Lab report #2

CAA - EdDSA attacks

Alexis Martins

November 21, 2023

Contents

1 Challenge 1

1.1 Implementation mistakes

There are two bad implementations in the first code. The first mistake is in the `sign` function when we calculate S. This is the calculation done by the program.

$$S = r + r \cdot h \mod l$$

According to the RFC 8032, this should be the code normally executed to calculate S.

$$S = r + s \cdot h \mod l \, with \, s = H_{msb}(k)$$

The second difference with the recommended implementation is in the `verify` function. This time, when we calculate `rhs` the provided code does it this way.

$$rhs = R + R \cdot h$$

Once again according to the previous RFC, the correct implementation should be the following.

$$rhs = R + A \cdot h$$

1.2 Signature forging

The forging is pretty easy. The signature is a 64 byte array, we just have to pass a 64 byte array full of `0`.

1.3 Attack explanation

To explain the attack, I will describe how does the sign and verify function work. This is how the signature works on the provided code.

$$\begin{aligned} r &= H(H_{lsb}(k)||M) \mod l \\ R &= r \cdot B \\ h &= H(R||A||M) \mod l \\ S &= r + r \cdot h \end{aligned}$$

Note : k is the private key, A is the public key, B a point on the EC and H a hashing function.

The signature returned to the user is $R||S$. The verification works this way.

$$
\begin{aligned}
h &= H(R||A||M) \mod l \\
rhs &= R + R \cdot h \\
lhs &= B \cdot S
\end{aligned}
$$

Then the program will compare `lhs` and `rhs`, if they are equal then the signature is valid. Now to illustrate the vulnerability, we should rewrite the equality.

$$rhs = lhs$$

$$R + R \cdot h = S \cdot B$$

As an attacker, we know the message to verify, the public key and the signature matching the message. The signature is what we should focus on, I noticed with the rewritten version of the equality it was possible to pass the test for all the cases.

If `R` and `S` are equal to `0`, then the test will **always** be **`true`**. Let's replace the value in the equation.

$$R + R \cdot h = S \cdot B$$

$$0 + 0 \cdot h = 0 \cdot B$$

$$0 = 0$$

In practice, we just have to pass a `bytearray(64)` as a signature, which is going to be a byte array full of `0`.

```python
def attack():
    ed = Ed25519()
    pub = base64.b64decode(b'QeSCHHMAr7w2wp+t49jHK7v19btFu42CGfdcClwKlKg=')
    msg = b"Grade of Alexandre Duc at CAA = 6.0"
    print(ed.verify(pub, msg, bytearray(64))) # ALWAYS True
```

2 Challenge 2

2.1 Implementation mistakes

The developer forgot to `hash` the value `k` before the concatenation with the M when calculating `r`.

$$r = H_{msb}(k||M)$$

Normally, this value should be calculated by hashing the value `k` before as follows :

$$r = H(H_{lsb}(k)||M)$$

2.2 Signature forging

See the code provided with this report. The value of the signature was too long for page width… Signature should be commented at the end of the file.

2.3 Attack explanation

We should first write down how does the signature work to have a clear view on the vulnerability

$$\begin{aligned} r &= H_{msb}(k||M) \mod l \\ R &= r \cdot B \\ s &= H_{msb}(k) \\ h &= H(R||A||M) \mod l \\ S &= (r + h \cdot s) \mod l \end{aligned}$$

I noticed this implementation was special when the message was **empty**, for this special case `r` and `s` have the same value. This is true because, the hashing of `k` was missing in the calculation of `r`.

$$r = s = H_{msb}(k)$$

For the following explanation, we assume the message is empty and won't be displayed in the equations. It's possible to recover the value of `r`and `s`, this will allow us to sign any message. Let's start by writing what are `R`and `S`. It's the entry point for this attack. For the equation of `S` we can replace the `s` by `r`, because they have the same value.

$$\begin{aligned} R &= r \cdot B \\ S &= r + h \cdot r = r \cdot (1 + h) \end{aligned}$$

To recover r (and s), we can manipulate the second equation. We know all the variables, except r.

$$r = s = \frac{S}{1+h} \mod l$$

Below, the code that retrieves these values before the call of the sign function.

```
def attack():
    # Define Edwards Curve
    ed = Ed25519()

    # Two constants of the system and the message
    l = hexi("1000000000000000000000000000000014
       def9dea2f79cd65812631a5cf5d3ed")
    b = 256

    # Provided pubkey
    pub = b64decode(b'T0vUg8CHzYIJupRYQEMWQeLy6bgEkJYJngFUpwbTg1w=')

    # Signature of an empty message
    empty_sign = b64decode(b'T0vUg8CHzYIJu...yJQnSeVTOBA==')

    new_r = ((int.from_bytes(empty_sign[32:], byteorder="little")) *
              pow(1 + (int.from_bytes(hashlib.sha512(empty_sign[:32] + pub)
              .digest(), byteorder="little") % l), -1, l) % l)
    print(new_r)
    forged_sign = signWithr(pub, flag, new_r, l, b)
    print("Signature forged :", ed.verify(pub, flag, forged_sign))
    print("Signature value :", b64encode(forged_sign))
```

Now that we have this value, we can calculate a signature for all the messages. I recoded the signature function to take the value of r as an argument instead of calculating it.

```
def signWithr(pubkey, msg, r, l, b):
    R = (Edwards25519Point.stdbase() * r).encode()
    # Calculate h.
    s = r
    h = int.from_bytes(hashlib.sha512(R + pubkey + msg).digest(), byteorder=
       "little") % l
    # Calculate s.
    S = to_bytes(((r + h * s) % l), b // 8, byteorder="little")
    # The final signature is a concatenation of R and S.
    return R + S
```

3 Challenge 3

3.1 Implementation mistakes

The problem with this implementation comes from the wrong usage of the date. It's totally possible to have a correct implementation of the date in EdDSA, but the developer this time didn't follow the CAA course.

It's possible to recover the parameters `r` and `s` using two signatures. This is because in the `S` part of the signature, the `r` will be equal, but the `h * s` will be different. Thus, we can recover `s`, then `r` using a simple substraction of signatures.

3.2 Signature forging

Like the second challenge, the signature was too long to print in this report. Check the corresponding code, you can execute it or just copy/paste from the comment.

3.3 Attack explanation

The first step to understand this attack is to write how does the signature work. And then to give two examples for a same message.

$$
\begin{aligned}
r &= H(H_{lsb}(k)||M) \\
s &= H_{msb}(k) \\
R &= r \cdot B \\
S &= r + H(R||A||M||date) \cdot s \mod l
\end{aligned}
$$

$R||S$ is the signature for the message M and the date $date$. Now we can sign two times the same message, but with different dates.

$$
\begin{aligned}
r &= H(H_{lsb}(k)||M) \\
s &= H_{msb}(k) \\
R &= r \cdot B \\
S_1 &= r + H(R||A||M||date_1) \cdot s \mod l
\end{aligned}
\qquad
\begin{aligned}
r &= H(H_{lsb}(k)||M) \\
s &= H_{msb}(k) \\
R &= r \cdot B \\
S_2 &= r + H(R||A||M||date_2) \cdot s \mod l
\end{aligned}
$$

These two signatures are pretty similar except for `S` where the `date` changes and so does `h = H(...)`. Noticing that, I had the idea to substract these signatures to isolate `s`.

$$
\begin{aligned}
S_1 - S_2 &= (r + h_1 \cdot s) - (r + h_2 \cdot s) \mod l \\
S_1 - S_2 &= (h_1 - h_2) \cdot s \mod l \\
s &= \frac{S_1 - S_2}{h_1 - h_2} \mod l
\end{aligned}
$$

Now that we have `s`, it's possible to also recover `r` using one of the signatures we have.

$$
\begin{aligned}
S_1 &= r + h_1 \cdot s \mod l \\
r &= S_1 - h_1 \cdot s \mod l
\end{aligned}
$$

Finally, to end this attack, we have to do something similar to the previous challenge. We will manually sign our forged message. We will use a "target" message which will be the message we want to create a signature for and we will reuse the date of the first signature.

$$
\begin{aligned}
R_{forged} &= r \cdot B \\
h &= H(R||A||M_{target}||date_1) \mod l \\
S_{forged} &= r + h \cdot s \mod l
\end{aligned}
$$

The valid signature we create is the concatenation of `R`and `S`.

3.4 Fixing the mistakes

To fix this mistake, I added the `date` to the hash made to calculate `r`.

$$
r = H(H_{lsb}(k)||M) \mod l \rightarrow r = H(H_{lsb}(k)||M||date) \mod l
$$

This modification totally prevents the previous exploit, because now `r` is also different for two similar messages. If we do a subtraction, `r` won't disappear, and it won't allow us to isolate `s`. Otherwise, I saw it was possible to use context strings with `ed25519ctx`. We could potentially remove the date from the calculus and use it as a context string. This method has various advantages, it uses the standard implementation of EdDSA, and it authenticates the date. The only problem arises from the context strings, which are not intended for this exact purpose according to the RFC.

Note : At the end of the day, don't roll your own crypto.