

OLLVM (7) ——OLLVM 去除控制流平坦化

——王泉成 2020.3.2

简单说明：

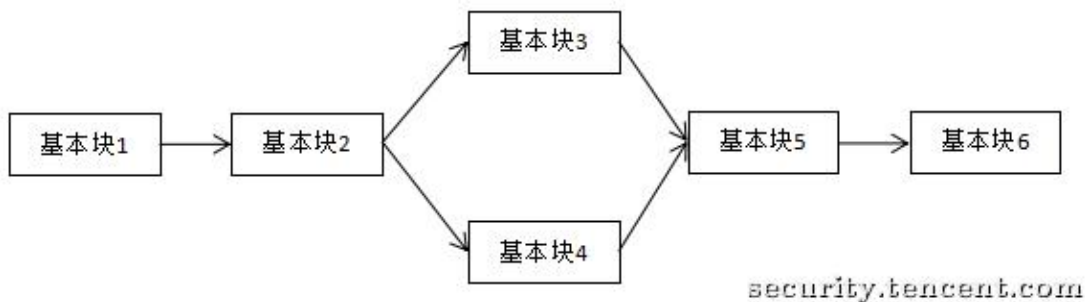
我对这方面也不太懂，只是看了一些网上的文章，然后参考别人的路线，然后尝试复现一遍，并且找了一道 CTF 中的相关题目做了练习。

主要涉及两个方面有关控制流平坦化，一个是有关 IDA microcode，还有一个有关符号执行。

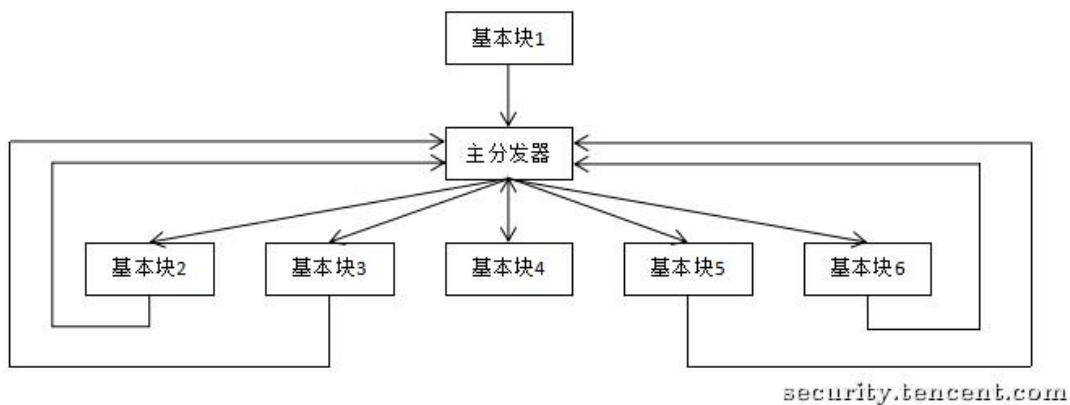
一、控制流平坦化

1. 简单介绍

控制流平坦化(control flow flattening)的基本思想主要是通过一个主分发器来控制程序基本块的执行流程，例如下图是正常的执行流程。



经过控制流平坦化后的执行流程就如下图。



这样可以模糊基本块之间的前后关系，增加程序分析的难度，同时这个流程也很像 VM 的执行流程。

2. 编写一个简单的程序

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

int main()
```

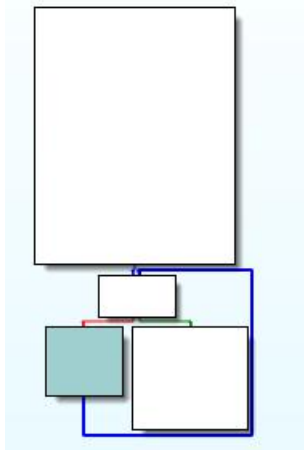
```

{
    printf("Plz input a number.\n");
    int n;
    scanf("%d", &n);
    printf("You have input: %d\n", n);
    int sum = 1;
    for (int i = 1; i <= n; i++)
    {
        sum *= i;
    }
    printf("f(n): %d\n", sum);
    return 0;
}

```

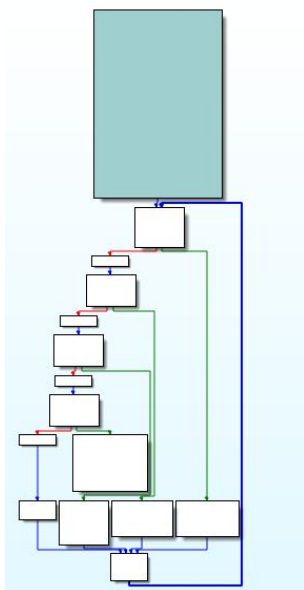
3. 未经过控制流平坦化的流程图

其实之前也以及介绍过这个图片，不多说。



4. 经过控制流平坦化以后的流程图

可以看到添加 flr 后的程序结构有了很大的不同。



二、使用 IDA microcode 去除控制流平坦化

5. microcode 和 ctree

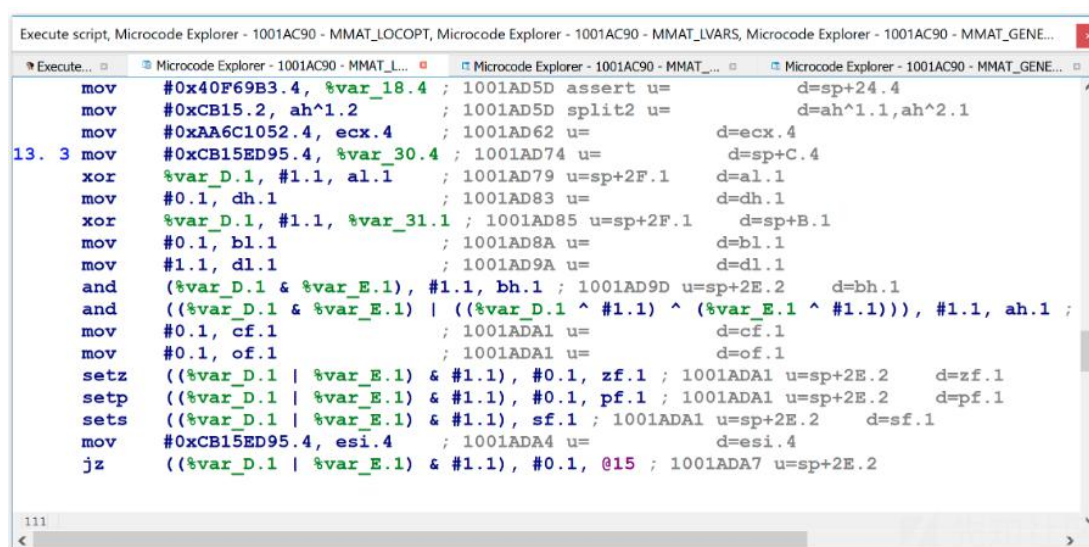
IDA 7.1 中 IDA 发布了反编译中使用的中间语言 microcode

IDA 7.2 和 7.3 中又新增了相关的 C++和 python API

IDA 反编译器中二进制代码有两种表现形式

microcode: 处理器指令被翻译成 microcode, 反编译器对其进行优化和转换。

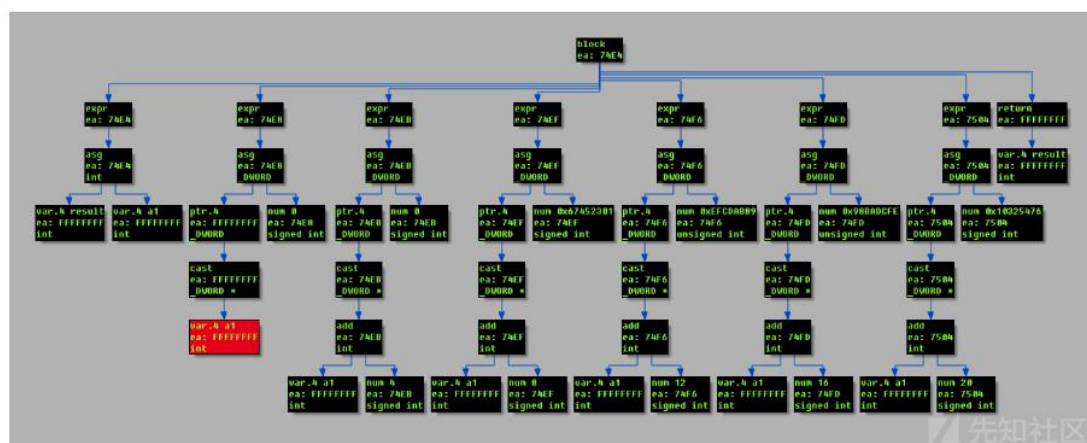
使用 HexRaysDeob 插件除了处理 ollvm 混淆也可以查看 microcode。



```
mov #0x40F69B3.4, %var_18.4 ; 1001AD5D assert u= d=sp+24.4
mov #0xCB15.2, ah^1.2 ; 1001AD5D split2 u= d=ah^1.1,ah^2.1
mov #0xAA6C1052.4, ecx.4 ; 1001AD62 u= d=ecx.4
13. 3 mov #0xCB15ED95.4, %var_30.4 ; 1001AD74 u= d=sp+C.4
xor %var_D.1, #1.1, al.1 ; 1001AD79 u=sp+2F.1 d=al.1
mov #0.1, dh.1 ; 1001AD83 u= d=dh.1
mov %var_D.1, #1.1, %var_31.1 ; 1001AD85 u=sp+2F.1 d=sp+B.1
mov #0.1, bl.1 ; 1001AD8A u= d=bl.1
mov #1.1, dl.1 ; 1001AD9A u= d=dl.1
and (%var_D.1 & %var_E.1), #1.1, bh.1 ; 1001AD9D u=sp+2E.2 d=bh.1
and ((%var_D.1 & %var_E.1) | ((%var_D.1 ^ #1.1) ^ (%var_E.1 ^ #1.1))), #1.1, ah.1 ;
mov #0.1, cf.1 ; 1001ADA1 u= d=cf.1
mov #0.1, of.1 ; 1001ADA1 u= d=of.1
setz ((%var_D.1 | %var_E.1) & #1.1), #0.1, zf.1 ; 1001ADA1 u=sp+2E.2 d=zf.1
setp ((%var_D.1 | %var_E.1) & #1.1), #0.1, pf.1 ; 1001ADA1 u=sp+2E.2 d=pf.1
sets ((%var_D.1 | %var_E.1) & #1.1), sf.1 ; 1001ADA1 u=sp+2E.2 d=sf.1
mov #0xCB15ED95.4, esi.4 ; 1001ADA4 u= d=esi.4
jz ((%var_D.1 | %var_E.1) & #1.1), #0.1, @15 ; 1001ADA7 u=sp+2E.2
```

ctree: 由优化的 microcode 构建而成, 用 C 语句和表达式表示像 AST 一样的树。

使用 HexRaysCodeXplorer 插件或者 IDA python 中的示例 vds5.py 可以查看 ctree。



6. microcode 数据结构

这里有四个比较重要的数据结构

- mbl_array_t

保存关于反编译代码和基本块数组的信息。

- `mblock_t`

保存关于反编译代码和基本块数组的信息。

- `minsn_t`

表示一条指令。

- `mop_t`

表示一个操作数，根据它的类型可以表示不同的信息（数字，寄存器，堆栈变量等等）。

在先知社区的文章上有一些概念图，可以自行查看理解。

7. 反混淆器的简单说明

`// HexRaysDeob`

IDA 的插件入口一般会有三个函数是 `init`，`term` 和 `run`。

作用分别是初始化，清理和调用插件。

`init` 函数中调用了 `install_optinsn_handler` 函数和 `install_optblock_handler` 函数。

进行指令级别的优化 (`PatternDeobfuscate`) 和块级别的优化 (`Unflattener`)。

HexRays 会自动调用注册的回调对象。

8. 确定平坦块编号到 `mblock_t` 的映射

9. 确定每个平坦块的后继者

10. 直接将控制流从源块转移到目标块

<https://xz.aliyun.com/t/6749#toc-6>

三、 利用符号执行去除控制流平坦化

11. 符号执行

符号执行是一种重要的形式化方法和软件分析技术，通过使用符号执行技术，将程序中变量的值表示为符号值和常量组成的计算表达式，符号是指取值集合的记号，程序计算的输出被表示为输入符号值的函数，其在软件测试和程序验证中发挥着重要作用，并可以应用于程序漏洞的检测。

符号执行的发展是从静态符号执行到动态符号执行到选择性符号执行，动态符号执行会以具体数值作为输入来模拟执行程序，是混合执行 (concolic execution) 的典型代表，有很高的精确度，目前较新的符号执行工具有 Triton 和 angr。

12. Angr

`// 百度一下，一个符号执行工具。`

`// 安装方式 pip install angr 即可。在 github 上也有 demo，可以借鉴简单学习。`

13. 获取真实块、序言、retn 块和无用块

// 这个地方其实还蛮有意思的，我还没有深究含义

文章的作者说使用的 BARF 来进行获取。

<https://github.com/programa-stic/barf-project>

```
def get_retn_predispatcher(cfg):
    global main_dispatcher
    for block in cfg.basic_blocks:
        if len(block.branches) == 0 and block.direct_branch == None:
            retn = block.start_address
        elif block.direct_branch == main_dispatcher:
            pre_dispatcher = block.start_address
    return retn, pre_dispatcher

def get_relevant_nop_blocks(cfg):
    global pre_dispatcher, prologue, retn
    relevant_blocks = []
    nop_blocks = []
    for block in cfg.basic_blocks:
        if block.direct_branch == pre_dispatcher and len(block.instrs) != 1:
            relevant_blocks.append(block.start_address)
        elif block.start_address != prologue and block.start_address != retn:
            nop_blocks.append(block)
    return relevant_blocks, nop_blocks
```

14. 确定真实块、序言和 retn 块的前后关系

这个步骤主要是使用符号执行。为了方便，这里把真实块、序言和 retn 块统称为真实块。

符号执行从每个真实块的起始地址开始，直到执行到下一个真实块。

如果遇到分支，就改变判断值执行两次来获取分支的地址。

这里用 angr 的 inspect 在遇到类型为 ITE 的 IR 表达式时，改变临时变量的值来实现。

```
for relevant in relevants_without_retn:
    block = cfg.find_basic_block(relevant)
    has_branches = False
    hook_addr = None
    for ins in block.instrs:
        if ins.asm_instr.mnemonic.startswith('cmov'):
            patch_instrs[relevant] = ins.asm_instr
            has_branches = True
        elif ins.asm_instr.mnemonic.startswith('call'):
            hook_addr = ins.address
    if has_branches:
        flow[relevant].append(symbolic_execution(relevant, hook_addr,
```

```

claripy.BVV(1, 1), True))
    flow[relevant].append(symbolic_execution(relevant, hook_addr,
claripy.BVV(0, 1), True))
    else:
        flow[relevant].append(symbolic_execution(relevant, hook_addr))

```

15. Patch 二进制程序

无用块处理

```

for nop_block in nop_blocks:
    fill_nop(origin_data, nop_block.start_address - base_addr,
nop_block.end_address - base_addr + 1)

```

没有分支的真实块 jmp 到下一个真实块

```

last_instr = cfg.find_basic_block(parent).instrs[-1].asm_instr
file_offset = last_instr.address - base_addr
origin_data[file_offset] = opcode['jmp']
file_offset += 1
fill_nop(origin_data, file_offset, file_offset + last_instr.size - 1)
fill_jump_offset(origin_data, file_offset, childs[0] - last_instr.address - 5)

```

处理分支的真实块

```

instr = patch_instrs[parent]
file_offset = instr.address - base_addr
fill_nop(origin_data, file_offset, cfg.find_basic_block(parent).end_address -
base_addr + 1)
origin_data[file_offset] = opcode['j']
origin_data[file_offset + 1] = opcode[instr.mnemonic[4:]]
fill_jump_offset(origin_data, file_offset + 2, childs[0] - instr.address - 6)
file_offset += 6
origin_data[file_offset] = opcode['jmp']
fill_jump_offset(origin_data, file_offset + 1, childs[1] - (instr.address + 6)
- 5)

```

16. 实例去除控制流平坦化演示

第一步，安装 BARF

```

root@kali:~/桌面/barf-project# python setup.py install
running install
running bdist_egg
running egg_info
writing requirements to barf.egg-info/requirements.txt

```

安装 angr

```

root@kali:~/桌面/deflat# pip install angr
Collecting angr
Using cached https://files.pythonhosted.org/packages/1a/0a/6b2463555ac1cd6a40be49d20a10d94da22cbba/angr-7.8.9.26-py

```

安装 simuvex


```

import simuvex
ImportError: No module named simuvex
root@kali:~/桌面/deflat# pip install simuvex
Collecting simuvex

```

第二步，得到 main 函数地址

两种方式，objdump 或者 gdb。我采用 gdb。

```

Dump of assembler code for function main:
0x0000000000401140 <+0>:    push    %rbp
0x0000000000401141 <+1>:    mov     %rsp,%rbp
0x0000000000401144 <+4>:    sub     $0x40,%rsp
0x0000000000401148 <+8>:    movabs  $0x402004,%rdi
0x0000000000401152 <+18>:   movl    $0x0,-0x4(%rbp)
0x0000000000401159 <+25>:   mov     $0x0,%al

```

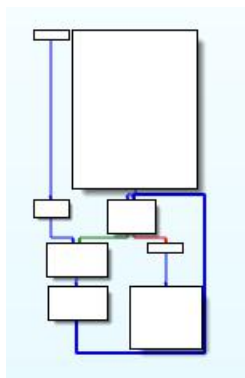
第三步，执行 python 脚本

```

root@kali:~/桌面/deflat-angr/flat_control_flow# python3 deflat.py test-fla 0x401140
*****relevant blocks*****
prologue: 0x401140
main_dispatcher: 0x4011a7
pre_dispatcher: 0x401271
retn: 0x401250
relevant_blocks: ['0x401225', '0x40123b', '0x40120a', '0x401200']
*****symbolic execution*****
-----dse 0x401225-----
-----dse 0x40123b-----
-----dse 0x40120a-----
-----dse 0x401200-----
-----dse 0x401140-----
CRITICAL | 2020-03-02 17:33:22,321 | angr.sim_state | The name state.se is deprecated; please use state.solver.
*****flow*****
0x401225:  ['0x40123b']
0x40123b:  ['0x40120a']
0x40120a:  ['0x401225', '0x401250']
0x401200:  ['0x401225']
0x401140:  ['0x40120a']
0x401250:  []
*****patch*****
Successful! The recovered file: test-fla_recovered

```

第四步，用 IDA 看恢复后的程序流程结构



第五步，反编译出伪 C

效果非常好。

```

int __cdecl main(int argc, const char **argv, const char **envp)
{
    signed int i; // [sp+30h] [bp-10h]@1
    signed int v5; // [sp+34h] [bp-Ch]@1
    unsigned int v6; // [sp+38h] [bp-8h]@1
    int v7; // [sp+3Ch] [bp-4h]@1

    v7 = 0;
    printf("Plz input a number.\n", argv, envp);
    __isoc99_scanf("%d", &v6);
    printf("You have input: %d\n", v6);
    v5 = 1;
    for ( i = 1; i <= (signed int)v6; ++i )
        v5 *= i;
    printf("f(n): %d\n", (unsigned int)v5, (unsigned int)i);
    return 0;
}

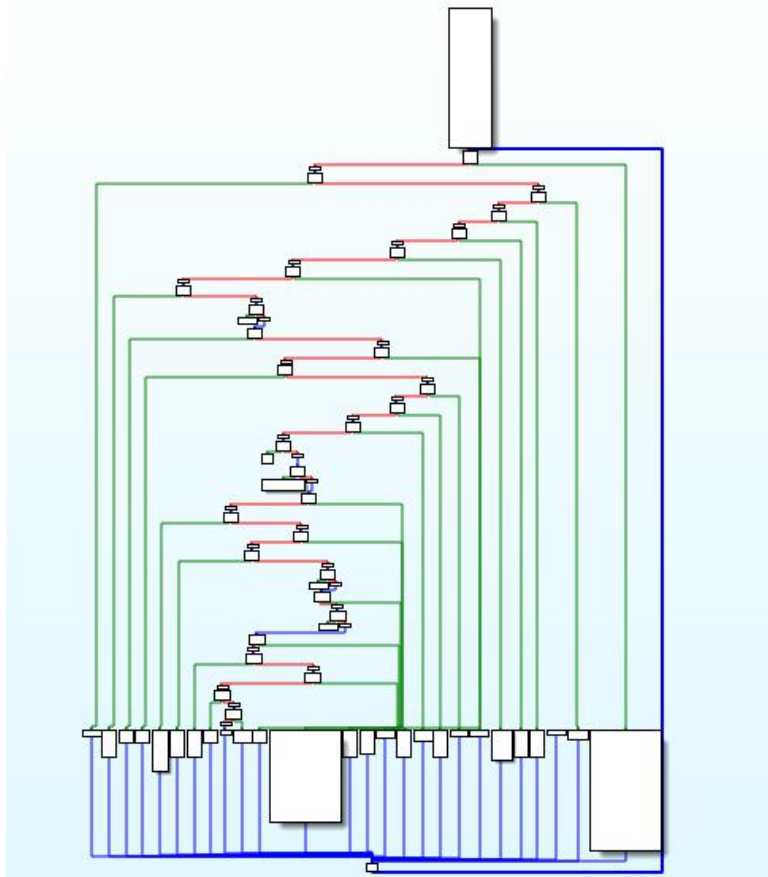
```

四、一道 CTF 中控制流平坦化题目

SUCTF-2019 hardCpp:

<https://github.com/team-su/SUCTF-2019/tree/master/Rev/hardCpp>

17. 首先把程序放到 IDA 中分析



发现这个程序的整体结构是 ollvm 混淆以后的，非常明显。

```
case 739060228:
    v33 = strlen(s);
    v47 = v33 != 21;
    v5 = 2137009843;
    if (y < 10 || (((_BYTE)x - 1) * (_BYTE)x & 1) == 0 )
        v5 = -1234402024;
    v24 = v5;
    break;
case 1011555671:
    v10 = 1299792205;
    if (y < 10 || (((_BYTE)x - 1) * (_BYTE)x & 1) == 0 )
        v10 = -2000540314;
    v24 = v10;
    break;
case 1132336453:
    v8 = 623475433;
    v32 = 1;
    if (y < 10 || (((_BYTE)x - 1) * (_BYTE)x & 1) == 0 )
        v8 = -1257245689;
    v24 = v8;
```

18. 去 fla 和 bcf

首先，gdb 找到 main 函数地址

```
Dump of assembler code for function main:
0x00000000004007e0 <+0>:    push    %rbp
0x00000000004007e1 <+1>:    mov     %rsp,%rbp
0x00000000004007e4 <+4>:    sub     $0x130,%rsp
0x00000000004007eb <+11>:   xor     %eax,%eax
```

然后执行 python 脚本去除伪控制流

```
root@kali:~/桌面/deflat-angr/bogus_control_flow# python3 debugus.py hardCpp 0x4007e0
*****symbolic execution*****
```

去除控制流平坦化

```
root@kali:~/桌面/deflat-angr/flat_control_flow# python3 deflat.py hardCpp-erase-bcf 0x4007e0
*****relevant blocks*****
prologue: 0x4007e0
main_dispatcher: 0x400876
```

19. 再次使用 IDA 分析 C 代码

首先看到 puts

```
puts("func(?)=\\"01abfc750a0c942167651c40d088531d\\"?");
```

这是#的 md5

后面读入 21 个字符

```
s[0] = getchar();
fgets(&s[1], 21, stdin);
```

核心的地方在后面的 while 中

```

    }
    while (v19 < 21)
    {
        if (y >= 10 && (((_BYTE)x - 1) * (_BYTE)x & 1) != 0)
        {
            v18 = v21 ^ s[v19];
            v17 = main::$_0::operator() const((__int64)&v26, v18);
            v16 = main::$_1::operator() const((__int64)&v24, s[v21 - 1 + v19]);
            v8 = main::$_1::operator() const(char):: { lambda(int) #1 }
                ::operator() const(&v16, 7);
        }
    }
}

```

接下来不妨看看 main::\$_0 这几个函数的具体作用

其实非常简单就是四种不同的操作，分别是 xor，add，mul，mod

然后我们不妨将 while 这个简化一下：

```

v18 = v21 ^ s[v19];
v17 = v18;
v18 = v21 ^ s[v19];
v17 = v18;
v16 = s[v21 - 1 + v19];
v3 = v16[0] % 7;
v18 = v17 + v3;
v15 = v18;
v14 = s[v21 - 1 + v19];
v4 = 18 ^ v14[0];
v13 = v4;
v5 = v13[0] * 3;
v12 = v5;
v6 = v12[0] + 2;
v18 = v15[0] ^ v6;

```

这样，结合 time=0，我们就可以得到

```

// res
v18 = ((v21 ^ s[v19]) + (s[v21 - 1 + v19] % 7)) ^ ((18 ^ s[v21 - 1 + v19]) * 3 + 2);
c = ((time ^ input[i]) + (input[time - 1 + i] % 7)) ^ ((18 ^ input[time - 1 + i]) * 3 + 2);
time = 0;
// so
c = ((0 ^ input[i]) + (input[i - 1] % 7)) ^ ((18 ^ input[i - 1]) * 3 + 2);

```

很显然，可以由密文得到明文

密文很容易获得：

```

.data:0000000000602060 public enc
.data:0000000000602060 ; char enc[]
.data:0000000000602060 enc db 0F3h ; DATA XREF: main+70A1r
.data:0000000000602061 a_6slM@v db '.',18h,'6酷"样嬉v'
.data:000000000060206F db 5
.data:0000000000602070 db 0A3h ;
.data:0000000000602071 db 90h ;
.data:0000000000602072 db 0Eh ;
.data:0000000000602073 db 0A5h ;
.data:0000000000602073 _data ends
.data:0000000000602073

```

然后反解到明文:

```

flag = [0 for i in range(21)]
flag[0] = 0x23
enc = 'F3 2E 18 36 E1 4C 22 D1 F9 8C 40 76 F4 0E 00 05 A3 90 0E A5'.split(' ')
c = [int(i,16) for i in enc]
for i in range(1,len(c)+1):
    flag[i] = (((c[i-1] ^ ((flag[i-1] ^ 18) * 3 + 2)) - (flag[i-1]%7)) ^ 0) & 0xff)
print(''.join(map(chr,flag)))

```

```

@y_wang python flag.py
#flag{mY-CurRled_Fns}

```

五、 主要参考链接

先知社区:

<https://xz.aliyun.com/t/6749>

腾讯安全应急响应中心:

<https://security.tencent.com/index.php/blog/msg/112>