Return-Oriented Exploitation

Dino A. Dai Zovi
Independent Security Researcher
Trail of Bits
Context

- Full control of EIP no longer yields immediate arbitrary code execution
  - Primarily due to increasing availability and utilization of exploit mitigations such as DEP and ASLR

- Attackers must identify other supplementary vulnerabilities to enable exploitation of memory corruption issues
  - Memory address/layout disclosure vulnerabilities
  - Availability of known executable code at static, predictable, or chosen locations
    - i.e. non-ASLR DLLs, JIT sprays, IE .NET user controls
Agenda

- Current State of Exploitation
- Return-Oriented Exploitation
- Borrowed Instructions Synthetic Computer
  - Or, ROP in Evenings and Weekends
- Return-Oriented Exploitation Strategies
- Exploiting Aurora on Windows 7
- Conclusion
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 stack buffer overflows all over the universe
- Many believe that only stack corruption is exploitable…
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 or heap memory corruption
A Brief History of 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 commercial operating system to incorporate exploit mitigations
  - Protected stack metadata (Visual Studio compiler /GS flag)
  - Protected heap metadata (Heap Safe Unlinking)
  - SafeSEH (compile-time exception handler registration)
  - Software and hardware-enforced Data Execution Prevention (DEP)

- Windows Vista and 7 include Address Space Layout Randomization (ASLR) and other mitigations
Mitigations Make Exploitation Harder

- ASLR
- DEP/NX
- SafeSEH
- Heap Metadata Protection
- Stack Cookies
Exploitation Techniques
Rendered Ineffective

- Stack return address overwrite
- Heap free block metadata overwrite
- SEH Frame Overwrite
- Direct jump/return to shellcode
- App-specific data overwrite
- ???
Mitigations requires OS, Compiler, and Application Participation and are additive

- OS run-time mitigations: Heap protections, SEH Chain Validation
- Compiler-based mitigations: Stack cookies, SafeSEH
- Application opt-in mitigations: DEP, ASLR
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
Return-Oriented Exploitation
EIP != Arbitrary Code Execution

- Direct jump or “register spring” (jmp/call <reg>) into injected code is not always possible
- ASLR and Library Randomization make code and data locations unpredictable
- EIP pointing to attacker-controlled data does not yield arbitrary code execution
- DEP/NX makes data pages non-executable
- On platforms with separate data and instruction caches (PowerPC, ARM), the CPU may fetch old data from memory, not your shellcode from data cache
EIP => Arbitrary Code Execution

- It now requires extra effort to go from full control of EIP to arbitrary code execution

- We use control of EIP to point ESP to attacker-controlled data
  - “Stack Pivot”

- We use control of the stack to direct execution by simulating subroutine returns into existing code

- Reuse existing subroutines and instruction sequences until we can transition to full arbitrary code execution
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 or simply act as a temporary payload stage that enables subsequent execution of a traditional machine-code payload
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
  ...

Stack Growth
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:

```assembly
push ebp
mov ebp, esp
sub esp, 0x100
...
mov eax, [ebp+8]
...
leave
ret
```
Return Chaining

6842e84f:

pop edi
pop ebp
ret

stack growth
Return Chaining

6842e84f:

pop edi
pop ebp
ret

Argument 2
Argument 1
&(pop-pop-ret)
Function 2
Argument 2
Argument 1
&(pop-pop-ret)
ebp

Stack Growth
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 out letters from magazines, you are cutting out instructions from text segments.

Credit: Dr. Raid’s Girlfriend
Return-Oriented Gadgets

- 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 POKE Gadget

684a0f4e:
  pop eax
  ret

684a2367:
  pop ecx
  ret

684a123a:
  mov [ecx], eax
  ret

Stack Growth
Return-Oriented POKE Gadget

684a0f4e:
  pop eax
  ret

684a2367:
  pop ecx
  ret

684a123a:
  mov [ecx], eax
  ret
Return-Oriented POKE Gadget

684a0f4e:
  pop eax
  ret

684a2367:
  pop ecx
  ret

684a123a:
  mov [ecx], eax
  ret
### Return-Oriented POKE Gadget

**684a0f4e:**

```
pop eax
ret
```

**684a2367:**

```
pop ecx
ret
```

**684a123a:**

```
mov [ecx], eax
ret
```
Return-Oriented POKE Gadget

684a0f4e:
  pop eax
  ret

684a2367:
  pop ecx
  ret

684a123a:
  mov [ecx], eax
  ret

Stack Growth

0x684a0f4e
0xfeedface
0x684a2367
0xdeadbeef
0x684a123a
Return-Oriented POKE Gadget

684a0f4e:
    pop eax
    ret

684a2367:
    pop ecx
    ret

684a123a:
    mov [ecx], eax
    ret
Return-Oriented POKE Gadget

684a0f4e:
    pop eax
    ret

684a2367:
    pop ecx
    ret

684a123a:
    mov [ecx], eax
    ret

Stack Growth:
- 0x684a123a
- 0xfeedface
- 0x684a2367
- 0xdeadbeef
- 0x684a0f4e
Generating a Return-Oriented Program

- Scan executable memory regions of common shared libraries for useful instructions 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.
- “Preventing the introduction of malicious code is not enough to prevent the execution of malicious computations.”
- “The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)”, Hovav Shacham (ACM CCS 2007)
BISC

Borrowed Instructions Synthetic Computer
BISC

BISC is a ruby library for demonstrating how to build borrowed-instruction\textsuperscript{1} programs

Design principles:

\begin{itemize}
\item Keep It Simple, Stupid (KISS)
\item Analogous to a traditional assembler
\item Minimize behind the scenes “magic”
\item Let user write simple “macros”
\end{itemize}

\textsuperscript{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

<table>
<thead>
<tr>
<th><strong>Return-Oriented Programming</strong></th>
<th><strong>BISC</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuses single instructions followed by a return</td>
<td>Reuses single instructions followed by a return</td>
</tr>
<tr>
<td>Composes reused instruction sequences into gadgets</td>
<td>Programs are written using the mnemonics of the borrowed instructions</td>
</tr>
<tr>
<td>Requires a Turing-complete gadget catalog with conditionals and flow control</td>
<td>Opportunistic based on instructions available</td>
</tr>
<tr>
<td>May be compiled from a high-level language</td>
<td>Nowhere near Turing-complete</td>
</tr>
<tr>
<td></td>
<td>Supports user-written macros to abstract common operations</td>
</tr>
</tbody>
</table>
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
BISC Scanner

Core scanner functionality is implemented through binary regular expressions for known instruction encoding formats.

Regular expressions for all known instruction formats are combined into one complex regular expression.

Handler procedure is called on each match to parse identified instruction instances and produce a symbol representing the borrowable instruction.

i.e.: /\x89[\x00-\x3f\xc0-\xff]\xc3/

A match of \x8b\x01\xc3 produces the symbol “MOV EAX, [ECX]”
BISC Borrowable Instructions

$ ./bisc.rb EXAMPLE
ADD EAX, ECX
ADD EAX, [EAX]
ADD ESI, ESI
ADD ESI, [EBX]
ADD [EAX], EAX
ADD [EBX], EAX
ADD [EBX], EBX
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 [EDI], ECX
MOV [ESI], ECX
OR EAX, ECX
OR EAX, [EAX]
OR [EAX], EAX
OR [EDI], ESI
POP EAX
POP EBX
POP ECX
POP EDI
POP EDX
POP ESI
POP ESP
SUB EAX, EBX
SUB ESI, ESI
SUB [EBX], EAX
SUB [EBX], EDI
XCHG EAX, EBX
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" ]
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 Macros

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

```ruby
def set(variable, value)
    return [
        "POP EAX", value,
        "POP ECX", variable,
        "MOV [ECX], EAX"
    ]
end
```
#!/usr/bin/env ruby -I/opt/msf3/lib -I../lib
require 'bisc'

bisc = BISC::Assembler.new(ARGV)

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)
ROP Faster, Not Harder

- BISC intentionally uses simplest (dumbest) implementation and approach at every opportunity
- aka, “Return-Oriented Programming in Evenings and Weekends”
- Effective, but still requires some manual work

- ROP, Zynamics style (i.e. the smart way)
  - “Everybody be cool, this is a roppery!” by Iozzo, Kornau, and Weinmann
  - Searches for gadgets in architecture-independent manner using REIL meta assembly language
  - Compiles virtual assembly language into sequence of ARM returns
Return-Oriented Exploitation Strategies
Bridge to Execution of Traditional Payload

- Copy payload to executable memory
  - Allocate new RWX memory
  - Use existing RWX memory at known location
  - WriteProcessMemory(WriteProcessMemory())

- Build payload in executable memory
  - Copy 1-N byte chunks found at known locations
  - Sequence of returns to perform 4-byte writes

- Make memory containing payload executable
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()\textsuperscript{1}
- XP SP3, Vista SP1, and Windows 7 support “Permanent DEP” that once enabled, cannot be disabled at run-time

\textsuperscript{1} “Bypassing Windows Hardware-Enforced Data Execution Prevention”, skape and Skywing (Uninformed Journal, October 2005)
Example Return-Oriented Payload Stage

- DEP/NX prevents execution of data in non-executable memory, but does not prevent dynamic creation of new executable code
- Whereas iOS’s code signing enforcement does not

- Four basic steps to obtain arbitrary code execution:
  - GetESP – Records value of ESP for use later
  - Allocate – Allocates new executable memory
  - Copy – Copies traditional machine code payload into newly allocated executable memory
  - Jump – Executes payload from newly allocated memory
GetESP

```
pop ecx ; ECX = &(lpESP)
ret
...
xchg eax, esp
ret
...
mov [ecx], eax
ret
```
GetESP

Machine Code Payload

Jump

Copy

Allocate

GetESP

Stack Growth

pop ecx
ret
...

xchg eax, esp ; EAX = ESP
ret
...

mov [ecx], eax
ret
GetESP

```
pop ecx
ret
...
xchg eax, esp
ret
...
mov [ecx], eax ; *lpESP = EAX
ret
```
Allocate

```
pop eax ; EAX = &IAT_VA
ret

mov eax, [eax]
ret
...
```

```
push eax
ret
```
Allocate

Machine Code Payload

Jump

Copy

Allocate

GetESP

Stack Growth

...
Allocate

Stack Growth

Allocate

GetESP

Copy

Jump

Machine Code Payload

pop eax
ret
...
mov eax, [eax]
ret
...
push eax ; VirtualAlloc()
ret
Allocate

Machine Code Payload
Jump
Copy
Allocate
GetESP

Stack Growth

pop ecx ; ECX = lpMem
ret
...
mov [ecx], eax
ret
...

Allocate
Allocate

```
pop ecx
ret
...

mov [ecx], eax ; Store ret val
ret
...
```
Machine Code Payload
Jump
Copy
Allocate
GetESP

Stack Growth

Copy

pop eax  ; EAX = &lpESP
ret
...
mov eax, [eax]
ret
...
pop ecx
ret
Copy

Stack Growth

Machine Code Payload
Jump
Copy
Allocate
GetESP

pop eax
ret
...

mov eax, [eax] ; EAX = *lpESP
ret
...

pop ecx
ret
Copy

Machine Code Payload
Jump
Copy
Allocate
GetESP

Stack Growth

pop eax
ret
...
mov eax, [eax]
ret
...

pop ecx ; ECX = offset
ret

offset from saved ESP to argument to overwrite
Copy

Machine Code Payload
Jump
Copy
Allocate
GetESP

Stack Growth

add ecx, eax ; ECX = &arg0
ret
...
pop eax
ret
...
mov [eax], eax
ret
Copy

Machine Code Payload

Jump

Allocate

GetESP

Stack Growth

add ecx, eax
ret

...;

pop eax
ret

...;

mov [eax], eax
ret

; EAX = &lpMem
Copy

Machine Code Payload

Jump

Copy

Allocate

GetESP

add ecx, eax
ret
...
pop eax
ret
...

mov [eax], eax ; EAX = lpMem
ret

Stack Growth
Copy

mov [ecx], eax ; *arg0 = lpMem
ret

...;

;; do similar to set arg1 on
;; stack to address of embedded
;; machine code payload

...

;; call memcpy through IAT
mov [ecx], eax
ret
...

;; do similar to set arg1 on

;; stack to address of embedded

;; machine code payload

...  

;; call memcpy through IAT
Machine Code Payload

Jump

Copy

Allocate

GetESP

mov [ecx], eax ; *arg0 = lpMem
ret
...
;; do similar to set arg1 on
;; stack to address of embedded
;; machine code payload
...
;; call memcpy through IAT
Jump

Machine Code Payload

Jump

Copy

Allocate

GetESP

Stack Growth

pop ecx ; ECX = &lpMem
ret
...

mov eax, [ecx]
ret
...

push eax
ret
Jump

Machine Code Payload

Jump

Copy

Allocate

GetESP

Stack Growth

pop ecx
ret

...  

mov eax, [ecx] ; EAX = lpMem
ret  

...  

push eax
ret
Jump

```
pop ecx
ret
...
mov eax, [ecx]
ret
...
push eax ; jmp lpMem
ret
```
Alternative Strategies

Variations


2. Call VirtualProtect on the stack and execute payload directly from there


4. WriteProcessMemory()

5. "Séance" Technique
WriteProcessMemory()

- WriteProcessMemory(), instead of being used to write into a debugged process, can be used to write into the caller’s process.
- If the destination memory page is not writable, WriteProcessMemory() will make the page writable temporarily in order to perform the memory write.
- WriteProcessMemory() can be used to overwrite itself with new executable code at a precise location so that it executes the new code instead of returning to the caller.
“Séance” Technique

- For when you don’t know the location in memory of your buffer, but you can call WriteProcessMemory()
- Chain a sequence of returns into WPM() to build your shellcode in an existing .text segment from 1-N byte chunks elsewhere in memory
- Split desired payload into 1-N byte chunks identified in readable memory at known or static locations
Do the Math

Stack Pivot + Return-Oriented Payload Stage + Traditional Payload = Permanent DEP Bypass Exploit
Exploiting Aurora on Win7
The “Aurora” IE Vulnerability

EVENTPARAMs copied by `createEventObject(oldEvent)` don’t increment CTreeNode ref count

```
EVENTPARAM

m_pSrcElement
```

CTreeNode
The “Aurora” IE Vulnerability

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

EVENTPARAM

- m_pSrcElement

- CTreeNode

- CElement
The “Aurora” IE Vulnerability

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

EVENTPARAM

m_pSrcElement

CElement

CTreeNode
The “Aurora” IE Vulnerability

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

EVENTPARAM

m_pSrcElement

CTreeNode
Exploiting The Aurora Vulnerability

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

EVENTPARAM

m_pSrcElement

Crafted CTreeNode

0c0c0c0c04
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.

CElement vtable

```
xchg eax, esp
&(pop; ret)
0c0c0c08
&(ret)
```

\&(ret)
\& (ret)
\& (ret)
\& (ret)
\& (ret)
\& (ret)
Return-oriented payload stage
Exploit Demo
Conclusions
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
Take-Aways

“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)
Questions

@dindaizovi
ddz@theta44.org
http://trailofbits.com / http://theta44.org