FCSC 2020 CTF 代码混淆逆向题 Keykoolol 的 writeup

原创

systemino 于 2020-06-04 09:39:39 发布

通过基于虚拟机的混淆保护了keygenme，这是一种基于Miasm动态符号执行（DSE）的解决方案，可自动反汇编VM字节码。

几周前，我参加了法国网络安全挑战赛（简称FCSC）。由法国国家网络安全局（ANSSI）组建的Jeopardy CTF，以选择将在2020年底参加欧洲网络安全挑战赛（ECSC）的法国团队。

在提出的挑战（加密，反向，pwn，网络，取证，硬件）中，我真的很喜欢做一个名为keykoolol的反向追踪。

挑战的目的是分析一个以用户名和序列作为输入的二进制文件，并为其编写一个密钥。然后，我们必须使用此密钥生成器为多个用户名生成良好的序列以获取标志。

二进制文件是ELF x86-64可执行文件，非常小，奇热只有14KB。

```
$ file keykoolol
keykoolol: ELF 64-bit LSB pie executable, x86-
64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-
64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=1422aa3ad6edad4cc689ec6ed5d9fd4e6263cd72, stripped
```

如果使用伪输入执行它，则会得到以下输出。

```
$ ./keykoolol
[+] Username: aaaaaa
[+] Serial:   bbbbbbbbb
[!] Incorrect serial.
```

让我们在我们最喜欢的反汇编程序中将其打开，以查看代码的外观。这是IDA反编译器的main函数。

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  __int64 v3; // rdx
  const char *output; // rdi
  char username[512]; // [rsp+8h] [rbp-420h]
  char serial[512]; // [rsp+208h] [rbp-220h]
  unsigned __int64 v8; // [rsp+408h] [rbp-20h]

  v8 = __readfsqword(0x28u);
  __printf_chk(1LL, "[+] Username: ", a3);
  fgets(username, 512, stdin);
  username[strcspn(username, "\n")] = 0;
  __printf_chk(1LL, "[+] Serial:   ", v3);
  fgets(serial, 512, stdin);
  serial[strcspn(serial, "\n")] = 0;
  output = "[!] Incorrect serial.";
  if ( (unsigned int)check((__int64)dword_24E0, 1024LL, username, strlen(username), serial, strlen(serial)) )
  {
    puts("[>] Valid serial!");
    output = "[>] Now connect to the remote server and generate serials for the given usernames.";
  }
  puts(output);
  return 0LL;
}
```

以上是反编译器输出的IDAmain函数，如你所见，main函数的代码易于阅读，并包含以下步骤。

1. 从stdin读取用户名和序列；

2. 删除两个输入的换行符；

3. 调用包含6个参数的函数（此处重命名为check），包括用户名，序列及其各自的长度。

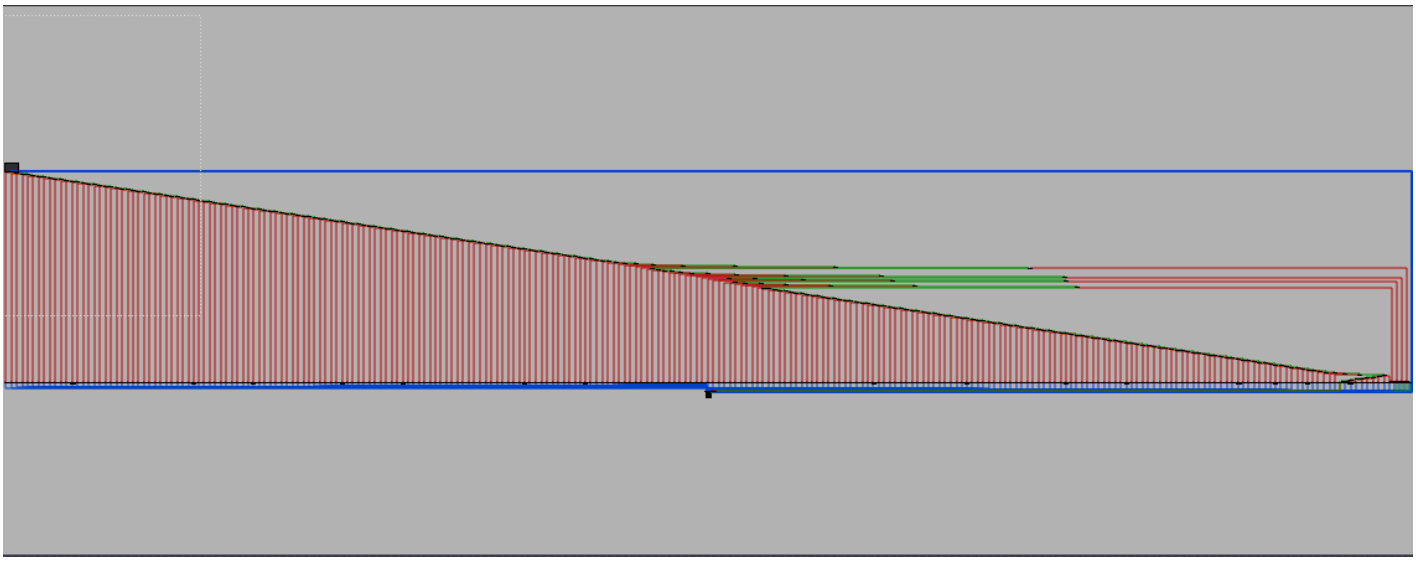
4. 如果函数的返回值不同于0，则说明该用户名/序列是正确的。

现在，让我们看一下函数check。

串口校验函数分析

在IDA反编译器中打开check函数时，我们注意到的第一件事是IDA对其进行反编译所花费的时间。

如果我们查看反汇编控制流程图，我们会很快理解为什么，函数非常强大！



让我们深入研究代码，以了解函数的结构。

```
v8 = dword_203040;
for ( i = 16LL; i; --i )
{
  *v8 = 0;
  ++v8;
}
v11 = &unk_203080;
v12 = 512LL;
v13 = a2 & 0xFFFFFFF0;
while ( v12 )
{
  *v11 = 0;
  ++v11;
  --v12;
}
```

函数首先在0x203040处初始化前40个字节，在0x203080到0处初始化前2048个字节。

```
v14 = (_DWORD *)__memcpy_chk(&unk_203080, a1, a2, 2048LL);
dword_203064 = username_length;
dword_203060 = v13 + 16;
username_dest = (char *)v14 + (unsigned int)(v13 + 16);
v16 = v13 + 16 + ((unsigned int)username_length & 0xFFFFFFF0) + 16;
LODWORD(v18) = dword_20302C;
v19 = 0;
qmemcpy(username_dest, username, username_length);
dword_203068 = v16;
dword_20306C = serial_length;
v1 = dword_205880;
v21 = 0;
v22 = 0;
qmemcpy((char *)v14 + v16, serial, serial_length);
```

然后，它依次复制：

1. 将位于0x24E0(函数的第一个参数)处的缓冲区发送到地址0x203080；

2. 用户名缓冲区地址为0x203490（0x203080 + 1024 + 16）；

3. 用户名缓冲区地址加16之后的序列缓冲区。

然后，它进入一个无限循环，并从0x24E0(现在是0x203080)处的缓冲区读取一个32位的整数。

```
while ( 1 )
{
  v1 = *(_DWORD *)((char *)v14 + v3);
  v2 = *(_DWORD *)((char *)v14 + v3) >> 24;
  switch ( v2 )
  {
    case 0:
      dword_203040[v1 >> 20] = dword_203040[(v1 >> 16) & 0xF];
      v26 = v3 + 4;
      goto LABEL_10;
    case 31:
      dword_203054 ^= 0xF7E1560A;
      v26 = v3 + 4;
      goto LABEL_10;
    case 32:
      dword_203048 ^= 0x6DDC660Cu;
      v26 = v3 + 4;
      goto LABEL_10;
    case 33:
      dword_203074 ^= 0x13E40C56u;
      v26 = v3 + 4;
      goto LABEL_10;
```

读取32位整数中最重要的字节，并根据该值将控制流移至交换机的256个条目中的一个。如果仔细看一下交换机的不同分支，我们几乎可以在交换机的每个分支中观察到以下代码模式：

1. 在0x203040和0x203080之间的内存上进行一个简单的操作；

2. 一个变量增加了4(这恰好是在输入开关之前读取的整数的大小)；

3. 如果开关值不等于255，则控制流程返回while循环的开始。

这种结构使我们得出的结论是，我们在本文处理的是虚拟机，而不是一个很小的虚拟机，因为它实现了256条指令。

虚拟机的结构

由此，我们可以对虚拟机的结构做出以下几个假设：

1. VM有16个32位寄存器，存储在0x203040；

2. 从0x24E0复制的神秘缓冲区是VM的字节码；

3. VM程序计数器存储在ESI中；

4. VM标志（或多或少等同于EFLAGS寄存器的非常简化的版本）存储在R9中；

5. 操作码255的执行意味着序列不正确，因为它将check的返回值设置为0；

6. 0x203880处的内存对应于VM的堆栈。

同样，我们可以通过查看字节码中存在的不同操作码来评估VM执行的实际操作码。但是，这意味着VM的代码本身不会打补丁（SPOILER：它会打补丁）。

尽管如此，下面是字节码中出现的55个不同操作码的列表，其前面是出现的次数。

```
23 15
22 212
16 0
14 216
13 8
13 19
12 29
12 12
8 24
8 18
7 2
6 6
6 202
5 223
5 21
5 11
4 25
4 221
4 220
4 214
4 10
3 28
3 27
3 26
3 23
3 215
3 206
3 17
3 1
2 9
2 3
2 255
2 254
2 219
2 218
2 210
2 207
2 204
2 20
2 14
1 98
1 63
1 42
1 35
1 244
1 217
1 213
1 201
1 200
1 197
1 195
1 194
1 193
1 192
1 188
```

反汇编VM字节码

为了反汇编VM字节码并继续进行分析，我们可以采用以下几种策略：

1. 在Python脚本中实现每个操作码并读取字节码；

2. 编写VM的体系结构插件，由IDA或Binary Ninja等反汇编程序支持，并使用它打开字节码；

3. 追踪VM执行并从中提取VM指令（例如，使用Triton）。

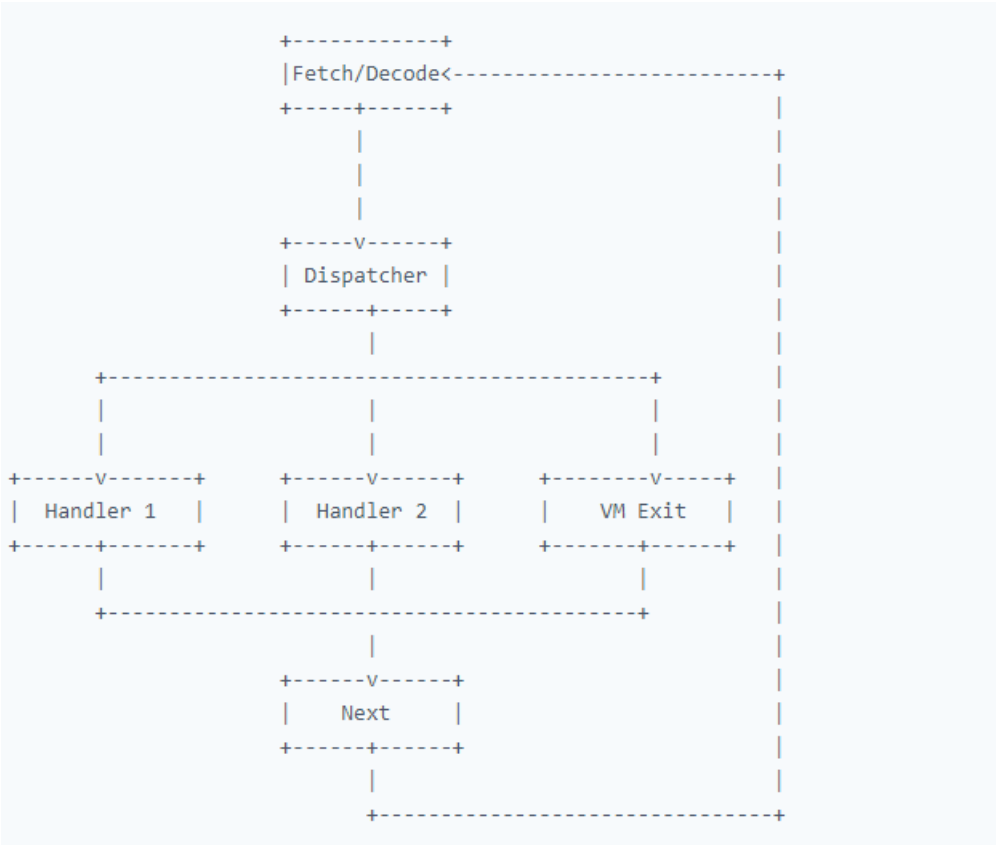
由于我有点懒，并且我不想重新执行每条指令，因此我选择了某种类似于上一个方案的解决方案，该方案包含使用Miasm动态符号执行（简称DSE）自动反汇编每条已执行的VM指令。最终目标是清晰地了解VM执行情况，该解决方案的优势在于，如果VM进行自我修改，我们可以观察到它并反汇编修改后的VM字节码。

Miasm DSE

Miasm动态符号执行的用法并没有真正的使用说明(实际上是整个项目)，但是你可以在Miasm博客上找到示例和代码中的伪文档。

此外，如果你遇到问题，可以通过Gitter或直接通过GitHub存储库与Miasm开发人员联系。

以下是在我们的案例中VM周期的图示：



为了反汇编一个给定的VM指令，我们需要在每个VM周期中获得关于VM状态(寄存器+标志+堆栈)的约束。

1. 在dispatcher中，我们从具体的执行中更新DSE状态，并将与VM状态对应的内存符号化；

2. 执行与VM指令相对应的处理程序代码；

3. 在下一步，我们评估在DSE状态上所做的修改并打印它们。

4. 如果指令不是VM退出，则返回1。

实际操作

在Miasm中，具体的执行函数由Jitter提供，我们可以通过导入miasm.analysis.sandbox中定义的类Sandbox_Linux_x86_64来激活ELF x86-64可执行文件。

```
from miasm.analysis.sandbox import Sandbox_Linux_x86_64

parser = Sandbox_Linux_x86_64.parser("Disassembler for keykoolol challenge")
parser.add_argument("filename", help="Challenge filename")
options = parser.parse_args()

sb = Sandbox_Linux_x86_64(options.filename, options, globals())
sb.run()
```

在执行上面的Python代码时，我们会遇到以下错误。

由于有对外部API的调用，因此我们必须在代码中处理它们。

```
def xxx___printf_chk(jitter):
    ret_ad, args = jitter.func_args_systemv(["flag", "format", "arg"])
    print(jitter.get_c_str(args.format))
    return jitter.func_ret_systemv(ret_ad, 1)

def xxx_fgets(jitter):
    ret_ad, args = jitter.func_args_systemv(["dest", "size", "stream"])
    s = input()
    jitter.vm.set_mem(args.dest, s.encode())
    return jitter.func_ret_systemv(ret_ad, len(s))

def xxx_strcspn(jitter):
    ret_ad, args = jitter.func_args_systemv(["s", "rejected"])
    s = jitter.get_c_str(args.s)
    jitter.vm.set_mem(args.s, s.strip().encode())
    return jitter.func_ret_systemv(ret_ad, len(s))

def xxx___memcpy_chk(jitter):
    ret_ad, args = jitter.func_args_systemv(["dest", "src", "len", "destlen"])
    src = jitter.vm.get_mem(args.src, args.len)
    jitter.vm.set_mem(args.dest, src)
    return jitter.func_ret_systemv(ret_ad, args.dest)
```

添加句柄后，我们就可以输入我们想要的用户名和序列(这里是aaaaaaa和aaaaaaaaa)，并观察二进制文件的“执行”。

```
$ python keykoolol.py keykoolol
[...]
[INFO    ]: xxx___libc_start_main(main=0x730, argc=0x13371acc, ubp_av=0x140000, init=0x23a0, fini=0x2410, rt
ld_fini=0x0, stack_end=0x13fff8) ret addr: 0x88a
[INFO    ]: xxx___printf_chk(flag=0x1, format=0x2424, arg=0x99ccd668) ret addr: 0x76a
[+] Username:
[INFO    ]: xxx_fgets(dest=0x13fbc8, size=0x200, stream=0x71111064) ret addr: 0x77e
aaaaaaa
[INFO    ]: xxx_strcspn(s=0x13fbc8, rejected=0x2433) ret addr: 0x78d
[INFO    ]: xxx___printf_chk(flag=0x1, format=0x2435, arg=0x71111064) ret addr: 0x7a5
[+] Serial:
[INFO    ]: xxx_fgets(dest=0x13fdc8, size=0x200, stream=0x71111064) ret addr: 0x7b9
aaaaaaaaaaa
[INFO    ]: xxx_strcspn(s=0x13fdc8, rejected=0x2433) ret addr: 0x7c8
[INFO    ]: xxx___memcpy_chk(dest=0x203080, src=0x24e0, len=0x400, destlen=0x800) ret addr: 0x9c5
[INFO    ]: xxx_puts(s=0x24a9) ret addr: 0x834
[!] Incorrect serial.
```

不过，要确保具体的执行效果良好，我们就必须通过实例化DSEEngine类来添加DSE。另外，我们要求它存根外部API（类似于Sandbox的方式）。

```
from miasm.analysis.dse import DSEEngine

[...]

dse = DSEEngine(sb.machine)
dse.add_lib_handler(sb.libs, globals())
```

但是，这还不够，因为DSE也需要连接到Jitter上。为此，可以使用对__memcpy_chk的调用将其附加，如下所示。

```
def xxx___memcpy_chk(jitter):
    ret_ad, args = jitter.func_args_systemv(["dest", "src", "len", "destlen"])
    src = jitter.vm.get_mem(args.src, args.len)
    jitter.vm.set_mem(args.dest, src)

    global dse
    dse.attach(jitter)

    return jitter.func_ret_systemv(ret_ad, args.dest)
```

然后，我们在dispatcher上设置一个断点来符号化与VM寄存器对应的内存，我们还创建了2个字典：

1. vm_registers_symb：包含VM寄存器的符号；

2. already_disass：必须确保VM指令已经反汇编，以避免打印未滚动的循环。

```
[...]
DISPATCHER_ADDR = 0xa5d
NEXT_ADDR = 0xa77

vm_registers_symb = {}
already_disass = {}

dse.add_instrumentation(DISPATCHER_ADDR, symbolize_vm)
[...]
```

symbolize_vm回调实现如下，并且与上述策略相对应。唯一的区别是执行aesenc3指令的opcode 30，由于后者目前还没有在Miasm jitter中实现，所以我添加了一个脏补丁来绕过相应的处理程序的执行。

```
from miasm.expression.expression import *

[...]

def symbolize_vm(dse):
    global vm_registers_symb, already_disass

    # update the DSE state (including the memory) from the concrete state
    dse.update_state_from_concrete(mem=True)

    # symbolize the memory corresponding to the VM registers (16 registers of 32 bits at 0x203040)
    for i in range(16):
        vm_registers_symb[ExprMem(ExprInt(0x203040 + i*4, 64), 32)] = ExprId("VM_R{}".format(i), 32)

    # symbolize the VM registers that correpond to real registers
    vm_registers_symb[dse.ir_arch.arch.regs.R9] = ExprId("VM_FLAGS", 64)
    vm_registers_symb[dse.ir_arch.arch.regs.RSI] = ExprId("VM_PC", 64)

    # update the DSE state with the VM registers symbols
    dse.update_state(vm_registers_symb)

    # get the VM state (PC, instruction bytes and opcode)
    vm_pc = int(dse.jitter.cpu.RSI)
    vm_instr = int(dse.jitter.cpu.RCX)
    vm_opcode = int(dse.jitter.cpu.RAX)

    # if the VM instruction was not already disassembled, we print the state and add a breakpoint at NEXT_AD
DR
    if not vm_pc in already_disass or (vm_pc in already_disass and vm_instr != already_disass[vm_pc]):
        print("\n{:x}:".format(vm_pc), end=" ")

        already_disass[vm_pc] = vm_instr

        # VM opcode 0xFF exits the VM
        if vm_opcode == 0xFF:
            print("EXIT")

        # VM opcode 30 executes aesenc instruction but this instruction is not implemented in miasm jitter
        if vm_opcode == 30:
            arg0 = vm_registers_symb[ExprMem(ExprInt(0x203040+(((vm_instr >> 16) & 0xF)*4), 64), 32)]
            arg1 = vm_registers_symb[ExprMem(ExprInt(0x203040+(((vm_instr >> 12) & 0xF)*4), 64), 32)]
            dest = vm_registers_symb[ExprMem(ExprInt(0x203040+(((vm_instr >> 20) & 0xF)*4), 64), 32)]
            print("@128[{} + 0x203080] = AESENC(@128[{} + 0x203080], @128[{} + 0x203080])".format(dest, arg0
, arg1))

        dse.add_instrumentation(NEXT_ADDR, disass_vm_instruction)

    # as we do not want miasm to raise an exception when aesenc is jitted, we jump after the instruction and
 update the DSE state accordingly
    if vm_instr >> 24 == 30:
        dse.jitter.pc = 0x232d
        dse.jitter.cpu.RIP = 0x232d
        dse.update_state({
            dse.ir_arch.arch.regs.RIP: ExprInt(0x232d, 64),
            dse.ir_arch.arch.regs.RAX: ExprInt(vm_pc+4, 64) # update pc
        })

    return True
```

如你所见，如果之前没有看到VM指令，则在下一步添加断点。

回调disass_vm_instruction通过提取dispatcher和下一步之间对DSE状态所做的修改来反汇编VM指令。在Miasm中，这些修改可以在dse.symb.modified中获得。

```
def disass_vm_instruction(dse):
    global vm_registers_symb

    vm_instr = ""

    # get memory modifications
    for dst, src in dse.symb.modified(ids=False):
        # do not print vm registers unchanged
        if dst in vm_registers_symb and src == vm_registers_symb[dst]:
            continue
        vm_instr += "{} = {}\n".format(dst.replace_expr(vm_registers_symb), dse.eval_expr(src))

    # get register modifications
    for dst, src in dse.symb.modified(mems=False):
        # dst = ExprMem(VM_REG)
        if src in vm_registers_symb:
            vm_instr += "{} = {}\n".format(dst, dse.eval_expr(src))
        # VM_REG != VM_REG_ID
        elif dst in vm_registers_symb and src != vm_registers_symb[dst] and vm_registers_symb[dst] != ExprId
("VM_PC", 64):
            vm_instr += "{} = {}\n".format(vm_registers_symb[dst], dse.eval_expr(src))

    # if no modifications then print ZF and VM_PC changes
    if not vm_instr:
        for dst, src in dse.symb.modified(mems=False):
            if dst == dse.ir_arch.arch.regs.zf:
                vm_instr += "ZF = {}\n".format(dse.eval_expr(src))
            elif dst in vm_registers_symb and vm_registers_symb[dst] == ExprId("VM_PC", 64):
                vm_instr += "VM_PC = {}\n".format(dse.eval_expr(src))

    print(vm_instr.strip())

    # remove callback
    del dse.instrumentation[NEXT_ADDR]
```

完整的脚本可以在这里找到。

虚拟机追踪分析

首先，让我们在执行第一条指令之前定义VM寄存器的状态。

如果我们使用DSE和虚拟输入执行脚本，则得到的追踪如下：

```
$ python keykoolol.py keykoolol
[...]
0: VM_R11 = VM_R11 + 0xFFFFFFFF

4: VM_FLAGS = {VM_R11 + 0xFFFFFF01 0 32, 0x0 32 64}

8: ZF = VM_FLAGS[0:32]?(0x0,0x1)
VM_PC = 0x74

74: VM_R0 = 0x0

78: EXIT
Traceback (most recent call last):
[...]
RuntimeError: Symbolic stub 'b'xxx_puts_symb'' not found
```

如你所见，VM反汇编程序运行良好，但是VM很快退出，因为VM_R11的值不同于256。由于VM_R11与序列的长度相对应，则我们可以得出结论，序列长度必须等于256。

另外，由于未实现符号存根xxx_puts_symb，因此引发了异常。由于我们并不真正关心函数输出的内容，因此可以如下这样实现。

```
def xxx_puts_symb(dse):
    raise RuntimeError("Exit")
```

让我们再次执行256个字符的脚本：

```
$ python keykoolol.py keykoolol
[...]
0: VM_R11 = VM_R11 + 0xFFFFFFFF

4: VM_FLAGS = {VM_R11 + 0xFFFFFF01 0 32, 0x0 32 64}

8: ZF = VM_FLAGS[0:32]?(0x0,0x1)
VM_PC = {(VM_PC + 0x4)[0:32] 0 32, 0x0 32 64}

c: VM_R0 = VM_R10

10: VM_R1 = VM_R12

14: @32[0x203880] = VM_PC[0:32] + 0x4

374: VM_R3 = 0x0

378: VM_R2 = VM_R0

37c: VM_R2 = VM_R2 + VM_R3

380: VM_R2 = {@8[{VM_R2 0 32, 0x0 32 64} + 0x203080] 0 8, 0x0 8 32}

384: VM_FLAGS = {VM_R2 0 32, 0x0 32 64}
[...]
```

这一次，执行的指令要多得多。因为它们太多了，如果我把它们全部打印在这里，它将是不可读的，完整的跟踪可以在这里找到。

我不会详细分析每一条VM指令，以下是执行VM字节码的步骤，你可以自己分析。

1. 如果不退出VM，则检查序列的长度是否等于256；

2. 使用十六进制解码序列；

3. 通过用户名计算一个长度为16的自定义“哈希”（确切的算法在下面详述），，并将其复制到已解码的序列之后的内存中。

```
def custom_hash(username):
    hash = [0] * 16
    for i, c in enumerate(username):
        for j in range(16):
            hash[(i+j) % 16] ^= ((((ord(c) + j) * 0xD) ^ 0x25) % 0xFF)
    return hash
```

4. 使用XOR密钥0xF4E3D2C1解密0xC8处的字节码（VM字节码中的偏移）；

5. 如下扩展自定义哈希，以获取长度为96的缓冲区；

```
def expand_custom_hash(custom_hash):
    expanded_buffer = custom_hash + [0] * 80
    for i in range(80):
        expanded_buffer[i+16] = ((expanded_buffer[i] * 3) ^ 0xFF) & 0xFF
    return expanded_buffer
```

6. 使用XOR密钥0xA1B2C3D4解密0x148（VM字节码中的偏移）处的字节码；

7. 将序列分成8个16字节的缓冲区，并在序列的前6个缓冲区上使用aesenc执行32轮AES加密，如下所示。

```
def aesenc(buffer, key):
    # call aesenc instruction

# serial = buf1 + buf2 + ... + buf8
for _ in range(32):
    buf1 = aesenc(buf6, buf1)
    buf6 = aesenc(buf5, buf8)
    buf5 = aesenc(buf4, buf7)
    buf4 = aesenc(buf3, buf4)
    buf3 = aesenc(buf2, buf7)
    buf2 = aesenc(buf1, buf7)
```

8. 使用XOR密钥0xAABBCCDD解密0x334（VM字节码中的偏移量）处的字节码；

9. 如果加密的序列与从用户名计算得出的扩展缓冲区相匹配，则将它们进行比较。

正如你所看到的，VM字节码在其执行期间修补了至少3次。

在十六进制解码后，序列的长度为128字节，但不检查最后两个16字节的缓冲区。

我们已经知道了如何验证序列，现在可以实现自己的keygen来为任何用户名生成序列。

密钥生成

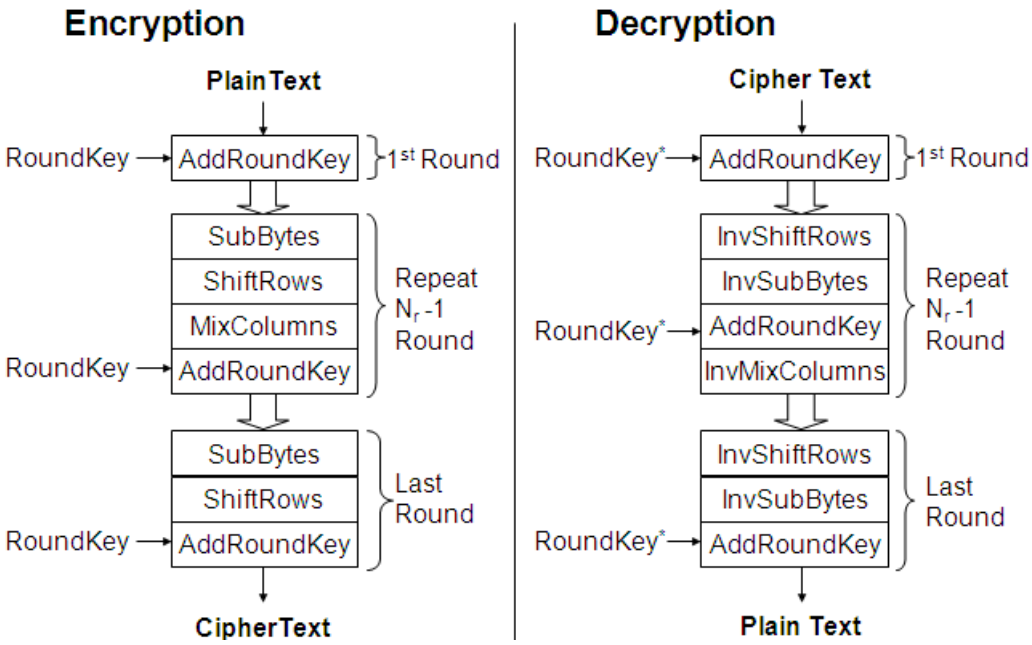
我之所以选择在Rust中实现keygen，除了提高我在使用该语言进行编程的技能外，没有特别的理由。

无论选择哪种语言来实现密钥生成，都必须执行以下步骤：

1. 使用上述算法从用户名生成自定义哈希；

2. 扩展自定义哈希以获取AES加密序列；

3. 在扩展的自定义哈希上执行32轮反向AES加密（我们不能使用aesdec指令，因为一轮解密不同于另一轮加密）来获取序列。



这是用户名admin的一个可能序列，序列的最后2个缓冲区设置为1。

```
$ ./keygen_keykoolol admin 1
b40e0b81eb1d09c017b3c6d9001118a63b6a2377d1e14470531ee487fe9de34b86c949836a5d789baf503680717547b7910facdc11bd
56c22626326ca7053d6ce72e2e638c1d0881c2e699c412485b567128c297e5c7cfa02b6f10b18dbbee140101010101010101010101010101
0101010101010101010101010101010101010101010101
```

我的注册机代码可在此处获得。

拥有函数完备的密钥生成器后，你仍然必须与服务器进行通信以获取标志。

```
$ nc challenges2.challenge-anssi.fr 3000
Give me two valid serials for username: Michael Barnett
>>> afaeff615f362ffbe36eccf4f2e80e6b18404cf0f96398e4881789c0c2b4310a58733ccd2273f48e4983fe171fdfed95d9867c67
742609d24a4dbf6917742c41ba804a642b96c6792e8264454e120e26860480c292ab29537820ada4cb4b8edc0101010101010101010101
0101010101010101010101010101010101010101010101
>>> 8e60f3338de9499d5bd3b9b2ab1371b11d61775ccf9575d47b5f669a04b60be01bf9299819c7f6eee12471fffba41f88d4908548
10aa62c7c23c554d65fbbdecf6eec3ebcb00f4126f09eee7281d694650942ab7e4b33a500343e83ca5d232720000000000000000000000
0000000000000000000000000000000000000000000000000000
Give me two valid serials for username: Shelly Heilman
>>>
```

不幸的是，我很快就意识到我需要自动化与服务器的通信，因为不只有一个用户名来生成的序列。

我选择了pwntools，因为它为此类内容提供了一个简单的界面。

```
$ python get_flag.py
[...]
[DEBUG] Received 0x64 bytes:
    b'Well done! Here is the flag: FCSC{REDACTED}\n'
```

脚本可以在这里找到。