Description

DSA can be hacked if you have access to the size of the random key k.

Flag

flag{25903ADB-15B6-44D7-A027-CAE500675EA5}

Writeup

此题基于19年年底的一个研究发现：https://tpm.fail/

研究人员在TPM（Trusted Platform Module）中发现了漏洞，能够让攻击者利用Timing-information leakage和Lattice attacks获取到存储于TPM中用于ECDSA数字签名算法的私钥。

随后研究人员申请了两个CVE编号：CVE-2019-11090、CVE-2019-16863。

关于攻击的细节部分，可以参考研究人员发表的paper：https://tpm.fail/tpmfail.pdf

在这里简单的讲述一下。

1996年，Boneh和Venkatesan提出了hidden number problem（HNP），并提供了一个基于lattice的多项式算法来解决这个难题。随后，研究者们利用这个思路分析了很多数字签名算法，发现了很多漏洞。

这一次，研究人员研究的是ECDSA数字签名，其密钥生成和签名的流程如下：

ECDSA Key Generation:

1. Randomly choose a private key $d \in \mathbb{Z}_n^*$.
2. Compute the curve point $Q = dP \in \mathcal{E}$.

The private, public key pair is (d, Q).

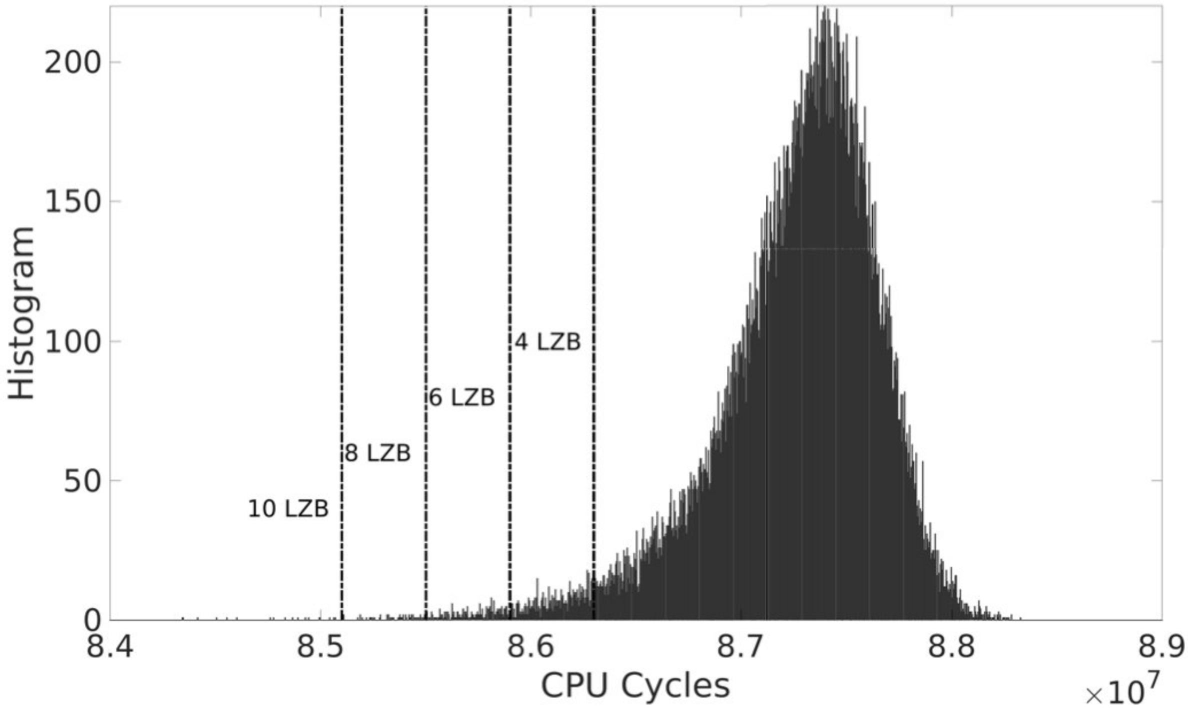
ECDSA Signing: To sign a message $m \in \{0,1\}^*$

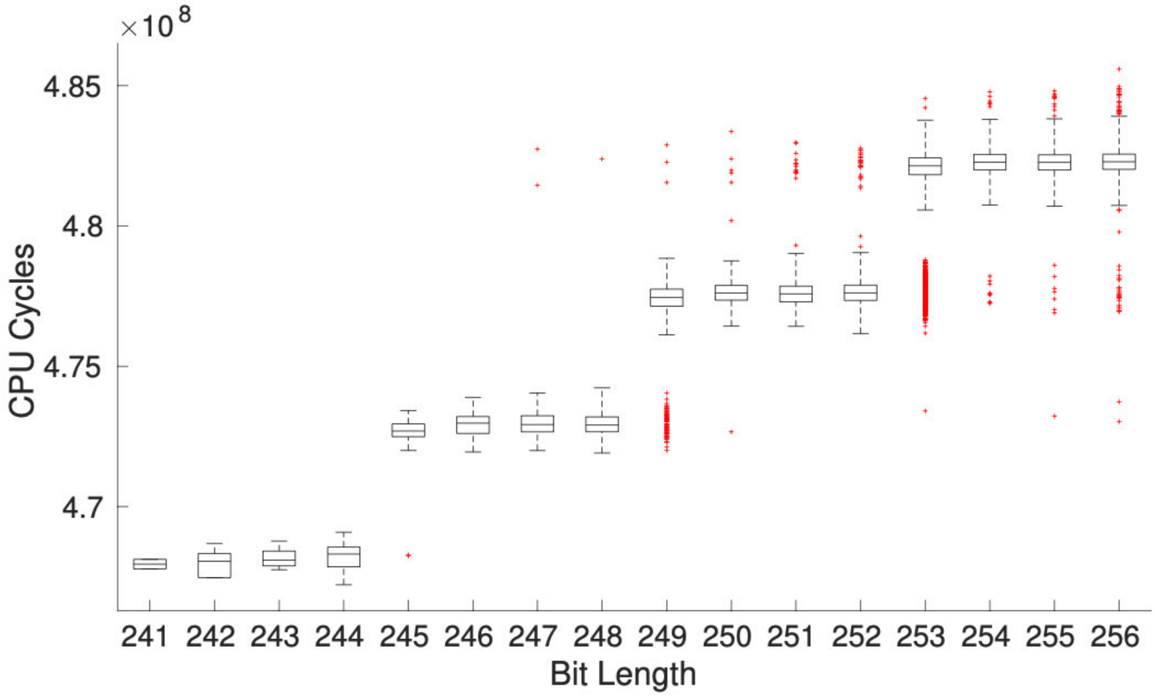
1. Choose a nonce/ephemeral key $k \in \mathbb{Z}_n^*$.
2. Compute the curve point kQ, and compute the x coordinate $r = (kQ)_x$.
3. Compute $s = k^{-1}(H(m) + dr) \bmod n$, where $H(.)$ represents a cryptographic hash function such as SHA-256.

The signature pair is (r, s).

如果我们知道了k的MSB（most significant bits），那么我们就可以利用HNP里的思路来攻击ECDSA签名算法。

研究人员就是利用在签名时会计算$r = (kQ)_x$，其中k的大小会使得这一步耗时不同来攻击的：





k越大，这一步所需的时间就越长；相反，k越小，这一步所需的时间就越短。

因此，可以根据签名的耗时，来判断k的大小。

签名时间越短，就说明k越小，也就是说，k的MSB全都是0。

这就是Timing-imformation leakage。

获取到足够多的由比较小的k（从中可以知道k的MSB有很多0）生成的签名后，我们就可以利用Lattice Attacks来求解出k和私钥d。

我们先获取t组签名(r_i, s_i)。

观察签名中的这一步：

$$s_i = k_i^{-1}(\mathcal{H}(m_i) + dr_i) \pmod n$$

我们将其变形一下：

$$k_i - s_i^{-1} r_i d - s_i^{-1}\mathcal{H}(m_i) \equiv 0 \pmod n$$

其中仅有k_i和私钥x未知。

令

$$A_i = -s_i^{-1} r_i \pmod n \quad B_i = -s_i^{-1} H(m_i) \pmod n$$

进而转化为：

$$k_i + A_i d + B_i = 0 \pmod n$$

可以针对这一个式子来构建lattice。

令K是k_i的一个上限，我们现在考虑由下面这个矩阵所形成的lattice：

$$M = \begin{bmatrix} n & & & & & \\ & n & & & & \\ & & \ddots & & & \\ & & & n & & \\ A_1 & A_2 & \dots & A_t & K/n & \\ B_1 & B_2 & \dots & B_t & & K \end{bmatrix}$$

不难发现向量$v_k = (k_1, k_2, \cdots, k_t, Kx/n, K)$就在这个lattice中，且$v_k$是一个长度相当小的向量。

（这个v_k就是倒数第二行乘上d再加上最后一行，最后再加上n的某个倍数）

因而，我们可以使用LLL算法来找到这个v_k，进而获取到密钥x。

（LLL能够在多项式时间内找到一个长度$\|v\| \leq 2^{(\dim L-1)/4}(\det L)^{1/\dim L}$的向量）

研究人员利用这个思路，先在本地上做了一些测试。

简单地摘录一些测试的结果：

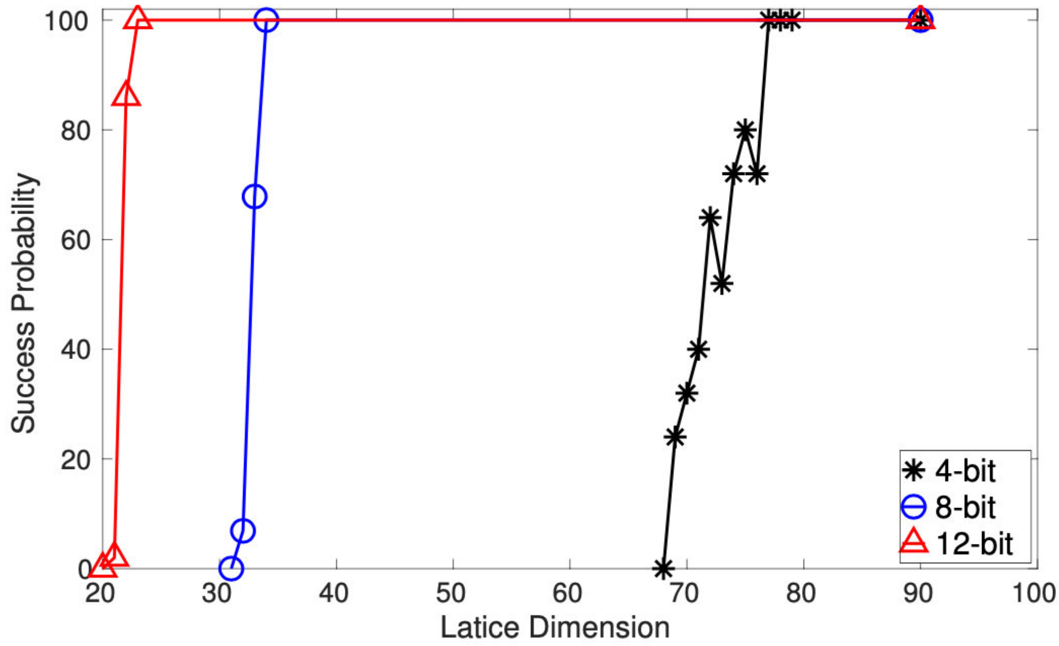


Figure 6: **System Adversary:** Key recovery success probabilities plotted by lattice dimension for 4-, 8-, and 12-bit biases for ECDSA (NIST-256p) with administrator privileges.

首先n是一个256-bit的数，k是区间$[1, n-1]$中随机分布的一个数。

如果k的前4bit都是0，即$k < 2^{256-4} = 2^{252}$，那么这样的k出现的概率大概是$2^{252}/2^{256} = 1/2^4 = 1/16$，研究人员通过计算发现大概选取78组由这样的$k$形成的签名就可以100%利用LLL算法找到$v_k$，因此平均需要获取$16 * 78 = 1248$组签名。

如果k的前8bit都是0时，概率为$1/256$，$t = 35$，需要8784组签名。

随后研究人员在真实世界中进行了分析，研究了一个开源VPN软件服务器，并成功地获取到了相关的密钥。

考虑到CTF比赛的原因，很难通过Timing-imformation leakage来判断出k的大小，因此本题中就直接给出了相应的k的位数。

此外由于ECDSA实现起来过于麻烦，所以本题改用DSA来实现，但原理实际上都是一样的。

本题中q为128位，k是一个在区间$[1, q-1]$中的一个数。

可以不断地与服务器交互，得到若干组签名。设定一个阈值，比如说121，扔去位数比121大的k，保留位数小于等于121的k。直到获取到t组这样的签名。

然后就可以开始Lattice Attacks：

通过

$$r = g^k \pmod q$$
$$s = k^{-1}(\mathcal{H}(m) + xr) \pmod q$$

可以推得

$$k_i = s_i^{-1} r_i \cdot x + s_i^{-1}\mathcal{H}(m_i) \pmod q$$
$$k_i = A_i x + B_i \pmod q$$
$$k_i = A_i x + B_i + l_i q$$

其中，$A_i = s_i^{-1} r$，$\quad B_i = s_i^{-1}\mathcal{H}(m)$

构建lattice：

$$M=\begin{bmatrix} q & & & & & \\ & q & & & & \\ & & \ddots & & & \\ & & & q & & \\ A_1 & A_2 & \dots & A_t & K/q & \\ B_1 & B_2 & \dots & B_t & & K \end{bmatrix}$$

（其中K是k的上界，例如k的位数小于等于121时，那么$K=2^{122}$）

不难发现，存在一个M的线性组合v，可以得到我们想要的v_k。

$$vM=[l_1 \quad l_2 \quad \cdots \quad l_t \quad x \quad 1]\begin{bmatrix} q & & & & & \\ & q & & & & \\ & & \ddots & & & \\ & & & q & & \\ A_1 & A_2 & \dots & A_t & K/q & \\ B_1 & B_2 & \dots & B_t & & K \end{bmatrix}=[k_1 \quad k_2 \quad \cdots \quad k_t \quad Kx/q \quad K]=v_k$$

因此v_k即为M上的一个格点，且长度很短，可以用LLL算法求出。

我们可以大致估量一下t和阀值的取值范围：

Lattice的determinant为：

$$\det L=q^tK/qK=q^{t-1}K^2$$

LLL算法可以找到这样一个向量：

$$\|v\|<2^{(\dim L-1)/4}(detL)^{1/dimL}=2^{(t+1)/4}q^{\frac{t-1}{t+2}}K^{\frac{2}{t+2}}$$

而v_k的长度为：

$$\|v_k\|=(k_1k_2\cdots k_tKx/qK)^{1/(t+2)}$$

因此只需要

$$\|v_k\|<\|v\|\Leftrightarrow(k_1k_2\cdots k_tKx/qK)^{1/(t+2)}<2^{(t+1)/4}q^{\frac{t-1}{t+2}}K^{\frac{2}{t+2}}$$

后面的不太好算。。

但是可以本地自己测试一下：

```
# sage 8.9

# Keygen
q = next_prime(2^128)
while True:
    s = ZZ.random_element(2^(1024 - 129))
    p = (s * 2 * q + 1)
    if p.is_prime():
        break

Zq = Zmod(q)
g = ZZ(pow(2, (p-1) // q, p))
x = ZZ(Zq.random_element())
# print x
y = ZZ(pow(g, x, p))

```

```
# Test
t = 34

yes = 0
for time in range(100):
    A = []
    B = []
    ks = []

    for i in range(0, t):
        Hm = ZZ(Zq.random_element())
        k = ZZ(Zmod(2^122).random_element())
        ks.append(k)
        r = ZZ(ZZ(pow(g, k, p)) % q)
        s = ZZ(inverse_mod(k, q) * (Hm + x*r) % q)
        # print (r, s)
        A.append(ZZ((inverse_mod(s, q) * r) % q))
        B.append(ZZ((inverse_mod(s, q) * Hm) % q))

    K = 2^122
    X = q * identity_matrix(QQ, t) # t * t
    Z = matrix(QQ, [0] * t + [K/q] + [0]).transpose() # t+1 column
    Z2 = matrix(QQ, [0] * (t+1) + [K]).transpose()    # t+2 column

    Y = block_matrix([[X],[matrix(QQ, A)], [matrix(QQ, B)]]) # (t+2) * t
    Y = block_matrix([[Y, Z, Z2]]) # (t+2) * (t+2)

    Y = Y.LLL()

    if abs(ZZ(Y[1, 0])) == ZZ(ks[0]):
        yes += 1

print yes, yes/100
```

在$t \geq 34$的时候，成功率就是100%。

解释下倒数第四行为什么取LLL后的第二行。因为有另一个短向量$v = (0, 0, \cdots, K, 0)$也在lattice上，且这个短向量比$(k_1, k_2, \cdots, k_t, Kx/n, K)$还要短。此外，多次测试发现，$v_k$总会出现在LLL后的第二行。

一些测试的数据：

MSB	K	成功率	成功率	成功率	需要的交互次数
5bit	2^123	55% (t=40)	93% (t=50)	100% (t=65)	2080
6bit	2^122	76% (t=29)	99% (t=32)	100% (t=34)	2176
7bit	2^121	17% (t=22)	68% (t=23)	99.5% (t=25)	3200

这里，我们选择6bit的MSB和$t = 40$组签名。

exp.py如下：

```
import re
import json
import string
import subprocess
from random import sample
from hashlib import sha256

from Crypto.Util.number import inverse
```

```python
from pwn import *


host, port = ('127.0.0.1', 10000)
r = remote(host, port)
# context.log_level = 'debug'


# Proof of Work
rec = r.recvline().decode()

suffix = re.findall(r'\+ ([0-9a-f]*?)\)', rec)[0]
digest = re.findall(r'== ([0-9a-f]*?)\n', rec)[0]
print(f"suffix: {suffix} \ndigest: {digest}")

for i in range(256**3):
    guess = i.to_bytes(3, 'big') + bytes.fromhex(suffix)
    if sha256(guess).hexdigest() == digest:
        print('[!] Find: ' + guess.hex())
        break
else:
    print('Not found...')

r.sendlineafter(b'Give me XXX in hex: ', guess[:3].hex().encode())

# DSA params
params = r.recvuntil(b'3. exit\n').decode()
p = int(re.findall(r'p = ([0-9]*?)\n', params)[0])
q = int(re.findall(r'q = ([0-9]*?)\n', params)[0])
g = int(re.findall(r'g = ([0-9]*?)\n', params)[0])
y = int(re.findall(r'y = ([0-9]*?)\n', params)[0])
print(f"p: {p}\nq: {q}\ng: {g}\ny: {y}")


# Interactive
Hm_s = []
r_s = []
s_s = []

s = string.ascii_letters + string.digits
cnt = 0
total = 0
while cnt < 40:
    total += 1
    name = ''.join(random.sample(s, 10)).encode()
    r.sendlineafter(b"$ ", b"1")
    r.sendlineafter(b"Please input your username: ", name)

    rec = r.recvuntil(b"3. exit\n").decode()
    k_bits = int(re.findall(r"== ([0-9]*?)\n", rec)[0])
    if k_bits < 122:
        cnt += 1

        data = re.findall(r"in hex: ([0-9A-Z]*?)\n", rec)[0]
        sig = bytes.fromhex(data)
        (name, sig_r, sig_s) = (sig[:-40], sig[-40:-20], sig[-20:])
        (sig_r, sig_s) = map(lambda x: int.from_bytes(x, 'big'), (sig_r, sig_s))

        print(f"\ncount: {cnt}\nk_bits: {k_bits}")
        print(f"sig_r: {sig_r}\nsig_s: {sig_s}")

        Hm = int.from_bytes(sha256(name).digest(), 'big')
        Hm_s.append(Hm)
```

```python
        r_s.append(sig_r)
        s_s.append(sig_s)

print(f"\nTotal times: {total}")

# save data
f = open('data', 'w')
json.dump([q, Hm_s, r_s, s_s], f)
f.close()

# solve HNP
print("\nSolving HNP...")
cmd = "sage solver.sage"
try:
    res = subprocess.check_output(cmd.split(' '))
except:
    print("Can't find x...")
    exit(1)
x = int(res)

# check
assert(y == pow(g, x, p))
print(f"find x: {x}")

# forge signature
admin = b"admin"
Hm = int.from_bytes(sha256(admin).digest(), 'big')
k = 0xdeadbeef
k_inv = inverse(k, q)
sig_r = pow(g, k, p) % q
sig_s = (k_inv * (Hm + x*sig_r)) % q

# sign in
sig = admin + sig_r.to_bytes(20, 'big') + sig_s.to_bytes(20, 'big')
print(f"Sending signature: {sig.hex().upper()}")
r.sendlineafter(b'$ ', b"2")
r.sendlineafter(b'Please send me your signature: ', sig.hex().upper().encode())


r.interactive()
```

其中用来求解私钥x的solver.sage如下：

```python
# sage 8.9
import json

t = 40

# Load data
f = open("data", "r")
(q, Hm_s, r_s, s_s) = json.load(f)


# Calculate A & B
A = []
B = []
for r, s, Hm in zip(r_s, s_s, Hm_s):
    A.append( ZZ( (inverse_mod(s, q)*r) % q ) )
    B.append( ZZ( (inverse_mod(s, q)*Hm) % q ) )


```

```
# Construct Lattice
K = 2^122   # ki < 2^122
X = q * identity_matrix(QQ, t) # t * t
Z = matrix(QQ, [0] * t + [K/q] + [0]).transpose() # t+1 column
Z2 = matrix(QQ, [0] * (t+1) + [K]).transpose()    # t+2 column

Y = block_matrix([[X],[matrix(QQ, A)], [matrix(QQ, B)]]) # (t+2) * t
Y = block_matrix([[Y, Z, Z2]])

# Find short vector
Y = Y.LLL()

# check
k0 = ZZ(Y[1, 0] % q)
x = ZZ(Y[1, -2] / (K/q) % q)
assert(k0 == (A[0]*x + B[0]) % q)
print x
```

考虑到网络延迟，可能需要几分钟的交互时间。

不过选手可以选择在本地上搭建环境，先本地跑好exp，再选择远程交互。

最终可以得到flag：flag{25903ADB-15B6-44D7-A027-CAE500675EA5}