Waiting List

by lanthan / ssh@uzl

Rating: 5.0

Add your writeup

The `verify` function does not really havy anything in common with the original verify function of ECDSA (you need the private key for verification o.O).

The `sign` function receives a message m as input and hashes it into value h. Then, it chooses a random k, computes $r = g^k \bmod n$ and $s = k^{-1}(h + d \cdot r) \bmod n$ as well as the seven least significant bits lsb of k. We are given 200 tuples of the form (m, h, r, s, lsb). The task is to find the private key d so we can sign the message `william;yarmouth;22-11-2021;09:00` and send it to the server. If the signature verifies correctly, the server provides us with the flag.

The security of ECDSA completely relies on k as well as d being chosen perfectly random. If an attacker is able to predict only a single bit of k with a probability bigger than 50% they can break ECDSA. We refer the interested reader to the *Ladderleak* paper for more information.

In our case we are given seven bits of the secret k per signature. So breaking ECDSA is possible for sure. Our attacker scenario (we know LSBs) and how to solve it using lattice attacks is described in the paper Recovering cryptographic keys from partial information, by example.

We are given 200 message signature pairs so we have 200 equations of the form

$$s_i = k_i^{-1}(h_i + dr_i) \mod n$$

with $0 \leq i \leq 199$.

Let's take the first equation and transform it.

$$\begin{aligned}
s_0 &= k_0^{-1}(h_0 + dr_0) \mod n && | \cdot k_0 s_0^{-1} \\
k_0 &= s_0^{-1}(h_0 + dr_0) \mod n && | - s_0^{-1}(h_0 + dr_0) \\
0 &= k_0 - s_0^{-1}(h_0 + dr_0) \mod n \\
0 &= k_0 - s_0^{-1} h_0 - r_0 s_0^{-1} d \mod n && | \cdot r_0^{-1} r_i s_0 s_i^{-1} \\
0 &= r_0^{-1} r_i s_0 s_i^{-1} k_0 - r_0^{-1} r_i s_i^{-1} h_0 - r_i s_i^{-1} d \mod n
\end{aligned}$$

Now we substract the first equation from all remaining 199 equations ($1 \leq i \leq 199$):

$$0 = k_i - s_i^{-1} h_i - r_i s_i^{-1} d - (r_0^{-1} r_i s_0 s_i^{-1} k_0 - r_0^{-1} r_i s_i^{-1} h_0 - r_i s_i^{-1} d) \mod n$$

$$0 = k_i - s_i^{-1} h_i - r_i s_i^{-1} d - r_0^{-1} r_i s_0 s_i^{-1} k_0 + r_0^{-1} r_i s_i^{-1} h_0 + r_i s_i^{-1} d \mod n$$

By doing so we get rid of the unknown private key d.

$$0 = k_i - s_i^{-1} h_i - r_0^{-1} r_i s_0 s_i^{-1} k_0 + r_0^{-1} r_i s_i^{-1} h_0 \mod n$$

We reorder the summands such that summands with unknown values k are written first.

$$0 = k_i - r_0^{-1} r_i s_0 s_i^{-1} k_0 - s_i^{-1} h_i + r_0^{-1} r_i s_i^{-1} h_0 \mod n$$

By getting rid of d, the only remaining unknons are the values $k_0, \ldots, k_{199}$. However, we also only have 199 equations left. Simply solving for the unknowns is therefore not possible.

Here, the additional information about the LSBs comes into play. Let's write every k_i as a number $k_i = 2^7 b_i + a_i$ of which we know the value a_i (this is the last seven bits). The b_i remain unknowns. We rearrange the equations further.

$$0 = (2^7 b_i + a_i) - s_i^{-1} h_i - r_0^{-1} r_i s_0 s_i^{-1} (2^7 b_0 + a_0) + r_0^{-1} r_i s_i^{-1} h_0 \mod n$$

$$0 = 2^7 b_i + a_i - s_i^{-1} h_i - 2^7 r_0^{-1} r_i s_0 s_i^{-1} b_0 - r_0^{-1} r_i s_0 s_i^{-1} a_0 + r_0^{-1} r_i s_i^{-1} h_0 \mod n$$

Again, we write the summands with unknowns first.

$$0 = 2^7 (b_i - r_0^{-1} r_i s_0 s_i^{-1} b_0) + a_i - s_i^{-1} h_i - r_0^{-1} r_i s_0 s_i^{-1} a_0 + r_0^{-1} r_i s_i^{-1} h_0 \mod n \quad | \cdot 2^{-7}$$

$$0 = b_i - r_0^{-1} r_i s_0 s_i^{-1} b_0 + 2^{-7} (a_i - s_i^{-1} h_i - r_0^{-1} r_i s_0 s_i^{-1} a_0 + r_0^{-1} r_i s_i^{-1} h_0) \mod n$$

We move everything except the b_i to the other site...

$$b_i = r_0^{-1} r_i s_0 s_i^{-1} b_0 - 2^{-7} (a_i - s_i^{-1} h_i - r_0^{-1} r_i s_0 s_i^{-1} a_0 + r_0^{-1} r_i s_i^{-1} h_0) \mod n$$

... and bracket common terms.

$$b_i = r_0^{-1} r_i s_0 s_i^{-1} b_0 - 2^{-7} (a_i - s_i^{-1} (h_i + r_0^{-1} r_i (s_0 a_0 - h_0))) \mod n$$

To ease the writing, we define

$A_i = r_0^{-1} r_i s_0 s_i^{-1}$ and $B_i = 2^{-7} (a_i - s_i^{-1} (h_i + r_0^{-1} r_i (s_0 a_0 - h_0)))$.

So our equations are

$$b_i = A_i b_0 + B_i \mod n.$$

But there are still 200 unknowns but only 199 equations. The paper by De Micheli and Henninger tells us that we can construct a lattice from these equations. A basis of this lattice is represented by the following (201×201) matrix:

```
    | n                                  |
    |       n                            |
    |             ...                    |
L = |                   n                |
    | A_1   A_2   ...   A_199   1        |
    | B_1   B_2   ...   B_199       K    |
```

Basicall, L is a matrix with n on its diagonal. Additionally, the vectors $(A_1, A_2, \cdots, A_{199}, 1, 0)$ and $(B_1, B_2, \cdots, B_{199}, 0, K)$ form the second last and last row respectively. The collumns of L form the vectors of a basis. It is important that the vector $Y = (b_1, \ldots, b_{199}, b_0, K)$ is a point on the lattice. That is the case here because $X \cdot L = Y$ with $X = (x_1, x_2, \ldots, x_{199}, b_0, 1)$ for some integer values x_i that hide themselves behind the " $\mod n$".

The value K is chosen as the smalles value that is still guaranteed to be bigger than each b_i. We know that the b_i are smaller than n since everything is computes modulo n. For this challenge, n is a 256 bit integer. We further know that the b_i are integers with at most $256 - 7 = 249$ bits since they are still smaller than n after multiplying them by 2^7 and adding a_i. This is why we choose $K = 2^{256-7+1} = 2^{250}$ and know that $K > b_i$ holds for all $0 \leq i \leq 199$.

Each entry of a vector of the lattice L is expected to be of similar bit size as n. With all entries being roughly the same size, this means that most vectors are about the same length. On the other hand, we know that vector Y is part of the lattice. But since all entries of Y are bounded by K, which is much smaller than n, this means that Y is way shorter than most vectors in the lattice. This is something we can exploit!

The LLL algorithm, similar to the Gram-Schmidt process for vector spaces, finds a "good" basis. A basis is considered to be "good" if its vectors are as short and pairwise as orthogonal as possible. Since our vector Y is very short, it is part of a "good" basis with high probability. Actually, chances are high that Y is the shortest vector in the lattice.

We can find the basis vector with the unknown b_i by searching for the value K in the last entry. If the LLL algorithm finds a basis with a vector where the last entry is K, we get b_0 from the second last entry. Together with a_0, which we are given, we can then compute $k_0 = 2^7 b_0 + a_0$. This then allows us to solve for $d = (s_0 \cdot k_0 - h_0) \cdot r_0^{-1} \bmod n$. We can verify that d is actually correct by verifying some of the remaining given signatures.

If the secret key d is correct, we can go ahead and sign the given message and send it to the server, which will provide us with the flag.

We implemented our solution in Sagemath. Sagemath supports modular arithmetics and implements the LLL algorithm. So the only steps left for us is to enter the matrix L correctly and craft a valid signature. Our code is given at the end. You can run it with `ncat`. Here is an example output:

```
$ ncat -c "sage solve.sage" IP PORT
[+] Computed 199 coefficients.
[+] Constructing lattice...
[+] Performing base reduction...
[+] Key candidate found!
[+] Candidate verifies correctly!
[+] Crafting signature...
[+] Getting flag...
Welcome to the SteamShake transplant clinic where our mission is to deliver the most vintage an
d high tech arms worldwide.
Please use your signature to verify and confirm your appointment.
Estimated waiting for next appointment: 14 months
> {"pt": "william;yarmouth;22-11-2021;09:00", "r": "92da2be8475d17bb8d0a79e271c59dd55b392de7c85
d6127748fd56bf392bb2f", "s": "488625aaaad30d5b9af3cd4924d49558d8d8928eb62a62a1013ca5aa39a23ae
7"}
Your appointment has been confirmed, congratulations!
Here is your flag: HTB{t3ll_m3_y0ur_s3cr37_w17h0u7_t3ll1n9_m3_y0ur_s3cr37_15bf7w}
```

Sagemath code:

```
import sys
import csv
import json
import challenge

DEBUG=True
```

```
def dbg(line, force=False, end='\n'):
    if DEBUG or force:
        print(line, flush=True, file=sys.stderr, end=end)
        pass
    pass

def prnt(line, force=False):
    dbg(line, force=force)
    print(line, flush=True)
    pass

def inp(lines=1, chars=0, force=False):
    line = ''
    for _ in range(lines):
        line = input()
        dbg(line, force=force)
    if chars > 0:
        dbg(sys.stdin.read(chars), force=force, end='')
    return line

def hash_msg(msg, n):
    from hashlib import sha1
    h = sha1(msg).digest()
    h = int.from_bytes(h, 'big')
    h = bin(h)[2:]
    h = int(h[:len(bin(n)[2:])], 2)
    return h

def inverse(x, n):
    return int(pow(x, -1, n))

def ppow(x, e, n):
    return int(pow(x, e, n))

def sign(dsa, msg):
    from random import randint
    h = hash_msg(msg, dsa.n)
    k = randint(1, dsa.n-1)
    r = pow(dsa.g, k, dsa.n)
    s = (pow(k, -1, dsa.n) * (h + dsa.key * r)) % dsa.n
    lsb = k % (2 ** 7)
    return {"h": hex(h)[2:], "r": hex(r)[2:], "s": hex(s)[2:], "lsb": bin(lsb)[2:] }

def verify_h(dsa, h, r, s):
    c = inverse(s, dsa.n)
    k = (c * (h + dsa.key * r)) % dsa.n
    return r == pow(dsa.g, k, dsa.n)

def verify_m(dsa, m, r, s):
    h = hash_msg(m, dsa.n)
    return verify_h(dsa, h, r, s)


if __name__ == '__main__':
```

```
dsa = challenge.ECDSA()
msg = b'william;yarmouth;22-11-2021;09:00'
sig_r = 1
sig_s = 0
known_bits = 7

mm = list()
hrsa = list()
# Read provided information
with open('appointments.txt', 'r') as csvf:
    csvr = csv.reader(csvf)
    header = next(csvr)
    for row in csvr:
        mm.append(row[0].encode('utf8'))
        pass
    pass
with open('signatures.txt', 'r') as csvf:
    csvr = csv.reader(csvf, delimiter=';')
    header = next(csvr)
    for h, r, s, a in csvr:
        hrsa.append( (int(h,16), int(r,16), int(s,16), int(a,2)) )
        pass
    pass

# Compute coeffs
bigB = 2**(len(bin(dsa.n)[2:]) - known_bits + 1) # Upper bound for all bi
AA = list()
BB = list()
Fn = GF(dsa.n)
h0, r0, s0, a0 = [Fn(x) for x in hrsa[0]]
for i, (hi, ri, si, ai) in enumerate(hrsa):
    if i == 0:
        continue
    hi, ri, si, ai = Fn(hi), Fn(ri), Fn(si), Fn(ai)
    Ai = ri * si^-1 * r0^-1 * s0
    Bi = -1 * Fn(2)^-known_bits * (ai - si^-1 * (hi + r0^-1 * ri * (s0 * a0 - h0)))
    AA.append(Ai.lift())
    BB.append(Bi.lift())
    pass
dbg(f"[+] Computed {len(AA)} coefficients.\n[+] Constructing lattice...")

# Construct lattice
size = len(AA)
MM = Matrix(ZZ, size + 2)
for ii in range(size):
    MM[ii, ii] = dsa.n
    MM[-2, ii] = AA[ii]
    MM[-1, ii] = BB[ii]
    pass
MM[-2,-2] = 1
MM[-1,-1] = bigB

# Perform LLL
dbg("[+] Performing base reduction...")
```

```
MB = MM.LLL()                # LLL sorts the basis vectors by size
if Fn(MB[0, -1]) == Fn(bigB): # so we only check the first basis vector
    k0 = Fn(2)**known_bits * MB[0, -2] + a0
    dsa.key = ((s0 * k0 - h0) * r0^-1).lift()
    dbg('[+] Key candidate found!')
    pass
else:
    dbg('[!] No candidate found :(')
    sys.exit()

# Verify a signature to know that the key candidate is correct
m1 = mm[1]
_, r1, s1, a1 = hrsa[1]
if verify_m(dsa, m1, r1, s1):
    wohoo = True
    dbg(f'[+] Candidate verifies correctly!')
    pass
else:
    dbg("[!] Key candidate was not the key :(")
    sys.exit()

# Craft signature for message
dbg(f"[+] Crafting signature...")
dic = sign(dsa, msg)

# Print flag
dbg(f"[+] Getting flag...\n")
msg2server = json.dumps({'pt': msg.decode('ascii'), 'r': dic['r'], 's': dic['s']})
inp(3, 2)
prnt(msg2server)
try:
    inp(2)
    pass
except:
    pass
pass
```

Comments

Comment

Send