# TokenCard

## Security Assessment

**May 3, 2019**

Prepared For:
Mischa Tuffield | *TokenCard*
mischa@tokencard.io

Prepared By:
Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

John Dunlap | *Trail of Bits*
john.dunlap@trailofbits.com

## Executive Summary

From April 15 through May 3 2019, TokenCard engaged with Trail of Bits to review the security of their smart contract wallet and iOS application. Trail of Bits conducted this assessment over the course of four person-weeks with two engineers working from commit hash `b99b7d1670f9ad7b90335e8391fe63fd7e20de9b` from the TokenCard `contracts` repository.

During the first week, we began to familiarize ourselves with the Solidity smart contracts used by the TokenCard system. This consisted of manual review focused on the Wallet contract as well as automated analysis of some of the libraries used by the smart contracts. In the second week, we continued our review of the smart contracts and conducted a brief review of the provided development build of the iOS application. For the final week of the assessment, Trail of Bits concluded our manual review of the TokenCard system, continued to build out a suite of Echidna tests of various properties as well as reviewed a fix for TOB-TokenCard-001 and a work in progress migration from v0.4.x to the v0.5.x branch of the Solidity compiler.

In total, Trail of Bits identified 12 issues, ranging from medium- to low-severity with one additional finding of undetermined severity. Six issues—two medium- and four low-severity—were the result of improper data validation by the contracts. A medium-severity finding in the address whitelist contract allows the contract owner to add her own address. Two low-severity potential denials of service were found relating to a malicious price source via the oracle as well as the arithmetic computations of two wallet functions. An additional low-severity finding relates to the potential for interoperability issues when working with large JSON integer values. (Integer handling is not well defined by the JSON specification.) The final low-severity finding is the result of including external dependencies directly in the Github repository which makes tracking upstream changes difficult or impossible. The remaining finding of undetermined severity pertains to compiling the Solidity code with compiler optimizations enabled.

The code reviewed shows that the TokenCard team is aware of the most common smart contract pitfalls and performed numerous tests of the codebase. We recommend TokenCard remedy all of the reported issues and improve testing around untrusted input validation. This was a common class of issue encountered in the code. This testing could also be supplemented with Manticore and Echidna tests to verify correctness. Additionally, continue to work toward migrating the codebase to use the v0.5.x branch of the Solidity compiler before launch; the v0.4.x branch used by the application is no longer officially supported.

## Project Dashboard

### Application Summary

| | |
|---|---|
| Name | TokenCard |
| Version | `b99b7d1670f9ad7b90335e8391fe63fd7e20de9b` |
| Type | Solidity Smart Contracts & iOS Application |
| Platforms | Ethereum, iOS |

### Engagement Summary

| | |
|---|---|
| Dates | April 15 - May 3, 2019 |
| Method | Whitebox |
| Consultants Engaged | 2 |
| Level of Effort | 4 person-weeks |

### Vulnerability Summary

| | | |
|---|---|---|
| Total High-Severity Issues | 0 | |
| Total Medium-Severity Issues | 3 | ■■■ |
| Total Low-Severity Issues | 8 | ■■■■■■■■ |
| Total Informational-Severity Issues | 0 | |
| Total Undetermined-Severity Issues | 1 | ■ |
| Total | 12 | |

### Category Breakdown

| | | |
|---|---|---|
| Data Validation | 6 | ■■■■■■ |
| Undefined Behavior | 2 | ■■ |
| Patching | 1 | ■ |
| Denial of Service | 2 | ■■ |
| Access Controls | 1 | ■ |
| Total | 12 | |

## Engagement Goals

The engagement was scoped to provide a security assessment of the TokenCard Wallet and ancillary smart contracts in the `contracts` repository as well as the companion iOS mobile application.

Specifically, we sought to answer the following questions:

- Is it possible to trap funds in the Wallet?
- Can TokenCard access user wallet funds?
- Are access controls well-defined?
- Are there any potential arithmetic issues when converting tokens to base units?
- What influence does the oracle have over user funds?

## Coverage

This review included the Wallet contract and all other TokenCard protocol smart contracts it interacts with. The specific version of the codebase used for the assessment came from commit hash `b99b7d1670f9ad7b90335e8391fe63fd7e20de9b` of the TokenCard `contracts` Github repository. Additionally, we briefly reviewed a development build of the companion iOS application.

Contracts were reviewed for common Solidity flaws, such as integer overflows, reentrancy vulnerabilities, and unprotected functions.

**Access controls.** Many parts of the system expose privileged functionality, such as adding to the wallet's payee whitelist or transferring funds to be loaded to a TokenCard card. These functions were reviewed to ensure they can be triggered only by the intended actors.

**Arithmetic.** We reviewed these calculations for logical consistency as well as potential scenarios where reverts due to overflow may negatively impact use of the wallet.

**Interactions with external contracts.** The TokenCard wallet has explicit support for holding both ether as well as any ERC20 token. We reviewed the contract's interactions with these external, untrusted token contracts, to ensure proper behavior even in the case of non-ERC20-compliant tokens.

**Parsing third-party input.** The TokenCard oracle uses the Oraclize service to query and parse exchange rates on-chain. To support this, the TokenCard team has had to include libraries for parsing JSON, base64 and dates as part of the system. We reviewed this code and considered potential scenarios that may occur upon parsing malicious input data.

**Access to wallet funds.** TokenCard bills its application as having a non-custodial wallet. We sought to verify that TokenCard cannot exert undue influence or otherwise access a user's wallet funds aside from when the user loads her card.

## Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

### Short Term

❑ **Adopt the [OpenZeppelin SafeERC20 library](OpenZeppelin SafeERC20 library) or otherwise add explicit support for ERC20 tokens with incorrect return values.** This will allow the TokenCard wallet to support tokens which deviate from the ERC20 standard.

❑ **Properly document the potential for JSON interoperability issues** and make sure users are aware if they are meant to interact with contracts that parse JSON on-chain.

❑ **Properly document that the `_base64decode` function does not perform input validation** and make sure users are aware that they should provide only valid base64 strings.

❑ **Consider disabling compiler optimizations when building the contracts.** Measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.

❑ **Add contract existence checks for transactions with non-empty data.** This will help prevent users from interacting with contracts they may otherwise believe exist.

❑ **Review all the code based on third-party contracts to check that it includes proper information regarding where it was taken from and whether it was modified.** If possible, include the contract sources as a submodule in your Git repository so that internal path consistency can be maintained and updated periodically.

❑ **Do not allow tokens to be added to the whitelist without carefully reviewing the token `string` value.** This will prevent URL parameter injection in the calls to CryptoCompare and ensure the appropriate exchange rate is returned.

❑ **Add a validation check when initially setting `_licenceAmountScaled`.** This will ensure that the licence amount is always in a safe range, including immediately after contract initialization.

❑ **Properly document that arithmetic overflow may block certain function calls in the wallet from being carried out with large input values** and make sure that users are aware of it.

❑ **When transferring ownership of a wallet, check that the new owner is not on the whitelist.**

❑ **Use Solidity constants (minutes, hours, days, years) to compute timestamps.**

❑ **Develop a plan for a backup oracle** in the event the current approach fails or becomes malicious.

## Long Term

❑ **Beware the idiosyncrasies of ERC20 implementations and document the necessary criteria for a token to be added to the whitelist.**

❑ **Provide a recommended JSON implementation for use when interacting with the TokenCard contracts.** Use Echidna and Manticore to prove that both the recommended implementation and the Solidity implementation handle numbers consistently.

❑ **Implement a thorough validation check in the `_base64decode` function.** The function should revert if it is provided with an invalid input. Use Echidna and Manticore to verify that the validation is working as expected.

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.**

❑ **Avoid low-level calls.** If they are unavoidable, carefully review the Solidity documentation, in particular the Warnings section.

❑ **Use an Ethereum development environment and NPM to manage packages as part of your project.** This will allow the project to more easily keep track of upstream changes in external dependencies.

❑ **Properly document how to sanitize the token names or implement a Solidity function to remove special characters used in URLs.**

❑ **Ensure validation is carried out for relevant parameters when developing new contracts.** This will prevent contracts from being initialized with unsafe parameters.

❑ **Alert the user when transactions fail due to overflow caused by very large input values.** For example, consider refactoring the code to detect this overflow, and using a suitable error message in the revert call.

❑ **Use Manticore and Echidna to ensure the `AddressWhitelist` contract cannot contain the address of the owner.**

❑ **Implement proper validation of dates to avoid accepting impossible dates.** Use Echidna to verify that no invalid inputs are accepted.

❑ **Consider decentralizing the oracle and supporting multiple oracle price sources** to avoid having a single point of failure.

## Findings Summary

| # | Title | Type | Severity |
|---|---|---|---|
| 1 | Wallet and Licence are incompatible with non-standard ERC20 tokens | Data Validation | Low |
| 2 | Parsing large JSON integers could result in interoperability issues | Undefined Behavior | Low |
| 3 | Base64 decoding does not validate its input | Data Validation | Low |
| 4 | Solidity compiler optimizations can be dangerous | Undefined Behavior | Undetermined |
| 5 | Lack of contract existence check may mislead the user about the transaction's result | Data Validation | Medium |
| 6 | Contracts used as dependencies do not track upstream changes | Patching | Low |
| 7 | No sanitization is performed when the Oraclize query is constructed | Data Validation | Medium |
| 8 | _licenceAmountScaled can be incorrectly initialized | Data Validation | Low |
| 9 | Multiplication overflow can block certain wallet operations | Denial of Service | Low |
| 10 | AddressWhitelist owner can be added to the whitelist | Access Controls | Medium |
| 11 | Date validation is insufficient and returns imprecise timestamps | Data Validation | Low |
| 12 | Malicious price source can block withdrawal of funds | Denial of Service | Low |

## 1. Wallet and Licence are incompatible with non-standard ERC20 tokens

| | |
|---|---|
| Severity: Low | Difficulty: Low |
| Type: Data Validation | Finding ID: TOB-TokenCard-001 |
| Target: `wallet.sol, licence.sol` | |

**Description**
The TokenCard contracts are meant to work with any ERC20 token, but several high-profile ERC20 tokens do not correctly implement the ERC20 standard. Therefore, the `Wallet` and `Licence` contracts will not work with these tokens.

The ERC20 standard defines, among others, two transfer functions:

- `transfer(address _to, uint256 _value) public returns (bool success)`
- `transferFrom(address _from, address _to, uint256 _value) public returns (bool success)`

However, several high-profile ERC20 tokens do not return a boolean on one or both of these two functions. Starting from Solidity 0.4.22, the return data size of external calls is checked. As a result, any call to `transfer` or `transferFrom` will fail for ERC20 tokens that have an incorrect return value.

The ERC20 tokens that will not work include `BNB` (#1 in terms of market cap), `OMG` (#6 in terms of market cap) and Oyster Pearl (#33 in terms of market cap).

**Exploit Scenario**
Alice adds `OMG` to her TokenCard wallet. She attempts to `load` these tokens onto her card, but due to the lack of a `return` in the token code, the transaction fails. Once Alice realizes she will not be able to load these tokens, she attempts to withdraw them from her wallet. However, this will also fail using the wallet's built-in `transfer` function. She will instead need to handcraft a call to the token's `transfer` function via the wallet's `executeTransaction` function. This is likely to be an error-prone operation that may result in Alice accidentally sending her tokens to the wrong address.

**Recommendation**
Short term, consider using the OpenZeppelin SafeERC20 library or otherwise adding explicit support for ERC20 tokens with incorrect return values.

Long term, beware the idiosyncrasies of ERC20 implementations and document the necessary criteria for a token to be added to the whitelist.

**References**

- Missing return value bug - At least 130 tokens affected
- Explaining unexpected reverts starting with Solidity 0.4.22

## 2. Parsing large JSON integers could result in interoperability issues

Severity: Low
Type: Undefined Behavior
Target: `internals/parseIntScientific.sol`

Difficulty: Medium
Finding ID: TOB-TokenCard-002

**Description**
The parsing of strings as JSON integers differs from mainstream implementations such as NodeJS and jq.

The JSON standard warns about certain "interoperability problems" in numeric types outside the range [-(2**53)+1, (2**53)-1]. This issue is caused by some widely used JSON implementations employing IEEE 754 (double precision) numbers to implement integer numbers. For instance, the `_parseIntScientific` function shown in Figure 1 parses `"1152921504606846976" (2**60)` as the expected value `1152921504606846976`.

```
/// @notice ParseIntScientific parses a JSON standard - floating point number.
/// @param _inString is input string.
/// @param _magnitudeMult multiplies the number with 10^_magnitudeMult.
function _parseIntScientific(string _inString, uint _magnitudeMult) internal pure returns
(uint) {

    bytes memory inBytes = bytes(_inString);
    uint mint = 0; // the final uint returned
    uint mintDec = 0; // the uint following the decimal point
    …
```

**Figure 1.** Function to parse floating point numbers as integers from strings

However, NodeJS 10 and jq 1.5 parse it as `1152921504606847000`.

**Exploit Scenario**
Alice's code interacts with the TokenCard contracts by sending a string with a very large number. Alice's code interprets this number differently than the TokenCard contract's interpretation, causing unexpected behavior for Alice.

**Recommendation**
Short term, properly document this behavior and make sure users are aware if they are meant to interact with contracts that parse JSON on-chain.

Long term, provide a recommended JSON implementation to use when interacting with the TokenCard contracts. Use Echidna and Manticore to prove that both the recommended implementation and the Solidity implementation handle numbers in a consistent way.

**References**

- RFC 8259, Section 6: numbers in JSON

## 3. Base64 decoding does not validate its input

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-TokenCard-003
Target: `externals/base64.sol`

**Description**
The `_base64decode` function that implements decoding of base64 strings does not properly validate its input.

The function `_base64decode` function shown in Figure 1 is used to decode a base64 string into a list of bytes.

```
/// @return decoded array of bytes.
/// @param _encoded base 64 encoded array of bytes.
function _base64decode(bytes _encoded) internal pure returns (bytes) {
    byte v1;
    byte v2;
    byte v3;
    byte v4;
    uint length = _encoded.length;
    bytes memory result = new bytes(length);
    uint index;

    // base64 encoded strings can't be length 0 and they must be divisble by 4
    require(length > 0  && length % 4 == 0, "invalid base64 encoding");
    …
```

**Figure 1.** Function to decode base64 strings

However, this function does not fail in any way if an invalid base64 input is provided. The only check provided will verify the length of the input as shown in the `require` ( Figure 1). This check is insufficient to properly validate all possible inputs.

**Exploit Scenario**
Alice's code interacts with the TokenCard contracts by sending a base64 string to decode. Her code has a bug and produces an invalid base64 string. This string is incorrectly decoded into a list of bytes by the TokenCard contract, causing unexpected behavior for Alice.

**Recommendation**
Short term, properly document this behavior and make sure users are aware that they should provide only valid base64 strings.

Long term, implement a thorough validation check in the `_base64decode` function. The function should revert if an invalid input is provided. Use Echidna and Manticore to verify that the validation is working as expected.

## 4. Solidity compiler optimizations can be dangerous

Severity: Undetermined
Type: Undefined Behavior
Target: `build.sh`

Difficulty: Low
Finding ID: TOB-TokenCard-004

**Description**
The compilation of the TokenCard smart contracts has optional Solidity compiler optimizations enabled.

There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated `solc-js` compiler used by Truffle and Remix persisted until just a few months ago. The fix for this bug was not reported in the Solidity CHANGELOG.

A compiler audit of Solidity from November, 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is “implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function.” Similar code in other large projects has resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.\

**Exploit Scenario**
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the TokenCard contracts.

**Recommendation**
Short term, measure the gas savings from optimizations, and carefully weigh that against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

## 5. Lack of contract existence check may mislead the user about the transaction's result

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-TokenCard-005
Target: `Wallet.sol`

**Description**

A failure to check for a contract's existence may mislead a user into thinking that a failed transaction was successful.

`Wallet` uses assembly code to execute external transactions as shown in Figure 1.

```
function _externalCall(address _destination, uint _value, uint _dataLength, bytes _data)
private returns (bool) {
    bool result;
    assembly {
        let x := mload(0x40)   // "Allocate" memory for output (0x40 is where "free memory"
pointer is stored by convention)
        let d := add(_data, 32) // First 32 bytes are the padded length of data, so exclude
that
        result := call(
            sub(gas, 34710),   // 34710 is the value that solidity is currently emitting
                               // It includes callGas (700) + callVeryLow (3, to pay for
SUB) + callValueTransferGas (9000) +
                               // callNewAccountGas (25000, in case the destination address
does not exist and needs creating)
            _destination,
            _value,
            d,
            _dataLength,        // Size of the input (in bytes) - this is what fixes the
padding problem
            x,
            0                  // Output is ignored, therefore the output size is zero
        )
    }

    return result;
}
```

**Figure 1.** Function `_externalCall` to execute external transactions.

However, the Solidity documentation warns:

> *The low-level functions call, delegatecall and staticcall return true as their first return value if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.*

As a result, a call to a nonexistent or destructed contract will return success.

A transaction with non-empty data is likely to be meant for a contract. If the contract is destructed or nonexistent, the transaction will be considered successful by mistake.

**Exploit scenario**
Alice and Bob use `Wallet`. Bob has a contract with a payable `receiveFund()` function. Bob submits a call to `receiveFund` with 10 ethers to the multisig wallet. Bob's contract is destructed. Alice is not aware of the state of Bob's contract, and confirms the transaction. As a result, the transaction is executed and the funds are trapped.

**Recommendation**
In the short term, check the contract's existence for transactions with non-empty data.

In the long term, avoid low-level calls. If they are unavoidable, carefully review the Solidity documentation, in particular the Warnings section.

**References**

- Lack of contract's existence check may mislead the user about the transaction's result

## 6. Contracts used as dependencies do not track upstream changes

Severity: Low
Difficulty: Low
Type: Patching
Finding ID: TOB-TokenCard-006
Target: `Wallet.sol, oracle.sol,` contracts in `external`

**Description**
Several third-party contracts or functions are copy-pasted into the TokenCard contracts, including: the strings library, Oraclize contracts (version 0.4.25), Ownable from OpenZeppelin and several functions from Stack Overflow responses. Moreover, in a few cases, the code documentation does not specify the exact revision that was used and if it was modified.
This makes receiving updates and security fixes on these dependencies unreliable as they must be updated manually.

**Exploit Scenario**
A third-party contract used in TokenCard receives an update with a critical fix for a vulnerability. If an attacker detects the use of such contract, it could use the vulnerability against TokenCard.

**Recommendations**
In the short term, review all the code based on third-party contracts to check that it includes proper information regarding where it was taken and whether it was modified. If possible, include the contract sources as a submodule in your Git repository so that internal path consistency can be maintained and updated periodically.

In the long term, use an Ethereum development environment and NPM to manage the package as part of your project.

## 7. No sanitization is performed when the Oraclize query is constructed

Severity: Medium
Type: Data Validation
Target: `oracle.sol`

Difficulty: High
Finding ID: TOB-TokenCard-007

**Description**
An Oraclize URL query is not carefully constructed using the name of the token and can be manipulated using special characters (e.g. `&`, `%`, `#`, etc).

The `oracle` contract defines `_updateTokenRatesList`, a function to construct an Oraclize URL query, execute it and save its result as shown in Figure 1. In order to construct the query, the token symbol is directly concatenated with the URL parameters.

```
/// @notice Re-usable helper function that performs the Oraclize Query for a specific list
of tokens.
/// @param _gasLimit the gas limit is passed, this is used for the Oraclize callback.
/// @param _tokenList the list of tokens that need to be updated.
function _updateTokenRatesList(uint _gasLimit, address[] _tokenList) private {
    // Check if there are any existing tokens.
    if (_tokenList.length == 0) {
        // Emit a query failure event.
        emit FailedUpdateRequest("empty token list");
    // Check if the contract has enough Ether to pay for the query.
    } else if (oraclize_getPrice("URL") * _tokenList.length > address(this).balance) {
        // Emit a query failure event.
        emit FailedUpdateRequest("insufficient balance");
    } else {
        // Set up the cryptocompare API query strings.
        strings.slice memory apiPrefix =
"https://min-api.cryptocompare.com/data/price?fsym=".toSlice();
        strings.slice memory apiSuffix = "&tsyms=ETH&sign=true".toSlice();

        // Create a new oraclize query for each supported token.
        for (uint i = 0; i < _tokenList.length; i++) {
            //token must exist, revert if it doesn't
            (string memory tokenSymbol, , , bool available , , , ) =
_getTokenInfo(_tokenList[i]);
            require(available, "token must be available");
            // Store the token symbol used in the query.
            strings.slice memory symbol = tokenSymbol.toSlice();
            // Create a new oraclize query from the component strings.
            bytes32 queryID = oraclize_query("URL",
apiPrefix.concat(symbol).toSlice().concat(apiSuffix), _gasLimit);
            // Store the query ID together with the associated token address.
            _queryToToken[queryID] = _tokenList[i];
            // Emit the query success event.
            emit RequestedUpdate(symbol.toString());
        }
    }
}
```

**Figure 1.** Function _updateTokenRatesList queries oraclize and saves the result

The token symbol is specified when a new token is added into the whitelist by a controller. However, it is unclear how that string should be sanitized or which characters are allowed in the name.

**Exploit Scenario**
An attacker manages to add a new token into the whitelist. He can manipulate the price of that token if he uses a specially crafted token name. For instance, using "`DAI&tsyms=ETH#`" will trick the CryptoCompare web service to return the value of DAI, instead of his token.

**Recommendation**
Short term, do not allow any user to add tokens into the whitelist without carefully reviewing the token string.

Long term, properly document how to sanitize the token names or implement a Solidity function to remove special characters used in URLs.

## 8. _licenceAmountScaled can be incorrectly initialized

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-TokenCard-008
Target: `licence.sol`

### Description

The `licence` contract collects a percentage fee when a user loads her card. The fee percentage is recorded in the `_licenceAmountScaled` state variable.

This state variable can be initialized during the contract deployment as shown in Figure 1 or updated using `updateLicenceAmount` as shown in Figure 2.

```
   constructor(address _owner_, bool _transferable_, uint _licence_, address _float_,
address _holder_, address _tknAddress_) Ownable(_owner_, _transferable_) public {
        _licenceAmountScaled = _licence_;
        _cryptoFloat = _float_;
        _tokenHolder = _holder_;
        if (_tknAddress_ != address(0)) {
            _tknContractAddress = _tknAddress_;
        }
    }
```

*Figure 1:* `constructor` of `licence` *contract*

```
   function updateLicenceAmount(uint _newAmount) external onlyDAO {
        require(1 <= _newAmount && _newAmount <= MAX_AMOUNT_SCALE, "licence amount out of
range");
        _licenceAmountScaled = _newAmount;
        emit UpdatedLicenceAmount(_newAmount);
    }
```

*Figure 2:* `updateLicenceAmount` *function*

When `_licenceAmountScaled` is updated after initialization, the contract performs a sanity check to make sure the new value is a reasonable amount. However, `_licenceAmountScaled` can be set to any unsigned integer value in the call to the contract constructor. This allows for no licence fee to be set upon initialization, which is explicitly disallowed when updating the licence amount.

### Exploit Scenario

When deploying the contract, there is a typo in the `licence` parameter. Instead of a 0.5% fee, the contract is deployed with a 50% fee. Any users loading their cards before this mistake is noticed will pay excessively high fees.

### Recommendation

Short term, add a validation check when initially setting `_licenceAmountScaled`.

Long term, ensure validation is carried out for relevant parameters when developing new contracts.

## 9. Multiplication overflow can block certain wallet operations

Severity: Low
Type: Denial of Service
Target: `wallet.sol`

Difficulty: High
Finding ID: TOB-TokenCard-009

**Description**
Some wallet transfers involving a very large number of tokens may trigger an integer overflow that will revert the transaction.

The `convertToEther` and `convertToStablecoin` functions are designed to compute the value of certain amounts of ERC20 tokens in terms of tokens or stablecoins, respectively, as shown in Figure 1 and Figure 2.

```
    function convertToEther(address _token, uint _amount) public view returns (uint) {
         // Store the token in memory to save map entry lookup gas.
         (,uint256 magnitude, uint256 rate, bool available, , , ) = _getTokenInfo(_token);
         // If the token exists require that its rate is not zero.
         if (available) {
             require(rate != 0, "token rate is 0");
             // Safely convert the token amount to ether based on the exchange rate.
             return _amount.mul(rate).div(magnitude);
         }
         return 0;
     }
```

*Figure 1:* `convertToEther` *function*

```
    function convertToStablecoin(address _token, uint _amount) public view returns (uint) {
         //avoid the unnecessary calculations if the token to be loaded is the stablecoin
itself
         if (_token == _stablecoin()) {
             return _amount;
         }
         //0x0 represents ether
         if (_token != address(0)) {
             //convert to eth first, same as convertToEther()
             // Store the token in memory to save map entry lookup gas.
             (,uint256 magnitude, uint256 rate, bool available, , , ) =
_getTokenInfo(_token);
             // require that token both exists in the whitelist and its rate is not zero.
             require(available, "token is not available");
             require(rate != 0, "token rate is 0");
             // Safely convert the token amount to ether based on the exchange rate.
             _amount = _amount.mul(rate).div(magnitude);
         }
         ...
   }
```

*Figure 1:* `convertToStablecoin` *function*

However, they can trigger similar integer overflows when they multiply the amount of tokens to transfer by its `rate`. Since the `Wallet` contract is using SafeMath for its arithmetic operations, the integer overflow triggers a revert, as shown in Figure 3.

```
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
  // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
  // benefit is lost if 'b' is also tested.
  // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522
  if (a == 0) {
    return 0;
  }

  uint256 c = a * b;
  require(c / a == b);

  return c;
}
```

*Figure 2:* `mul(int256 a, int256 b)` *function*

**Exploit Scenario**
A user wants to transfer a very large number of tokens using his `Wallet` contract. However, he is unable to do so because that transaction will trigger a revert. He may not realize that transferring a smaller amount will work around this limitation. He stops using the `Wallet` contract.

**Recommendation**
Short term, properly document this limitation and make sure that users are aware of it.

Long term, alert the user that the transfer fails because of a very large number of tokens. For instance, refactoring the code to detect this overflow and using a suitable error message in revert call.

## 10. AddressWhitelist owner can be added to the whitelist

Severity: Medium
Difficulty: High
Type: Access Controls
Finding ID: TOB-TokenCard-10
Target: `wallet.sol`

**Description**
The `AddressWhitelist` contract implements a payee whitelist used to bypass the daily limits imposed by the `Wallet` contract. `AddressWhitelist` contains a modifier, `hasNoOwnerOrZeroAddress`, intended to prevent the contract owner or the zero address from being added to the whitelist. However, it is possible to add the owner to the whitelist by first by adding another address controlled by the current owner address to the whitelist and then transferring ownership of the contract to that address.

**Exploit Scenario**
Alice controls two Ethereum addresses: she uses the first one to send transactions and the second one to keep her funds. In order to use a `AddressWhitelist` contract, she uses the first address as owner and adds the second one to her whitelist. Later, she decides to keep only the second account, so she transfers the ownership of her `AddressWhitelist` contract to the second account. If Bob, an attacker, gains access to Alice's private key then he will be able to exfiltrate all of Alice's funds immediately.

**Recommendation**
When transferring ownership, check that the new owner is not on the whitelist.

In the long term, use Manticore and Echidna to ensure the `AddressWhitelist` contract cannot contain the address of the owner.

## 11. Date validation is insufficient and returns imprecise timestamps

Severity: Low
Type: Data Validation
Target: `oracle.sol`

Difficulty: Low
Finding ID: TOB-TokenCard-011

**Description**
The `_verifyDate` function that implements date parsing from strings does not properly validate its input.

The function `_verifyDate` shown in Figure 1 is used to compute timestamps from a string representing a date (e.g. "`12 Sep 2018 15:18:14`").

```
function _verifyDate(string _dateHeader, uint _lastUpdate) private pure returns (bool, uint)
{

        // called by verifyProof(), _dateHeader is always a string of length = 20
        assert(abi.encodePacked(_dateHeader).length == 20);

        // Split the date string and get individual date components.
        strings.slice memory date = _dateHeader.toSlice();
        strings.slice memory timeDelimiter = ":".toSlice();
        strings.slice memory dateDelimiter = " ".toSlice();

        uint day = _parseIntScientific(date.split(dateDelimiter).toString());
        require(day > 0 && day < 32, "day error");

        uint month = _monthToNumber(date.split(dateDelimiter).toString());
        require(month > 0 && month < 13, "month error");

        uint year = _parseIntScientific(date.split(dateDelimiter).toString());
        require(year > 2017 && year < 3000, "year error");

        uint hour = _parseIntScientific(date.split(timeDelimiter).toString());
        require(hour < 25, "hour error");

        uint minute = _parseIntScientific(date.split(timeDelimiter).toString());
        require(minute < 60, "minute error");

        uint second = _parseIntScientific(date.split(timeDelimiter).toString());
        require(second < 60, "second error");

        uint timestamp = year * (10 ** 10) + month * (10 ** 8) + day * (10 ** 6) + hour *
(10 ** 4) + minute * (10 ** 2) + second;

        return (timestamp > _lastUpdate, timestamp);
}
```

**Figure 1.** Function to compute a timestamp from a string

However, this function does not fail in any way if an invalid (e.g. "`31 Feb 2018 15:18:14`") or even malformed date is provided (e.g. "`1.2 Sep 2018 15:18:0`").

Moreover, the computation of the timestamp is based on very imprecise coefficients. For instance, the number of seconds in a year is approximated as `10**10` when Solidity provides a more precise constant using `"1 years"`, which is equal to `31536000` seconds.

**Exploit Scenario**
The Oraclize API returns an invalid date with an otherwise valid signature. `_verifyDate` parses this malformed date string as sometime far in the future. Future calls to `_verifyDate` will fail since their correct timestamps will appear to be from the past. Token rate updates will revert until the Oracle is redeployed, potentially leading to a disruption in service for users trying to load their cards.

**Recommendation**
Short term, use a more suitable approximation to compute the timestamp using Solidity constants (`minutes`, `hours`, `days`, `years`).

Long term, implement proper validation of dates to avoid accepting impossible dates. Use Echidna to verify that no invalid inputs are accepted.

## 12. Malicious price source can block withdrawal of funds

Severity: Low
Difficulty: High
Type: Denial of Service
Finding ID: TOB-TokenCard-12
Target: `oracle.sol`

**Description**
The TokenCard `oracle` interacts with an Oraclize smart contract to facilitate off-chain queries to CryptoCompare in order to obtain token exchange rates. In the event that CryptoCompare is compromised or becomes malicious, transfers of whitelisted tokens out of users' wallets could become impossible until a new oracle is deployed.

**Exploit Scenario**
Alice holds `$TKN` in her TokenCard wallet. She wishes to withdraw it and initiates a transfer. Unbeknown to her, CryptoCompare's signing key had been compromised. During the most recent exchange rate update, an attacker returned an incredibly high exchange rate to the oracle. When Alice attempts to withdraw these tokens, the contract believes this will cross over her daily limit and prevents the transfer.

**Recommendation**
Short term, develop a plan for a backup oracle in the event the current approach fails or becomes malicious.

Long term, consider decentralizing the oracle and supporting multiple oracle price sources to avoid having a single point of failure.

## A. Vulnerability Classifications

| **Vulnerability Classes** | |
|---|---|
| **Class** | **Description** |
| Access Controls | Related to authorization of users and assessment of rights |
| Auditing and Logging | Related to auditing of actions or logging of problems |
| Authentication | Related to the identification of users |
| Configuration | Related to security configurations of servers, devices or software |
| Cryptography | Related to protecting the privacy or integrity of data |
| Data Exposure | Related to unintended exposure of sensitive information |
| Data Validation | Related to improper reliance on the structure or values of data |
| Denial of Service | Related to causing system failure |
| Error Reporting | Related to the reporting of error conditions in a secure fashion |
| Patching | Related to keeping software up to date |
| Session Management | Related to the identification of authenticated users |
| Timing | Related to race conditions, locking or order of operations |
| Undefined Behavior | Related to undefined behavior triggered by the program |

| **Severity Categories** | |
|---|---|
| **Severity** | **Description** |
| Informational | The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth |
| Undetermined | The extent of the risk was not determined during this engagement |
| Low | The risk is relatively small or is not a risk the customer has indicated is important |
| Medium | Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal |

| | |
|---|---|
| | implications for client |
| High | Large numbers of users, very bad for client’s reputation, or serious legal or financial implications |

| **Difficulty Levels** | |
|---|---|
| **Difficulty** | **Description** |
| Undetermined | The difficulty of exploit was not determined during this engagement |
| Low | Commonly exploited, public tools exist or can be scripted that exploit this flaw |
| Medium | Attackers must write an exploit, or need an in-depth knowledge of a complex system |
| High | The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue |

## B. Code Quality

Code-quality findings are included to detail findings which are not immediate security concerns, but could lead to the introduction of vulnerabilities in the future.

**Wallet.sol**

- The addition used to calculate the daily limit time window in the `DailyLimitTrait` contract's `_getAvailableLimit` function does not use `SafeMath` addition.
- The `_bytesToAddress` function does not seem properly tested in any unit tests. Additionally, its origin is unclear. A very similar version can be found in this Stack Overflow answer.

## C. Property-based testing using Echidna

Trail of Bits used Echidna, our property-based testing framework, to find logic errors in the Solidity components of TokenCard.

Trail of Bits developed custom Echidna testing harnesses for the TokenCard contracts. These harnesses initialize a variety of contracts, define a set of important invariants and perform a random sequence of smart contract transactions in an attempt to cause anomalous behavior and break the invariants.

These harnesses include ownership tests (*e.g.*, only the owner can perform certain operations) and on-chain input parsing and validation (*e.g.*, parsing dates, base64 and JSON numbers). Upon completion of the engagement, these harnesses and their related tests will be delivered to the TokenCard team.

For instance, Figure C.1 shows the Solidity source code used to define, initialize, and test ownership of the `licence` contract. The script defines a simple `licence` with an intransferable ownership. The `crytic_nontransferable_owner` function is used as an invariant to check that the ownership was not transferred. An example of how to run this test with Echidna appears in Figure C.2.

```
import "contracts/licence.sol";

contract CryticInterface{
    address internal crytic_owner = address(0x41414141);
    address internal crytic_user = address(0x42424242);
    address internal crytic_attacker = address(0x43434343);
}

contract TEST is CryticInterface, Licence {

    constructor() Licence(crytic_user, false, 10, address(0x0), address(0x0), address(0x0))
public {
    }

    function crytic_nontransferable_owner() public returns (bool) {
        return owner() == crytic_user;
    }

}
```

**Figure C.1:** *`test/verification/echidna/cryticOwnable`*`.sol`, *which initializes the* `licence` *contract test harness.*

```
$ echidna-test test/verification/echidna/cryticOwnable.sol --config
test/verification/echidna/cryticOwnable.yaml TEST
...
Analyzing contract: test/verification/echidna/cryticOwnable.sol:TEST
```

```
crytic_nontransferable_owner: passed! 🎉
```

**Figure C.2:** *An example run of Echidna with the* `licence.sol` *ownership test, including test results.*