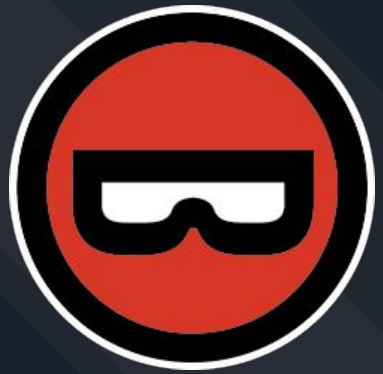
Static Analysis with Binary Ninja



DawgCTF

March 11th! Teams of 4! (or less)

http://umbccd.umbc.edu/dawgctf

Lunch and dinner provided

Prizes for top teams

SIGN UP

Last week we talked about...

- What RE is
- What you can do with RE
- Basic assembly overview
- Using objdump

Objdump sucks, but we have better tools

You may have noticed that objdump kinda sucked to read and I was tripping over myself while demoing

Luckily we have better tools, like Binary Ninja!

```
080484de <main>:
 80484de:       8d 4c 24 04             lea    ecx,[esp+0x4]
 80484e2:       83 e4 f0                and    esp,0xfffffff0
 80484e5:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 80484e8:       55                      push   ebp
 80484e9:       89 e5                   mov    ebp,esp
 80484eb:       51                      push   ecx
 80484ec:       83 ec 14                sub    esp,0x14
 80484ef:       83 ec 0c                sub    esp,0xc
 80484f2:       68 c9 86 04 08          push   0x80486c9
 80484f7:       e8 74 fe ff ff          call   8048370 <puts@plt>
 80484fc:       83 c4 10                add    esp,0x10
 80484ff:       83 ec 0c                sub    esp,0xc
 8048502:       68 e6 86 04 08          push   0x80486e6
 8048507:       e8 54 fe ff ff          call   8048360 <printf@plt>
 804850c:       83 c4 10                add    esp,0x10
 804850f:       c7 45 f0 00 00 00 00    mov    DWORD PTR [ebp-0x10],0x0
 8048516:       c7 45 ec 00 00 00 00    mov    DWORD PTR [ebp-0x14],0x0
 804851d:       83 ec 08                sub    esp,0x8
 8048520:       8d 45 e8                lea    eax,[ebp-0x18]
 8048523:       50                      push   eax
 8048524:       68 f9 86 04 08          push   0x80486f9
 8048529:       e8 72 fe ff ff          call   80483a0 <__isoc99_scanf@plt>
 804852e:       83 c4 10                add    esp,0x10
 8048531:       8b 45 e8                mov    eax,DWORD PTR [ebp-0x18]
 8048534:       3d fe ca 37 13          cmp    eax,0x1337cafe
 8048539:       75 0d                   jne    8048548 <main+0x6a>
 804853b:       83 ec 0c                sub    esp,0xc
 804853e:       6a 01                   push   0x1
 8048540:       e8 76 ff ff ff          call   80484bb <fail>
 8048545:       83 c4 10                add    esp,0x10
 8048548:       8b 45 e8                mov    eax,DWORD PTR [ebp-0x18]
 804854b:       25 80 80 80 80          and    eax,0x80808080
 8048550:       85 c0                   test   eax,eax
 8048552:       74 0d                   je     8048561 <main+0x83>
 8048554:       83 ec 0c                sub    esp,0xc
 8048557:       6a 02                   push   0x2
 8048559:       e8 5d ff ff ff          call   80484bb <fail>
 804855e:       83 c4 10                add    esp,0x10
 8048561:       8b 45 e8                mov    eax,DWORD PTR [ebp-0x18]
 8048564:       25 7f 7f 7f 7f          and    eax,0x7f7f7f7f
 8048569:       85 c0                   test   eax,eax
```

```
main:
lea     ecx, [esp+0x4 {argc}]
and     esp, 0xfffffff0 {__return_addr}
push    dword [ecx-0x4 {__return_addr}]
push    ebp
mov     ebp, esp
push    ecx
sub     esp, 0x14 {var_20}
sub     esp, 0xc
push    0x80486c9  {"I'm thinking of a number...."}
call    puts
add     esp, 0x10 {var_20}
sub     esp, 0xc
push    0x80486e6  {"Give me a number! "}
call    printf
add     esp, 0x10 {var_20}
mov     dword [ebp-0x10 {var_18}], 0x0
mov     dword [ebp-0x14 {var_1c}], 0x0
sub     esp, 0x8 {var_28}
lea     eax, [ebp-0x18 {var_20}]
push    eax
push    0x80486f9
call    __isoc99_scanf
add     esp, 0x10 {var_20}
mov     eax, dword [ebp-0x18 {var_20}]
cmp     eax, 0x1337cafe
jne     0x8048548
```

```
mov     eax, dword [ebp-0x18 {var_20}]
and     eax, 0x80808080
test    eax, eax
je      0x8048561
```

```
mov     eax, dword [ebp-0x18 {var_20}]
and     eax, 0x7f7f7f7f
```

```
sub     esp, 0xc
push    0x2
```

Benefits of a modern disassembler

- Recognizes and lets you rename variables that the program uses
- Resilient to some obfuscation techniques
  - Symbol stripping, odd assembly code, etc
- Let’s you visualize the control flow easily
  - By rendering the assembly as a graph of basic blocks

```
cmp     byte [ebp-0x9], 0x6f
je      0x80485e5
```

```
You guessed my number!\nYou win!"}

r_20}
_28}
```

```
sub     esp, 0xc
push    0x6
call    fail
{ Does not return }
```

```
movzx   eax, byte [ebp-0xa]
lea     edx, [eax-0x1]
mov     byte [ebp-0xa], dl
test    al, al
jne     0x80485ba
```

```
movzx   eax, byte [ebp-0x9]
add     eax, 0x1
mov     byte [ebp-0x9], al
```

```
cmp     byte [ebp-
je      0x80485e5
```

```
sub     esp, 0xc
```

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
```

Complicated 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 its 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 can be very helpful in this step

Demo! Chris reverses a simple binary

I'll do most of it for you, you can finish it though

Lab

3 more binaries for you to reverse, on your own or in groups

I will be walking around to help

Scoreboard and challenges are at: http://52.91.53.202/

http://52.91.53.202/