Token Group Ltd. Wallet Contract Audit

Nov, 2018

Contents

Chapter 1

Introduction

1.1 Scope of Work

Token Group Ltd. requested technical feedback on the Token Group Ltd. Wallet, an Etherum smart contract implemented in Solidity, along with an auxilliary Oracle contract that serves as a crypto-asset-to-Ether exchange rate "oracle" for deployed Wallet contracts.

The purpose of this document is to describe architectural and implementation-specific defects present in the source files for the aforementioned smart contracts, along with appropriate mitigations and/or alternative architectural solutions, where appropriate. This source code review, like any source code review, is not an endorsement of the code or its suitability for any legal/regulatory regime, and it does not claim to be a definitive or exhaustive list of defects. Rather, this audit is an attempt to document discrepancies between the code's stated purpose (as described in written documentation and through verbal conversations with Token Group Ltd.'s engineering staff) and its behavior when correctly compiled and executed on the Ethereum blockchain.

1.2 Source Files

The sources files necessary to compile Token Group Ltd.'s Oracle and Wallet contracts are listed below underneath the SHA256 checksum of the source at the time the audit began.

Files in the `externals/` directory are vendored third-party source files, and files in the `internals/` directory are additional dependencies that Token Group Ltd. authored.

External (third-party) dependencies are not within the scope of this audit's line-by-line review. Nonetheless, this audit did uncover some issues in third-party code.

```
acea06418fedeacc6d3c4555c38dd8282e766b7e995774dc45bc7ebf3dc9c982
        externals/SafeMath.sol
45c0ccd7b640a592fcaaf982abde16f4a577673eb3660ea5d01156a104d5d85b
```

externals/base64.sol
8ecf13946c04b18aa819fd5aee8bd2c81ac020a00bd96a70a52bd95c3103c9ab
externals/oraclizeAPI_0.4.25.sol
0176b1f335e6a278d7fe7af64f1840ba779317c252d374b92a67d89f841bc9b1
externals/oraclizeConnector.sol
0f0937d876de0d11a5615dace4422f4540606c7b731b7a1c98ba0ad585f7a2cf
externals/strings.sol
b0f348f6e5a1f67e2eb8fa510248cc09ffd47324f26be3c7422adb9538e0060d
internals/controllable.sol
634f647a8659768037e8eab16e677632462058eea59348f39dc31e3e26119f65
internals/controller.sol
0e78a7c7a690a6dc102c9e8571504ef16df37c61de1a32eb14a44b3e920ca448
internals/date.sol
bd0055b323f816b2c3902164b06b419ba788ec47bc77ba44da0305d69ee23c23
internals/json.sol
4a26094440af7cb0ea7619c623bcc43904330142ed0fc5e6a36e80dd954acf67
internals/oraclize-resolver.sol
ec419c0475459c7d18fc187b1b62d7caf4df1708e67df9079441ee890bf0abe1
internals/ownable.sol
4f36ab8908006688348295ccb56df43b3a4acfc5f23133b3d32729e97ee3f117
oracle.sol
4746558e3cd9801190791ff7a2501f3013e3ced9a8290176b3f33619dee9c3de
wallet.sol

Chapter 2

Summary

2.1 Purpose

Token Group Ltd.'s `Wallet` contract, defined in `wallet.sol`, is the core of the "Token Wallet" implementation. Its documented purpose is to reproduce a European consumer banking experience by implementing a handful of features that have analogs in the consumer banking world:

- Daily spending limits
- A "whitelist" of accounts that can receive assets in excess of the daily spending limit

The `Wallet`'s "consumer" features are, in practice, key disclosure mitigation mechanisms: the impact of a key disclosure is reduced if the addresses to which assets can be sent is predetermined.

2.2 Architecture

2.2.1 Current Architecture

Each `Wallet` contract serves as an asset repository for one entity, and the contract is jointly administrated by the address that deployed the contract (the `owner`) and by a set of addresses controlled by Token Group Ltd. collectively referred to as the `controller`.

The `onlyController` modifier, which guards functionality intended to be used exclusively by Token Group Ltd., is implemented by looking up a `Controller` *contract* in ENS, and then checking whether or not the sender address is present in the list of authorized addresses in the controller. In other words, there are two layers of indirection present in this access control mechanism: first, ENS is used to look up a contract, and then that contract is consulted regarding the capabilities of an address. (It is a bit confusing that the term "controller" has been overloaded here; an address that

passes the `onlyController()` modifier is determined by the `Controller` contract, but the `Controller` itself does not pass `Controller.isController()`.)

When a `Wallet` is deployed, the entity deploying the contract has the opportunity to set preliminary values for the address whitelist and spending limit. After those preliminary values have been set, subsequent modifications must be approved by Token Group Ltd. by way of a two-phase commit in which the `owner` proposes a change to the contract and the `controller` approves it.

The `Wallet` contract's daily spending limit is partly implemented by an `Oracle` contract, which attempts to provide token-to-Ether exchange rates for tokens held in `Wallet` contracts. Thus, the daily spending limit can be denominated solely in Ether. The vast majority of the vendored contracts covered by this audit exist to support the exchange rate oracle. Wallets look up the address of the `Oracle` contract through ENS, much like they do with the `Controller` contract.

2.2.2 Architectural Comments

A substantial amount of the source code (and therefore, attack surface) upon which the `Wallet` contract implicitly depends is devoted to the implementation of the exchange rate oracle. In order to obtain third-party exchange rate information, the source code has to partially implement a number of complex and error-prone data processing steps, such as HTTP Date header parsing, JSON parsing, and DER-encoded ECDSA signature unpacking. The implementation of those routines is not robust to adversarial input (and relies upon the input being "trusted" due to the fact that it is part of the signed transaction payload), which substantially increases the risk that an insufficiently hardened routine could be exploited as part of a larger exploit chain. Moreover, many of the defects uncovered in this review were in the `Oracle` and its dependencies, despite the fact that external dependencies were not subject to the same level of scrutiny as Token Group Ltd.'s *de-novo* code.

The architectural purpose of the `Oracle` contract is to "decentralize" the source of exchange rates so that Token Group Ltd. isn't directly capable of manipulating them. However, through the layer of indirection inserted into the `Wallet` contract via ENS, the `Oracle` implementation can be swapped on-demand by anyone who has administrative privileges on the ENS name for the `Oracle`. In other words, the `Wallet` exchange rate source isn't meaningfully more decentralized than if Token Group Ltd. maintained an on-chain dictionary of rates themselves, since there exists a simple "back door" through which the code that produces exchange rates can be mutated. The additional risk and complexity of introducing a third-party exchange rate oracle is not worth a purely symbolic gesture towards "decentralization."

Section 6.1 describes a much simpler counter-signing implementation that confers the same security benefits as the current `Wallet` implementation, but with significantly less Solidity source and reduced complexity.

2.3 Testing

Token Group Ltd. has taken a considered, albeit unconventional, approach to testing the `Wallet` and `Oracle` contracts: the code is tested using a wrapper of the `geth` Ethereum Virtual Machine, and test code is written in the Go programming language. Just over three thousand lines of Go test code accompany the contract source code, which is an above-average ratio of test code to implementation code for this particular programming domain. Moreover, it appears that the tests cover virtually all of the `Wallet` functionality, and the tests include both negative and positive tests.

Chapter 3

Critical Defects

3.1 Submit/Confirm Pattern Race Condition

The `Wallet` contract uses a pattern to modify state variables wherein the owner of the wallet (the end-user) requests a change by calling a function like `submitWhitelist()`, and Token Group Ltd. approves the changes by calling `approveWhitelist()`.

The purpose of this pattern is to give Token Group Ltd. the opportunity to disallow suspicious or illegal state changes before. However, an attacker in posession of the end-user's private key can still make arbitrary state changes to the contract by carrying out the following series of operations:

1. The attacker submits a legal change via a method such as `submitWhitelistAddition()`.
2. The transaction in step 1 is mined.
3. Token Group Ltd.'s infrastructure evaluates the `SubmittedWhitelistAddition()` event emitted by the transaction in step 1 and concludes that the proposed whitelist change is a valid state transition.
4. Token Group Ltd. submits a `confirmWhitelistAddition()` transaction.
5. The attacker submits an illegal change via `submitWhitelistAddition()` as soon as the `confirmWhitelistAddition()` transaction becomes a pending transaction. This new `submitWhitelistAddition()` transaction is submitted with a much higher gas price than Token Group Ltd.'s `confirmWhitelistAddition()` transaction.
6. The attacker's second `submitWhitelistAddition()` is mined before the `confirmWhitelistAddition()`, and thus the illegal state change is successfully committed to the contract.

The following functions are vulnerable to this race condition:

- `Wallet.submitTopUpLimit()`, `Wallet.confirmTopUpLimit()`
- `SpendLimit.submitSpendLimit()`, `SpendLimit.confirmSpendLimit()`
- `Whitelist.submitWhitelistAddition()`, `Whitelist.confirmWhitelistAddition()`
- `Whitelist.submitWhitelistRemoval()`, `Whitelist.confirmWhitelistRemoval()`

Since the `Wallet` contract's spending limit and address whitelist are both guarded by this vulnerable pattern, it is possible for a moderately sophisticated attacker to bypass all of the security features of the `Wallet` contract.

3.1.1 Mitigation: Lamport Clocks

The mitigation for this race condition is to introduce a Lamport Clock that allows the state-change confirmation to evaluate if the state has changed since the confirmation was issued:

Lamport Clocks are sometimes colloquially referred to as "sequence numbers."

```
event SubmittedFoo(uint value, uint counter);
event ConfirmedFoo(uint value, uint counter);
uint private counter;
uint private maybeFoo;
uint private foo;

function submitFoo(uint value) public onlyOwner {
        counter++;
        maybeFoo = value;
        emit SubmittedFoo(value, counter);
}

function confirmFoo(uint clock) public onlyController {
        require(counter == clock);
        foo = maybeFoo;
        maybeFoo = 0;
        emit ConfirmedFoo(foo, clock);
}
```

In the example above, the off-chain infrastructure can easily associate a sequence number with each `submitFoo()` transaction by examining the emitted `Subbmitted-Foo()` event, and then approve *only that submitted value* by calling `confirmFoo()` with the same sequence number.

Note that this mitigation is only air-tight if events are examined only after they are *finalized*, which, in practice, occurs after six or more blocks have been mined after the block in which the transaction was mined. Without a strong (probabilistic) guarantee

of finality, it is possible that a transaction that uses clock value `A` could be mined, re-org'ed out of the canonical chain, and a different transaction with the same clock value `A` could be mined.

The probability of a transaction being removed from the chain due to a re-org can be approximated by using the current "uncle rate" u of the chain and the number of blocks that have elapsed since the block containing the transaction was mined (here denoted as z).

$$Pr(R_{n-z}) = u^{z+1}$$

For example, if the current uncle rate was 5%, and 6 blocks had been mined on top of the block with which we are concerned, there is a $Pr(R_{n-6}) = (0.05)^{6+1} = 7.8125 * 10^{-10}$ likelihood that the block isn't final.

Chapter 4

Moderate Defects

4.1 Unmitigated ECDSA Signature Malleability

The `ecrecover` "pre-compiled contract" offered by the Ethereum Virtual Machine returns the Ethereum address associated with a `secp256k1` ECDSA signature, as specified by a triple (R, S, V), where R and S are the values from a conventional ECDSA signature tuple (R, S), and V is a value that indicates both the transaction format and the signedness of the Y-component of the `secp256k1` curve point that corresponds to R.

One weakness of the ECDSA signature scheme is that the signatures are malleable. Specifically, for the field order p, the signatures (R, S) and $(R, p - S)$ are equivalent: they are both valid signatures for the same public key and hashed input. What this means is that code that presumes unique signatures correspond to unique invocations of the ECDSA signature scheme can be tricked by "replaying" old signed input.

The code in `externals/oraclizeAPI_0.4.25.sol` that wraps the `ecrecover` pre-compiled contract, such as `ecrecovery()`, `verifySig()`, `safer_ecrecover()`, etc., does not implement any countermeasures against ECDSA signature malleability. Consequently, calling code can mis-use these APIs if it makes incorrect assumptions the uniqueness of signatures accepted by these validation functions.

It does not appear that Token Group Ltd.'s code is vulnerable due to misuse of these APIs; however, it is always advisable to proactively reduce the possibility of API misuse where practical.

4.1.1 Mitigation: Nonce Clamping

Between the Frontier and Homestead Ethereum hard-forks, the Etheruem transaction format was changed so that valid transactions must have $S <= Z_p/2$. The inversion $S' = p - S$ over the field will invalidate the precondition that S must be under the curve half-order, and thus only one of the two possible signature tuples is accepted as a valid signature.

However, the Etheruem `ecrecover` contract behavior was *not* changed, in order to preserve backwards compatibility. Consequently, code that accepts `secp256k1` sig-

The `secp256k1` elliptic curve is over a prime field Z_p and of the form $y^2 = x^3 + 7$, which is symmetric over the x-axis, and thus there are two y-coordinates that are on the curve for a given x-coordinate. The Ethereum V value indicates which y-coordinate pairs with the x-coordinate given in the ECDSA signature R component, and thus we can compute from the signature the precise public key of the signer. For this reason, ordinary ECDSA signatures of the form (R, S) correspond to one of two possible public keys.

natures from off-chain signing programs must either implement the same malleability checks, or be absolutely certain that signature malleability doesn't violate any assumptions in the code's security model.

A simple proactive fix is to use the following code immediately before passing it to `ecrecover`:

```
require(sigS < CURVE_HALF_ORDER);
```

4.2 Oracle: `removeTokens()` is $O(n^2)$

The `removeTokens()` function in `oracle.sol` executes in linear time for each token removed, or polynomial time for removing a list of tokens where the list of tokens to be removed is proportional to the size of the list of tokens overall. Consequently, it would be relatively easy for the list of tokens reach a length such that the gas cost of executing `removeTokens()` would exceed the block gas limit, and thus the function could never be called successfully. In general, Solidity code should carefully avoid operations that do not run in constant time, as there is a practical limit to the execution "time" available for a transaction.

As of this writing, the consensus block gas limit is 8,000,000 gas.

The code should store the index of each entry in the `_tokenAddresses` list in the `tokens` mapping so that the entry can be removed in constant time.

For example:

This code is for demonstration purposes only. It may contain bugs.

```
function addToken(address token) onlyController {
        require(!tokens[token].exists);
        _tokenAddresses.push(token);
        tokens[token] = Token({
                exists: true,
                index: _tokenAddresses.length - 1,
                // ... other fields
        });
}

function removeToken(address token) onlyController {
        require(tokens[token].exists);
        uint i = tokens[token].index;
        uint l = _tokenAddresses.length;
        _tokenAddresses[i] = _tokenAddresses[l-1];
        tokens[_tokenAddresses[i]].index = i;
        delete tokens[token];
        _tokenAddresses.length--;
}
```

Chapter 5

Minor Defects

5.1 Oracle: `_updateTokenRates()` wants JSON

Oraclize supports a feature whereby the oracle will deserialize parts of a JSON response body so that the smart contract doesn't have to parse the complete returned response. For example, asking for `"json(https://foo.com).bar"` will yield the `"bar"` entry of the returned JSON object. In order to reduce the implementation complexity in the `Oracle` contract, Token Group Ltd. should make use of this feature.

5.2 Oracle: `verifyProof()` Parses Untrusted Input

The `verifyProof()` function in `oracle.sol` parses untrusted text before checking the signature associated with the input. As a general rule of thumb, input validation should happen as early as possible, in order to reduce the amount of code that is reachable with corrupt or malicious input. In this particular case, the code doesn't `revert()` when it encounters invalid input (instead, it just returns `false`), which is particularly troubling: a vulnerability in the assembly-heavy string processing code through which the input passes could permanently corrupt contract storage. The code should always `require()` correct input to ensure that unexpected conditions within the execution of a transaction cause the transaction to be rolled back in its entirety.

5.3 Oracle: Unhandled Error Conditions in String Parsing

The `verifyDate()` function within `verifyProof()` uses the `parseInt()` function to unpack an HTTP `Date` string. The `parseInt()` function does not do any input validation, and so any string at all will yield an integer of some sort. Combined with the weakness in section 5.2, this creates a frighteningly large (and unnecessary) attack surface. The `parseInt()` function should eagerly `revert()` when it detects that the input is formatted incorrectly.

5.4 Oracle: `verifyDate()` not RFC2616-compliant

RFC2616 defines the acceptable formats for HTTP `Date` headers. The `verifyDate()` function does not handle the legacy formats specified in the RFC. It is unclear whether or not this could present a compatibility problem in practice, as those formats are explicitly discouraged in the RFC. Moreover, making the code more permissive in terms of acceptable inputs could increase the surface area for exploitable bugs. Nonetheless, Token Group Ltd. should be aware of this incompatibility.

5.5 All: Unnecessary Accessor Functions

A number of trivial accessor functions in the source code can be eliminated entirely. For example, the `Ownable.owner()` function is a trivial accessor for the state variable `_owner`. If the `_owner` variable were declared as `address public owner;`, then Solidity would automatically generate a public `owner()` accessor function.

Declaring state variables as `public` does not meaningfully change contracts' attack surface area, since those variables can still be read by examining the contract storage directly. Consequently, state variables that need to be read by other contracts or off-chain software should be declared `public`, rather than accessed through a superfluous function.

5.6 Date: Many Unused Constants

There is a fair bit of dead code in `internals/date.sol`. For clarity, it should be removed.

5.7 Controller: Key Disclosure Attack Surface

As discussed in section 2.2.2, the `Controller` contract is referenced through a layer of indirection via the Ethereum Name Service. Additionally, the `Controller` contract maintains a list of addresses that all have the same elevated level of privilege: each can add or remove other addresses from the `onlyController()` privilege level. Consequently, a disclosure of the key associated with the address that manages the `Controller` ENS name, *or* the disclosure of any key associated with an address that passes `onlyController()` compromises Token Group Ltd.'s wallet administration infrastructure. A compromise of this nature would be able to lock funds in some wallets by inflating exchange rates such that no reasonable quantity of funds would pass the daily spend limit validation.

In order to reduce the impact of a key disclosure, Token Group Ltd. should segregate the capabilities of privileged addresses such that the impact of any one key disclosure is minimized. Similarly, to the extent that the disclosure of one key could prove problematic, the architecture should rarely require the key to be used. Note that key management is beyond the scope of this evaluation.

5.8 Oraclize: DER-encoded Signatures

The `verifySig()` function in `externals/oraclizeAPI_0.4.25.sol` takes DER-encoded ECDSA signatures rather than Ethereum-format (R, S, V) signatures. Consequently, the signature validation code in that function has to compute the address associated with two curve points corresponding to the signature. Calling `ecrecover` more than once wastes gas and creates some address malleability concerns, since the signature does not correspond to a unique Ethereum address.

5.9 Vault: Access of `_spendAvailable`

The `Vault` contract accesses a private state variable `_spendAvailable` that it inherited from the `SpendLimit` contract. The practice of accessing inherited state variables is generally discouraged; it makes the code harder to follow, and it breaks encapsulation. In particular, it is important that mutation of variables like `_spendAvailable` be highly localized, since it regulates one of the primary security features of the `Wallet` contract.

Overall, the `Wallet` contract might be easier to read if the code for `Whitelist`, `SpendLimit`, and `Vault`, etc. was flattened into its implementation. No other code inherits from these contracts, and the `Wallet` code is not at all oblivious to the implementation of those contracts, so it's not clear that the code benefits substantially from encapsulation. Moreover, ABI shadowing can become a major risk as contract inheritance grows, so inheritance is best reserved for when it results in substantial code deduplication. Instead of encapsulating at the contract level, the code should prefer encapsulation at the function level: mutation of security-critical state should occur in as few places as possible, and the internal APIs exposed by those functions should be difficult or impossible to mis-use.

5.10 Oraclize: Many Compiler Warnings

When `externals/oraclizeConnector.sol` is compiled with the requisite compiler version for the rest of the code (`solc 0.4.25`), many compiler warnings are generated. Moreover, the copyright header in the file suggested that it has not been updated in at least a year. Neither of these facts are positive findings in terms of the security posture of the contract. See section 6.1 for a description of an alternative architecture that doesn't depend on Oraclize.

Chapter 6

Alternatives

6.1 Transfer Counter-signatures

One way to introduce an explicit "co-signer" for a wallet is to wrap `transfer()` functions in a function that requires the transaction arguments include a signature of the co-signer.

The example below implements a `transfer()` function that requires that a signature from Token Group Ltd. be present in order for assets to be transferred. Thus, Token Group Ltd.'s validation logic (address whitelisting, fraud/anomaly detection, spending limits, etc.) can be performed in an out-of-band manner. An additional benefit of this architecture is that Token Group Ltd. never needs to pay gas for transactions; it merely needs to provide signatures for transactions that meet its published criteria.

A small variation on this proposed architecture would be to invert the roles of transaction submitter and counter-signer: Token Group Ltd. would submit the Ethereum transaction (and thus pay for gas), while the end-user would only need to provide signatures.

A production solution would have to include a feature to detect inactivity so that key disclosure (for either the end-user's key, or Token Group Ltd's key) would not indefinitely freeze funds.

This example is for demonstration purposes only. It may contain bugs.

```
uint counter;      // newest approved transaction number
address tokencard; // TokenCard-controlled signing key

function transfer(address asset, address dst,
                  uint amount, uint nonce,
                  bytes32 sigR, bytes32 sigS, uint8 sigV)
{
        require(msg.sender == owner);

        // sanity check for signature paramters
```

```
        require(uint256(sigR) < CURVE_ORDER);
        require(uint256(sigS) < CURVE_HALF_ORDER);

        // every transaction submitted must be 'older'
        // than the previous one
        require(nonce > counter);

        // check the signature, which encompasses all of
        // the parameters of the transaction
        // NOTE: a hardened implementation would
        // include a hard-coded signature prefix here
        // to mitigate signing oracle attacks
        bytes32 hash = keccak256(msg.sender, asset, dst, amount, nonce);
        require(ecrecover(hash, sigV, sigR, sigS) == tokencard);

        // require future transfers to be signed with a higher nonce
        counter = nonce;

        // actually do the asset transfer
        if (asset == 0) {
                dst.transfer(amount);
        } else {
                require(Token(asset).transfer(dst, amount));
        }
}
```