Code Reuse Attacks (I)

Slide credits: some slides and figures adapted from H. Shacham, V. Pappas, Luca Davi, David Brumley and Nora Sovarel
Administrivia

- **Summaries**
  - They are due before class (iLearn link will expire by class time)
  - The point is to be ready for discussion

- **Next week, I will assign presentations**
  - Email me preference if you don’t want to get stuck with something you don’t like

- **There are 18 of us, so even with one 30 minute presentation each, that is 3 weeks**
  - Would like very much to make the presentations in the flow of the topics rather than standalone classes
Participation is essential

- This is a seminar class
  - It's really boring if I just lecture at you

- Totally fine if you don’t understand something
  - It happens to all of us; questions lead to interesting discussions
  - Broad background required. No one has it all (least of all me!)

- What can I do to make it easier to participate?
Recall Buffer Overflow and Code Injection Attack

main (int argc, char **argv) {
    ...
    vulnerable(argv[1]);
    ...
}

vulnerable(char *str1) {
    char str2[100];
    strcpy(str2, str1);
    return;
}
Possible solutions to code injection attacks

- We talked about stack canaries, shadow stacks and ASLR

- How about Harvard architecture (separate instructions and data memories)?

- How about instruction set randomization?

- Of course, we also talked about the NX (or $W^{⊕}X$) bit
  - A page is either writeable or executable

- Two general approaches:
  1. make sure that malicious code cannot be injected, or
  2. make sure control flow cannot be hijacked
Aside: Architecture/Hardware solutions

- Look up, look down, look back, look forward
- Can it be done in software? Even if not quite as good
  - Remember the strengths and weaknesses of HW
- Over-designing for security
  - How complicated?
  - Does it touch critical datapath elements?
    - What is the performance penalty?
- How useful?
  - Does it solve a problem completely? Does it significantly reduce the attack surface?
- Other factors
  - Is it backward compatible? Does it protect legacy binaries?
  - Does it require major changes to the code base?
Existing Protection from Code Injection Attacks: No Execute Bit (NX)

- Mark memory pages as
  - Either WRITABLE
  - Or EXECUTABLE
  - But not both

- Standard technique in current processors and operating systems
  - Intel XD bit
  - AMD XN bit
  - Windows DEP
  - Linux PaX

```assembly
xor ecx, ecx
mul ecx
lea ebx, [esp+8]
mov al, 11
int 0x80
```
What Does W⊕X Not Prevent?

- Can still corrupt stack ...
  - ... or function pointers or critical data on the heap, but that’s not important right now
- As long as “saved EIP” points into existing code, W⊕X protection will not block control transfer
- This is the basis of return-to-libc exploits
  - Overwrite saved EIP with address of any library routine, arrange memory to look like arguments
- Does not look like a huge threat
  - Attacker cannot execute arbitrary code
  - ... especially if system() is not available
**New Idea: Return-to-libc Attack**

- Overwrite return address with address of libc function
  - setup fake return address and argument(s)
  - ret will “call” libc function

No injected code! How did we exploit?

ret transfers control to `system`, which finds arguments on stack
Code Reuse attacks timeline
H. Shacham

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

(CCS 2007)
Next Frontier: Code Reuse Attacks (CRAs)

- **Key Idea:** Reuse existing library code instead of code injection

- **CRA timeline**
  - Return-into-libc attack (1997)

- All bypass NX
Code Reuse Attacks

give me coffee and no one will get hurt!
Chaining RETs for Fun and Profit

- Can chain together sequences ending in RET
  - Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation using no injected code at all!
Return Oriented Programming (ROP)

Stack frame for `main()`

Stack frame for `vulnerable()`

Address A

Address B

Address C

Address D

Page marked as EXECUTABLE

Stack growth

0x0000

0xFFFF

malicious return

Stack frame for vulnerable()
Instruction pointer (EIP) determines which instruction to fetch and execute.

Once processor has executed the instruction, it automatically increments EIP to next instruction.

Control flow by changing value of EIP.
Return-Oriented Programming

- **Stack pointer** (ESP) determines which instruction sequence to fetch and 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 a NOP sled (what’s that?)
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: Multi-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
“The Gadget”: July 1945

http://rarehistoricalphotos.com/gadget-first-atomic-bomb/
Gadget Design

- **Testbed:** libc-2.3.5.so, Fedora Core 4
- **Gadgets built from found code sequences:**
  - Load-store, arithmetic & logic, control flow, syscalls
- **Found 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
Conditional Jumps

- **cmp** compares operands and sets a number of flags in the EFLAGS register
  - Luckily, many other ops set EFLAGS as a side effect
- **jcc** jumps when flags satisfy certain conditions
  - But this causes a change in EIP... not useful *(why?)*
- **Need conditional change in stack pointer (ESP)**
- **Strategy:**
  - Move flags to general-purpose register
  - Compute either delta (if flag is 1) or 0 (if flag is 0)
  - Perturb ESP by the computed delta
Phase 1: Perform Comparison

- **neg** calculates two’s complement
  - As a side effect, sets carry flag (CF) if the argument is nonzero
- Use this to test for equality
- **sub** is similar, use to test if one number is greater than another
Phase 2: Store 1-or-0 to Memory

1. Clear ECX
2. EDX points to destination
3. \texttt{adc} adds up its operands & the carry flag; result will be equal to the carry flag (why?)
4. Store result of \texttt{adc} into destination
Phase 3: Compute Delta-or-Zero

Bitwise AND with delta (in ESI)

Two’s-complement negation:
0 becomes 0...0;
1 becomes 1...1
Phase 4: Perturb ESP by Delta
Finding Instruction Sequences

- Any instruction sequence ending in RET is useful
- Algorithmic problem: recover all sequences of valid instructions from libc that end in a RET
- At each RET (C3 byte), look back:
  - Are preceding i bytes a valid instruction?
  - Recur from found instructions
- Collect found instruction sequences in a trie
**Example**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Code</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb8800000</td>
<td>pop eax</td>
<td>eax = 1</td>
</tr>
<tr>
<td>0x0000001</td>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>0xb880010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0000002</td>
<td></td>
<td>eax += ebx</td>
</tr>
<tr>
<td>0xb880020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xb880010</td>
<td></td>
<td>ebx = 0x400000</td>
</tr>
<tr>
<td>0x0040000</td>
<td></td>
<td>*ebx = eax</td>
</tr>
<tr>
<td>0xb880030</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Code:

- **0xb8800000**: `pop eax; ret`...
- **0xb8800010**: `pop ebx; ret`...
- **0xb8800020**: `add eax, ebx; ret`...
- **0xb8800030**: `mov [ebx], eax; ret`...
Unintended Instructions

Actual code from ecb_crypt()

movl $0x00000001, -44(%ebp)

test $0x00000007, %edi

setnzb -61(%ebp)

add %dh, %bh

movl $0x0F000000, (%edi)

xchg %ebp, %eax

inc %ebp

ret
Automating Gadget finding

- Galileo algorithm – walk backwards from the return (or indirect branch)
- Q – Map gadgets into functional equivalent
- Everything is automated using compilers
- Demonstrated against many platforms—seems very general
Protecting Against ROP: Discussion

- What could we do?

- There are several approaches, including some that we have seen.

- Take a couple of minutes to come up with one.
Example defense: Secure Call Stack

- Semantics of return instruction
  - `ret` expects an address pushed by a `call`

- Keep a copy of return addresses in a separate memory space
  - Secured by OS/Hardware

- On each return:
  - Compare two copies of the return addresses
  - Generate exception if they mismatch
Example of a Secure Call Stack

```c
main (int argc, char **argv)
{
    ...
    vulnerable(argv[1]);
    ...
}

vulnerable(char *str1)
{
    char str2[100];
    strcpy(str2,str1);
    return;
}
```
Defeating Secure Call Stacks

- **Initiating an attack**
  - Overwrite a function pointer,
  - Overwrite a longjmp buffer,
  - Exploit a format string vulnerability

- **Performing an attack**
  - Use indirect jumps instead of returns
    - Jump Oriented Programming
    - Need a special gadget to orchestrate control
Dispatcher Gadget/trampoline

- Find a gadget that can serve as a trampoline
- Find next gadget, jump to it
- Gadgets now must end in a jump to trampoline
Jump Oriented Programming

Page marked as EXECUTABLE

Dispatcher Gadget

Stack frame for vulnerable()

Stack frame for main()

xor ecx, ecx
mul ecx
lea ebx, [esp+8]
mov al, 11
int 0x80

malicious return

popa ecx, ecx
jmp esi
pop edi
jmp edi
xor ecx, ecx
jmp esi
mul ecx
jmp esi
lea ebx, [esp+8]
jmp esi
mov al, 11
int 0x80

<register values>

<C>
<D>
<F>
<E>

Stack growth

0xFFFF

0x0000
Dispatcher gadget

- `pop %eax; jmp *(%eax)`
  - Other sequences possible
  - Much more rare than rets
    - Trampoline idea – we only need one

- Normal gadgets now have to jump back to trampoline
  - Not that hard to do, but also need indirect branches

- What are the implications for defenses?
  - Those that focus on call-return semantics fail
Hmm....So now what??

- Revisit our ROP ideas...
  - Many break or need to be significantly modified

- Rest of today: randomization based defenses
Today – Randomization based defenses

- Can we randomize “things” about the program to make it harder to attack?
- This class of solutions is called moving-target solution
  - Idea to make the attack surface change for every instance the attacker is attacking
- What can be randomized? Remember ASLR?
  - Unfortunately, ASLR not that hard to defeat
  - Only need to see one code pointer
  - Entropy is low in some implementations
  - Some side channel attacks possible
In-Place Code Randomization [S&P ’12]

• Extend ASLR to a finer-grained level
• Applicable on third-party applications
• (Practically) Zero performance overhead

• Source code (Python):  
  http://nsl.cs.columbia.edu/projects/orp
Why in-place?

• Randomization usually changes the code size
  – Need to update the control-flow graph (CFG)

• But, accurate static disassembly of stripped binaries is hard
  ➔ Incomplete CFG (data vs. code)
  ➔ Code resize not an option

• Must randomize in-place!
Randomizations

• Instruction Substitution

• Instruction Reordering
  – Intra Basic Block
  – Register Preservation Code

• Register Reassignment
Instruction Substitution

```
add [edx],edi
ret

mov al, 0x1
cmp al, bl
    lea eax, [ebp-0x80]

B0 01 3A C3 8D 45 80 50 68
```

```
add [eax],edi
    fmul [ebp+0x68508045]

B0 01 38 D8 8D 45 80 50 68
```

```
mov al, 0x1
    cmp bl, al
    lea eax, [ebp-0x80]
```
Instruction Reordering (Intra BBL)

8B 41 10  mov eax,[ecx+0x10]
53          push ebx
8B 59 0C  mov ebx,[ecx+0xC]
3B C3       cmp eax,ebx
89 41 08  mov [ecx+0x8],eax
7E 4E       jle 0x5c

59          push ebx
0C 3B       or al,0x3B
C3          ret
Instruction Reordering (Intra BBL)

8B 41 10  mov eax,[ecx+0x10]
53       push ebx
8B 59 0C  mov ebx,[ecx+0xC]
3B C3    cmp eax,ebx
89 41 08  mov [ecx+0x8],eax
7E 4E     jle 0x5c

41       inc ecx
10 89 41 08 3B C3
        adc [ecx-0x3CC4F7BF],cl
Register Preservation Code Reordering

Prolog

push ebx
push esi
mov ebx, ecx
push edi
mov esi, edx
.
.
.

Epilog

pop edi
pop esi
pop ebx
ret

push edi
push ebx
push esi
mov ebx, ecx
mov esi, edx
.
.
.

pop esi
pop ebx
pop edi
ret
Register reassignment

function:
push esi
push edi
mov edi,[ebp+0x8]
mov eax,[edi+0x14]
test eax,eax
jz 0x4A80640B
mov ebx,[ebp+0x10]
push ebx
lea ecx,[ebp-0x4]
push ecx
push edi
call eax
...

Live regions

function:
push esi
push edi
mov eax,[ebp+0x8]
mov edi,[edi+0x14]
test edi,edi
jz 0x4A80640B
mov ebx,[ebp+0x10]
push ebx
lea ecx,[ebp-0x4]
push ecx
push eax
call edi
...


Evaluation

• Correctness and performance
  – Execute Wine’s test suite using randomized versions of Windows DLLs

• Randomization coverage
• Effectiveness against real-world exploits
• Robustness against ROP compilers
Randomization Coverage

Dataset: 5,235 PE files (~0.5GB code) from Windows, Firefox, iTunes and Reader
# Real-World Exploits

<table>
<thead>
<tr>
<th>Exploit/Reusable Payload</th>
<th>Unique Gadgets</th>
<th>Modifiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe Reader v9.3.4</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>Integard Pro v2.2.0</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>Mplayer Lite r33064</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>msvcr71.dll (White Phosphorus)</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>msvcr71.dll (Corelan)</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>mscore.dll (White Phosphorus)</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>mfc71u.dll (Corelan)</td>
<td>11</td>
<td>6</td>
</tr>
</tbody>
</table>

Modifiable gadgets were not always directly replaceable!
ROP Compilers

• *Is it possible to create a randomization-resistant payload?*

• **mona.py** constructs DEP+ASLR bypassing code
  – Allocate a WX buffer, copy shellcode and jump

• **Q** is the state-of-the-art ROP compiler [SAB11]
  – Designed to be robust against small gadget sets
# ROP Compilers Results

<table>
<thead>
<tr>
<th>Non-ASLR Code Base</th>
<th>Mona</th>
<th>Q</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig.</td>
<td>Rand.</td>
<td>Orig.</td>
</tr>
<tr>
<td>Adobe Reader v9.3.4</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Integard Pro v2.2.0</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Mplayer Lite r33064</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>msvcr71.dll</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>mscorei.dll</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>mfc71u.dll</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

Both failed to construct payloads from non-randomized code!
This class of solutions – defeated (1)
This class of solutions – defeated (2)

(2)
Other Randomization ideas? ISR

- Instruction Set Randomization ~2003
  - Older idea, can protect against code injection attacks too
  - What do you think?
Can the key be guessed?

32 bit key => 4,294,967,296 possibilities

32 bit key, guess 16 bits and 16 bits =>
2 * 65,536 = 131,072 possibilities

32 bit key, guess 8 bits at a time =>
4 * 256 = 1,024 possibilities
Problems

<table>
<thead>
<tr>
<th>Outcome</th>
<th>Count</th>
<th>Percent</th>
<th>Cumulative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signalled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>25,162</td>
<td>99.82%</td>
<td>99.82%</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4,504</td>
<td>0.18%</td>
<td></td>
</tr>
<tr>
<td>SIGFPE</td>
<td>178</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>101</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>Looped (timeout)</td>
<td>55</td>
<td>0.18%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Acquired shell</td>
<td>0</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

Total tests: 30000

[Randomized instruction set emulation to disrupt binary code injection attacks, Barrantes & all]
Demonstrated a lovely attack in usenix 2005 (Where’s the FEEB?)

• Attacked a multi-threaded network server
• Try to guess the return instruction—only 256 possibilities
  – Unfortunately program crashes if you guess wrong
• https://www.usenix.org/legacy/event/sec05.tech/full_papers/sovarel/sovarel.pdf
• Leaves the stack compromised when successful
  – Popped an extra value
• Guess jump instruction
  – 2 bytes, so 16K possibilities
  – But can cause an infinite loop when successful
Conclusions

• Moving target defenses are generally a good idea
  – Can be hardware or software supported
  – Examples?
• However, they can be circumvented sometimes
  – Low entropy
  – Disclosure attacks
• Fundamental limitation or limitation of current solutions?