technically_correct

LaCTF 2024 writeup

MadrHacks

Roberto Van Eeden

@rw-r-r-0644

CyberCup keynote speech
July 6, 2024

MadrHacks

We are a team of ethical hackers born during CyberChallenge.IT 2020 at the University of Udine

Come visit us at snakeCTF 2024!

Let's have a look at one of the challenges!

technically_correct

LaCTF 2024 // rev // @aplet123 // t.ly/WqLOC

All we are given is a binary that verifies if a given flag is correct

- we can execute the binary normally
- the ELF binary is good enough for the linux kernel

- we can execute the binary normally
- the ELF binary is good enough for the linux kernel

- ...but nearly nothing else, apparently :(
- RIP readelf, gdb, Ghidra, etc.

```
$ readelf -a ./technically_correct
ELF Header:
  Magic:   7f 45 4c 46 01 02 a8 9e b6 21 74 80 06 55 b8 e5
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           168 <unknown>
  OS/ABI:                            <unknown: 9e>
  ABI Version:                       182
  Type:                              <unknown>: 200
  Machine:                           <unknown>: 0x3e00
  Version:                           0x6ed7b4c7
  Entry point address:               0x37c184d0
  Start of program headers:          3338993664 (bytes into file)
  Start of section headers:          973078528 (bytes into file)
  Flags:                             0x0
  Size of this header:               36406 (bytes)
  Size of program headers:           8300 (bytes)
  Number of program headers:         15801
  Size of section headers:           35328 (bytes)
  Number of section headers:         56772
  Section header string table index: 5298
readelf: Warning: The e_shentsize field in the ELF header is larger than the size of an ELF section header
readelf: Error: Reading 2005641216 bytes extends past end of file for section headers
readelf: Error: Section headers are not available!
readelf: Error: Too many program headers - 0x3db9 - the file is not that big

There is no dynamic section in this file.
readelf: Error: Too many program headers - 0x3db9 - the file is not that big
$ 
```

Option 1

- carefully analyze ELF header, attempt manual repair
- annoying

Option 1

- carefully analyze ELF header, attempt manual repair
- annoying

Option 2

- slap some `printk()` within `linux/fs/binfmt_elf.c`
- build new ELF binary from captured load addresses / offsets
- profit?

```
ELF LOAD ADDR:0xD3A94076000 OFFS:0x24000 LEN:0x1000
ELF LOAD ADDR:0xD80A68EF000 OFFS:0x14000 LEN:0x1000
ELF LOAD ADDR:0xDC7D1B35000 OFFS:0x15000 LEN:0x1000
ELF LOAD ADDR:0xDCE3460C000 OFFS:0x2D000 LEN:0x1000
ELF LOAD ADDR:0xE58854D6000 OFFS:0x1E000 LEN:0x1000
ELF LOAD ADDR:0xE5FF5ADC000 OFFS:0x31000 LEN:0x1000
ELF LOAD ADDR:0xE885987B000 OFFS:0x30000 LEN:0x1000
ELF LOAD ADDR:0xF2FA2AEE000 OFFS:0x7000 LEN:0x1000
ELF LOAD ADDR:0xF84BC1F8000 OFFS:0x36000 LEN:0x1000
ELF LOAD ADDR:0xF8C4F8B0000 OFFS:0x1F000 LEN:0x1000
```

- seems to work!
- resulting binary still looks somewhat odd, maps 60 page-sized sections (0x1000 bytes) to seemingly arbitrary and discontiguous places
- (turns out this is intended!)

Analyzing the resulting binary

We can now throw the binary in gdb, Ghidra, ...

- usual anti-debugging techniques
- most annoyingly, self-modifying code sections become available only right before being executed

Analyzing the resulting binary

We can now throw the binary in gdb, Ghidra, ...

- usual anti-debugging techniques
- most annoyingly, self-modifying code sections become available only right before being executed

- fortunately, dynamic analysis techniques bypass most of these problems :)

Qiling Framework

- based on Unicorn cpu emulator (itself based on QEMU)
- high level syscall emulation for multiple targets
- API provides numerous hooks to analyze and/or modify what the binary is doing:
  - executed instructions, blocks
  - memory reads/writes
  - syscalls
  - ...and much more!

Reverse engineering

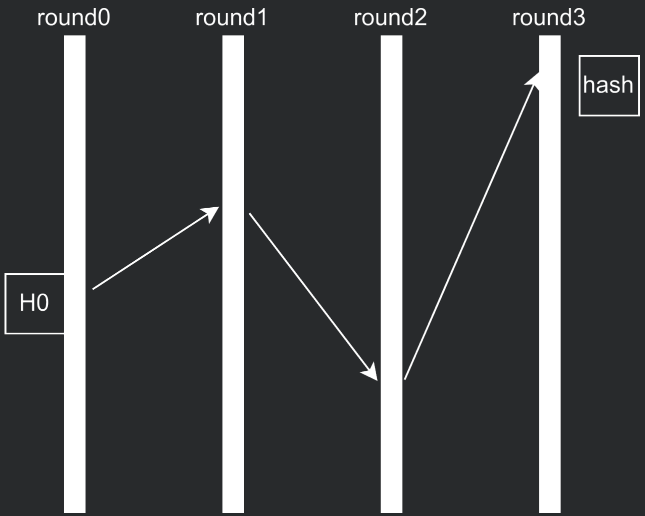
It appears ELF sections contain static data used to compute a custom hash.

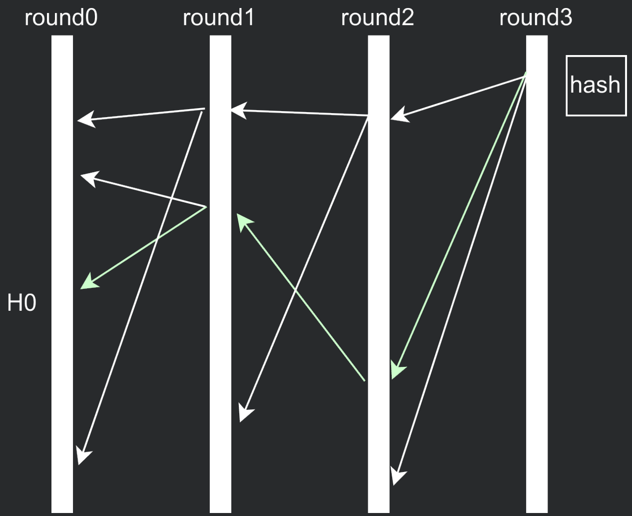
```
const uint64_t H0 = 0xf84bc1f88e8;
uint64_t h_update(uint64_t state, unsigned char data) {
  uint64_t p = state + data*8;
  state = read64(p) ^ p;
  state *= 0xb216cb3c48c1e693;
  state += 0xc200c6d3267c529d;
  return state;
}
```

- hash state is a 64 bit value
- non-injective transforms for state updates
- **hash state is a pointer to an address within one of previously mentioned data sections (arry with next hash base for each input byte)**

Inverting the hash

- **hash state is a pointer to an address within one of previously mentioned data sections (arry with next hash base for each input byte)**
- state space is relatively contained, only 30720 possible states (60 pages of size 0x1000 containing 64 bit values)
- build hash update inverse map, work backwards from correct hash until we reach H0 keeping tracks of the set of possible states that could reach target checksum
- bitsets are great


round0
round1
round2
round3
hash
H0


round0
round1
round2
round3
hash
H0

Thanks for your attention

Questions? Comments?