Reverse Engineering 101

Theory of Reverse Engineering and Static Analysis

Agenda

- WTF IS RE
- Myths and misconceptions
- All you need to know about Assembly
- The 4 steps of Reverse Engineering
- How to be a binary ninja using Binary Ninja (and other tools)
- If we have time: Using Python to make your life easier

WTF IS RE

Reverse engineering

verb (used with object)

to study or analyze (a device, as a microchip for computers) in order to learn details of design, construction, and operation, perhaps to produce a copy or an improved version.

In this class, we will be dealing with compiled programs that were written in C, and we will be trying to find some information hidden in the program (a **flag**)

We'll use several different names for programs in this course - binaries, assemblies, services, etc

WTF IS RE

We accomplish this goal using a variety of tools:

Disassemblers - Convert the machine code of a program to (somewhat) human readable assembly code

Debuggers - You should know what these are, but we use these to get information about the program's state given some input (we'll cover these next week)

Other tools - provide information about the compiled program, edit the binary, convert the binary to another format, etc

A web browser and a connection to Google - arguably as important as a disassembler

Disassemblers are the focus of today

WTF IS RE

You might think that Reverse Engineering is reading all the assembly in a binary, and then figuring out what is hidden.

This is wrong.

If you do this, you will never RE anything non-trivial in a reasonable amount of time.

RE is not about reading assembly code.

RE is the art of **NOT** reading assembly.

WTF IS RE

~90% of the assembly in a given binary is irrelevant to our task

So our job is to figure out what parts of the binary are relevant, then read them.

We want to reduce the amount of assembly we need to look at (reduction of work). I'll tell you how to do this efficiently

Once you've found the assembly, reading it is easy and you can figure out what you need to do to get the flag

Myths about RE

- Myth: Reverse Engineering is hard
  - Wrong. Math is hard, crypto is hard, exploitation is hard. RE is just tedious.
- Myth: You need to know assembly really well to be able to RE
  - I can't write assembly for the life of me, but I can read it despite forgetting what half the instructions do every time I open a disassembler
  - Reading assembly is usually pretty straightforward
- Myth: You need $1000s worth of software to RE effectively
  - This used to be true, but not anymore. Binary Ninja is pretty affordable, and radare2 is free (but it sucks)

ALL YOU NEED TO KNOW ABOUT x86 ASSEMBLY

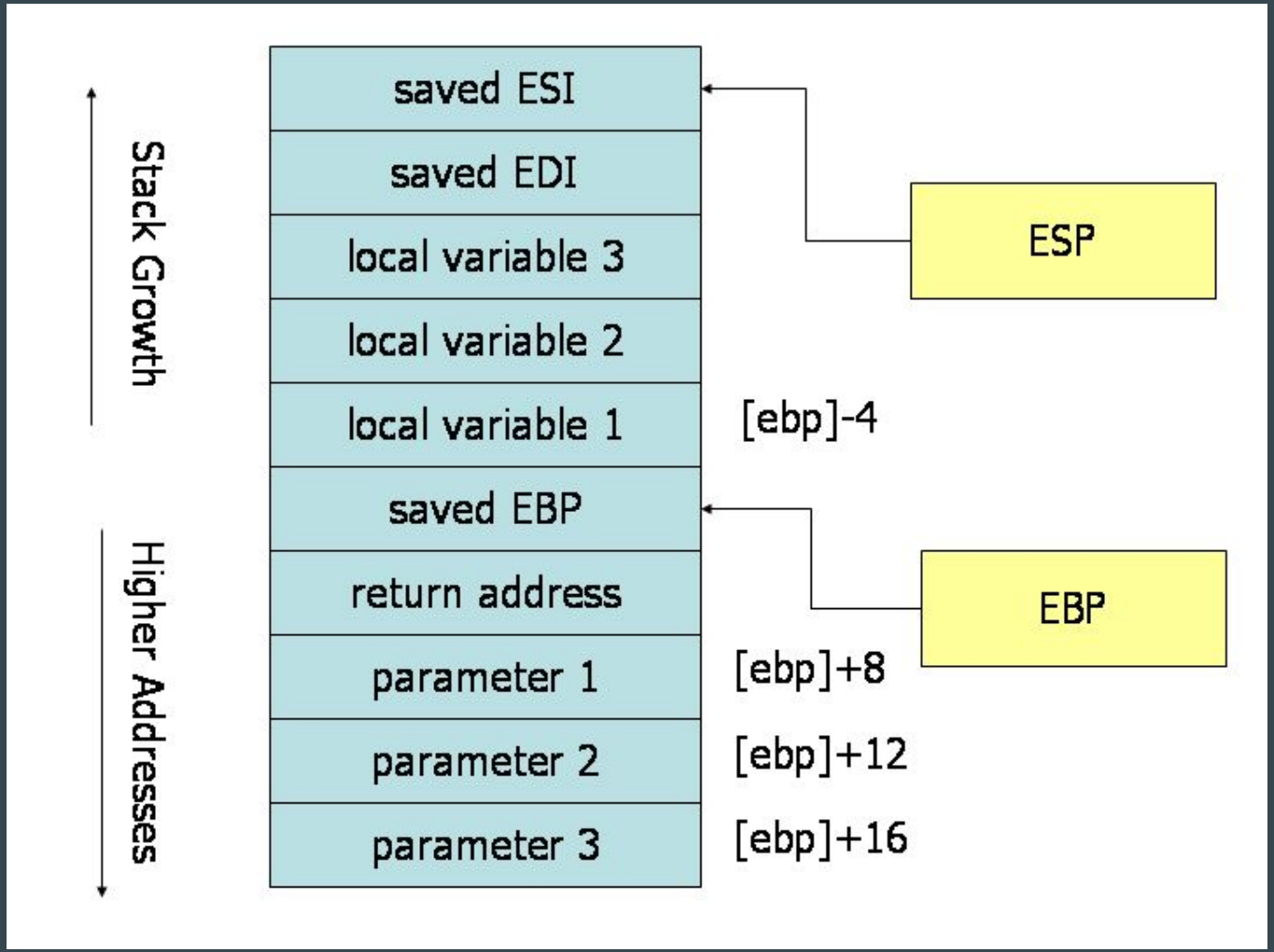
- Instructions are of the form “opcode destination, source”
  - mov eax, ebx
  - add ebx, ecx
  - Numbers are usually expressed in hexadecimal, in these two forms: 0x1A or 1Ah
- There are registers: eax, ebx, ecx, edx, esi, edi, esp, ebp, some others....
  - They are all 32 bit, however the word size is 16 bits (2 bytes). Each register is a doubleword
  - The first four registers can be referenced as 16 bit and 8 bit registers:
    - ax = last 16 bits of eax, al = last 8 bits of eax, ah = first 8 bits of ax (not often used)
- [something] is a pointer dereference treating something as a pointer
  - *some_ptr in C
  - You can put expressions in the something
- “lea eax, [some_expr]” computes the result of some_expr and stores it in eax
  - It doesn’t modify memory at all

ALL YOU NEED TO KNOW ABOUT x86 ASSEMBLY

- Conditionals start with a cmp instruction, then a conditional jump
  - cmp subtracts the two operands and updates flags, doesn't actually update the registers
  - test works similarly, but ANDs the operands together
- The conditional jumps are simple acronyms:
  - je = Jump Equal
  - jne = Jump Not Equal
  - jge = Jump Greater Than or Equal
  - jz = Jump Zero (same as je)

ALL YOU NEED TO KNOW ABOUT x86 ASSEMBLY

- There is a stack, which is modified with the push and pop instructions
  - Unlike conventional stacks, the stack grows downwards. So push eax **decrements** the stack pointer (esp)
- Local variables are stored on the stack and are accessed like this: [ebp-some_offset]
  - Or like this: [esp+some_offset-some_other_offset]
  - Function arguments are accessed like [ebp+some_offset] (note the addition instead of subtraction)
- The call instruction pushes the current address onto the stack, then jumps to the operand
  - The ret instruction pops an address off the stack and jumps to it
  - This detail is very important for binary exploitation (week 4 & 5)


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

ALL YOU NEED TO KNOW ABOUT x86 ASSEMBLY

- Function calls look like this:
- The return value is placed in eax

```
push arg3
push arg2
push arg1
push arg0
call function
```

```
mov [esp+8], arg2
mov [esp+4], arg1
mov [esp], arg0
call function
```

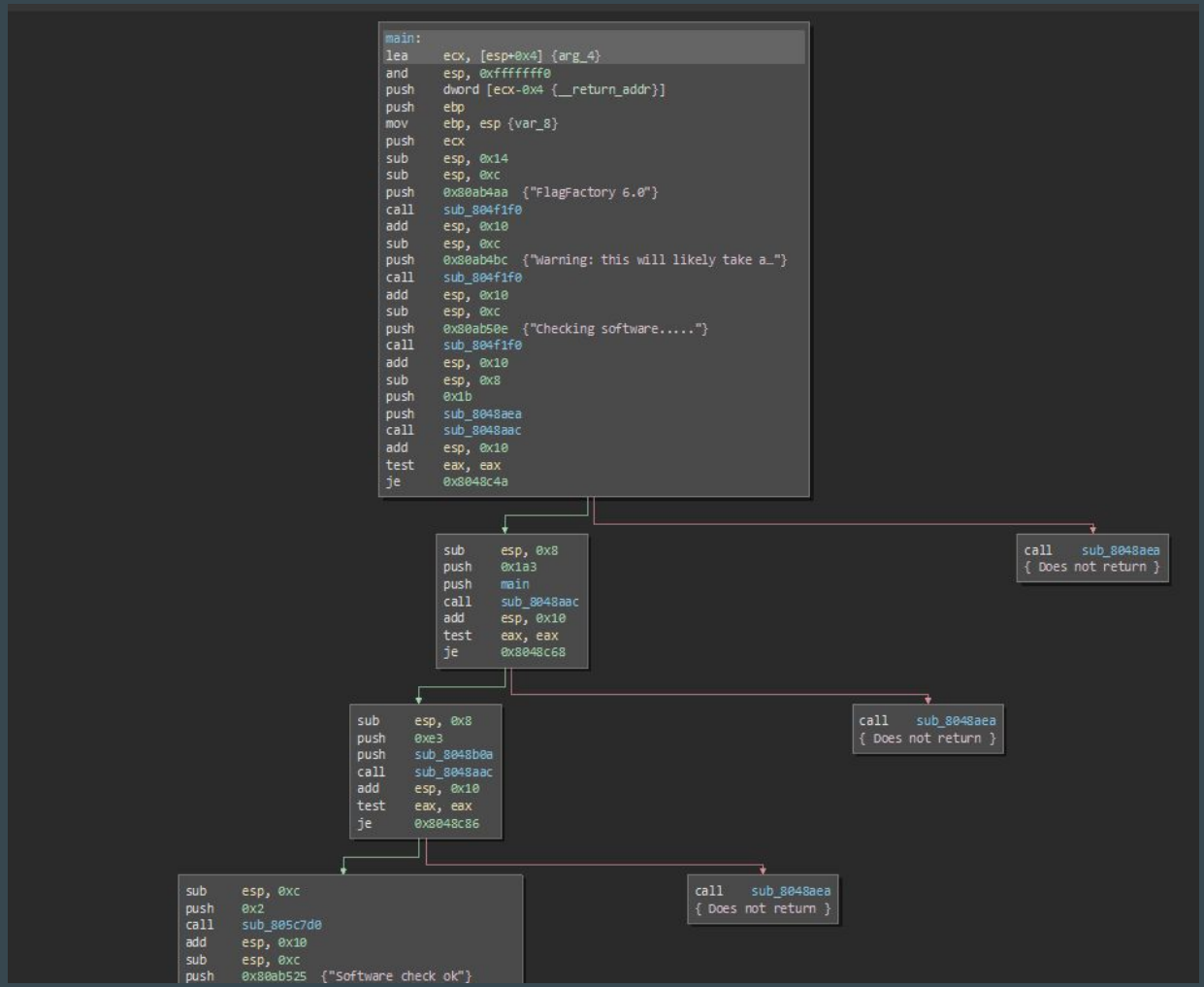
- Most assembly instructions are straightforward, you can Google them if you don't understand what an instruction does
  - No one knows what "repne scasb" does off to the top of their head, they just google it every time.
- 99% of the instructions that directly modify esp or ebp aren't worth looking at

Static Analysis

Static analysis involves using some tools to discover things about the binary, without actually running it. We will focus on disassemblers today.

Basic disassemblers just dump the assembly code into a text file, but the ones used in reverse engineering do some more analysis and allow you to browse the disassembly visually. They do this by splitting up code into “basic blocks”

Since looking at all the assembly code is a waste of time, we will break this process up into 4 steps (3 of which we will do inside of a disassembler)

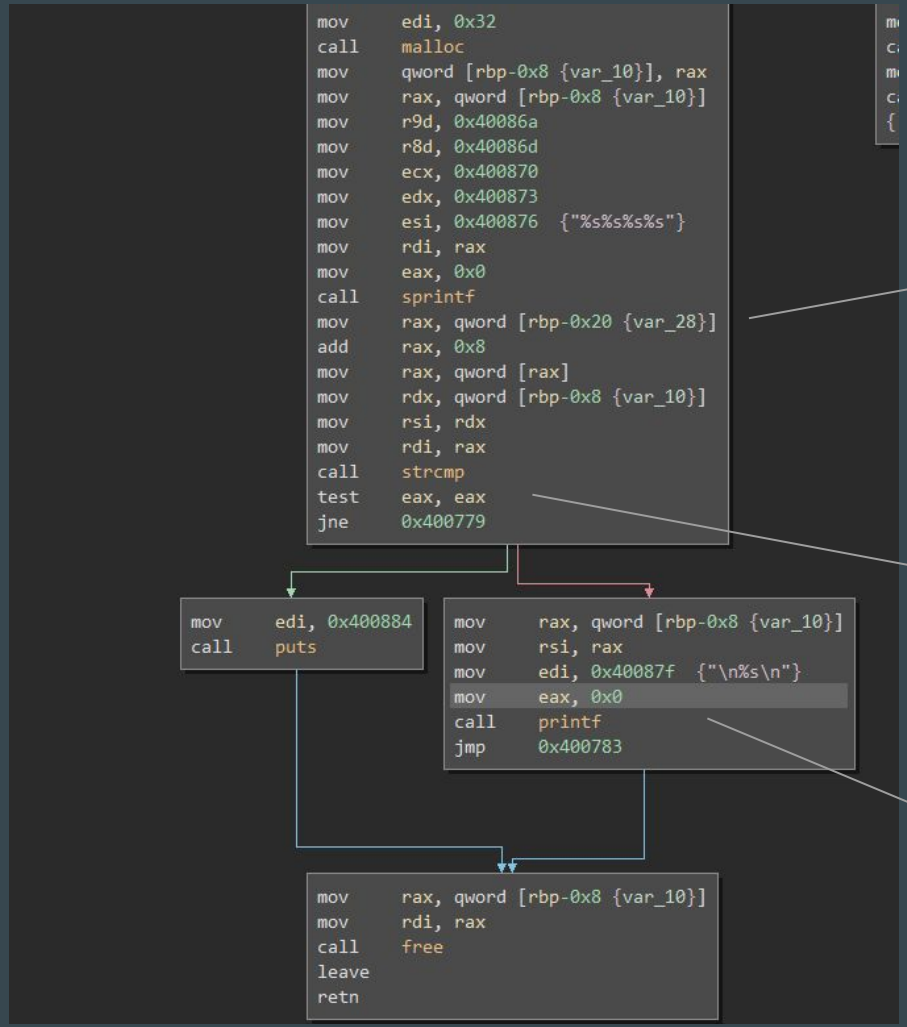

main:
lea ecx, [esp+0x4] {arg_4}
and esp, 0xfffffff0
push dword [ecx-0x4 {__return_addr}]
push ebp
mov ebp, esp {var_8}
push ecx
sub esp, 0x14
sub esp, 0xc
push 0x80ab4aa {"FlagFactory 6.0"}
call sub_804f1f0
add esp, 0x10
sub esp, 0xc
push 0x80ab4bc {"Warning: this will likely take a…"}
call sub_804f1f0
add esp, 0x10
sub esp, 0xc
push 0x80ab50e {"Checking software....."}
call sub_804f1f0
add esp, 0x10
sub esp, 0x8
push 0x1b
push sub_8048aea
call sub_8048aac
add esp, 0x10
test eax, eax
je 0x8048c4a
call sub_8048aea
{ Does not return }
sub esp, 0x8
push 0x1a3
push main
call sub_8048aac
add esp, 0x10
test eax, eax
je 0x8048c68
call sub_8048aea
{ Does not return }
sub esp, 0x8
push 0xe3
push sub_8048b0a
call sub_8048aac
add esp, 0x10
test eax, eax
je 0x8048c86
call sub_8048aea
{ Does not return }
sub esp, 0xc
push 0x2
call sub_805c7d0
add esp, 0x10
sub esp, 0xc
push 0x80ab525 {"Software check ok"}

The 4 steps of Reverse Engineering

- Identify Code
- Identify Input
- Analysis!
- Implementation

Identify Code

- ‘Start at the bottom, stop when you see math’
- In this step we start at our goal state and work backwards to find relevant code that we will need to understand.
- The goal state can be a variety of things, such as flag being printed out or a “Login Successful message”
    - Sometimes working backwards from the failure state can be helpful too
- Once you encounter some assembly code that looks like it will take longer than a couple seconds to understand, stop and mark that block.
- Look at that block and try to figure out if it directly influences whether the code will go to the goal state or not. (do not fully reverse it)
    - If it does, stop. Try to figure out what input values there are to this block, and move on to the next step.
    - If not, keep working backwards


mov edi, 0x32
call malloc
mov qword [rbp-0x8 {var_10}], rax
mov rax, qword [rbp-0x8 {var_10}]
mov r9d, 0x40086a
mov r8d, 0x40086d
mov ecx, 0x400870
mov edx, 0x400873
mov esi, 0x400876 {"%s%s%s%s"}
mov rdi, rax
mov eax, 0x0
call sprintf
mov rax, qword [rbp-0x20 {var_28}]
add rax, 0x8
mov rax, qword [rax]
mov rdx, qword [rbp-0x8 {var_10}]
mov rsi, rdx
mov rdi, rax
call strcmp
test eax, eax
jne 0x400779
mov edi, 0x400884
call puts
mov rax, qword [rbp-0x8 {var_10}]
mov rsi, rax
mov edi, 0x40087f {"\n%s\n"}
mov eax, 0x0
call printf
jmp 0x400783
mov rax, qword [rbp-0x8 {var_10}]
mov rdi, rax
call free
leave
retn
Confusing math
Direct result
Goal

Identify inputs

Here, we skip over the block we identified earlier and continue to work backwards through the program, marking interesting/hard to understand assembly blocks as we go.

While doing this, keep track of how data flows between the start of the program and the blocks you marked earlier.

Trace back to the start of the program, and figure out what part of your input goes to the assembly blocks you looked at earlier

Analysis

Now that we have reduced the amount of assembly we need to look at, start reversing the blocks you marked along the way, and figure out what input you need to supply to send the program to it's goal state.

If you don't know what to do here, you didn't actually finish the last two steps.

'Think hard'

Write things down! Either in notepad, or in comments

GUESS. AND. CHECK

Implementation

At some point during the analysis stage, you will realize how to get the program to the goal state. So do it!

You might just have to enter a couple inputs extracted from the program, or you might have to write some code to calculate some stuff.

People usually use Python for this, but you can use whatever you are comfortable in.

The python library pwntools is extremely helpful in this stage.

How to be a binary ninja

So now that we know the theory behind static analysis, how do we actually do it?

There are a few different disassemblers to choose from:

- IDA Pro - The industry standard, costs several arms and several legs
  - Demo exists, but pretty limited
- Radare2 - Free, but sucks
  - This is the vi equivalent for reverse engineering - but it's also lacking a TON of needed features
- Hopper - useless if you aren't on OSX
- Binary Ninja - newcomer to the area, several cool features, only $100
  - Also has a pretty good demo, we will be using this

What to RE

I’m going to be demonstrating how to use Binary Ninja and the 4 steps of RE by reverse engineering part of the CMU Binary Bomb. This is a virtual bomb that you have to ‘defuse’ by completing 7 reverse engineering challenges.

It is avaliable at
https://github.com/petroav/CMU-assembly-challenge/raw/master/binary

(I’ll put the link in slack)