Lab2 - ICR

Titus Abele

Chall 1

In the correct EdDSA signing process, s should be calculated as $$((r + h * a) % self.l)$$ where a is the private key. However, in the provided code, s is calculated as $$((r + h * r) % self.l)$$ This introduces a vulnerability because a is not included in the calculation of s, which means the private key is not being utilized as it should be.

To exploit this vulnerability and forge a signature for a new message we can simply extract all known values from the given parameters and create our own signature completely bypassing the need for the signer's private key.

The exploit was defined in the method `exploit()` in the `chall1.py` file. The method relies on following step-by-step pseudo code:

1. Obtain a valid signature (R, S) for a known message.
2. Calculate the value of h for the known message.
3. Calculate the value of s for the known message using the provided incorrect calculation.
4. Forge a new signature for the new message using the calculated s and the original R.

The output of the code:

```
msg_known = b'Grade of Alexandre Duc at ICR = 6.0'
sig_known = b'eHM1...KCQ=='
True
msg_new = b'My grade in ICR is 6.0'
sig_new = b'HvYY...sBA=='
True
```

Chall 2

While looking at the new implementation, we discovered a vulnerability in the `sign()` function on line 360:

```
khashMessage = self.H(privkey + msg, None, None)
```

This new value is used to calculate the first component of the signature, namely R. However, if we give an empty string to the signing algorithm (the hosted one), we should get an R value from which we can extract the `khashMessage` value. Why this is a vulnerability comes to light when lookint at line 367:

```
khash = self.H(privkey, None, None)
```

Because of the empty message, both `khashMessage` and `khash` are equal. This means that the resulting s and r values are also the same. We can use those to sign a new message, the private key is "stored" in them so we don't need it.

Procedure

Sending "" to be signed:

```
b'T0vUg8CHzYIJupRYQEMWQeLy6bgEkJYJngFUpwbTg1wFAGppiiUmn40t32ALaQqVUjFpsGgBtQWyJQnS
eVTOBA=='
```

The signature is defined as:

$$ sig \equiv r + H(R || A || M)s \mod l $$

We know:

1. R the "left" half of the signature, which is defined as `khashMessage` in the implementation and would be equivalent to `khash` for an empty message
2. A is the public key
3. M is the empty string
4. $H()$ is `sha512`
5. $r = s$ because ln 364 is equivalent to ln 368 for an emtpy string (see point 1).
6. l is some large prime

By factoring r, we find:

$$ r \equiv s \equiv \frac{sig}{1 +H(R || A || M)} \mod l $$

So to forge a new signature we can use this as our s. In the code, we need to first grab all the values from the empty message's signature:

```
    Rraw, Sraw = sig[:pEd25519.b // 8], sig[pEd25519.b // 8:]
    R, S = pEd25519.B.decode(Rraw), from_le(Sraw)
    A = pEd25519.B.decode(pubkey)

    # Calculate h = H(R || A || M) where M is the empty string
    l = Edwards25519Point.stdbase().l()
    h = from_le(pEd25519.H(Rraw + pubkey + empty_msg, None, False)) % l
    h_inv = pow(1 + h, -1, l)

    s = from_le(Sraw) * h_inv
```

While respecting every value's type and base. To find all the necessary functions (to encode, decode or convert) applied to the various values, we had a look at the implementation and copied some code from the `sign()` and `verify()` functions.

```
    # Now we grab s, which is equal to r following the equation explained in point
6
    s = from_le(Sraw) * h_inv

    # The private key does not matter here
    a = os.urandom(32)
    kh_mess_new = pEd25519.H(a + flag, None, None)
    r_new = from_le(pEd25519._PureEdDSA__clamp(kh_mess_new[:pEd25519.b // 8])) % l

    # R of sig_new
    R_new = (pEd25519.B * r_new).encode()

    # Calculate h.
    h_new = from_le(Ed25519_inthash(R_new + pubkey + flag, None, False)) % l
    S_new = to_bytes(((r_new + h_new * s) % l), pEd25519.b // 8,
byteorder="little")

    # Forge signature for the flag
    forged_sig = R_new + S_new
```

It's important to note that the private key used in the equation for the new `khashMessage` value is irrelevant. This is due to the fact that we have the valid s value extracted from the empty message's S component.

The complete code is located in the `exploit()` function and should result in this:

```
$ python .\chall2.py
Message b'' is verified: True
Message b'My grade in ICR is 6.0' is verified: True
```

Chall 3

The last implementation includes a timestamp with the signature. The timestamp is added as an additional security to the signature algorithm here:

```
    date = str(datetime.datetime.utcnow()).encode()
    ...

    # Calculate h with date
    h = from_le(self.H(R + pubkey + msg + date, ctx, hflag)) % self.l
```

This may seem like a smart move but what the coder failed to realise was that we can extract s (similar to what we did before) by sending two identical messages at different times instead of an empty one. This is thanks to the following equation:

We know that:

$$sig \equiv r + H(R || A || M)s \mod l$$

So from that:

$$sig_1 \equiv r_1 + H(R_1 || A || M)s \mod l$$

$$sig_2 \equiv r_2 + H(R_2 || A || M)s \mod l$$

NB: The r and consequently R values differ because they incorporate the timestamp.

Now we can solve for s the private key:

$$s \equiv (sig_1 - r_1) \cdot (H(R_1 || A || M))^{-1} \mod l$$

$$s \equiv (sig_2 - r_2) \cdot (H(R_2 || A || M))^{-1} \mod l$$

Which can be transformed into:

$$s \equiv (sig_1 - r_1) \cdot (H(R_1 || A || M))^{-1} \equiv (sig_2 - r_2) \cdot (H(R_2 || A || M))^{-1} \mod l$$

$$(sig_1 - r_1) \cdot (H(R_1 || A || M))^{-1} \equiv (sig_2 - r_2) \cdot (H(R_2 || A || M))^{-1} \mod l$$

And finally:

$$s \equiv ((sig1 - sig2) \cdot (H(R_1 || A || M) - H(R_2 || A || M))^{-1})$$

First we can extract the `a` value defined here:

```
    # Expand key.
    khash = self.H(privkey, None, None)
    a = from_le(self.__clamp(khash[:self.b // 8]))
```

by looking at the difference between two signatures, specifically:

```
    # Calculate h = H(R || A || M) where M is the message
    l = Edwards25519Point.stdbase().l()
    h1 = from_le(pEd25519.H(Rraw1 + pubkey + msg + sig_1_date, None, False)) % l
    h2 = from_le(pEd25519.H(Rraw2 + pubkey + msg + sig_2_date, None, False)) % l

    # And then s using the formula specified above
    s = ((S1 - S2) * pow((h1 - h2), -1, l)) % l
```

We can then proceed to trivially sign a new message using the extracted private key. The code for this can be found in the `exploit()` function.

And when running it:

```
$ python chall3.py
Message 1: b'U29tZSBtZXNzYWdl' is verified: True
Message 2: b'U29tZSBtZXNzYWdl' is verified: True
Message b'My grade in ICR is 6.0' is verified: True
```

Improving the implementation

To improve the implementation and eliminate the vulnerability, we can simply hash the date with the message content before calculating our hash.

```
def sign_date(...):
    ...
    # Calculate the unique deterministic message date hash
    msg_date_hash = self.H(R + msg + date)

    # Calculate h with date
    h = from_le(self.H(pubkey + msg_date_hash, ctx, hflag)) % self.l
```

And to verify we do the same:

```
def verify_date(...):
    ...
    # Calculate the unique deterministic message date hash
    msg_date_hash = self.H(R + msg + date)

    # Calculate h.
    h = from_le(self.H(pubkey + msg_date_hash, ctx, hflag)) % self.l
```

This way the crucial values from our formula earlier are "hidden" and cannot be used to determine the private key.