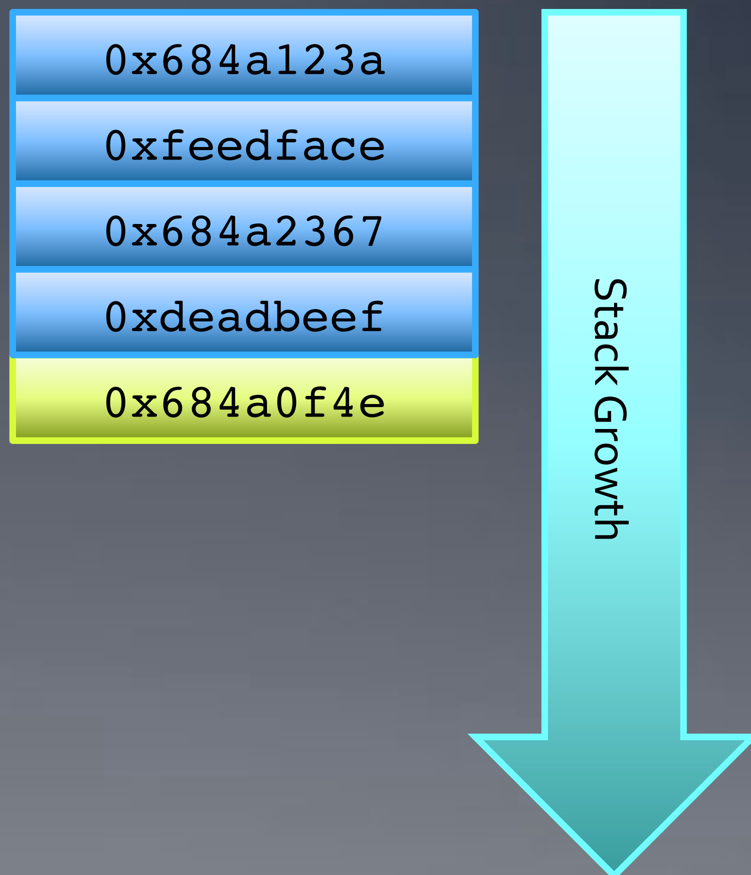
```
mov
[ecx],eax
ret
```

=

STORE IMMEDIATE VALUE

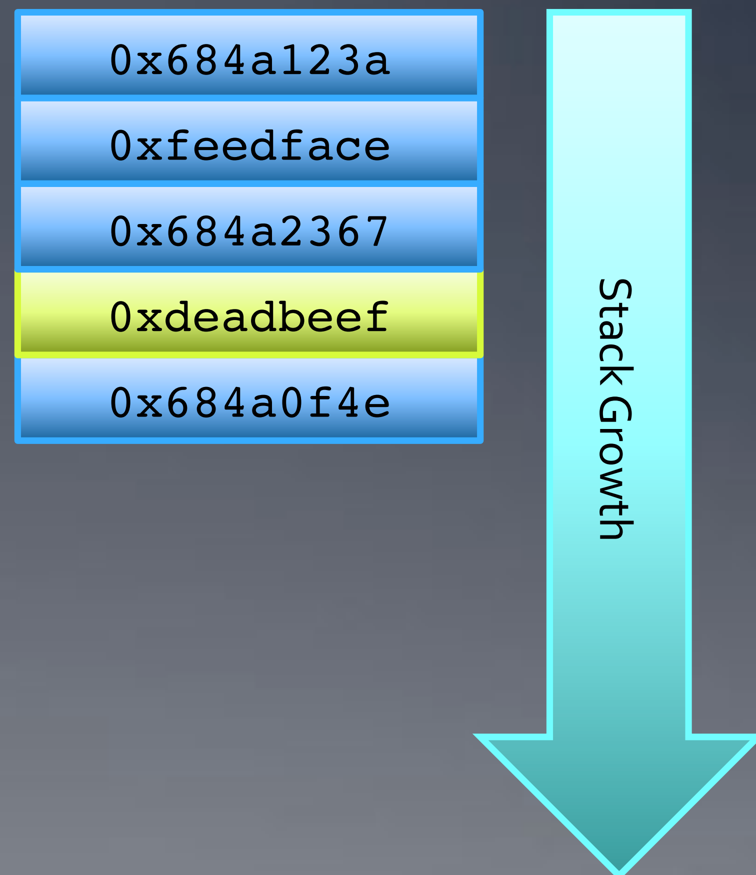
Return-Oriented Write4 Gadget

```
684a0f4e:
  pop eax
  ret
684a2367:
  pop ecx
  ret
684a123a:
  mov [ecx], eax
  ret
```



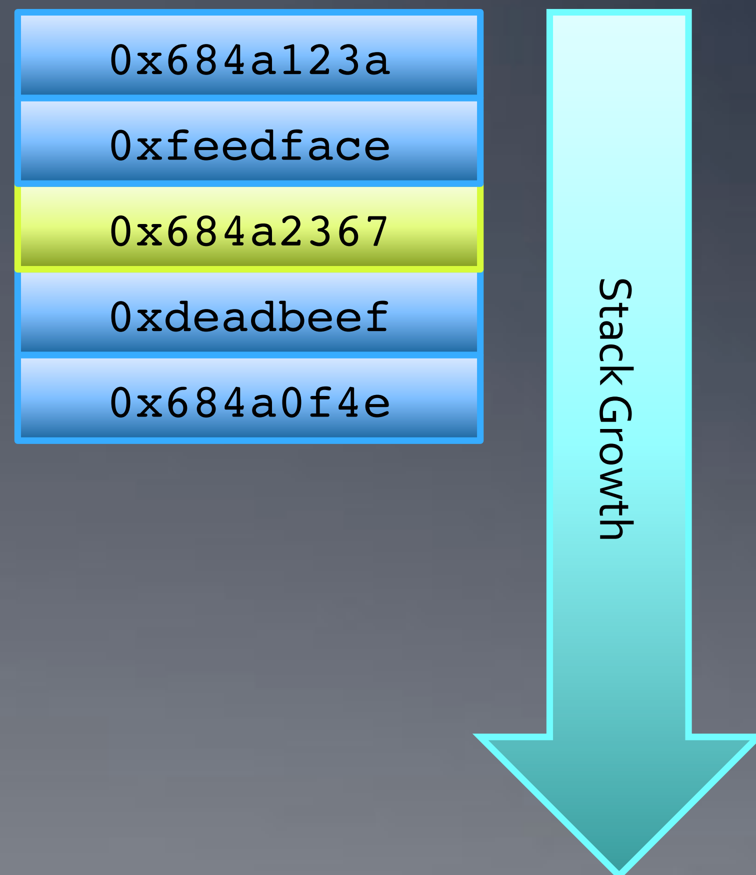
Return-Oriented Write4 Gadget

```
684a0f4e:
  pop eax
  ret
684a2367:
  pop ecx
  ret
684a123a:
  mov [ecx], eax
  ret
```



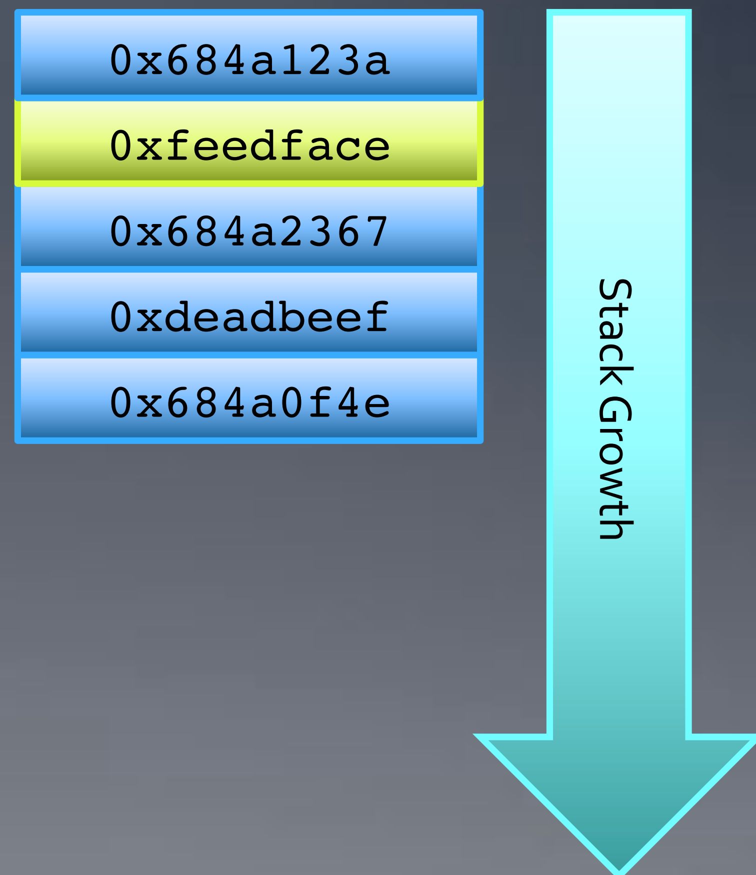
Return-Oriented Write4 Gadget

```
684a0f4e:
  pop eax
  ret
684a2367:
  pop ecx
  ret
684a123a:
  mov [ecx], eax
  ret
```



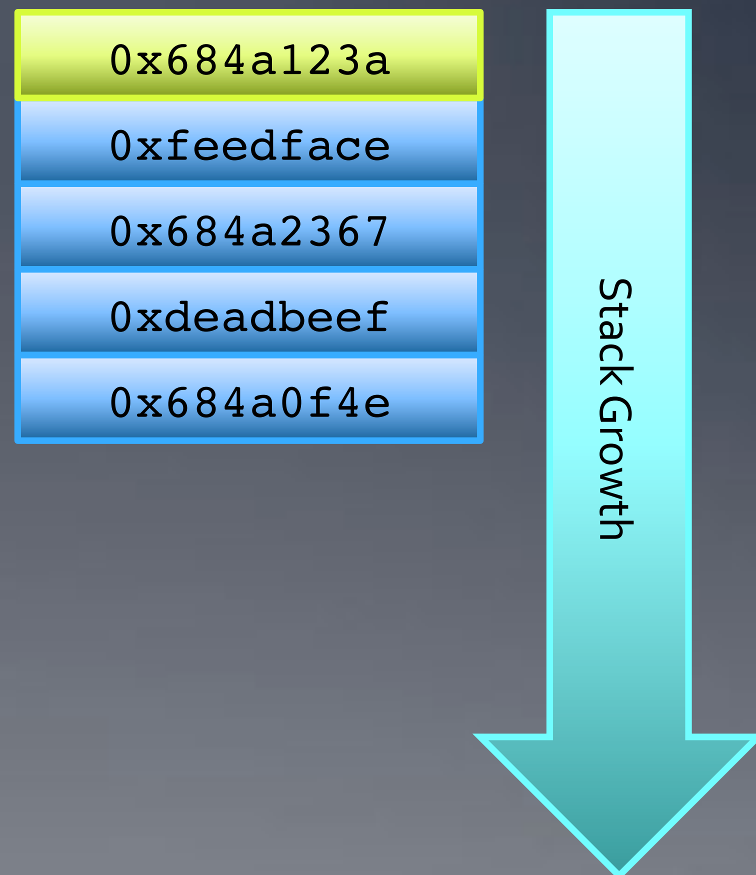
Return-Oriented Write4 Gadget

```
684a0f4e:
  pop eax
  ret
684a2367:
  pop ecx
  ret
684a123a:
  mov [ecx], eax
  ret
```



Return-Oriented Write4 Gadget

```
684a0f4e:
  pop eax
  ret
684a2367:
  pop ecx
  ret
684a123a:
  mov [ecx], eax
  ret
```

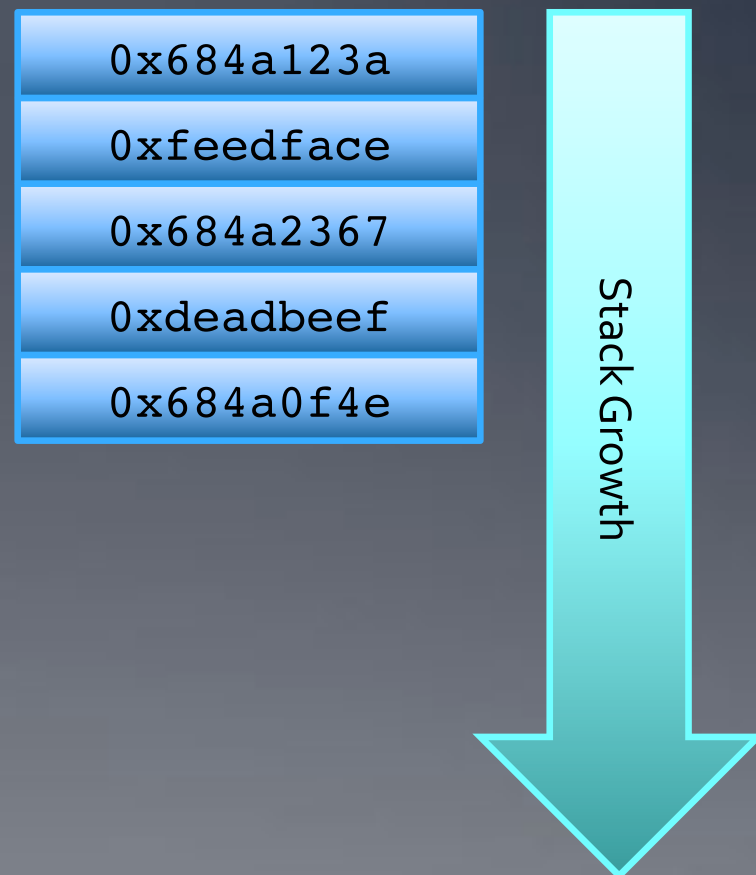


Return-Oriented Write4 Gadget

```
684a0f4e:
  pop eax
  ret
684a2367:
  pop ecx
  ret
684a123a:
  mov [ecx], eax
  ret
```

Return-Oriented Write4 Gadget

```
684a0f4e:
  pop eax
  ret
684a2367:
  pop ecx
  ret
684a123a:
  mov [ecx], eax
  ret
```



Generating a Return-Oriented Program

- Scan executable memory regions of common shared libraries for useful instruction sequences followed by return instructions

- Chain returns to identified sequences to form all of the desired gadgets from a Turing-complete gadget catalog

- The gadgets can be used as a backend to a C compiler
  - See Hovav Shacham's paper for details on their compiler and demonstration of return-oriented quicksort

- **Preventing the introduction of malicious *code* is not enough to prevent the execution of malicious *computations***

Bypassing DEP

Data Execution Prevention

- DEP uses the NX/XD bit of x86 processors to enforce the non-execution of memory pages without PROT_EXEC permission
  - On non-PAE processors/kernels, READ => EXEC
  - PaX project cleverly simulated NX by desynchronizing instruction and data TLBs
- Requires every module in the process (EXE and DLLs) to be compiled with /NXCOMPAT flag
- DEP can be turned off dynamically for the whole process by calling (or returning into) NtSetInformationProcess()[1]
- XP SP3, Vista SP1, and Windows 7 support "Permanent DEP" that once enabled, cannot be disabled at run-time

1. "Bypassing Windows Hardware-Enforced Data Execution Prevention", skape and Skywing (Uninformed Journal, October 2005)

Return-Oriented Exploits

- First, attacker must cause stack pointer to point into attacker-controlled data
  - This comes for free in a stack buffer overflow
  - Exploiting other vulnerabilities (i.e. heap overflows) requires using a *stack pivot* sequence to point ESP into attacker data
    - ```
      mov esp, eax
      ret
      ```
    - ```
      xchg eax, esp
      ret
      ```
    - ```
      add esp, <some amount>
      ret
      ```
- Attacker-controlled data contains a return-oriented exploit payload
  - These payloads may be 100% return-oriented programming or simply act as a temporary payload stage that enables subsequent execution of a traditional machine-code payload
      ```

Return-Oriented Payload Stage

- HEAP_CREATE_ENABLE_EXECUTE method[1]

```
hHeap = HeapCreate(HEAP_CREATE_ENABLE_EXECUTE, 0, 0);
pfnPayload = HeapAlloc(hHeap, 0, dwPayloadLength);
CopyMemory(pfnPayload, ESP+offset, dwPayloadLength);
(*pfnPayload)();
```

- VirtualAlloc() method

```
VirtualAlloc(lpAddress, dwPayloadSize, MEM_COMMIT,
   PAGE_EXECUTE_READWRITE);
CopyMemory(lpAddress, ESP+offset, dwPayloadSize);
(*lpAddress)();
```

- VirtualProtect(ESP) method

```
VirtualProtect(ESP+offset & ~(4096 - 1),
   dwPayloadSize, PAGE_EXECUTE_READWRITE);
(*ESP+offset)();
```

1. "DEPLIB", Pablo Sole (H2HC November 2008)

Do the Math

Stack Pivot + Return-Oriented Payload Stage + Traditional Payload = Permanent DEP Bypass Exploit

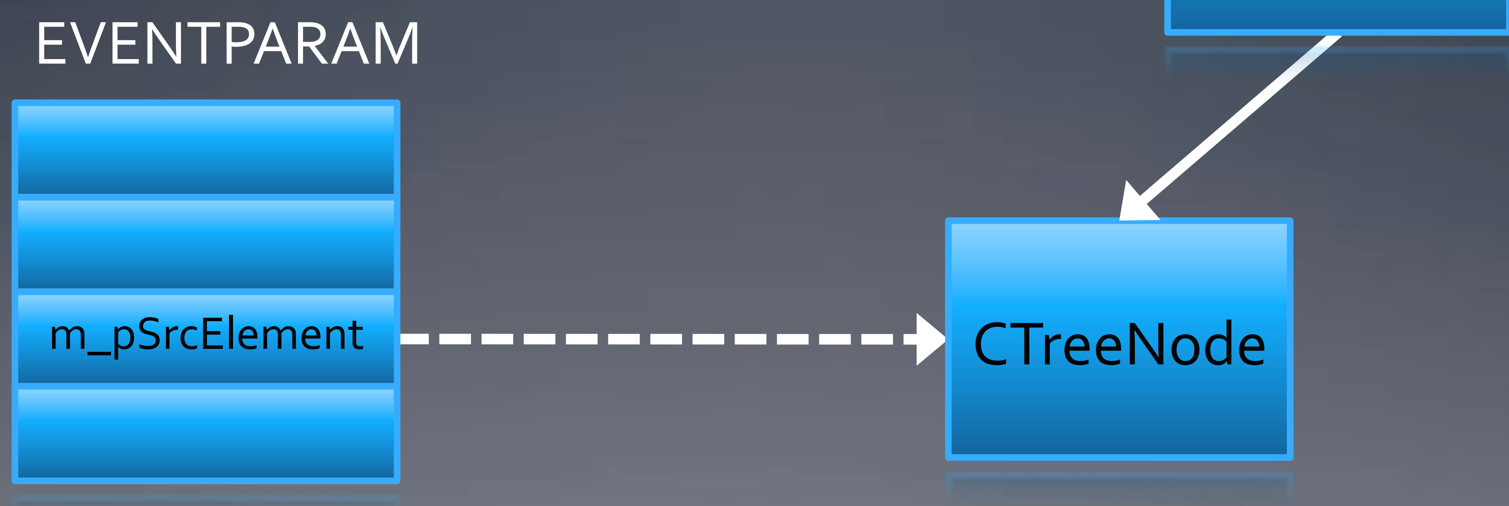
DEP w/o ASLR is Weak Sauce™

- No ASLR:
  - Exploitation requires building a reusable return-oriented payload stage from any common DLL
- One or more modules do not opt-in to ASLR:
  - Exploitation requires building entire return-oriented payload stage from useful instructions found in non-ASLR module(s)
- All executable modules opt-in to ASLR:
  - Exploitation requires exploiting a memory disclosure vulnerability to reveal the load address of one DLL and dynamically building the return-oriented payload stage

Exploiting Aurora on Win7

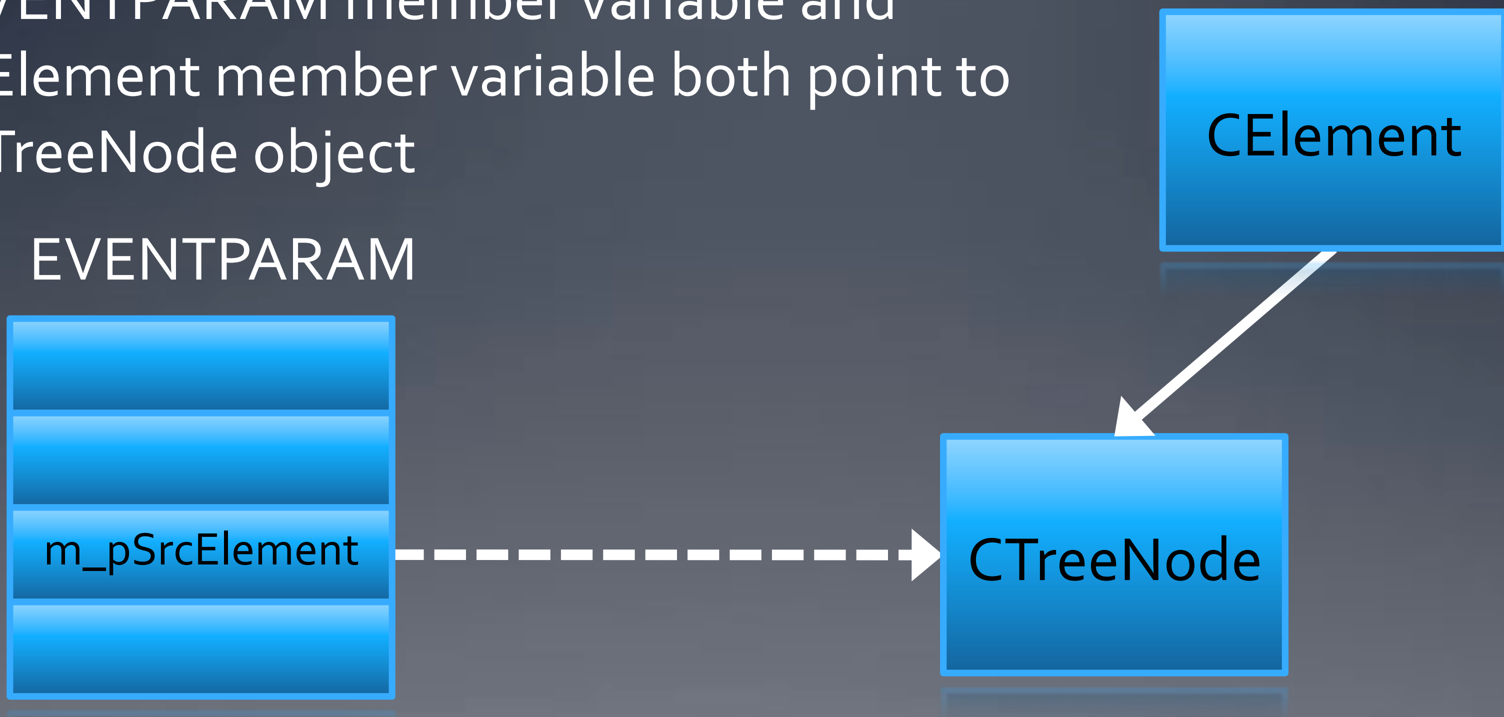
The "Aurora" IE Vulnerability

- EVENTPARAMs copied by createEventObject (oldEvent) don't increment CTreeNode ref count



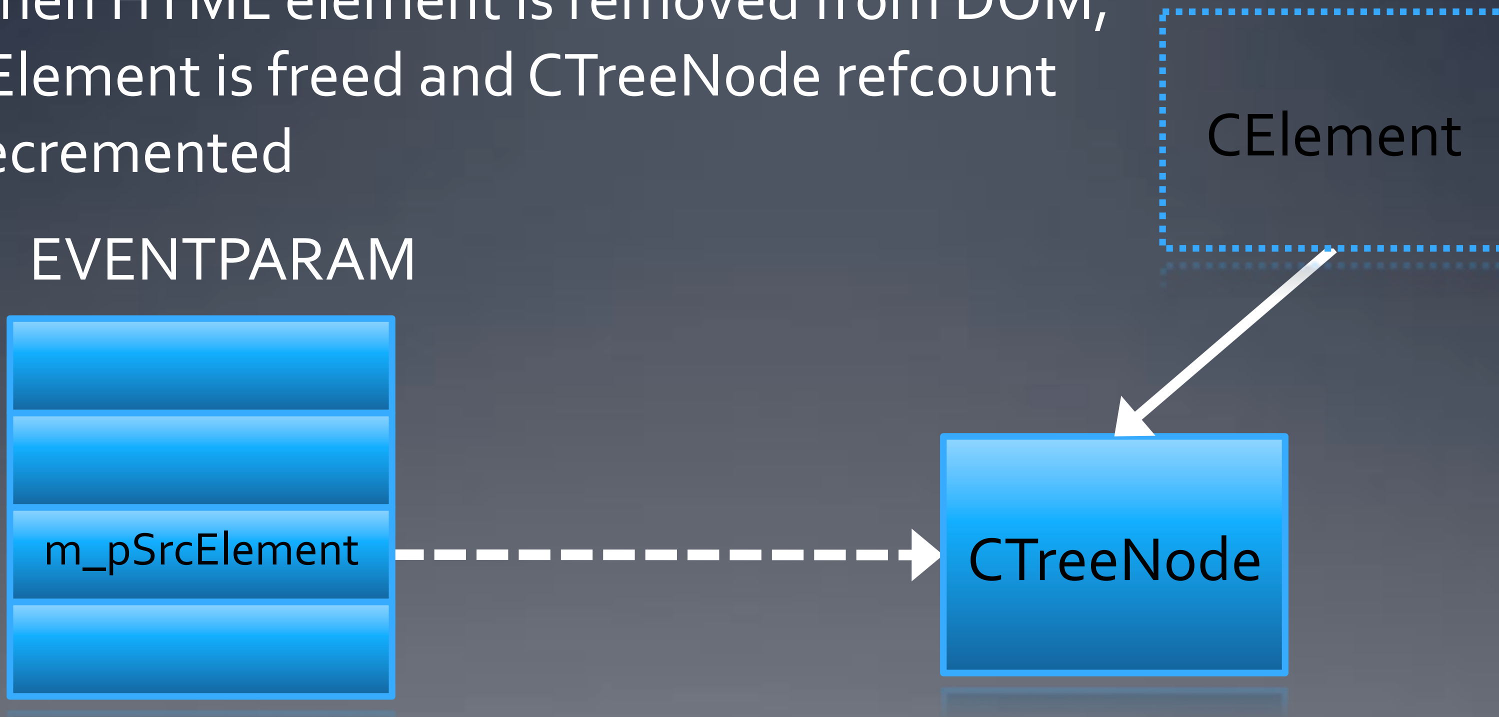
The "Aurora" IE Vulnerability

- EVENTPARAM member variable and CElement member variable both point to CTreeNode object



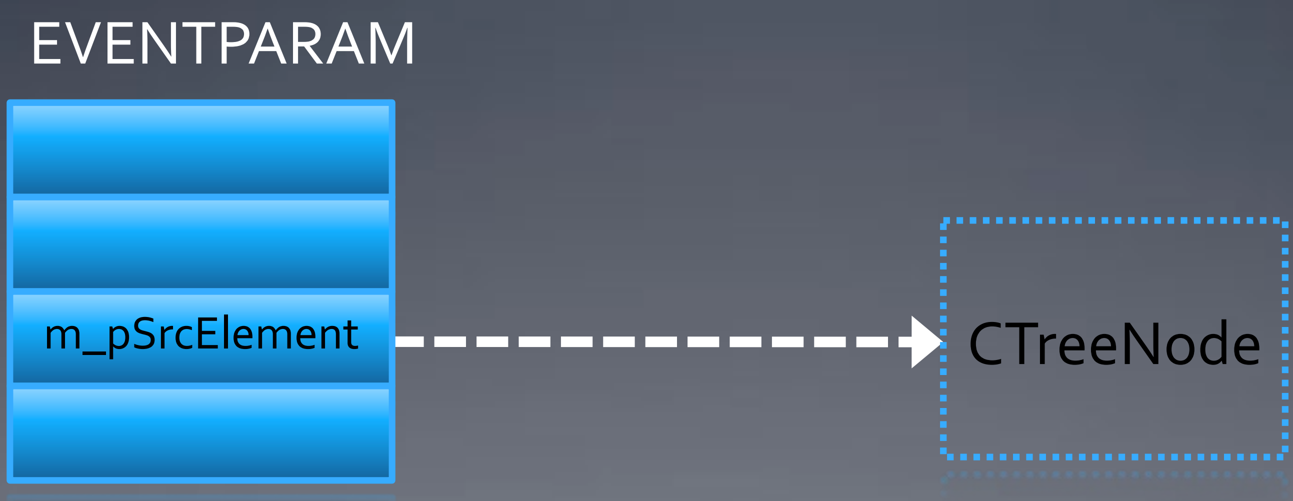
The "Aurora" IE Vulnerability

- When HTML element is removed from DOM, CElement is freed and CTreeNode refcount decremented



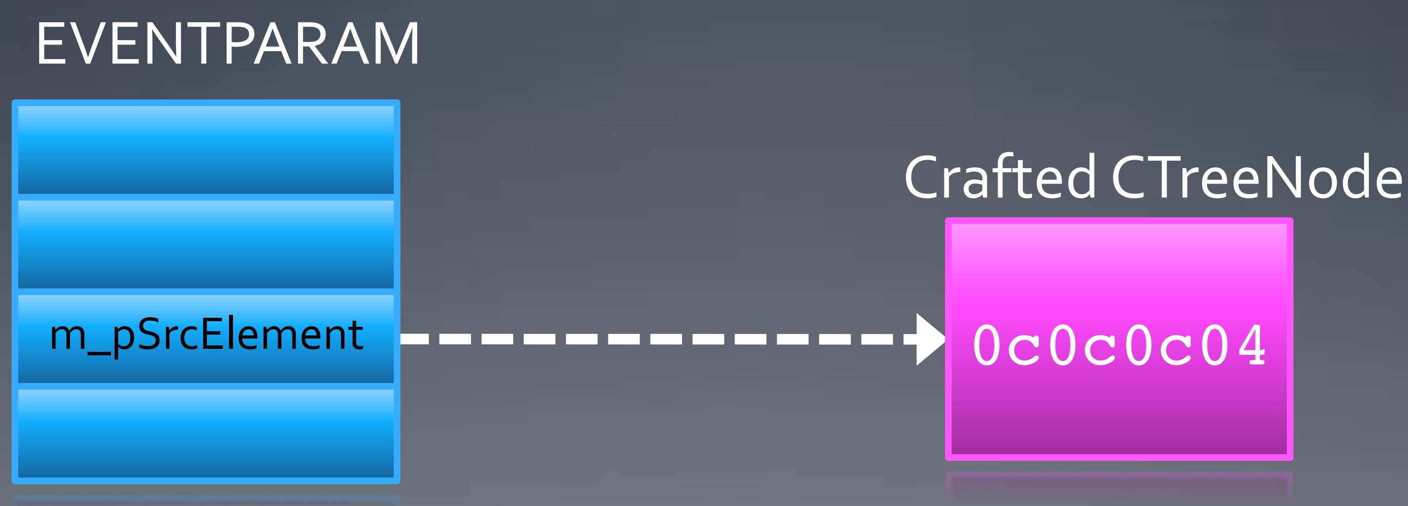
The "Aurora" IE Vulnerability

- If CTreeNode refcount == 0, the object will be freed and EVENTPARAM points free memory



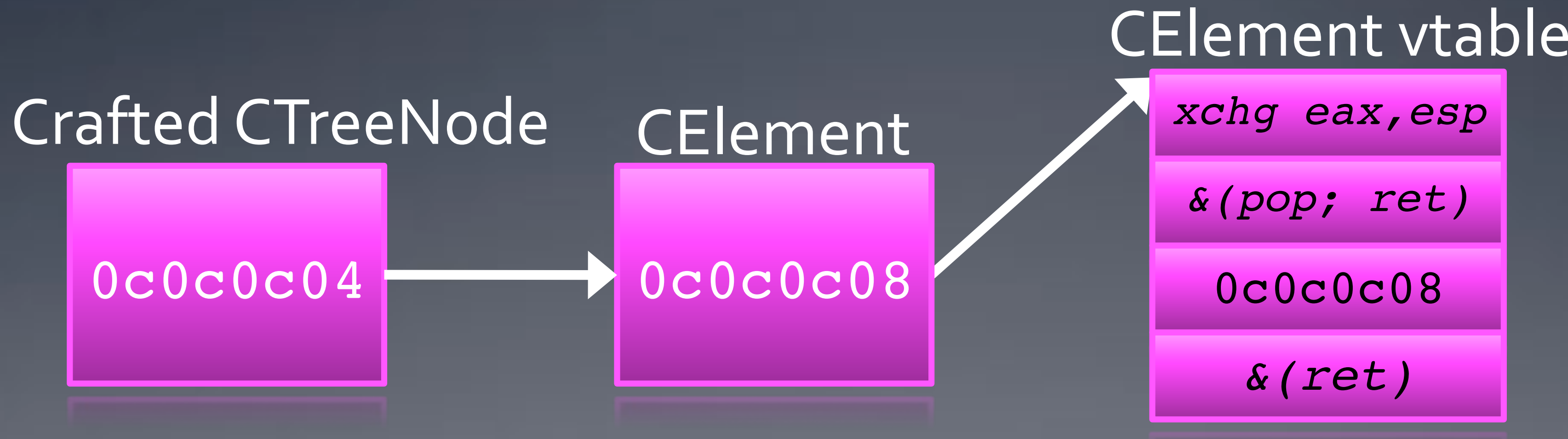
Exploiting The Aurora Vulnerability

- Attacker can use controlled heap allocations to replace freed heap block with crafted heap block



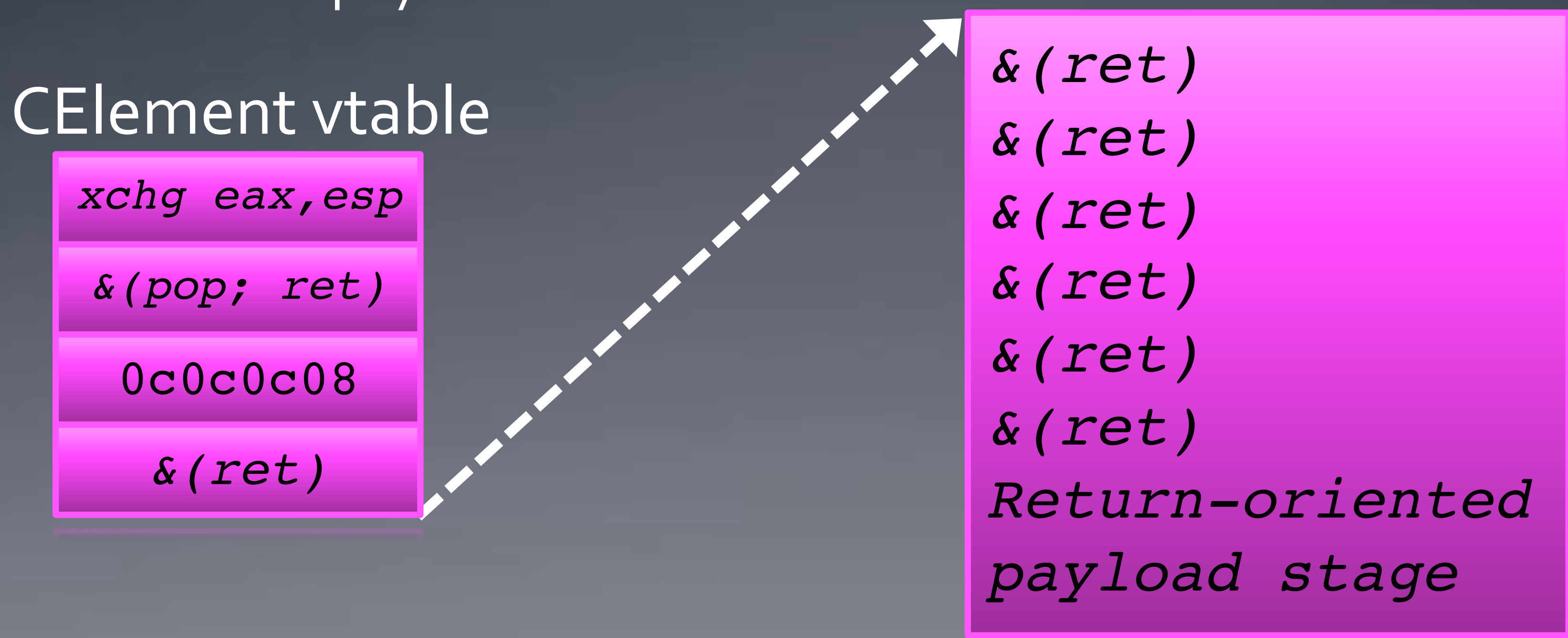
Exploiting The Aurora Vulnerability

- The crafted heap block points to a crafted CElement object in the heap spray, which points back to itself as a crafted vtable



Exploiting The Aurora Vulnerability

- Attacker triggers virtual function call through crafted CElement vtable, which performs a stack pivot through a return to an 'xchg eax, esp; ret' sequence and runs return-oriented payload



Exploit Demo

BISC

Borrowed Instructions Synthetic Computer

BISC

- BISC is a ruby library for demonstrating how to build borrowed-instruction[1] programs
- Design principles:
  - Keep It Simple, Stupid (KISS)
  - Analogous to a traditional assembler
  - Minimize behind the scenes "magic"
  - Let user write simple "macros"

1. Sebastian Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique". http://www.suse.de/~krahmer/no-nx.pdf

ROP vs. BISC

Return-Oriented Programming

- Reuses single instructions followed by a return
- Composes reused instruction sequences into gadgets
- Requires a Turing-complete gadget catalog with conditionals and flow control
- May be compiled from a high-level language

BISC

- Reuses single instructions followed by a return
- Programs are written using the mnemonics of the borrowed instructions
- Opportunistic based on instructions available
- Rarely Turing-complete
- Supports user-written macros to abstract common operations

Borrowed-Instruction Assembler

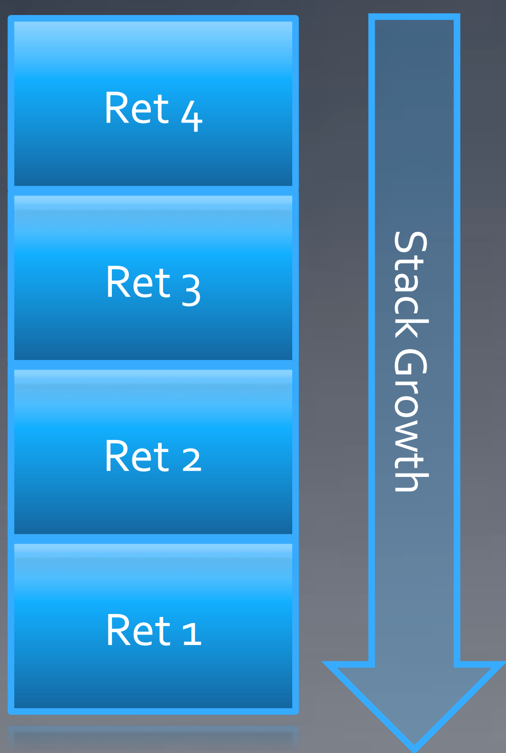
- We don’t need a full compiler, just an assembler
  - Writing x86 assembly is not scary
  - Only needs to support a minimal subset of x86
- Our assembler will let us write borrowed-instruction programs using familiar x86 assembly syntax
  - Source instructions are replaced with an address corresponding to that borrowed instruction
- Assembler will scan a given set of PE files for borrowable instructions
- No support for conditionals or loops

MSF PeScan-Based Scanner

```
$ ./scanner.rb dirapi.dll
ADD EAX, ECX
ADD EAX, [EAX]
ADD ESI, ESI
ADD ESI, [EBX]
ADD [EAX], EAX
ADD [EBX], EAX
ADD [EBX], EBP
ADD [EBX], EDI
ADD [ECX], EAX
ADD [ESP], EAX
AND EAX, EDX
AND ESI, ESI
INT3
MOV EAX, ECX
MOV EAX, EDX
MOV EAX, [ECX]
MOV [EAX], EDX
MOV [EBX], EAX
MOV [ECX], EAX
MOV [ECX], EDX
MOV [EDI], EAX
MOV [EDX], EAX
MOV [EDX], ECX
MOV [ESI], ECX
OR EAX, ECX
OR EAX, [EAX]
OR [EAX], EAX
OR [EDX], ESI
POP EAX
POP EBP
POP EBX
POP ECX
POP EDI
POP EDX
POP ESI
POP ESP
SUB EAX, EBP
SUB ESI, ESI
SUB [EBX], EAX
SUB [EBX], EDI
XCHG EAX, EBP
XCHG EAX, ECX
XCHG EAX, EDI
XCHG EAX, EDX
XCHG EAX, ESP
XOR EAX, EAX
XOR EAX, ECX
XOR EDX, EDX
XOR [EBX], EAX
```

Programming Model

Stack unwinds “upward”



We write borrowed-instruction programs “downward”

```
RET 1
RET 2
RET 3
RET 4
```

Me Talk Pretty One Day

- Each unique return-oriented instruction is a word in your vocabulary
- A larger vocabulary is obviously better, but not strictly necessary in order to get your point across
- You will need to work with the vocabulary that you have available

```
MOV EDX, [ECX]
MOV EAX, EDX
MOV ESI, 3
ADD EAX, ESI
MOV [ECX], EAX
```

⇔

```
ADD [ECX], 3
```

BISC Programs

- Programs are nested arrays of strings representing borrowed instructions and immediate values

```
Main = [ “POP EAX”, 0xdeadbeef ]
```

- Arrays can be nested, which allows macros:

```
Main = [
  [ “POP EAX”, 0xdeadbeef ],
  “INT3”
]
```

BISC Macros

- Macros are ruby functions that return an array of borrowed-instructions and values

```
def set(variable, value)
  return [
    “POP EAX”, value,
    “POP ECX”, variable,
    “MOV [ECX], EAX”
  ]
end
```

BISC Sample Program

```
#!/usr/bin/env ruby -I/opt/msf3/lib -I../lib

require 'bisc'

bisc = BISC.new()
ARGV.each { |a|
  bisc.add_module(a)
}

def clear(var)
  return [
  “POP EDI”, 0xffffffff,
  “POP EBX”, var,
  “OR [EBX], EDI”,
  “POP EDI”, 1,
  “ADD [EBX], EDI”
]
end

v = bisc.allocate(4)
Main = [ clear(v) ]
print bisc.assemble(Main)
```

Higher-Order BISC

- Consider macros “virtual methods” for common high-level operations:
  - Set variable to immediate value
  - ADD/XOR/AND variable with immediate value
  - Call a stdcall/cdecl function through IAT
- Write programs in terms of macros, not borrowed instructions
- Macros can be re-implemented if they require unavailable borrowed instructions

BISC (Non) Availability

- Covered and included in “Assured Exploitation” training materials under an individual student personal use license
  - Training given with Alex Sotirov at CanSecWest 2010

- Not going to be made freely available (sorry)
  - I don’t want to contribute to the development of DEP-evading malware exploits
  - Your favorite pen-testing framework will likely implement something similar eventually

Wrapping Up

Other Applications of Return-oriented Programming

- iPhone's code signing enforcement prevents modification of code or introduction of new executable code
  - Exploit payloads must be 100% pure return-oriented
- Embedded processors often have separate instruction and data write-back caches, which make injecting code problematic
  - Return-oriented exploitation techniques can be used to flush the caches before executing the payload (Dai Zovi, 2003)
- x86-64 ABI requires non-executable (NX) data memory
  - "Borrowed code chunks" exploitation technique (Krahmer 2005)
- Some secure hardware designs keep code in ROM and refuse to execute code from RAM
  - Checkoway et al (Usenix 2008) demonstrated the use of ROP on the Z80-based Sequoia AVC Advantage secure voting machine

Conclusions

- Return-oriented techniques are increasingly required to exploit vulnerabilities on systems with non-executable data memory protections
- A return-oriented payload stage can be developed to bypass Permanent DEP
- Bypassing DEP under ASLR requires at least one non-ASLR module
- Bypassing DEP under full ASLR requires an executable memory address disclosure vulnerability in addition to memory corruption corruption
- iPhone's code signing enforcement requires attackers to develop fully return-oriented payloads
  - Attacker's actions are still limited by the application sandbox
- Preventing malicious actions is more important than preventing malicious code

Takeaways

- IT Security
  - Malware may eventually use these techniques to exploit DEP-enabled processes
  - Malware analysts must learn how to analyze return-oriented exploit payloads
- Software Vendors
  - Do not assume DEP/ASLR make vulnerabilities non-exploitable
  - Better to assume that all vulnerabilities yield full code execution
  - Restrict the actions that may be performed by application components that parse and handle potentially untrusted data
    - Privilege reduction (i.e. run under Low Integrity on Vista/7)
    - Sandboxing (see Chromium's sandboxed web renderers)
    - Virtualization?

Soapbox

- Stop defending only against tactics and start defending against larger attacker strategies
  - Code injection through memory corruption is a *tactic*
  - Malware persistence through various registry modifications are all *tactics*
  - Causing application/host/human misbehavior is the *strategy*

Otherwise…

- We run the risk of dealing with the volcanic ash cloud from a “Cyber Pompeii” or “Cyber Eyjafjallajökull”

Questions?

Mail: ddz@theta44.org
Twitter: @dinodaizovi
Web: http://www.theta44.org
Blog: http://blog.trailofbits.com