Reverse Engineering

Intro

Created by Ik0ri4n

09.11.2023

```
import pwn

pwn.context.arch = "amd64"
pwn.context.os = "linux"

SHELLCODE = pwn.shellcraft.amd64.linux.echo('Test') + pwn.shellcraft
EXPLOIT = 0x45*b"\x90" + pwn.asm(SHELLCODE, arch="amd64", os="linux"

PROGRAM = b""
length = 20 + 16
for i in EXPLOIT:
    PROGRAM += i*b'+' + b'>'

    if i == 1:
        length += 5
    elif i > 1:
        length += 6
    ength+= 13

     0x8000 - length) > 0x40:
     RAM += b"<>"
      h += 2*13


        b".["


          0 - length) + 7 -1

          F+0x10)*b"<"


            host", 1337) as conn:
            (b"Brainf*ck code: ")
            ROGRAM)
            e()
```

What's that?

Making a compiled program readable

Understanding what it does

Why would I need that?

- Security analysis
- Malware analysis
- No docs, source available
- Modding, Cracking

...plus it's fun!

Where do we start?

```
$ ls -l
total 16
-rwxrwxr-x 1 user user 16120 Nov  9 14:10 chal
```

```
$ hexdump -C chal | head
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  03 00 3e 00 01 00 00 00  e0 10 00 00 00 00 00 00  |..>.............|
00000020  40 00 00 00 00 00 00 00  38 37 00 00 00 00 00 00  |@.......87......|
00000030  00 00 00 00 40 00 38 00  0d 00 40 00 1f 00 1e 00  |....@.8...@.....|
00000040  06 00 00 00 04 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |@.......@.......|
00000060  d8 02 00 00 00 00 00 00  d8 02 00 00 00 00 00 00  |................|
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00  |................|
00000080  18 03 00 00 00 00 00 00  18 03 00 00 00 00 00 00  |................|
00000090  18 03 00 00 00 00 00 00  1c 00 00 00 00 00 00 00  |................|
```

What are we dealing with?

```
$ file chal
chal: ELF 64-bit LSB pie executable,
    x86-64,
    version 1 (SYSV),
    dynamically linked,
    interpreter /lib64/ld-linux-x86-64.so.2,
    BuildID[sha1]=e7f3e971abeb24c4d7cc7747b3274f3058e749af,
    for GNU/Linux 3.2.0,
    stripped
```

Making sense of op codes

http://ref.x86asm.net/coder64.html

Disassemblers

```
$ objdump -M intel -S chal
chal:     file format elf64-x86-64

Disassembly of section .init:

0000000000001000 <_init>:
    1000: f3 0f 1e fa           endbr64
    1004: 48 83 ec 08           sub    rsp,0x8
    1008: 48 8b 05 d9 2f 00 00  mov    rax,QWORD PTR [rip+0x2fd9]        # 3fe8 <__gmon_start__@Base>
        100f: 48 85 c0              test   rax,rax
        1012: 74 02                 je     1016 <_init+0x16>
            1014: ff d0                 call   rax
            1016: 48 83 c4 08           add    rsp,0x8
            101a: c3                    ret
```

Assembly

Recall ELF sections:

- .data: pre-initialized global writable data
- .rodata: pre-initialized global read-only data
- .bss: uninitialized global writable data

OST 2 - Architecture 1001: x86-64 Assembly

Decompilers

Ghidra

Binary Ninja

IDA pro

Rev player trust issues

Tool output is not always perfect!

- `file` checks magic bytes, use your own with `-m`
- Use `file --keep-going` or `binwalk` for multi-matches
- Decompilers make (wrong) assumptions all the time!

Static analysis tools

- file, binwalk
- nm, strings
- objdump
- checksec (check protections)
- Ghidra, Binary Ninja, IDA Pro, etc.

Dynamic approach

Debugging with gdb

```
gdb -ex 'set disassembly-flavor intel' chal
```

Useful extensions:

- pwndbg
- GEF

Ideally put such settings into .gdbinit

Overview

Function	Meaning
run args	Run the program
starti args	Run the program and break on first instruction
break expr	Break at the given address or symbol
watch expr	Break when a value is written to the given address
rwatch expr	Break when a value is read from the given address
continue	Continue program execution
si and ni	Step into and step over

Examine Memory

```
x/<amount><format><size> <expr>
```

Parameter	Meaning
amount	Number of things to read
format	Output format, notably x, a, s for hex, addresses, and strings
size	Size of the data blocks, b, h, w, g for 1, 2, 4, 8 bytes respectively
expr	C-like expression describing data location

Dynamic analysis tools

- strace
- ltrace
- gdb
- Emulators

Further reading

Processor ISA Manuals

Gdb and Pwndbg documentation

Ghidra Book

ost2.fyi

Helpful tools

angr (symbolic execution)

SMT solvers (e.g., z3)

SageMath (ask our crypto players 😉)

Lots plugins and tools for specific use cases

Start playing at intro.kitctf.de