Introduction to Reverse Engineering

Alan Padilla, Ricardo Alanis, Stephen Ballenger, Luke Castro, Jake Rawlins
Reverse Engineering (of Software)

- What is it?
  - Taking stuff apart and learning how it works. Specifically, we are taking apart programs

- What is it for?
  - Binary exploitation (the cool topic)
  - Malware analysis
  - Other stuff

- Binary exploitation
  - OG hacking. Way harder and cooler than web hacking.
    - But (mostly) kidding

- A word on “hacking”...
  - Learn the technology
  - Sprinkle in some ingenuity
Not Another Boring Text Slide

This stuff is cool. Not gonna make you take my word for it though. Demo time.

```c
#include <stdio.h>
#include <string.h>

void main (int argc, char*argv[]) {
    copier(argc);  // Fixed typo: argc to argv
    printf("Done\n");
}

int copier (char *str) {
    char buffer[100];
    strcpy(buffer,str);
    printf("You entered \"%s\" at %p\n", buffer, buffer);
}
```
Ok, this one is another boring text slide

Why did that happen? How did it happen?

Like any sort of hacking, learn how something works, sprinkle in some ingenuity, bend some rules, and all the root shells will be yours.

Hopefully you will be able to do this by the end of this presentation, and you will be a real life Mr. Robot.

...But first you have to learn the background of how stuff works, before you can exploit it.
Basics
What is a Program?

- A program is a collection of instructions that performs a specific task when executed by a computer.
  - At the lowest level, programs are a series of binary bits, 0 and 1.
Numbering Systems

- **Base 10 (Decimal)** - The representation of numbers we are most familiar with.
  - Each digit (0-9) is a product of a power of 10, for example:
    - $6197 = 7 \times 10^0 + 9 \times 10^1 + 1 \times 10^2 + 6 \times 10^3 = 7 \times 1 + 90 \times 10 + 1 \times 100 + 6 \times 1000 = 6197$

- **Base 2 (Binary)** - The representation of numbers processed by computers.
  - Each digit (0 and 1) is a product of a power of 2, for example:
    - $1011 = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 1 \times 1 + 1 \times 2 + 0 \times 4 + 1 \times 8 = 11$

- **Base 16 (Hexadecimal)** - The representation of numbers used by programmers to represent long binary numbers concisely.
  - Contains 0 - 9 and A - F as digits where each is a product of a power of 16. For example:
    - $0xC5 = 5 \times 16^0 + 12 \times 16^1 = 5 + 192 = 197$
  - Note: Many times hexadecimal numbers are preceded by “0x” to denote their base.
A **bit** is a single binary digit, 0 or 1.

A **byte** is a group of eight bits.

- For example, 00110101 = 0x35

A **word** is a group of 2 bytes, or 16 bits.

- For example, 01101001101101 = 0x69AD
Memory Layout

- **Code** - instructions fetched by the CPU to execute the program’s tasks
- **Heap** - used for dynamic memory during execution, creates (allocate) new values and eliminates (free) values that the program no longer needs
- **Stack** - used for local variables and parameters for functions, and to help control program flow. Last-In-First-Out
Little and Big Endianness

- Little Endian - “little end” is where the least significant byte of a word or larger is stored in the lowest address. Used for variables in memory.
- Big Endian - “big end” is how we read it sort of left to right. Typically used for Network Traffic

Big Endian: 0x12345678

Little Endian: 0x78563412
X86 Assembly
ASM

- Lowest-level programming language

```c
#include <stdio.h>
int main(){
    printf("Hello World!\n");
    return 0x1234;
}

push  ebp
mov   ebp, esp
push  offset aHelloWorld ; "Hello world\n"
call  ds:__imp__printf
add   esp, 4
mov   eax, 1234h
pop   ebp
retn
```
Intel vs AT&T

Intel

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

AT&T

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

Bytes—8 bits. Examples: **AL, BL, CL**

Word—16 bits. Examples: **AX, BX, CX**

Double word—32 bits. Examples: **EAX, EBX, ECX**

Quad word—64 bits. Not found in x86 architectures but instead combines two registers usually **EDX:EAX**.
## Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>Stores function return values</td>
</tr>
<tr>
<td>EBX</td>
<td>Base pointer to the data section</td>
</tr>
<tr>
<td>ECX</td>
<td>Counter for loop operations</td>
</tr>
<tr>
<td>EDX</td>
<td>I/O pointer</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>Holds single bit flags</td>
</tr>
<tr>
<td>ESI</td>
<td>Source pointer for string operations</td>
</tr>
<tr>
<td>EDI</td>
<td>Destination pointer for string operations</td>
</tr>
<tr>
<td>ESP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>EBP</td>
<td>Stack frame base pointer</td>
</tr>
<tr>
<td>EIP</td>
<td>Pointer to next instruction to execute (&quot;instruction pointer&quot;)</td>
</tr>
</tbody>
</table>
Evolution of Register

$$\begin{array}{c|c|c|c|c|c|c|c|c|c} & & & & & & & & & \\ \hline \text{EAX} & & & & & & & & & \\ \hline & 1010 & 1001 & 1101 & 1100 & 1000 & 0001 & 1111 & 0101 & \text{Binary} \\ & A & 9 & D & C & 8 & 1 & F & 5 & \text{Hex} \\
\hline \end{array}$$

$$\begin{array}{c|c|c|c|c|c|c|c|c|c} & & & & & & & & & \\ \hline \text{AX} & & & & & & & & & \\ \hline & 1000 & 0001 & 1111 & 0101 & \text{16 bits} \\ & 8 & 1 & F & 5 & \\
\hline \end{array}$$

$$\begin{array}{c|c|c|c|c|c|c|c|c|c} & & & & & & & & & \\ \hline \text{AH} & & & & & & & & & \\ \hline & 1000 & 0001 & \text{8 bits} \\ & 8 & 1 \\
\hline \end{array}$$

$$\begin{array}{c|c|c|c|c|c|c|c|c|c} & & & & & & & & & \\ \hline \text{AL} & & & & & & & & & \\ \hline & 1111 & 0101 & \text{8 bits} \\ & F & 5 \\
\hline \end{array}$$
**Important X86 Instructions**

**push** - “Pushes” DWORD onto Stack. decrements the stack pointer, esp, by 4 bytes.

```plaintext
push eax
```

**pop** - “pops” DWORD off Stack onto a register. Increments the stack pointer, esp, by 4 bytes.

```plaintext
pop eax
```
X86 Instructions continued

`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
X86 Arithmetic

add eax, 0x5
sub eax, 0x5
mul eax, edx: stores value in edx:eax
div eax, edx: stores dividend in eax, remainder in edx
inc edx: increments edx by 1
dec ecx: decrements edx by 1
push, pop, mov, add - In action

push    ebp
mov     ebp, esp
push    offset aHelloWorld ; "Hello world\n"
call    ds:__imp__printf
add     esp, 4
mov     eax, 1234h
pop     ebp
retn

- Push stack frame
- Move current stack frame
- Push “Hello world” onto stack for parameter to call
- Call print function
- Add 4 to stack pointer
- Move 1234h into aex
- Pop old stack frame pointer return
- Return to next instruction
X86 Instructions continued

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
cmp, jmp - In action

```assembly
sum = 0;
for (i = 0; i <= 10; i++)
    sum += i

    mov eax, 0
    mov ebx, 0

loop_start:
    cmp ebx, 10
    jg    loop_end
    add    eax, ebx
    inc    ebx
    jmp    loop_start

loop_end:
```

- eax will hold sum
- ebx will hold i

- Compare i with 10
- If greater than jump to the loop_end
- Else add i to sum
- Increment i
- Jump back to start of loop
Static Analysis
What is Static Analysis?

Analyzing the code and structure of a program without actually running the program.
What are you analyzing?

paint.exe? sketchy.exe?

Integrity - make sure the program you download/run is the one the trusted source created.

Hash it! Check it on VirusTotal. Verify.

Tools to use:

- **shasum**
  ```
  → in TexSaw shasum sketchy.exe
  b7f1c0ed73b98039819c1bb8118182802f465da1 sketchy.exe
  ```

- **md5**
  ```
  → in TexSaw md5 sketchy.exe
  MD5 (sketchy.exe) = f02f45007a0dc907bc487b35b5b314fe
  ```
Strings

“Any word or phrase is a string just like this one”

Searching through the strings can be a simple way to get hints about the functionality of a program.

Strings can give you:

- URLs
- PASSWORDS
- Standard library calls

*Diagrams from Practical Malware Analysis*
Strings: Tools

GNU Strings:
  - ASCII
  - UNICODE: UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE

FLOSS:
  - More powerful String finder: Obfuscated Strings (purposely garbled strings)
  - ASCII, UTF-16LE
Decompilers

Turning 01’s into readable Assembly Language

Useful for analyzing a program’s structure and procedures.

Tools used:

- IDA Pro
- Binary Ninja
- Radare2
Dynamic Analysis
What is Dynamic Analysis

The analysis of a program while it is running, to observe its true functionality

This allows you to view the transfer of state within a program

Dynamic Analysis should only be performed after static analysis has been completed.
Tools

Linux: GDB, Immunity Debugger

Windows: OllyDBG, WinDBG
GDB Walkthrough

Command line interface

- Step through programs
- View stack
- Jump through memory addresses

GDB Cheat Sheet!
Dynamic Analysis Limitations

Not all functionalities may execute when a program is run

- Command line arguments
- Branches in code
Dynamic Analysis and Malware

Dynamic analysis techniques on malware can put your system and network at risk!

Virtual Machines and Sandboxes allow dynamic analysis on malware

- Cuckoo Sandbox
- Virtualbox/VMWare
Basic Dynamic Analysis on Malware

Process Monitoring

- Top

Virtual Networking

- FakeNet-NG / INetSim

Network Traffic Logging

- WireShark
- NetCat
Buffer Overflow Exploitation
Buffer Overflow

- Putting more data into a buffer than there is space allocated
- Changes program flow, sends stack pointer (SP) to another address
Buffer Overflow

- Four possibilities, SP is sent:
  - to a virtual address that isn’t mapped to a physical address
  - to a protected address (kernel)
  - to an address that has no executable instruction (NOP)
  - to an address that contains an instruction
```c
int copier (char *str) {
    char buffer[100];
    strcpy(buffer, str);
    printf("You entered \"%s\ at %p\n", buffer, buffer);
} // end function copier
```
int copier (char *str) {
    char buffer[100];
    strcpy(buffer, str);
    printf("You entered \"%s\ at %p\n", buffer, buffer);
} // end function copier
Buffer Overflow

```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);
}
```
GDB

```
EBP: 0xbfffeb0f8 -> 0x0
ESP: 0xbfffeb0f0 -> 0xb7fba3dc -> 0xb7fbd1e0 -> 0x0
EIP: 0x804847e (<main+19>: mov eax,DWORD PTR [eax+0x4])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)

[---code---]
0x8048478 <main+13>: push ecx
0x8048479 <main+14>: sub esp,0x4
0x804847c <main+17>: mov eax,ecx
0x804847e <main+19>: mov eax,DWORD PTR [eax+0x4]
0x8048481 <main+22>: add eax,0x4
0x8048484 <main+25>: mov eax,DWORD PTR [eax]
0x8048486 <main+27>: sub esp,0xc
0x8048489 <main+30>: push eax

[---stack---]
0000 | 0xbfffeb0f0 -> 0xb7fba3dc -> 0xb7fbd1e0 -> 0x0
0004 | 0xbfffeb0f4 -> 0xbfffeb10 -> 0x2
0008 | 0xbfffeb0f8 -> 0x0
0012 | 0xbfffeb0fc -> 0xb7f2e0637 (<libc_start_main+247>: add esp,0x10)
0016 | 0xbfffeb0f0 -> 0xb7fba000 -> 0xb1bd0b0
0020 | 0xbfffeb0f4 -> 0xb7fba000 -> 0xb1bd0b0
0024 | 0xbfffeb0f08 -> 0x0
0028 | 0xbfffeb0f0c -> 0xb7f2e0637 (<libc_start_main+247>: add esp,0x10)

Legend: code, data, rodata, value
```

Breakpoint 1, main (argc=0x2, argv=0xbffeca4) at overflow_example.c:5
5 copier(argv[1]);
gdb-peda$
<table>
<thead>
<tr>
<th>Register</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0xbffffec10</td>
<td>0x2</td>
</tr>
<tr>
<td>EBX</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>ECX</td>
<td>0xbffffec10</td>
<td>0x2</td>
</tr>
<tr>
<td>EDX</td>
<td>0xbffffec34</td>
<td>0x0</td>
</tr>
<tr>
<td>ESI</td>
<td>0xb7fba000</td>
<td>0x1b1db0</td>
</tr>
<tr>
<td>EDI</td>
<td>0xb7fba000</td>
<td>0x1b1db0</td>
</tr>
<tr>
<td>EBP</td>
<td>0xbffffebf8</td>
<td>0x0</td>
</tr>
<tr>
<td>ESP</td>
<td>0xbffffebf0</td>
<td>0xb7fba3dc 0xb7fbb1e0 0x0</td>
</tr>
</tbody>
</table>
GDB

EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)

```assembly
0x80484ab <copier>: push ebp
0x80484ac <copier+1>: mov ebp,esp
0x80484ae <copier+3>: sub esp,0x78
0x80484b1 <copier+6>: sub esp,0x8
0x80484b4 <copier+9>: push DWORD PTR [ebp+0x8]
0x80484b7 <copier+12>: lea eax,[ebp-0x6c]
0x80484ba <copier+15>: push eax
0x80484bb <copier+16>: call 0x8048330 <strcpy@plt>
```

Breakpoint 3, copier (str=0xbffeed '220' <repeats 64 times>, "$061\300\211\0271\232Rhn/shh/\bi/\211\343\211\341\215\v\n\'A' <repeats 16 times>, "$\353\377\277")

```c
at overflow_example.c:11
11     strcpy(buffer,str);
```

gdb-peda$
<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>0xbfffeeed</td>
<td>0x90909090</td>
</tr>
<tr>
<td>EBX</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>ECX</td>
<td>0xbfffc10</td>
<td>0x2</td>
</tr>
<tr>
<td>EDX</td>
<td>0xbfffc34</td>
<td>0x0</td>
</tr>
<tr>
<td>ESI</td>
<td>0xb7fba000</td>
<td>0x1b1db0</td>
</tr>
<tr>
<td>EDI</td>
<td>0xb7fba000</td>
<td>0x1b1db0</td>
</tr>
<tr>
<td>EBP</td>
<td>0xbfffebdb8</td>
<td>0xbfffebfb8</td>
</tr>
<tr>
<td>ESP</td>
<td>0xbfffeb60</td>
<td>0x0</td>
</tr>
</tbody>
</table>
gdb-peda$ x/40w $esp

0xbfffeb60:  0x00000000 0x00000001 0xb7ffff918 0x90909090
0xbfffeb70:  0x90909090 0x90909090 0x90909090 0x90909090
0xbfffeb80:  0x90909090 0x90909090 0x90909090 0x90909090
0xbfffeb90:  0x90909090 0x90909090 0x90909090 0x90909090
0xbfffeba0:  0x90909090 0x90909090 0x90909090 0xc389c031
0xbfffebb0:  0x80cd17b0 0x6852d231 0x68732f6e 0x622f2f68
0xbfffebc0:  0x52e38969 0x8de18953 0x80cd0b42 0x41414141
0xbfffebd0:  0x41414141 0x41414141 0x41414141 0xbfffeb84
0xbfffebe0:  0xbfffee00 0xbfffeeca4 0xbfffccb0 0x080f8501
0xbfffebf0:  0xb7fba3dc 0xbfffece10 0x00000000 0xb7e20637
Continuing.

process 3446 is executing new program: /bin/zsh5

Error in re-setting breakpoint 1: No source file named /home/seed/Downloads/bufferoverflowexamplefiles/overflow_example.c.

Error in re-setting breakpoint 2: No source file named /home/seed/Downloads/bufferoverflowexamplefiles/overflow_example.c.

Error in re-setting breakpoint 3: No source file named /home/seed/Downloads/bufferoverflowexamplefiles/overflow_example.c.

Error in re-setting breakpoint 4: No source file named /home/seed/Downloads/bufferoverflowexamplefiles/overflow_example.c.

Error in re-setting breakpoint 5: No source file named /home/seed/Downloads/bufferoverflowexamplefiles/overflow_example.c.

$
You entered 'AAAAAAAAAAAAAAAAAAAAAA????' at 0xbфффеб5с

whoami
root
ls
overflow_example  payload  peda-session-overflow_example.txt
overflow_example.c  peda-session-ls.txt  printBuffer.py
“Advanced” Topics
Other Attacks

Congrats! You are now a super l33t hacker!

...Of the 1980s. The attack demo’d here is old news

Some other attacks you may want to google on your own time:

- Printf arbitrary read/write
- Heap overflow
- Data leakage
More Stuff To Google

Protections

- Non-executable Stack
- Address Space Layout Randomization (ASLR)
- Stack Canaries

...And Circumventing Those Protections

- NOP-sledding
- Data leakage
- Return-to-libc attack
- ROP chaining
Takeaway

A stack overflow attack is just one (classic) example of exploiting program logic to do cool stuff.

Hacking is about learning the rules and coming up with a neat way to do unexpected things within those rules.

The example we showed today is just that: one example. Exploitation of logic flaws can take countless forms.

Get familiar with how stuff works and you’ll be ready to start hacking!
NOVEMBER 2 – 3

TexSAW
2018

8th ANNUAL
TEXAS SECURITY AWARENESS WEEK

ERIK JONSSON SCHOOL OF ENGINEERING AND COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

UT DALLAS

State Farm®