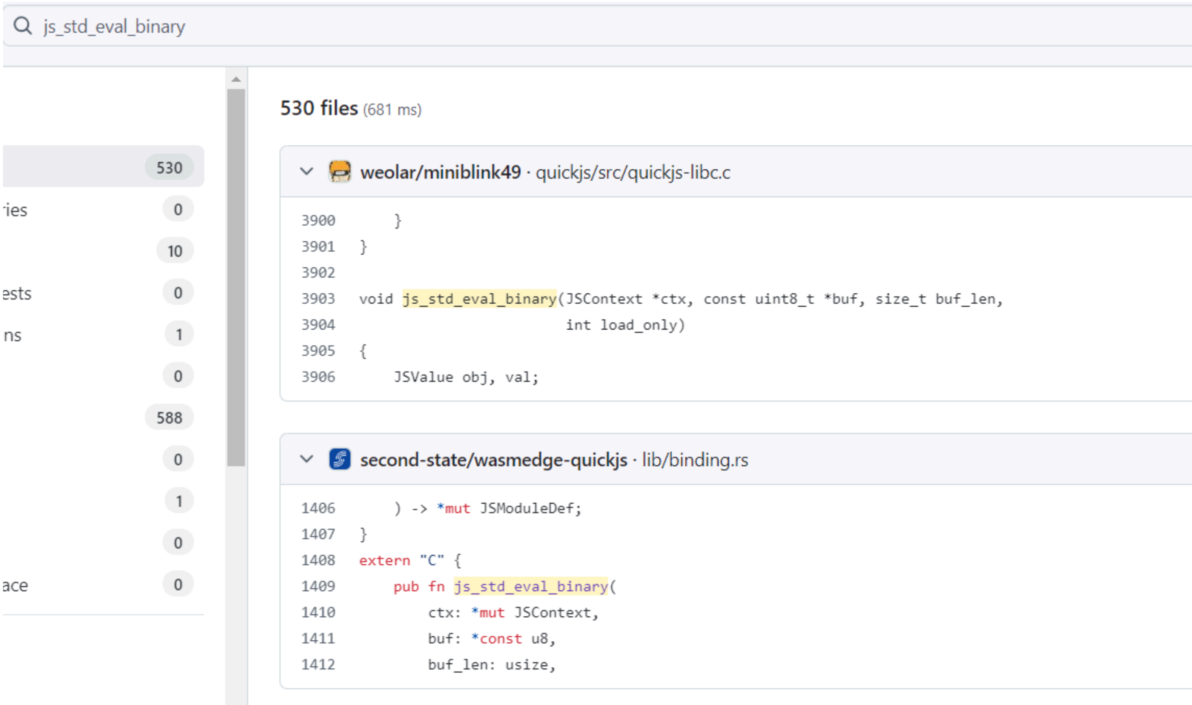
qjvmp

IDA反编译看一眼

```c
int __fastcall main(int argc, const char **argv, const char **envp)
{
  __int64 v3; // rbx
  __int64 v4; // rbp

  v3 = JS_NewRuntime(argc, argv, envp);
  js_std_set_worker_new_context_func(JS_NewCustomContext);
  js_std_init_handlers(v3);
  JS_SetModuleLoaderFunc(v3, 0LL, js_module_loader, 0LL);
  v4 = JS_NewCustomContext(v3);
  js_std_add_helpers(v4, (unsigned int)argc, argv);
  js_std_eval_binary(v4, &qjsc_ca, 7482LL, 0LL);
  js_std_loop(v4);
  js_std_free_handlers(v3);
  JS_FreeContext(v4);
  JS_FreeRuntime(v3);
  return 0;
}
```

github搜索得知时quickjs的东西



同时认为这段就是编译的字节码，程序加载在这一段字节码运行

```
rodata:00000000004D7020                 public qjsc_ca
rodata:00000000004D7020 qjsc_ca         db      43h,     18h,     0Ah,     63h,     61h,     2Eh
rodata:00000000004D7020                                          ; DATA XREF: main+5E↑o
rodata:00000000004D7026                 db      6Ah,     73h,       6,     65h,     65h,     65h
rodata:00000000004D702C                 db        6,     67h,     67h,     67h,       6,     6Eh
rodata:00000000004D7032                 db      6Eh,     6Eh,     0Ah,     6Eh,     6Eh,     6Eh
rodata:00000000004D7038                 db      6Eh,     6Eh,       6,     74h,     74h,     74h
rodata:00000000004D703E                 db        6,     66h,     66h,     66h,       2,     69h
rodata:00000000004D7044                 db        6,     78h,     78h,     78h,       2,     78h
rodata:00000000004D704A                 db        4,     78h,     78h,       2,     65h,     14h
rodata:00000000004D7050                 db      73h,     63h,     72h,     69h,     70h,     74h
rodata:00000000004D7056                 db      41h,     72h,     67h,     73h,     14h,     63h
rodata:00000000004D705C                 db      68h,     61h,     72h,     43h,     6Fh,     64h
rodata:00000000004D7062                 db      65h,     41h,     74h,     0Eh,     63h,     6Fh
rodata:00000000004D7068                 db      6Eh,     73h,     6Fh,     6Ch,     65h,       6
rodata:00000000004D706E                 db      6Ch,     6Fh,     67h,     0Ah,     70h,     72h
rodata:00000000004D7074                 db      69h,     6Eh,     74h,     30h,     50h,     75h
rodata:00000000004D707A                 db      74h,     20h,     79h,     6Fh,     75h,     72h
rodata:00000000004D7080                 db      20h,     66h,     6Ch,     61h,     67h,     20h
rodata:00000000004D7086                 db      69h,     6Eh,     20h,     61h,     72h,     67h
rodata:00000000004D708C                 db      76h,     5Bh,     31h,     5Dh,       8,     67h
rodata:00000000004D7092                 db      67h,     67h,     67h,       6,     72h,     72h
rodata:00000000004D7098                 db      72h,       6,     63h,     63h,     63h,       6
rodata:00000000004D709E                 db      79h,     79h,     79h,       8,     6Eh,     6Eh
rodata:00000000004D70A4                 db      6Eh,     6Eh,     0Ch,     6Eh,     6Eh,     6Eh
rodata:00000000004D70AA                 db      6Eh,     6Eh,     6Eh,     0Ch,       0,       6
rodata:00000000004D70B0                 db        0,    0A2h,       1,       0,       1,       0
rodata:00000000004D70B6                 db        1,       0,       1,       5,       4,       0
rodata:00000000004D70BC                 db        0,       0,       0,       0,       0,       0
rodata:00000000004D70C2                 db        1,    0A4h,       1,       0,       0,       0
rodata:00000000004D70C8                 db        2,       0,       1,       1,       1,     58h
rodata:00000000004D70CE                 db        6,     43h,       0,       0,       0,       0
rodata:00000000004D70D4                 db        0,       0,       0,       0,       0,       0
rodata:00000000004D70DA                 db        0,       0,       0,     61h,     18h,     43h
rodata:00000000004D70E0                 db        0,       0,       0,       0,       0,    0E2h
rodata:00000000004D70E6                 db      39h,     43h,       0,       0,       0,       0
rodata:00000000004D70EC                 db        0,    0B5h,     26h,     43h,       0,       0
rodata:00000000004D70F2                 db        0,       0,       0,    0C6h,       3,       1
rodata:00000000004D70F8                 db        4,       0,       0,     9Eh,       1,     0Ch
rodata:00000000004D70FE                 db        2,       6,       0,       0,       0,     0Bh
rodata:00000000004D7104                 db        0,     20h,       0,       2,    0E3h,       3
```

好！开找decompiler！好！没有！

那么去原始仓库找找字节码对应的指令总行吧，好！没有明确的对应！那么我们不得不采取一些原始的办法。尝试使用gdb辨认执行流程。

由于程序会在check失败时输出一个false，于是在write下断点，观察调用栈

```
  0x7ffff7d1b29c <write+28>    ret

  0x7ffff7d1b29d <write+29>    nop    dword ptr [rax]
  0x7ffff7d1b2a0 <write+32>    sub    rsp, 0x28
  0x7ffff7d1b2a4 <write+36>    mov    qword ptr [rsp + 0x18], rdx
───────────────────────────────────────[ STACK ]───────────────────────────────────────
0:0000│ rsp 0x7fffffffa818 —▸ 0x7ffff7c8eefd (_IO_file_write+45) ◂— test rax, rax
1:0008│     0x7fffffffa820 —▸ 0x50c070 ◂— 0x65736c6166 /* 'false' */
2:0010│     0x7fffffffa828 —▸ 0x7ffff7dff7a0 (_IO_2_1_stdout_) ◂— 0xfbad2a84
3:0018│     0x7fffffffa830 ◂— 6
4:0020│     0x7fffffffa838 —▸ 0x52bae0 ◂— 0xa65736c6166 /* 'false\n' */
5:0028│     0x7fffffffa840 —▸ 0x7ffff7dfd270 (_IO_file_jumps) ◂— 0
6:0030│     0x7fffffffa848 —▸ 0x7ffff7c8dbc8 (_IO_do_write+168) ◂— mov r13, rax
7:0038│     0x7fffffffa850 —▸ 0x7ffff7dff7a0 (_IO_2_1_stdout_) ◂— 0xfbad2a84
─────────────────────────────────────[ BACKTRACE ]─────────────────────────────────────
 ► 0   0x7ffff7d1b280 write
   1   0x7ffff7c8eefd _IO_file_write+45
   2   0x7ffff7c8dbc8 _IO_do_write+168
   3   0x7ffff7c8e4f3 _IO_file_overflow+227
   4   0x7ffff7c85796 putchar+230
   5         0x4bf95c js_print+211
   6         0x42f44d js_call_c_function+758
   7         0x42fe0c JS_CallInternal+623
───────────────────────────────────────────────────────────────────────────────────────
wndbg> backtrace
#0  0x00007ffff7d1b280 in write () from /home/ctf/Desktop/pwn/tmp/qjvmp/glibc-all-in-one/libs/2.38-1ubuntu6.3_amd64/libc.so.6
#1  0x00007ffff7c8eefd in _IO_file_write () from /home/ctf/Desktop/pwn/tmp/qjvmp/glibc-all-in-one/libs/2.38-1ubuntu6.3_amd64/libc.so.6
#2  0x00007ffff7c8dbc8 in _IO_do_write () from /home/ctf/Desktop/pwn/tmp/qjvmp/glibc-all-in-one/libs/2.38-1ubuntu6.3_amd64/libc.so.6
#3  0x00007ffff7c8e4f3 in _IO_file_overflow () from /home/ctf/Desktop/pwn/tmp/qjvmp/glibc-all-in-one/libs/2.38-1ubuntu6.3_amd64/libc.so.6
#4  0x00007ffff7c85796 in putchar () from /home/ctf/Desktop/pwn/tmp/qjvmp/glibc-all-in-one/libs/2.38-1ubuntu6.3_amd64/libc.so.6
#5  0x00000000004bf95c in js_print ()
#6  0x000000000042f44d in js_call_c_function ()
#7  0x000000000042fe0c in JS_CallInternal ()
#8  0x000000000043224c in JS_CallInternal ()
#9  0x0000000000431902 in JS_CallInternal ()
#10 0x000000000043ae59 in JS_CallFree ()
#11 0x00000000004621a2 in JS_EvalFunctionInternal ()
#12 0x00000000004622eb in JS_EvalFunction ()
#13 0x00000000004c02a6 in js_std_eval_binary ()
#14 0x0000000000403938 in main ()
#15 0x00007ffff7c28150 in ?? () from /home/ctf/Desktop/pwn/tmp/qjvmp/glibc-all-in-one/libs/2.38-1ubuntu6.3_amd64/libc.so.6
#16 0x00007ffff7c28209 in __libc_start_main () from /home/ctf/Desktop/pwn/tmp/qjvmp/glibc-all-in-one/libs/2.38-1ubuntu6.3_amd64/libc.so.6
#17 0x0000000000403991 in _start ()
wndbg>
```

进一步比对源代码可知，JS_CallInternal为关键，我认为他代表了一个函数的调用结构。

> JS_CallInternal部分内容如以下程序片段所示（使用gcc -E去掉了宏）

```c
            goto *dispatch_table[opcode = *pc++];;


        case_OP_get_loc8: *sp++ = JS_DupValue(ctx, var_buf[*pc++]); goto *dispatch_table[opcode = *pc++];;
        case_OP_put_loc8: set_value(ctx, &var_buf[*pc++], *--sp); goto *dispatch_table[opcode = *pc++];;
        case_OP_set_loc8: set_value(ctx, &var_buf[*pc++], JS_DupValue(ctx, sp[-1])); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_loc0: *sp++ = JS_DupValue(ctx, var_buf[0]); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_loc1: *sp++ = JS_DupValue(ctx, var_buf[1]); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_loc2: *sp++ = JS_DupValue(ctx, var_buf[2]); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_loc3: *sp++ = JS_DupValue(ctx, var_buf[3]); goto *dispatch_table[opcode = *pc++];;
        case_OP_put_loc0: set_value(ctx, &var_buf[0], *--sp); goto *dispatch_table[opcode = *pc++];;
        case_OP_put_loc1: set_value(ctx, &var_buf[1], *--sp); goto *dispatch_table[opcode = *pc++];;
        case_OP_put_loc2: set_value(ctx, &var_buf[2], *--sp); goto *dispatch_table[opcode = *pc++];;
        case_OP_put_loc3: set_value(ctx, &var_buf[3], *--sp); goto *dispatch_table[opcode = *pc++];;
        case_OP_set_loc0: set_value(ctx, &var_buf[0], JS_DupValue(ctx, sp[-1])); goto *dispatch_table[opcode = *pc++];;
        case_OP_set_loc1: set_value(ctx, &var_buf[1], JS_DupValue(ctx, sp[-1])); goto *dispatch_table[opcode = *pc++];;
        case_OP_set_loc2: set_value(ctx, &var_buf[2], JS_DupValue(ctx, sp[-1])); goto *dispatch_table[opcode = *pc++];;
        case_OP_set_loc3: set_value(ctx, &var_buf[3], JS_DupValue(ctx, sp[-1])); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_arg0: *sp++ = JS_DupValue(ctx, arg_buf[0]); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_arg1: *sp++ = JS_DupValue(ctx, arg_buf[1]); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_arg2: *sp++ = JS_DupValue(ctx, arg_buf[2]); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_arg3: *sp++ = JS_DupValue(ctx, arg_buf[3]); goto *dispatch_table[opcode = *pc++];;
        case_OP_put_arg0: set_value(ctx, &arg_buf[0], *--sp); goto *dispatch_table[opcode = *pc++];;
        case_OP_put_arg1: set_value(ctx, &arg_buf[1], *--sp); goto *dispatch_table[opcode = *pc++];;
        case_OP_put_arg2: set_value(ctx, &arg_buf[2], *--sp); goto *dispatch_table[opcode = *pc++];;
        case_OP_put_arg3: set_value(ctx, &arg_buf[3], *--sp); goto *dispatch_table[opcode = *pc++];;
        case_OP_set_arg0: set_value(ctx, &arg_buf[0], JS_DupValue(ctx, sp[-1])); goto *dispatch_table[opcode = *pc++];;
        case_OP_set_arg1: set_value(ctx, &arg_buf[1], JS_DupValue(ctx, sp[-1])); goto *dispatch_table[opcode = *pc++];;
        case_OP_set_arg2: set_value(ctx, &arg_buf[2], JS_DupValue(ctx, sp[-1])); goto *dispatch_table[opcode = *pc++];;
        case_OP_set_arg3: set_value(ctx, &arg_buf[3], JS_DupValue(ctx, sp[-1])); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_var_ref0: *sp++ = JS_DupValue(ctx, *var_refs[0]->pvalue); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_var_ref1: *sp++ = JS_DupValue(ctx, *var_refs[1]->pvalue); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_var_ref2: *sp++ = JS_DupValue(ctx, *var_refs[2]->pvalue); goto *dispatch_table[opcode = *pc++];;
        case_OP_get_var_ref3: *sp++ = JS_DupValue(ctx, *var_refs[3]->pvalue); goto *dispatch_table[opcode = *pc++];;
```

那么找到那些字节码对应的操作是关键

JS_CallInternal下断，输入ni，然后按住回车，让他一直执行。最终观察到程序会在"某一段区域"循环。放慢速度找到最开始的部分在： 0x43012d <JS_CallInternal+1424>即下图部分

```c
restart:
    for(;;) {
        int call_argc;
        JSValue *call_argv;

        SWITCH(pc) {
        CASE(OP_push_i32):
            *sp++ = JS_NewInt32(ctx, get_u32(pc));
            pc += 4;
            BREAK;
        CASE(OP_push_const):
            *sp++ = JS_DupValue(ctx, b->cpool[get_u32(pc)]);
            pc += 4;
            BREAK;
#if SHORT_OPCODES
        CASE(OP_push_minus1):
        CASE(OP_push_0):
        CASE(OP_push_1):
        CASE(OP_push_2):
        CASE(OP_push_3):
        CASE(OP_push_4):
        CASE(OP_push_5):
        CASE(OP_push_6):
        CASE(OP_push_7):
            *sp++ = JS_NewInt32(ctx, opcode - OP_push_0);
            BREAK;
        CASE(OP_push_i8):
            *sp++ = JS_NewInt32(ctx, get_i8(pc));
            pc += 1;
            BREAK;
        CASE(OP_push_i16):
            *sp++ = JS_NewInt32(ctx, get_i16(pc));
            pc += 2;
```

下断在0x43012d <JS_CallInternal+1424>，不断按c，pwndbg会告诉你RDX的变化

```
► 0x43012d <JS_CallInternal+1424>    mov    rax, qword ptr [rbp - 0x48]     RAX, [0x7fffffffcc68] => 0x520650 ◂— 0x100000002
  0x430131 <JS_CallInternal+1428>    mov    rcx, qword ptr [rax + 0x30]     RCX, [0x520680] => 0x52099b —▸ 0x430658 (JS_CallInternal+2747) ◂— mov rax, qword ptr [rbp - 0x48]
  0x430135 <JS_CallInternal+1432>    mov    rax, qword ptr [rbp - 0x58]     RAX, [0x7fffffffcc58] => 0x5207ca ◂— 0xf3036302008803
  0x430139 <JS_CallInternal+1436>    lea    rdx, [rax + 1]                  RDX => 0x5207cb ◂— 0x200f30363020088
  0x43013d <JS_CallInternal+1440>    mov    qword ptr [rbp - 0x58], rdx     [0x7fffffffcc58] => 0x5207cb ◂— 0x200f30363020088
  0x430141 <JS_CallInternal+1444>    mov    rdx, qword ptr [rbp - 0x48]     RDX, [0x7fffffffcc68] => 0x520650 ◂— 0x100000002
```

```
► 0x43012d <JS_CallInternal+1424>    mov    rax, qword ptr [rbp - 0x48]     RAX, [0x7fffffffcc68] => 0x520650 ◂— 0x100000002
  0x430131 <JS_CallInternal+1428>    mov    rcx, qword ptr [rax + 0x30]     RCX, [0x520680] => 0x52099b —▸ 0x430658 (JS_CallInternal+2747) ◂— mov rax, qword ptr [rbp - 0x48]
  0x430135 <JS_CallInternal+1432>    mov    rax, qword ptr [rbp - 0x58]     RAX, [0x7fffffffcc58] => 0x5207cd ◂— 0x31b0200f3036302
  0x430139 <JS_CallInternal+1436>    lea    rdx, [rax + 1]                  RDX => 0x5207ce ◂— 0xcf031b0200f30363
  0x43013d <JS_CallInternal+1440>    mov    qword ptr [rbp - 0x58], rdx     [0x7fffffffcc58] => 0x5207ce ◂— 0xcf031b0200f30363
  0x430141 <JS_CallInternal+1444>    mov    rdx, qword ptr [rbp - 0x48]     RDX, [0x7fffffffcc68] => 0x520650 ◂— 0x100000002
```

```
► 0x43012d <JS_CallInternal+1424>    mov    rax, qword ptr [rbp - 0x48]     RAX, [0x7fffffffcc68] => 0x520650 ◂— 0x100000002
  0x430131 <JS_CallInternal+1428>    mov    rcx, qword ptr [rax + 0x30]     RCX, [0x520680] => 0x52099b —▸ 0x430658 (JS_CallInternal+2747) ◂— mov rax, qword ptr [rbp - 0x48]
  0x430135 <JS_CallInternal+1432>    mov    rax, qword ptr [rbp - 0x58]     RAX, [0x7fffffffcc58] => 0x5207cf ◂— 0xcf031b0200f303
  0x430139 <JS_CallInternal+1436>    lea    rdx, [rax + 1]                  RDX => 0x5207d0 ◂— 0x200cf031b0200f3
  0x43013d <JS_CallInternal+1440>    mov    qword ptr [rbp - 0x58], rdx     [0x7fffffffcc58] => 0x5207d0 ◂— 0x200cf031b0200f3
  0x430141 <JS_CallInternal+1444>    mov    rdx, qword ptr [rbp - 0x48]     RDX, [0x7fffffffcc68] => 0x520650 ◂— 0x100000002
```

那么pc就找到了: rdx

~~然后根据pc值来对应字节码，然后ida dump字节码，然后逆出指令序列，然后就接近成功力~~

然后就浪费了很多时间，得出结论没法内找到源代码的一个操作对应的字节码

但是！我们可以花点时间从gdb身上找到并对应上源代码中的switch结构。

如下图所示，通过jmp rax，执行不同的指令。我们可以把他的跳转后的地址记录下来，作为一个代码块的标志。然后利用call定位这个代码块块对应源代码的哪个case块

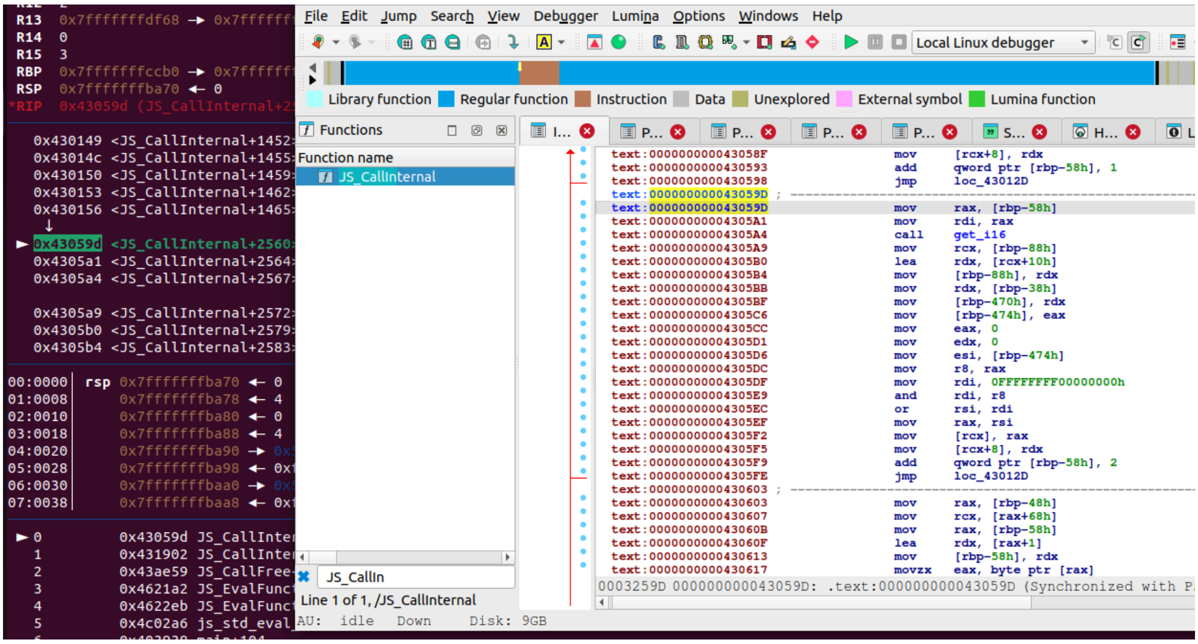
```
 0x430145 <JS_CallInternal+1448>    mov    rdx, qword ptr [rdx + 0x20]     RDX, [0x520670] => 0x5207b8 ◂— 0x103010102010002
 0x430149 <JS_CallInternal+1452>    sub    rax, rdx                        RAX => 23 (0x5207cf - 0x5207b8)
 0x43014c <JS_CallInternal+1455>    shl    rax, 3
 0x430150 <JS_CallInternal+1459>    add    rax, rcx                        RAX => 0x520a53 (0xb8 + 0x52099b)
 0x430153 <JS_CallInternal+1462>    mov    rax, qword ptr [rax]            RAX, [0x520a53] => 0x43059d (JS_CallInternal+2560) ◂— mov rax, qword ptr [rbp - 0x58]
►0x430156 <JS_CallInternal+1465>    jmp    rax                         <JS_CallInternal+2560>
  ↓
 0x43059d <JS_CallInternal+2560>    mov    rax, qword ptr [rbp - 0x58]     RAX, [0x7fffffffcc58] => 0x5207d0 ◂— 0x200cf031b0200f3
 0x4305a1 <JS_CallInternal+2564>    mov    rdi, rax                        RDI => 0x5207d0 ◂— 0x200cf031b0200f3
 0x4305a4 <JS_CallInternal+2567>    call   get_i16                     <get_i16>

 0x4305a9 <JS_CallInternal+2572>    mov    rcx, qword ptr [rbp - 0x88]
 0x4305b0 <JS_CallInternal+2579>    lea    rdx, [rcx + 0x10]
```

这样一来，我们建立了映射，并可得知操作类型：

> 代码块<--->case块（操作类型）

下面详细叙述一下如何利用call来定位这个代码块块对应源代码的哪个case块：

IDA中找到对应地址代码段



可以看到，以jmp loc_43012D分割为不同的代码块。上一个代码块call了一个get_i8，本块call了一个get_i16，下一个JS_DupValue

在源代码中搜索这些函数，并确认上下case

```c
        case OP_push_5:
        case OP_push_6:
        case OP_push_7:
            *sp++ = JS_NewInt32(ctx, opcode - OP_push_0);
            goto *dispatch_table[opcode = *pc++];;
        case OP_push_i8:
            *sp++ = JS_NewInt32(ctx, get_i8(pc));
            pc += 1;
            goto *dispatch_table[opcode = *pc++];;
        case OP_push_i16:
            *sp++ = JS_NewInt32(ctx, get_i16(pc));
            pc += 2;
            goto *dispatch_table[opcode = *pc++];;
        case OP_push_const8:
            *sp++ = JS_DupValue(ctx, b->cpool[*pc++]);
            goto *dispatch_table[opcode = *pc++];;
        case OP_fclosure8:
            *sp++ = js_closure(ctx, JS_DupValue(ctx, b->cpool[*pc++]), var_refs, sf);
            if (__builtin_expect(!!(JS_IsException(sp[-1])), 0))
                goto exception;
            goto *dispatch_table[opcode = *pc++];;
        case OP_push_empty_string:
```

最终找到地址0x43059d对应操作OP_push_i16。同时我们也可以确认上一块为OP_push_i8，下一块为OP_push_const8

叙述原理后，我们需要知道程序一共用了多少不同的代码块，来评估工作量（不然大约452个等着找吧）。

编写gdb脚本

```
set logging file disassembly_output.txt
set logging overwrite on
set logging on

break *0x43013d
break *0x430156

commands 1
    silent
    printf "rdx = 0x%lx\n", $rdx
    continue
end

commands 2
    silent
    x/i $rax
    continue
end

start nex{123456}
continue

set logging off
```

1. 断点1确认pc
2. 断点2确认跳转地址（jmp rax，打印rax即可）

编写ida脚本，分离出所有代码块(0x43AAA6是后来发现的返回地址，即JS_CallInternal跳转到done块然后返回)

```
import idautils
import idaapi

start = 0x430158
end = 0x43AAA6
target_jumps = ['loc_43AAA6', 'loc_43012D']

out = open('/home/ctf/Desktop/pwn/tmp/qjvmp/tool/idaout.txt', 'w')

for addr in idautils.Heads(start, end):
    if idaapi.is_code(idaapi.get_full_flags(addr)):
        mnem = idc.generate_disasm_line(addr,0)
        if mnem.startswith('jmp'):
            out.write(f'{hex(addr+5)}\n')

out.close()
```

编写python脚本处理数据，处理出[程序跳转到了第几个代码块]，方便在源代码中数数（call定不出来就挨个case数），再处理出不同的代码块数量

```
gdbout = open('disassembly_output.txt', 'r')
idaout = open('idaout.txt', 'r')

op_address = idaout.read().split('\n')
op_address = [int(_,16) for _ in op_address]

pc_list = []
inst_address_list = []
index_list = []
for i in range(2202220 // 2):
    context = []
    for j in range(2):
        context.append(gdbout.readline())

    pc = context[0].split(' ')[2]
    pc = int(pc, 16)
    pc_list.append(pc)


    inst = context[1].split(' ')[3]
    inst = int(inst, 16)
    inst_address_list.append(inst)

    try:
        index = op_address.index(inst)
        index_list.append(index)
    except:
        index_list.append(-1)

unique = list(set(index_list))
unique.sort()
print(unique)
print(len(unique))
```

pc_inst_index对应表:

qjvmp > tool > pc_inst_index3.txt

```
0x5156f9: 0x430658 at 14
0x5156fb: 0x431861 at 53
0x5207b9: 0x430658 at 14
0x5207bb: 0x43397e at 113
0x5207bc: 0x430658 at 14
0x5207be: 0x4339b0 at 114
0x5207bf: 0x43443a at 152
0x5207c2: 0x43443a at 152
0x5207c5: 0x430537 at 11
0x5207c7: 0x430537 at 11
0x5207c9: 0x430537 at 11
0x5207cb: 0x43059d at 12
0x5207ce: 0x430537 at 11
0x5207d0: 0x43059d at 12
0x5207d3: 0x430537 at 11
0x5207d5: 0x43059d at 12
0x5207d8: 0x430537 at 11
0x5207da: 0x430537 at 11
0x5207dc: 0x4304de at 10
0x5207dd: 0x430537 at 11
0x5207df: 0x430537 at 11
0x5207e1: 0x43059d at 12
0x5207e4: 0x430537 at 11
0x5207e6: 0x43059d at 12
0x5207e9: 0x430537 at 11
0x5207eb: 0x43059d at 12
```

```
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp/tool$ python bind.py
[-1, 2, 3, 4, 5, 7, 10, 11, 12, 14, 16, 17, 22, 23, 33, 36, 53, 55, 59, 65, 68, 71, 90, 105, 110, 111, 112, 113, 114, 115, 119, 120, 121, 131, 152, 153, 157, 172, 173, 175, 176, 177, 178, 196, 202, 212, 216, 223, 237, 245, 256, 273, 275, 277, 283, 285, 287, 299]
58
```

只需要手动找57个就好了呢。~~真是太好子~~

(-1是后来漏的，发现只出现了一次，就干脆不找了，标了个unknown)

然后我们就有了如下字典（第几个代码块对应什么操作）：

```
opdict = {
    -1:'unknown',
    2:'case_OP_push_minus1',
    3:'case_OP_push_0',
    4:'case_OP_push_1',
    5:'case_OP_push_2',
    6:'case_OP_push_3',
    7:'case_OP_push_4',
    8:'case_OP_push_5',
    9:'case_OP_push_6',
    10:'case_OP_push_7',
    11:'case_OP_push_i8',
    12:'case_OP_push_i16',
    14:'case_OP_closure8',
    16:'case_OP_get_length',
    17:'case_OP_push_atom_value',
    22:'case_OP_push_false',
    23:'case_OP_push_true',
    33:'case_OP_drop',
```

```
36:'case_OP_dup',
53:'case_OP_call0',
55:'case_OP_call1',
57:'case_OP_call2',
59:'case_OP_call3',
65:'case_OP_call_method',
68:'case_OP_array_from',
71:'case_OP_return',
90:'case_OP_get_var',
104:'case_OP_get_loc8',
105:'case_OP_put_loc8',
106:'case_OP_set_loc8',
107:'case_OP_get_loc0',
108:'case_OP_get_loc1',
109:'case_OP_get_loc2',
110:'case_OP_get_loc3',
111:'case_OP_put_loc0',
112:'case_OP_put_loc1',
113:'case_OP_put_loc2',
114:'case_OP_put_loc3',
115:'case_OP_set_loc0',
116:'case_OP_set_loc1',
117:'case_OP_set_loc2',
118:'case_OP_set_loc3',
119:'case_OP_get_arg0',
120:'case_OP_get_arg1',
121:'case_OP_get_arg2',
122:'case_OP_get_arg3',
123:'case_OP_put_arg0',
124:'case_OP_put_arg1',
125:'case_OP_put_arg2',
126:'case_OP_put_arg3',
127:'case_OP_set_arg0',
128:'case_OP_set_arg1',
129:'case_OP_set_arg2',
130:'case_OP_set_arg3',
131:'case_OP_get_var_ref',
152:'case_OP_get_var_ref_check?',
153:'case_OP_put_var_ref_check?',
157:'case_OP_get_loc_checkthis?',
172:'case_OP_goto16',
173:'case_OP_goto8',
175:'case_OP_if_true',
176:'case_OP_if_false',
177:'case_OP_if_true8',
178:'case_OP_if_false8',
196:'case_OP_post_inc',
197:'case_OP_post_dec',
202:'case_OP_define_field',
212:'case_OP_get_array_el',
216:'case_OP_put_array_el',
223:'case_OP_add',
237:'case_OP_mul',
245:'case_OP_pow',
256:'case_OP_post_inc',
273:'case_OP_xor',
```

```
    275:'case_OP_lt',
    277:'case_OP_lte',
    279:'case_OP_gt',
    281:'case_OP_gte',
    283:'case_OP_eq',
    285:'case_OP_neq',
    287:'case_OP_strict_eq',
    289:'case_OP_strict_neq',
    291:'case_OP_in',
    299:'case_OP_to_propkey2'
}
```

自然，我们也能够处理出pc: 操作（因为每一个pc跳到什么地址，以及什么地址对应什么操作我们都已得知）

qjvmp > tool > pc_inst_index.txt

```
0x5156f9: op_closure8
0x5156fb: op_call0
0x5207b9: op_closure8
0x5207bb: case_OP_put_loc2
0x5207bc: op_closure8
0x5207be: case_OP_put_loc3
0x5207bf: case_OP_get_var_ref_check?
0x5207c2: case_OP_get_var_ref_check?
0x5207c5: op_push_i8
0x5207c7: op_push_i8
0x5207c9: op_push_i8
0x5207cb: op_push_i16
0x5207ce: op_push_i8
0x5207d0: op_push_i16
0x5207d3: op_push_i8
0x5207d5: op_push_i16
0x5207d8: op_push_i8
0x5207da: op_push_i8
0x5207dc: case_OP_push_7
0x5207dd: op_push_i8
0x5207df: op_push_i8
0x5207e1: op_push_i16
0x5207e4: op_push_i8
0x5207e6: op_push_i16
0x5207e9: op_push_i8
0x5207eb: op_push_i16
0x5207ee: op_push_i8
0x5207f0: op_push_i8
0x5207f2: op_push_i8
0x5207f4: op_push_i16
0x5207f7: op_push_i8
0x5207f9: op_push_i8
0x5207fb: case_OP_push_2
0x5207fc: op_push_i8
0x5207fe: op_push_i8
0x520800: op_push_i8
```

master ⊗ 0 ⚠ 3 0

~~下面分析这段指令序列并找出程序逻辑吧！~~

找半天找不着，放弃了，栈机push来pop去的，具体操作什么数据也不知道(因为无法找到字节码与操作的一一对应)。

开始歪门斜道，下面登场的是测时间，基于对这东西应该运行很慢的猜想，如果他运行时动态check flag并退出，那么用时就会不同。当你输入了正确的flag片段越多，程序运行就越慢。

代码如下：

```
import subprocess
import time
import sys

executable = './qjvmp'
argument = 'flag'
```

```python
# 记录开始时间


time_list = []
out = open('/dev/null', 'w')
for _ in range(3000):
    start_time = time.perf_counter()
    # 执行可执行程序
    try:
        result = subprocess.run([executable, argument], check=True, stdout=out)
    except subprocess.CalledProcessError as e:
        print(f"执行失败: {e}")
        sys.exit(1)

    # 记录结束时间
    end_time = time.perf_counter()

    # 计算运行时长
    elapsed_time = end_time - start_time
    time_list.append(elapsed_time)

print(f"程序运行时长: {sum(time_list)/len(time_list)} 秒")
```

```
程序运行时长： 0.015184 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015278 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015206 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015272 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015053 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.018895 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015234 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015969 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015323 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015319 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015211 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.015089 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.014935 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.014869 秒
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
程序运行时长： 0.014897 秒
```

但是屁用没有，程序运行时间大致相同。考虑是统一check。

然后灵光一闪统计了一个不同操作类型调用的次数：

```
count: 1, op:unknown
count: 1, op:case_OP_push_7
count: 1, op:case_OP_push_atom_value
count: 1, op:case_OP_push_true
count: 1, op:case_OP_call0
count: 1, op:case_OP_set_loc0
count: 1, op:case_OP_if_false8
count: 2, op:case_OP_get_var
count: 3, op:case_OP_closure8
count: 10, op:case_OP_define_field
count: 11, op:case_OP_push_i16
count: 29, op:case_OP_push_i8
count: 41, op:case_OP_push_false
count: 42, op:case_OP_call1
count: 42, op:case_OP_get_loc3
count: 42, op:case_OP_goto16
count: 42, op:case_OP_xor
count: 42, op:case_OP_neq
count: 43, op:case_OP_call_method
count: 43, op:case_OP_if_true
count: 43, op:case_OP_post_inc
count: 336, op:case_OP_mul
count: 388, op:case_OP_array_from
count: 420, op:case_OP_push_4
count: 2352, op:case_OP_call3
count: 2352, op:case_OP_get_var_ref
count: 2352, op:case_OP_eq
count: 2395, op:case_OP_return
count: 2395, op:case_OP_put_loc0
count: 2395, op:case_OP_put_loc1
count: 2395, op:case_OP_put_loc2
count: 2395, op:case_OP_put_loc3
count: 2688, op:case_OP_put_array_el
count: 2688, op:case_OP_to_propkey2
count: 3655, op:case_OP_lt
count: 3943, op:case_OP_push_2
count: 4831, op:case_OP_get_length
count: 7056, op:case_OP_if_false
count: 9408, op:case_OP_push_minus1
count: 18816, op:case_OP_get_arg1
count: 18816, op:case_OP_get_arg2
count: 19235, op:case_OP_dup
count: 28224, op:case_OP_strict_eq
count: 28308, op:case_OP_get_arg0
count: 31374, op:case_OP_post_inc
count: 32460, op:case_OP_goto8
count: 33547, op:case_OP_push_0
count: 37632, op:case_OP_lte
count: 41112, op:case_OP_pow
count: 41139, op:case_OP_push_1
count: 47547, op:case_OP_put_loc8
count: 47797, op:case_OP_get_array_el
count: 50609, op:case_OP_get_loc_checkthis?
count: 50611, op:case_OP_drop
count: 61829, op:case_OP_get_var_ref_check?
count: 64807, op:case_OP_if_true8
```

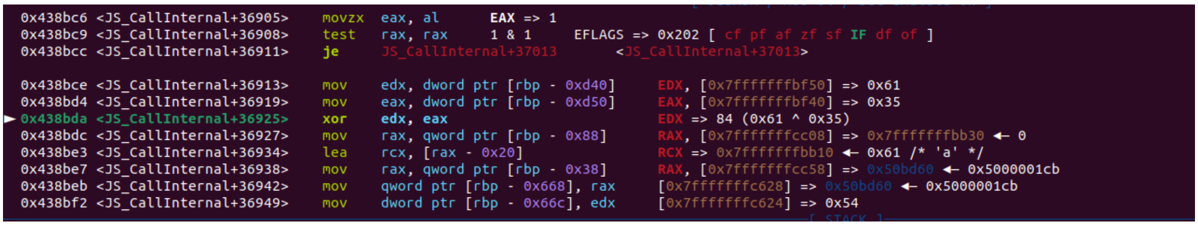
```
count: 95544, op:case_OP_add
count: 296817, op:case_OP_put_var_ref_check?
```

发现了什么？与加密和check相关的xor/eq/true/false在41,42,43这些数值

count: 41, op:case_OP_push_false
count: 42, op:case_OP_call1
count: 42, op:case_OP_get_loc3
count: 42, op:case_OP_goto16
count: 42, op:case_OP_xor
count: 42, op:case_OP_neq
count: 43, op:case_OP_call_method
count: 43, op:case_OP_if_true
count: 43, op:case_OP_post_inc

我们编写gdb脚本在case_OP_xor操作处下断，找到xor指令

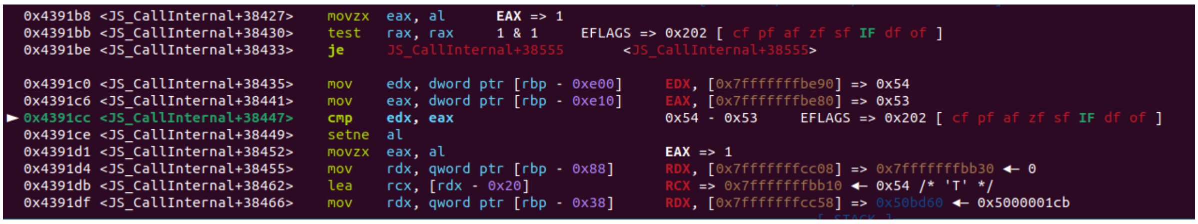
```
break *0x43013d if $rdx == 0x52093b

commands 1
    silent
    printf "rdx = 0x%lx\n", $rdx
end

start abcdefghijklmnopqrstuvwxyzabcdefghijklmnop
continue
```



```
 0x438bc6 <JS_CallInternal+36905>    movzx  eax, al        EAX => 1
 0x438bc9 <JS_CallInternal+36908>    test   rax, rax       1 & 1     EFLAGS => 0x202 [ cf pf af zf sf IF df of ]
 0x438bcc <JS_CallInternal+36911>    je     JS_CallInternal+37013       <JS_CallInternal+37013>

 0x438bce <JS_CallInternal+36913>    mov    edx, dword ptr [rbp - 0xd40]     EDX, [0x7fffffffbf50] => 0x61
 0x438bd4 <JS_CallInternal+36919>    mov    eax, dword ptr [rbp - 0xd50]     EAX, [0x7fffffffbf40] => 0x35
► 0x438bda <JS_CallInternal+36925>    xor    edx, eax                         EDX => 84 (0x61 ^ 0x35)
 0x438bdc <JS_CallInternal+36927>    mov    rax, qword ptr [rbp - 0x88]      RAX, [0x7fffffffcc08] => 0x7fffffffbb30 ◂— 0
 0x438be3 <JS_CallInternal+36934>    lea    rcx, [rax - 0x20]                RCX => 0x7fffffffbb10 ◂— 0x61 /* 'a' */
 0x438be7 <JS_CallInternal+36938>    mov    rax, qword ptr [rbp - 0x38]      RAX, [0x7fffffffcc58] => 0x50bd60 ◂— 0x5000001cb
 0x438beb <JS_CallInternal+36942>    mov    qword ptr [rbp - 0x668], rax     [0x7fffffffc628] => 0x50bd60 ◂— 0x5000001cb
 0x438bf2 <JS_CallInternal+36949>    mov    dword ptr [rbp - 0x66c], edx     [0x7fffffffc624] => 0x54
```

发现了'a' ^ 0x35（这个值是0x54）

同理检查case_OP_neq：



```
 0x4391b8 <JS_CallInternal+38427>    movzx  eax, al        EAX => 1
 0x4391bb <JS_CallInternal+38430>    test   rax, rax       1 & 1     EFLAGS => 0x202 [ cf pf af zf sf IF df of ]
 0x4391be <JS_CallInternal+38433>    je     JS_CallInternal+38555       <JS_CallInternal+38555>

 0x4391c0 <JS_CallInternal+38435>    mov    edx, dword ptr [rbp - 0xe00]     EDX, [0x7fffffffbe90] => 0x54
 0x4391c6 <JS_CallInternal+38441>    mov    eax, dword ptr [rbp - 0xe10]     EAX, [0x7fffffffbe80] => 0x53
► 0x4391cc <JS_CallInternal+38447>    cmp    edx, eax                         0x54 - 0x53     EFLAGS => 0x202 [ cf pf af zf sf IF df of ]
 0x4391ce <JS_CallInternal+38449>    setne  al
 0x4391d1 <JS_CallInternal+38452>    movzx  eax, al                          EAX => 1
 0x4391d4 <JS_CallInternal+38455>    mov    rdx, qword ptr [rbp - 0x88]      RDX, [0x7fffffffcc08] => 0x7fffffffbb30 ◂— 0
 0x4391db <JS_CallInternal+38462>    lea    rcx, [rdx - 0x20]                RCX => 0x7fffffffbb10 ◂— 0x54 /* 'T' */
 0x4391df <JS_CallInternal+38466>    mov    rdx, qword ptr [rbp - 0x38]      RDX, [0x7fffffffcc58] => 0x50bd60 ◂— 0x5000001cb
```

这有个cmp 0x54 0x53（正好就是检查刚刚异或出来的值）

检查指令序列很难不怀疑是在检查a0 ^ 0x35 == 0x53

```
0x52093b: case_OP_xor
0x52093c: case_OP_put_var_ref_check?
0x52093f: case_OP_put_var_ref_check?
0x520942: case_OP_get_array_el
0x520943: case_OP_neq
0x520944: case_OP_if_true8
0x520946: case_OP_push_false
0x520947: case_OP_dup
```

```
0x520948: case_OP_get_loc_checkthis?
0x52094b: case_OP_drop
0x52094c: case_OP_put_var_ref_check?
0x52094f: case_OP_post_inc
0x520950: case_OP_get_loc_checkthis?
0x520953: case_OP_drop
0x520954: case_OP_goto16
```

编写gdb脚本，处理出所有的xor数据和cmp数据

```
break *0x438bda

commands 1
    silent
    printf "%c ^ %c, 0x%x\n", $edx, $eax, $eax
    continue
end

start abcdefghijklmnopqrstuvwxyzabcdefghijklmnop
continue
```

```
break *0x4391cc

commands 1
    silent
    printf "0x%x\n", $eax
    continue
end

start abcdefghijklmnopqrstuvwxyzabcdefghijklmnop
continue
```

整理后结果如下：

```
xordata =
[0x35,0x7a,0x42,0xef,0x18,0xc1,0x78,0xf6,0x55,0x2a,0x66,0x56,0x2d,0xbb,0x35,0xe9,0x4d,0xeb,0x11,0x50,0x78,0x91,0x5c,0x4f,0x60,0x50,0x2b,0x49,0x35,0x7a,0x42,0xef,0x18,0xc1,0x78,0xf6,0x55,0x2a,0x66,0x56,0x2d,0xbb]

neqdata =
[0x53,0x16,0x23,0x88,0x63,0xf3,0x1b,0xcf,0x67,0x1e,0x7,0x64,0x48,0x96,0x54,0xd8,0x2c,0xd2,0x3c,0x64,0x1d,0xf4,0x6e,0x62,0x2,0x33,0x1b,0x28,0x18,0x1e,0x75,0x8e,0x7a,0xf3,0x4f,0xce,0x60,0x4f,0x5e,0x60,0x4e,0xc6]
```

编写脚本得到flag

```
xor_data = [i^j for i,j in zip(xordata, neqdata)]
print(''.join([chr(_) for _ in xor_data]))
```

```
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$ python final.py
flag{2c924a2e-a1a9-4ee2-bc0a-d7ab2785e86c}
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$ ./qjvmp flag{2c924a2e-a1a9-4ee2-bc0a-d7ab2785e86c}
true
ctf@ctf-virtual-machine:~/Desktop/pwn/tmp/qjvmp$
```