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

Bader AlBassam
bader@mail.usf.edu

University of South Florida

2015/October/05
Outline

1. Introduction
   - Background
   - Existing vulnerabilities

2. Building Blocks for Attack
   - Intel x86 ISA
   - Using Sequences in Crafting an Attack

3. Useful Instruction Sequences
   - Useful instructions
   - Boring instructions

4. Return-Oriented Programming
   - Progressing through instructions
   - Example gadgets

5. Return-Oriented Shellcode

6. Conclusion
Definition (Exploit)

[When an attacker] subverts a program’s control flow such that it performs actions of his choice with its credentials.
Traditional Vulnerabilities

- Buffer-overflow on the stack
Traditional Vulnerabilities

- Buffer-overflow on the stack
- Buffer-overflow on the heap
Traditional Vulnerabilities

- Buffer-overflow on the stack
- Buffer-overflow on the heap
- Integer-overflow
Traditional Vulnerabilities

- Buffer-overflow on the stack
- Buffer-overflow on the heap
- Integer-overflow
- Format string vulnerabilities
In each of the previous vulnerabilities, the attacker had two things that they must accomplish:
In each of the previous vulnerabilities, the attacker had two things that they must accomplish:

1. Find some way to subvert the program’s control flow from the normal course
In each of the previous vulnerabilities, the attacker had two things that they must accomplish:

1. Find some way to subvert the program’s control flow from the normal course
2. Cause the program to act the manner of choosing
In each of the previous vulnerabilities, the attacker had two things that they must accomplish:

1. Find some way to subvert the program’s control flow from the normal course
2. Cause the program to act the manner of choosing

This paper targets the second goal of an attacker.
Traditional Stack Smashing Attack
Traditional Stack Smashing Attack

- Unallocated Stack Space
- h e l l
- o \0
- Char c[12]
- Char *bar
- Saved Frame pointer
- Return Address
- Parent Routine's Stack
Traditional Stack Smashing Attack

Address 0x80C03508

Unallocated Stack Space

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

Stack Growth

Memory Addresses

Little Endian 0x80C03508

Parent Routine's Stack

\x08 \x35 \xC0 \x80
Defending Against Attacks

Common defense is to prevent attack from executing injected code
Defending Against Attacks

Common defense is to prevent attack from executing injected code

- Solar Designer’s StackPatch: Make stack non-executable
Defending Against Attacks

Common defense is to prevent attack from executing injected code

- Solar Designer’s StackPatch: Make stack non-executable
- \(W \oplus X\): No memory is both \(W\) and \(X\).
  - Cannot inject code to be executed
Defending Against Attacks

Common defense is to prevent attack from executing injected code

- Solar Designer’s StackPatch: Make stack non-executable
- $W \oplus X$: No memory is both $W$ and $X$.
  - Cannot inject code to be executed
  - Exists in OpenBSD and patched into Linux as part of PAX
Defending Against Attacks

Common defense is to prevent attack from executing injected code

- Solar Designer’s StackPatch: Make stack non-executable
- $W \oplus X$: No memory is both $W$ and $X$.
  - Cannot inject code to be executed
  - Exists in OpenBSD and patched into Linux as part of PAX
  - Added to AMD and Intel as ”NX” and ”XD” respectively as a per-page execution disable
Instead of injecting code, now attackers use existing code. Solar Designer was the first to suggest the approach of return-to-libc.
Return-to-libc

Instead of injecting code, now attackers use existing code. Solar Designer was the first to suggest the approach of return-to-libc

*Any Unix-like operating system needs a C library: the library which defines the “system calls” and other basic facilities such as open, malloc, printf, exit...*
Instead of injecting code, now attackers use existing code. Solar Designer was the first to suggest the approach of return-to-libc

*Any Unix-like operating system needs a C library: the library which defines the “system calls” and other basic facilities such as open, malloc, printf, exit...*

In principle any code available can be used
Ret-to-libc is more limited than code injection

1. Attacker can only execute straight-line code
2. Attacker can only invoke functions available in programs .text and loaded libraries.
Ret-to-libc is more limited than code injection

1. Attacker can only execute straight-line code
2. Attacker can only invoke functions available in programs .text and loaded libraries.

While the above can weaken attackers, this paper shows a new ret-to-libc attack that allows arbitrary executions
Compared to libc

- Traditional ret-to-libc attacks are built on functions that exist in the program.
- This attack is based on short code sequences, two or three instructions long.
- Some are present despite not being placed in the code without the compiler.
Intel x86 is a variable length instruction set CISC architecture.

- Processor knows where to start and continue
Intel x86 is a variable length instruction set CISC architecture.

- Processor knows where to start and continue
- A random byte stream is very likely to be interpreted as valid
Concrete example

Listing 1: Two instructions in ecb_crypt entrypoint

```
f7 c7 07 00 00 00       test $0x00000007 %edi
0f 95 45 c3             setnzb −61(%ebp)
```

Listing 2: Same instructions starting one byte later

```
c7 07 00 00 00 0f       movl $0x0f000000,(%edi)
95                        xchg %ebp,%eax
45                        inc %ebp
c3                         ret
```
Concrete example

Listing 3: Two instructions in ecb_crypt entrypoint

```
f7 c7 07 00 00 00  
test $0x00000007 %edi
0f 95 45 c3
setnz b −61(%ebp)
```

Listing 4: Same instructions starting one byte later

```
c7 07 00 00 00 0f
movl $0x0f000000 ,(%edi)
95
xchg %ebp, %eax
45
inc %ebp
c3
ret
```

Static analysis can find sequences of useful instructions
Building gadgets is difficult

**Definition (Gadget)**

*Short blocks placed on the stack that chain several of instruction sequences together*
Building gadgets is difficult

Definition (Gadget)

*Short blocks placed on the stack that chain several of instruction sequences together*

- Very short instruction sequences
Building gadgets is difficult

**Definition (Gadget)**

*Short blocks placed on the stack that chain several of instruction sequences together*

- Very short instruction sequences
- Neither function prologue nor epilogue
Building gadgets is difficult

Definition (Gadget)

Short blocks placed on the stack that chain several of instruction sequences together

- Very short instruction sequences
- Neither function prologue nor epilogue
- Sequences have haphazard interfaces (Do not follow ABI)
Building gadgets is difficult

Definition (Gadget)

*Short blocks placed on the stack that chain several of instruction sequences together*

- Very short instruction sequences
- Neither function prologue nor epilogue
- Sequences have haphazard interfaces (Do not follow ABI)

Previous attacks used `pop %reg; ret` to set registers to chain function calls.
Building gadgets is difficult

**Definition (Gadget)**

*Short blocks placed on the stack that chain several of instruction sequences together*

- Very short instruction sequences
- Neither function prologue nor epilogue
- Sequences have haphazard interfaces (Do not follow ABI)

Previous attacks used `pop %reg; ret` to set registers to chain function calls. Gadgets are Turing complete!
Definition (Useful)

A sequence is useful if it could be used in a gadget. i.e. it is a sequence of valid instructions ending in a ret.
Definition (Boring)

An instruction is "boring" if it is either

1. a leave and followed by a ret
2. a pop %ebp followed immediately by a ret
3. a return or an unconditional jump
Theorem

If “a;b;c;ret” exists then “b;c;ret” must also exist.
ESP is the new EIP

Instead of the instruction pointer being automatically incremented to gain the new instruction, the stack pointer determines the next gadget. ESP is incremented every time there is a “ret” instruction.
Memory Manipulation

Figure 2: Load the constant 0xdeadbeef into %edx.
Figure 3: Load a word in memory into %eax.
Memory Manipulation

Figure 4: Store %eax to a word in memory.
**ALU operations**

![Diagram of ALU operations with instructions](image)

Figure 5: Simple add into `%eax`. 
ALU operations

Figure 6: Simple add into %eax: State after push %edi is executed.
ALU operations

Figure 7: Repeatable add into %eax.
Figure 10: An infinite loop by means of an unconditional jump.
Getting to /bin/sh

Here is where we write our exploit
ShellCode

1. Sets %eax to zero.
2. Load %edx to address of second word in argv, load into %ecx 0b
4. Sets %eax to 0x0b
5. Points %ebx to "/bin/sh"
6. Sets %ecx to address of argv[], %edx to address of envp[]
7. Traps into kernel

Figure 16: Shellcode.
References I
