TL;DR

We've found a critical vulnerability in Loopring's wallet, which allows anyone to reproduce all private keys generated using Loopring's frontend. We've demonstrated this vulnerability by reproducing the private keys for over a dozen accounts. The vulnerability allows anyone to gain control of every single Loopring account created using their frontend.

A Tale of Two Keys

In Loopring a user has two types of keys:

- Ethereum key
- (Loopring) Account key

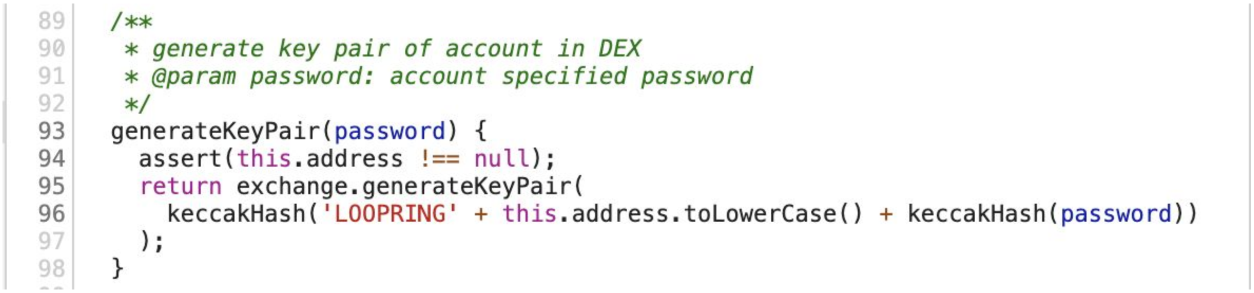
The Account key is needed for its SNARK-friendliness properties, as it is much (much) easier to prove a signature signed by the Account key than by the Ethereum key. The Account key, being SNARK-friendly, differs from the Ethereum key in several parameters (e.g. the elliptic curve). Because of that, it is not easily supported by other wallets.

When a user joins the system, he most likely has an Ethereum key, but not yet an Account key. So even though the user chooses MetaMask as his wallet when connecting to Loopring, it will only serve his Ethereum txs (for interactions with the contract, including deposits and withdrawals). But to submit orders, the user will have to use their Account key.

Loopring chose to run all the Account-key-related operations in the browser. Therefore, as part of the registration process, the user generates his Account key, and a map between the two key types is saved in the contract. From that moment on, every time the user needs to submit orders, he will use his Account key and not his Ethereum key.

The Account Key Derivation Process

To generate an Account key pair (public, private[1]), the user is required to enter a password. This password, together with the user's Ethereum address, is used to derive his Account key:



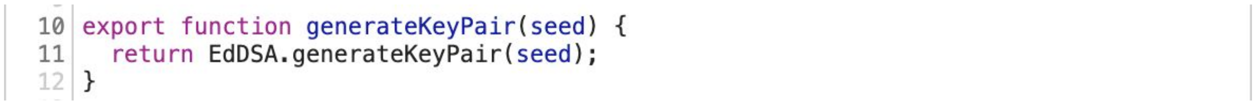
```
/**
 * generate key pair of account in DEX
 * @param password: account specified password
 */
generateKeyPair(password) {
  assert(this.address !== null);
  return exchange.generateKeyPair(
    keccakHash('LOOPRING' + this.address.toLowerCase() + keccakHash(password))
  );
}
```

`wallet.js`

[1]The code makes use of 'secret' instead of the commonly used 'private'.

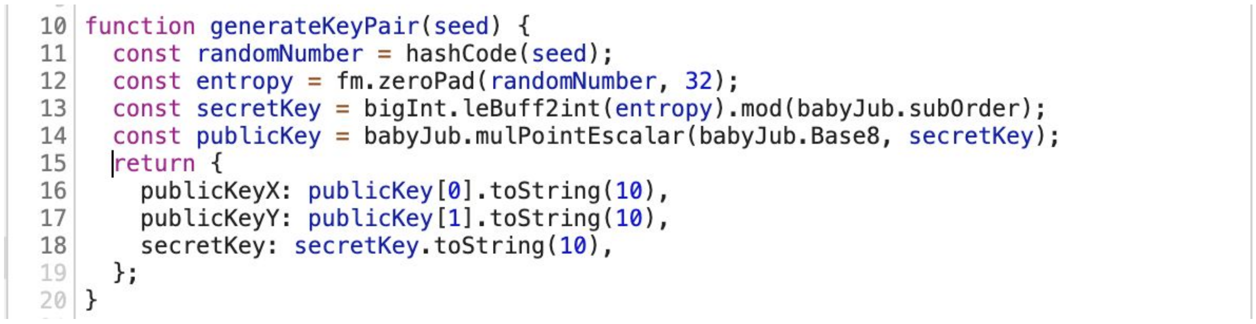
This in and of itself is bad practice: since the Ethereum addresses, as well as the Account Public Keys, are readable from the contract, a comprehensive search of passwords is feasible. And as history has taught us over and over again (see brain wallets[2]), most users are not good at choosing a strong password.

It doesn't end here. Let's inspect further:



```
export function generateKeyPair(seed) {
  return EdDSA.generateKeyPair(seed);
}
```

Exchange.js



```
function generateKeyPair(seed) {
  const randomNumber = hashCode(seed);
  const entropy = fm.zeroPad(randomNumber, 32);
  const secretKey = bigInt.leBuff2int(entropy).mod(babyJub.subOrder);
  const publicKey = babyJub.mulPointEscalar(babyJub.Base8, secretKey);
  return {
    publicKeyX: publicKey[0].toString(10),
    publicKeyY: publicKey[1].toString(10),
    secretKey: secretKey.toString(10),
  };
}
```

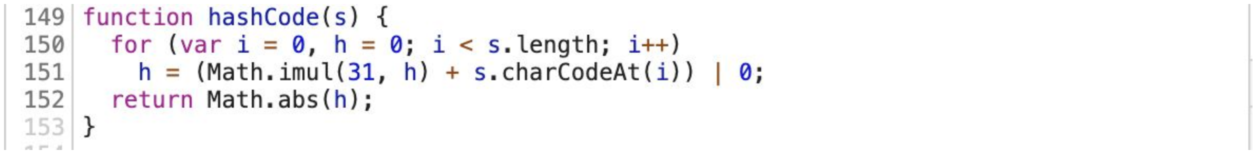
EdDSA.js

The Ethereum address and the password are used to generate the *seed*. Then, the *seed* is used to derive *randomNumber*. It's that random number that is later used to get entropy that will give the user the desired private key.

But how is *randomNumber* derived from the *seed*?

hashCode

Let's take a look at *hashcode*:



```
function hashCode(s) {
  for (var i = 0, h = 0; i < s.length; i++)
    h = (Math.imul(31, h) + s.charCodeAt(i)) | 0;
  return Math.abs(h);
}
```

bitstream.js

So in our case *hashCode* will go over the *seed* char by char, and will compute h from it.

[2] https://bitcoin.stackexchange.com/questions/8449/how-safe-is-a-brain-wallet

But *Math.imul* is doing 32-bit integer multiplication[3]! This means the result is a 32-bit integer. The other operation we see here (for addition) is | 0. But bitwise operations in JavaScript results in 32-bits integers[4] (all operands are transformed into 32-bit integers).
The result: *hashCode* returns a 32-bit integer! This means that no matter what the *seed* is, *randomNumber* will be a 32-bit integer.
The entire entropy to generate private keys, the entire keyspace, is 32-bits!

Implications

Regardless of the user's password, his Account private key will be derived from a keyspace of 32 bits. We can compute all the possible Account keys in the system (as long as they were created using their frontend).
Therefore, we can find the private keys of all the users registered through the API. We've actually reproduced the private keys for over a dozen accounts, to demonstrate this vulnerability.

By controlling those keys, an attacker can submit orders on behalf of the users. This can enable him to actually steal funds. One way to do it is to submit enough orders to an illiquid pair with a price that is far from the current price. For example, if an attacker wants to steal DAI from accounts, he can send many orders offering to "buy 1 ETH for 10000 DAI". At the same time, the attacker will place orders of "sell 1 ETH for 10000 DAI". If the pair is not liquid enough eventually those orders will be executed (once the order book was consumed) and the attacker's account will gain those DAI for a small amount of ETH.

Conclusions

Great care should be invested in designing secure wallets. In particular, the process of key generation and management should be done carefully. A system that doesn't use the standard Ethereum cryptography (e.g. the secp curve) has to work hard to produce the alternative wallet capability. In our fast-growing ecosystem this sometimes means conducting security-related operations in the browser (not so good practice); use passwords to generate keys (bad practice); and introduce critical bugs in the process (like the hashcode bug above - very bad).

Hopefully in the near future, Ethereum wallets will become more flexible and will support a broader set of cryptographic primitives. This will again separate the work of creating a secure wallet from the work of creating a dApp, and will prevent such bugs in the future.

[3] https://stackoverflow.com/questions/21052816/why-would-i-use-math-imul
[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators

Acknowledgement: Thanks to Louis Guthmann (@GuthL), a product manager and researcher on my team, for his help in exploiting the vulnerability.

Avihu Levy (@avihu28), Head of Product, StarkWare