Wallet and Oracle Audit - Phil - Rebuttal

In this document we present the various responses to the issues highlighted in the original version of the Nov 2018 Wallet Smart Contract Audit.

Specific Responses

In this section the numbers in the section headings are used to refer to section headings in the Smart Contract Audit Report.

3.1 Submit Confirm Pattern Race Condition

Valid point but misses the fact that we already accounted for this case by introducing a `require(!operationSubmitted)` before each of the 2FA submit methods.

https://github.com/tokencard/contracts/blob/master/test/wallet/whitelist_test.go#L171-L175

See unit tests for example that test for the race condition in the below lines:

https://github.com/tokencard/contracts/blob/master/test/wallet/whitelist_test.go#L192-L199

3.1.1 Mitigation: Lamport Clocks

Interesting solution for the above problem, but since it's already addressed, there's no need for a different solution.

It is noted though that our implementation will be exploitable by a re-org. Furthermore it is important to note that, similar to our solution, the Lamport Clocks proposed approach doesn't address the reorg issue either. The `counter` isn't enough to make this pattern robust to re-orgs, we will be extending this idea by including the actual amounts in the events and the subsequent `confirmation` transaction. We are going to have to come up with a solution to get an "amount" value for the `mapping` in the whitelist, some sort of "hash" of the `mapping`?

4.1 Unmitigated ECDSA Signature Malleability

This would require that we modify the `oraclizeAPI_0.4.25.sol` contract to make the `ecrecover()` wrapper implement the countermeasures mentioned in this section.

We have passed this onto the Oraclize team, they are looking into it.

Question: What the worst that can happen to us here?

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

Valid point, we should think of a way to not have nested for loops so that we know we won't get to a point where we run out of block gas limit or execution time.

Implementing extra logic as suggested in the audit document may need further auditing or introduce new bugs. From our perspective it all depends on how often we use the remove tokens function. We are not expecting to use it to remove all the tokens at any point in time. If we need to do it, we can always send more than one (expensive indeed) transactions.

5.1 Oracle: `updateTokens()` wants JSON

This is implemented as is so that we can verify the signature returned by the CryptoCompare API. In order to do this we need the whole JSON body returned in the callback.

By taking the advice of the auditor and selecting a specific field in the JSON response e.g. `json(https://foo.com).bar` would mean we can't verify the signature. And we would be trusting oraclize's integrity. We have taken this approach to minimise the entities we need to trust.

It is noted that we could always use the JSON parsing approach described and then by concating strings regenerate the string to verify the signature, but that didn't seem sensible.

5.2 Oracle: `verifyProof()` Parses Untrusted Input

When we receive the `__callback()` response from Oraclize, we first parse the HTTP headers and then verify the signature, we need to do it in this order since we can't verify the signature unless we have the individual components that were signed. That said, we could think about not returning `false` when the callback fails, and instead use `require()` which will `revert()` the callback transaction and not emit events - but arguably is less bug prone.

Given the nature of the _callback, we purposefully decided to implement the `verifyProof()` method like this so that we can emit events in the most informative way.

5.3 Oracle: Unhandled Error Conditions in String Parsing

The `parseInt()` function returns an integer regardless of input provided. This function comes from the `oraclizeAPI_0.4.25.sol` contract. We could either write tests for it to make sure

that nothing will go wrong when we pass bad input or alternatively we can add some input validation code, which would require modifying the Oraclize contract.

That said we have additional checks which are performed later on in the processing after that e.g. if (day > 31 || day < 1) {return (false, 0);}

But agreed, we should add some tests to see if the code is exploitable.

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

I don't think we need to support the full date standard, we are mostly constrained to what CryptoCompare returns. And if the API date header changes, we can easily deploy a new oracle.

We started off doing full RHC2616 DateTime parsing but we didn't most of the information in the DateTime standard, and we changed this to this simpler datetime parser on purpose as we didn't need more granularity for our purposes.

The information loss in our implementation was deemed worth it due to the gains we made in parsing simplicity.

5.5 All: Unnecessary Accessor Functions

Currently we use the pattern where data that is supposed to be encapsulated inside a contract class is declared as `private` and in order to access the variable from a child contract we call public getters and setters. This makes it clear that the variable isn't supposed to be accessed/modified by the child contract directly, but only through the specified methods.

5.6 Date: Many Unused Constants

Good point, we should probably remove the excessive constants that we don't use. This will also save on deployment costs.

They are there because they were used by the previous version of the datetime parser and were never eliminated, they should be removed.

5.7 Controller: Key Disclosure Attack Surface

I think we are already aware of this, that we should most likely have a controller key hierarchy of some sort and also make sure the ENS key is secure. We have plans to create a controller key hierarchy.

5.8 Oraclize: DER-encoded Signatures

The `verifySig()` function in the `oraclizeAPI_0.4.25.sol` contract takes DER-encoded signatures. But since we never call that function in the `oracle.sol` contract, this is likely not an issue for us.

This is true, we just call ecrecovery() which is a wrapper for ecrecover(). The sig is actually bruteforced (in order to obtain V) in the oraclize backend (inside info).

5.9 Vault: Access of `_spendAvailable`

As we discussed earlier this could be made into a setter method instead of making this variable available to the Vault contract directly - to be consistent with the above mentioned encapsulation approach.

5.10 Oraclize: Many Compiler Warnings

The Oraclize contract has been written more than a year ago, and creates a lot of warnings, we could potentially address some of them but that would require modifying the `oraclizeConnector.sol` contract.

We have spent some time eyeballing the warnings and we seem comfortable with them as is.

6.1 Transfer Counter-signatures

Off-chain-based approach for doing 2FA in the wallet contract - we could consider doing it this way. Thanks !