U2F-ETH Documentation

CHEN Xiwen
`victoria-x@sjtu.edu.cn`

May 12, 2019

1 Overview

The program is able to handle transactions for the users via a smart contract deployed on blockchain. It acts as a transaction manager and can choose to approve, reject, or delay a transaction based on a hidden policy. Besides, the users may be required to provide a second factor authentication, which is implemented as U2F in this program.

The Manager smart contract is deployed on blockchain and provides functions that can interact with the client side (Javascript). The functions can be divided in to U2F functions and manager functions (functions that help handle transactions for the user).

2 U2F Functions

The U2F functions are `enroll`, `bind`, `sign` and `verify`, which are further divided into two parts. `enroll` and `bind` are for U2F registration, while `sign` and `verify` are for U2F authentication. Besides, there is a function called `client` that is common for both processes.

2.1 Registration

To begin a registration, `enroll` function takes the `appId`, which is the origin of the website, as an input, and outputs a registration request in JSON string format. The registration request contains a 32-byte random string (generated based on the public key of the user, the current block number, current block miner's address, block difficulty, and a nonce that increments by one after each time a random string is required) as challenge, U2F version information (always `U2F_V2` in this program), and the `appId` (origin of the website). The server (Manager contract) will also store the registration request string for the specific user for later `bind` function.

Then the u2f-api, which is injected in Chrome browser, will check the registration request string, and contact the hardware token to get a signature. The hardware token will sign on the concatenation of `appParameter` (sha256 of `appId`), `challengeParameter` (sha256 of the `clientData`, described later), `keyHandle` (information generated by the hardware token to help decide which key to use in authentication), and `pubKey` (public key used to verify the authentication signature). The signature is 71- to 73-byte and is generated using ECDSA algorithm. Then the u2f-api will gather this signature and other information to generate an encoded JSON string, which is the output of the u2f-api.

The string is decoded using websafe encode/decode algorithm in client side (JavaScript), and the resulting dictionary contains a `registrationData` and a `clientData`. The `clientData` is another JSON string containing `type` (register), challenge (the 32-byte random string used in the registration process), and origin of the website. This information is passed to the Manager smart contract as the input of `client` function, together with other two inputs `appId` and `type` which are stored in the client side. The smart contract verifies the client information using these inputs by checking whether challenges, version and `appId` match. It also computes and outputs the `challengeParameter` for the `bind` function. The `registrationData` has the following format:

```
+------------------------------+
| 1 | 65 | 1 | L | implied | 64 |
+------------------------------+
0x05 : pubKey : keyHandleLength : keyHandle : cert : signature
```

Then the `registrationData` is parsed based on the format above in JavaScript, and the public key used to generate the signature is extracted from the certificate, called `certKey`. The parsed `pubKey`, `keyHandle`, `certKey` and `signature` are passed to the smart contract's `bind` function to complete the registration process.

The `bind` function first finds the registration request that was previously stored for the user. It then uses `appId` to obtain the `appParameter`, and uses the previous two data to concatenate with `keyHandle` and `pubKey` to get the message that is signed by the U2F token. Then `certKey` is used to verify the ECDSA signature. If the signature is successfully verified, the smart contract returns a JSON string containing the information about the registered key, including version, `keyHandle`, `appId`, and `pubKey`. The string is then stored in the smart contract for the user, announced as an event on blockchain, and is used in later authentication.

2.2 Authentication

The general structure of the authentication process is the same as registration. To begin authentication, `sign` function is called with input `appId` (origin of the website). Then the smart contract finds the registered key information for the user, generates a 32-byte random string as a challenge, and packs these data into a JSON string, which is the sign request. The sign request consists of `keyHandle`, version, challenge, `appId` and `pubKey`. This sign request is then stored in the smart contract for the user for `verify` function.

Then the u2f-api is used to check the sign request string, process the sign request data, and get message signed by the hardware token using ECDSA algorithm. The message to be signed is a concatenation of `appParameter` (sha256 of `appId`), `userPresence` (0 or 1 to indicate whether the user has tapped the device), counter (the number of times the hardware token is used to sign a message), and `challengeParamter` (sha256 of `clientData`). Then the u2f-api gathers the signature and other information to generate an encoded JSON string, which is passed as an input to the JavaScript client side.

The string is decoded using websafe encode/decode algorithm in client side, and the resulting dictionary contains a `signatureData` and a `clientData`. The `clientData` includes

the same information as in the registration client data (`type` (sign), challenge, and origin of the website). Similarly, by calling the `client` function of the smart contract, the client information is checked, and the `challengeParamter` is computed and returned. The `registrationData` has the following format:

```
+----------------+
| 1 | 4 | implied |
+----------------+
userPresence : counter : signature
```

Then this `registrationData`, together with the previously computed `challengeParameter`, is passed as an input to the `verify` function to verify the signature. The smart contract finds the sign request string for the user, computes the `appParameter` from `appId` and obtains the message that is signed by the U2F token. The signature is then verified in the smart contract using ECDSA algorithm. If the verification ends successfully, the smart contract generates a JSON string containing the verification data (`keyHandle`, `userPresence`, and `counter`). The verification data will be announced as an event on blockchain and be returned.

3   Manager Functions

Before a U2F hardware token is registered for the specific user in the smart contract, the manager functions are locked. Namely, the user is not able to transfer money to another public key via the Manager smart contract.

The main function that actually handles the transactions is the `transferViaManager` function. The inputs are the recipient's public key and amount. Based on a user customized U2F policy (will be described later), the smart contract decides whether to approve, reject, or delay a transaction, or require U2F authentication of the user. If the value transferred exceeds a preset limit, U2F policy needs to take charge. If the policy is "strict", then U2F authentication is outright required, and the user needs to go through a U2F authentication process to confirm the transaction. If the policy is "history", then the smart contract will look up the transaction history that is stored in its local storage. If there exists a valid transaction history (a transaction history that involves value exceeding the limit, and has not expired regarding a preset history lifetime), then the transaction is approved. Otherwise, a U2F authentication is required. If the value exceeds two times of the limit, in all cases the transaction will be delayed by a preset delay time.

Users are also able to lock/unlock their account in the smart contract. After locking the account, the manager functions are locked (similarly to the condition when the user has not registered a U2F hardware token). The lock/unlock operations both require U2F authentication.

4   U2F Verification Details

4.1   Certificate

The main purpose of a certificate in the registration response data is to include a public key, which is used to verify the U2F signature. Apart from this public key, a certificate includes other information such as issuer, period of validity, subject information, and signature. All of them are used to ensure that the certificate itself is valid. Considering the computational abilities of a smart contract, verifying a signature directly from certificate is hard to implement in Ethereum. Therefore, the validity of the certificate is instead ensured in the JavaScript client side, and the public key contained in the certificate (`certKey`) is extracted for verifying signature.

4.2   ECDSA Encryption

Blockchain uses elliptic curve encryption scheme for signatures. Compared with many other encryption schemes, elliptic curve is relatively cheap in computation. Furthermore, while ensuring the same level of security, elliptic algorithms provide shorter key pairs than many others such as RSA encryption scheme.

Ethereum provides a cheap function `ecrecover` to verify signatures that are signed by the user, which costs about 3000 Gas. The function takes the `sha3` hash of the message (`"\x19Ethereum Signed Message:\n"` + raw_message.length + raw_message), `v` (an id that is needed for public key recovery, equals 27 or 28), `r` and `s` as inputs, and outputs an Ethereum address (last 20 bytes of the `sha3` hash of the ECDSA public key). The elliptic curve used in this function is secp256k1.

However, in U2F signature, the elliptic curve is secp256r1. This means that `ecrecover` cannot be used to verify the U2F signature, both in registration and authentication. The ECDSA signature verification costs around 1600000 Gas, which is approximately 24% of the total Gas consumption of `bind` operation. Although this is not the major source of Gas consumption, changing the type of elliptic curve might be a potential improvement to this program.

5   Gas Cost

5.1   Aggregated Costs

Using a localhost RPC provided by Ganache CLI and MetaMask for testing, the gas consumption data and costs are shown in Table 1. Denote the transfer limit as l. (We use 1 ETH $\approx$ 164.11 USD, and suppose for the following tests, the U2F policy is "history" with delay.)

Table 1: Aggregated Costs

Category	Name	Gas Limit [Units]	Gas Used [Units]	Cost [ETH]	USD
Deployment	`Register.sol`	5606245	3737497	0.003737	0.61
	`Authenticate.sol`	7200000	5027873	0.005028	0.83
	`Manager.sol`	153211	102141	0.000102	0.0166
Settings	Set Limit	40672	27115	0.000027	0.0044
	U2F Policy	9026705	5838252	0.005839	0.934
U2F Register	Enroll	1290522	860348	0.00086	0.14
	Bind	7200000	7020296	0.00702	1.15
Transfer	$v \leq l$	48442	32295	0.000032	0.0053
	$l < v \leq 2l$, `history`	50257	33505	0.000034	0.0056
	$v > 2l$, `history`	161203	107469	0.000107	0.0172
	$l < v \leq 2l$, `!history`	9088775	5879184	0.005879	0.96
	$v > 2l$, `!history`	9062476	5861651	0.0059	0.9379
Lock Method	Lock	10350543	5827360	0.00583	0.96
	Unlock	10350998	5796939	0.00580	0.95

5.2   U2F Functional Costs

In this section, the functional costs of U2F methods are listed in Table 2. The operations are divided based on the things that will be done in the smart contract for each function. (We use 1 ETH ≈ 164.11 USD.)

From Table 2, the major gas consumption in the functions `bind` and `verify` are caused by websafe encode/decode in smart contract. The first significant gas consumption is to generate the device JSON string. This step is to gather the verified device information into a JSON string, which is then stored in local storage and announced on blockchain. In its data field, the `pubKey` and `keyHandle` require a websafe encode step. Similarly, in the `verify` function, the reason why verifying ECDSA is notably expensive is that the websafe encoded `pubKey` needs to be decoded in this step to complete the verification.

Therefore, one potential direction for improvements might be removing all the websafe encode/decode operations out of smart contract and be completed in JavaScript. But since these data are crucial part in the verification process, this might result in a decreased level of security (from personal point of view). Besides, removing encode/decode out of smart contract means more frequent interaction between the JavaScript client side and the smart contract, which is more complicated to implement than the current scheme. But this result gives an insight to possible revisions of the program, and these adjustments, although a little bit complicated, can be implemented if necessary.

Table 2: U2F Functional Costs by Task

Function	Task Description	Estimated Gas [Units]	USD
client	validate user information	380732	0.0609
	calculate and return `chalParam`	18120	0.0029
enroll	generate and websafe encode challenge	672725	0.1076
	generate request JSON string	59068	0.0095
	store request string, emit event	103714	0.0166
bind	get data that is signed on	147343	0.0236
	verify ECDSA signature	1594709	0.2552
	generate device JSON string	4556503	0.7290
	store device string, emit event	205980	0.033
sign	generate and websafe encode challenge	693815	0.111
	generate request JSON string	913015	0.1461
	store request, emit event	246575	0.0395
verify	parse `signatureData`	102040	0.0163
	get data that is signed on	28832	0.0046
	verify ECDSA signature	3159422	0.5055
	generate record JSON string	759839	0.1216
	store record string, emit event	237324	0.038