PUBLIC

Security Audit

of STACK's Smart Contracts

November 7, 2018

Produced for

by

Table Of Content

Foreword

We first and foremost thank STACK for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

When contracted by STACK, CHAINSECURITY conducted an extensive review of STACK's multi-token smart contracts with the help of several (internal and external) tools. Throughout this process, multiple issues including two critical security issues were uncovered. All of the security issues were promptly addressed by STACK and pose no further security threat.

Furthermore, in cooperation with CHAINSECURITY, STACK made additional optimizations to the contracts' design and their trust model resulting in a more efficient and more trustworthy set of smart contracts.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on September 8, 2018 and updated versions on October 4, 2018:

File	SHA-256 checksum
contracts/ERC20Token.sol	48cfde607ffaff703c8b62109bd58104e55eaa805da61ec24816bc285004b91f
contracts/MultiChannel.sol	28f855fcc00341de96455f516135dd825b3d6f76bd8af29644a41d0c6f2791a3
contracts/MultiLibrary.sol	6ac79f55089c435d6673756f5b0dfddde633b1eac1c1676f4e3aeeb51f5ba0e9
contracts/SafeMathLib.sol	d3544620437747ef2fe53fdc121f1e95085dc748fd129996892b3c2dc69a912b
contracts/StandardToken.sol	0113615c45ff18b8788125fb76db8bec5ffe696abf4563cab9cb5b1bb5b9aded
contracts/Token.sol	f02690d56a5ca284c54a17b9b537c9fba79e83aed6e0edf2a45e94b96356441e
contracts/WETH.sol	0fa579458de86fae9192e9ce429f2414b44e6896d406d19d589a34cae1ba9f0f

Depth of Audit

The scope of the security audit conducted by CHAINSECURITY was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorize the findings into 4 distinct categories, depending on their severities:

- Low: can be considered as less important
-  Medium: should be fixed
-  High: we strongly suggest to fix it before release
-  Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

[1] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

	IMPACT		
LIKELIHOOD	**High**	**Medium**	**Low**
High			
Medium			
Low			

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as ✓ No Issue. If during the course of the audit process, an issue has been addressed technically, we label it as ✓ Fixed, while if it has been addressed otherwise by improving documentation or further specification, we label it as ✓ Addressed. Finally, if an issue is meant to be fixed in the future without immediate changes to the code, we label it as ✓ Acknowledged.

Findings that are labelled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed auditing in order to discover as many vulnerabilities as possible.

System Overview

STACK aims to enable feeless real-time multi-currency transactions. This is achieved by using State Channels which allow the real time exchange of signed transactions off-chain before settling the final amount on the Blockchain.

Participants

A MultiChannel contract is always deployed by STACK and serves as a medium for executing payments in the form of token exchanges between a **user** and a **recipient**. The recipient of a Channel is always STACK. The interaction between the two parties works as follows.

Opening a channel

A State Channel is always opened by STACK upon request of the *user*. To do so, a new MultiChannel contract with the respective initialization parameters is deployed on the Blockchain and a first channel is opened. STACK can open more channels in a MultiChannel contract to support payments using a different token. A channel is always either in the open or closed state.

Operating with channels

As long as a State channel is open, signed transactions can be exchanged off-chain. Both parties keep a local copy of the new state off-chain.

Retrieving funds

If the *user* or STACK wants to retrieve funds, the party signs and publishes a close request to the State Channel. The channel transits into the closed state.

This initiates a contest period during which either party can submit remaining transactions before the final state of the channel is determined.

After the contest period is over, the final amount of the exchange is settled onto the Blockchain. On completion, the channel returns to the open state and is ready to exchange off-chain transactions again.

Stack decides which ERC-20 compliant tokens they add to a Multichannel contract. Additionally they have an implementation wrapping this functionality for Ether.

Best Practices in STACK's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when STACK's project fitted the criterion when the audit started.

Hard Requirements

These requirements ensure that the STACK's project can be audited by ChainSecurity.

- ☑ The code is provided as a Git repository to allow the review of future code changes.
- ☑ Code duplication is minimal, or justified and documented.
- ☑ Libraries are properly referred to as package dependencies, including the specific version(s) that are compatible with STACK's project. No library file is mixed with STACK's own files.
- ☑ The code compiles with the latest Solidity compiler version. If STACK uses an older version, the reasons are documented.
- ☑ There are no compiler warnings, or warnings are documented.

Soft Requirements

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable to STACK.

- ☑ There are migration scripts.
- ☑ There are tests.
- ☐ The tests are related to the migration scripts and a clear separation is made between the two.
- ☑ The tests are easy to run for ChainSecurity, using the documentation provided by STACK.
- ☐ The test coverage is available or can be obtained easily.
- ☑ The output of the build process (including possible flattened files) is not committed to the Git repository.
- ☑ The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that ChainSecurity should assume correct and out of scope.
- ☑ There is no dead code.
- ☑ The code is well documented.
- ☑ The high-level specification is thorough and allow a quick understanding of the project without looking at the code.
- ☑ Both the code documentation and the high-level specification are up to date with respect to the code version ChainSecurity audits.
- ☑ There are no getter functions for public variables, or the reason why these getters are in the code is given.
- ☑ Function are grouped together according either to the Solidity guidelines[2], or to their functionality.

[2] https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions

Security Issues

In the following, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

Unrestricted ether flow C ✓ Fixed

The execution of ether flows should be restricted to an authorized set of users. However, in the following code there are no checks in place on the `msg.sender`. Since the function is also marked as `public`, anyone can call it with freely chosen parameters and execute a WETH transfer from an arbitrary source address to an arbitrary destination.

```
65  function send(address src, address dst, uint wad) public returns(bool) {
66    require(balances[src]>=wad);
67    balances[src] -= wad;
68    dst.transfer(wad);
69    return true;
70  }
```

contracts/WETH.sol

Likelihood: High
Impact: High

Fixed: STACK added `require(msg.sender == src)`. This restricts the transfer of WETH using this function to the authorized user.

Outdated and mixed compiler versions M ✓ Fixed

STACK contracts do not make use of the newest compiler which would be the version `0.4.25`. Additionally, the contracts use different compiler versions, with one of them being very old (e.g. `0.4.18` in `WETH.sol`). Without a documented reason, the newest compiler version should be used homogenously.

Likelihood: Medium
Impact: Medium

Fixed: All files have been updated to compiler version 0.4.25.

Missing `transfer` check M ✓ Addressed

STACK's `StandardToken` implements the `transfer` function which is defined in the `ERC20` interface. However, following best practices, a check should be introduced to prevent users from accidentally burning tokens:

```
require(_to != address(0));
```

Likelihood: Medium
Impact: Medium

Addressed: STACK uses these as place-holders for the actual STK contract on mainnet. The actual STK Token Contracts deployed on mainnet have been audited by TokenMarket.

Users may unintentionally lose ETH M ✓ Fixed

Contract users may unintentionally transmit all Ether of their MultiChannel contract to an ERC-20 token contract: When using WETH tokens, one of the participants needs to deposit ETH into the WETH contract. This is done by sending ETH to the empty callback function of the `MultiChannel` contract, the only payable function. The intention is that this Ether is then sent to the WETH contract by calling `deposit(_address)`.

```
154 function deposit(address _addressOfWETH, uint gasAmount)
155 external
156 channelExists(_addressOfWETH)
157 {
158     require(recipientAddress == msg.sender || channels[_addressOfWETH].
          userAddress_ == msg.sender);
159     require(_addressOfWETH.call.value(address(this).balance).gas(gasAmount)());
```

MultiChannel.sol

`deposit()` however only checks if `channelExists(_address)` and (after it checks the `msg.sender` is authorized) transmits all the ETH to the given token contract.

Thus users may unintentionally transmit all the Ether of their contract to the ERC-20 token contract, as most token contracts won't handle this case correctly and instead the ETH will be lost.

Likelihood: Low
Impact: High

Fixed: STACK implemented a new design where users must validate an address, this prevents unintentional transfers of Ethers to an ERC-20 contract.

Unaddressed compiler warnings M ✓ Fixed

The STACK contracts contain several unaddressed compiler warnings.

- In the `MultiChannel` contract the variable `channel` is implicitly declared as a `storage` pointer. Given the use case, it should be explicitly declared as `memory` variable.
- In the `MultiLibrary`, for the usage of `keccak256` it is recommended to use the `abi.encodePacked` function to encode the input data.

These should be either be documented or fixed.
Likelihood: High
Impact: Low

Fixed: STACK fixed this by implementing the recommendations mentioned above.

Missing Input Validation M ✓ Addressed

The contracts implicitly make the assumption that `signerAddress_` and `userAddress_` are not `0x0` and are not equal to `recipientAddress_`. To ensure the correctness of this assumptions, these properties should be checked in the `constructor` and the `addChannel` of `MultiChannel`. If these properties do not hold, several problems can arise.

Likelihood: Low
Impact: High

Addressed: STACK addressed this issue in their backend, not in the smart contract, by adding following checks: `userAddress = address(0)`, `signerAddress = address(0)`, `recipientAddress != userAddress` and `recipientAddress != signerAddress`.

Settlement Timeout has to be set appropriately M ✓ Addressed

The settlement timeout has to be set sufficiently large to ensure that STACK has sufficient time to update a potentially outdated state. In particular, the user can always call `closeWithoutSignature()`, which keeps the `owedAmount` at `0` and thereby potentially returns the complete funds to the user.

In case of small timeouts, the user therefore has an incentive to flood the network with transactions in order to block STACK's transaction from succeeding. However, this quickly becomes non-economical if STACK is aware of the situation and sets a reasonable timeout.

Likelihood: Low
Impact: High

Addressed: STACK is aware of this and their back-end will choose a suitable timeout.

Missing input validation for Timeout L ✓ Addressed

The timeout is used to determine the status of a channel, e.g.

```
14      modifier channelExists(address addressOfToken)
15      {
16          require(channels[addressOfToken].timeout_ > uint(0));
17          _;
18      }
```

MultiChannel.sol

However, whenever the timeout is set, its value is not validated. Hence, an accidental mistake could have consequences for the security of the following channel operations.

Likelihood: Low
Impact: Medium

Addressed: STACK added a check in the backend to ensure `timeout > uint(0)`

Requirements for Used Tokens ✓ Acknowledged

STACK relies on the correctness of the used tokens for the correctness of their system. Any malicious tokens, where token balances can be manipulated or where ERC20 functions are incorrectly implemented would jeopardize the funds of users and STACK itself.

Furthermore additional care is required when dealing with the following token features:

- Inflationary/Deflationary Tokens (i.e. the balance can change without additional transfers)
- Tokens with transfer fees (break the current settlement process in case `shouldReturn_ == true`).
- Tokens with blacklists/whitelists (a blacklisted user blocks settlement for STACK)
- Built-in locking/vesting schemes

Acknowledged: STACK is aware of this and will take caution when adding addition tokens into the channels. Currently STACK uses a list of whitelisted tokens.

Trust Issues

The issues described in this section are not security issues but describe functionality which is not fixed inside the smart contract and hence requires additional trust into STACK, including in STACK's ability to deal with such powers appropriately.

STACK can steal all ETH H ✓ Fixed

In case the WETH token is used, STACK can steal all ETH. The sequence is as follows:

1. The WETH channel is added.
2. The user sends some ETH into the contract.
3. STACK adds another channel which is not really a token but just an Ethereum account under STACK's control.
4. STACK steals all the ETH by calling `deposit()` with the address of the new channel. They get transmitted to the previously specified account.

This issue is related to the issue, that users may unintentionally lose ETH. However, in this case (which requires malicious intent or loss of control by STACK) user funds can not only be stolen but lost.

Fixed: STACK introduced an additional signature verification inside the `deposit` function. This signature verification ensures that `deposit` can only be called on tokens that were previously authorized by the user and hence removes the problem that it can be called on arbitrary tokens.

STACK needs to alert users M ✓ Acknowledged

As most users will interact with the smart contracts through dApps controlled by STACK, they have to alert the user to important events and send the correct information.

In theory, STACK could close channels with outdated states and not notify users, which are unlikely to monitor the blockchain for the matching events. Hence, STACK has to ensure simple, correct and transparent settlement procedures for users.

Acknowledged: Events are emitted in the STACK's smart contracts and the UI will be planned accordingly.

In Case of Private Key Compromise ✓ Acknowledged

STACK asked ChainSecurity to investigate the case of key compromise.

Apart from the previously mentioned issues the following would happen in case STACK's private key would be compromised. Using the key, arbitrary statements could be signed and hence all channels could be closed in such a way that all tokens would flow to the users. Hence, STACK would use all funds.

This would be possible as users could use the key to sign that statements saying that their owed amounts are 0, consequently close and settle all channels and extract all funds.

Acknowledged: STACK understands the repercussions of private key compromise and will ensue to keep this private keys safe by storing them in cold storage.

Design Issues

The points listed here are general recommendations about the design and style of STACK's project. They highlight possible ways for STACK to further improve the code.

No SafeMath in WETH M ✓ Fixed

To further ensure the correctness of token balances SafeMath libraries can be used for token balance calculations. This is not the case in STACK's wrapped ETH token.

Fixed: STACK solved this by updating the WETH contract to use SafeMath for any interaction with `uint`.

Design of `transfer` in Wrapped ETH token M ✓ Addressed

It is unclear why the `transfer` function of the wrapped ETH token unwraps the token value and transfers underlying ETH. This does not match the expectations of a `transfer` function for a token and at the very least should be explicitly documented.

Fixed: STACK improved the documentation in `WETH.sol` clarifying the functionality of these functions.

Struct `MultiChannelData` can be optimized L

The following struct is declared in the `MultiLibrary`:

```
10    struct MultiChannelData {
11        ERC20Token token_;
12        address userAddress_;
13        address signerAddress_;
14        address recipientAddress_;
15        uint timeout_;
16        uint amountOwed_;
17        uint closedBlock_;
18        uint closedNonce_;
19        bool shouldReturn_;
20     }
```

Multilibrary.sol

The storage of Ethereum is organized in slots of 256 bits. STACK's current implementation uses nine storage slots to store one `struct MultiChannelData`. This has to be paid for by STACK in terms of gas every time a new channel is added.

`uint` is an alias for `uint256` which is 256-bit wide and can store numbers up to `(2^256)− 1` . The fields `timeout_` and `closedBlock_` which both contain block numbers, can be stored in a significantly smaller datatype, e.g. `uint32`.

Note that the solidity compiler is currently unable to optimize the order of the fields, thus this needs to be considered and optimized manually.[3]

Fields of the `address` datatype are 160-bit wide, leaving 96 bit unused if stored alone in a storage slot.

By using `uint32` for `closedBlock_` and `timeout_`, furthermore rearranging the fields, the storage slots required can be reduced from nine to six:

```
    struct MultiChannelData {
        ERC20Token token_;
        uint32 timeout_;
        address userAddress_;
        uint32 closedBlock_;
```

[3] https://medium.com/@novablitz/storing-structs-is-costing-you-gas-774da988895e

```
        address signerAddress_;
        bool shouldReturn_;
        address recipientAddress_;
        uint amountOwed_;
        uint closedNonce_;
    }
```

Note that code interacting with changed fields of this struct will need to be adapted as well, to work with `uint32` instead of `uint256`.

Further optimization are possible, e.g. STACK might reconsider the actual size needed for `closedNonce_`.

This simple optimizations result in significant amounts of gas saved for STACK over the lifetime of the `MultiLibrary` contract.

Fixed: STACK has performed the optimization by resizing the attributes `timeout_` and `closedBlock_` and reordering the attributes within the struct. Hence the storage consumption was reduced from nine slots to six slots, which implies significant gas savings.

Inconsistent events in Wrapped ETH Token M ✓ Acknowledged

Sometimes the wrapped ETH token emits `Transfer` events and sometimes it does not. It would make sense to make this consistent in order for external tools to show correct token balances.

Acknowledged: STACK explains that the `deposit` function returns `true/false` depending on execution which will waterfall onto the `deposit` function in `MultiLibrary`. The `send` function in `WETH.sol` now emits a `Transfer` event instead of returning a `boolean`.

Deprecated keyword `constant` used for functions L ✓ Addressed

In the `StandardToken` contract the visibility of the functions `balanceOf`, `allowance` and `totalSupply` is declared with the deprecated keyword `constant`. The same holds true for these functions in the corresponding interface `Token.sol`. Their visibility should be described with the supported keyword `view`.

Addressed: STACK states the `StandardToken` contract is a placeholder for the STK Token already deployed on mainnet.

`approve` race condition L ✓ Addressed

STACK does not mention the possibility of a race condition on the `approve` method of its `StandardToken` contract. Through reordering of transactions a successful attack can be performed, allowing to spend an allowance higher than thought[4]. STACK should follow best practices and provide an appropriate comment before the function, as is done in e.g. the OpenZeppelin library[5].

In addition, two extra functions - `increaseApproval` and `decreaseApproval` can be introduced as is currently common and recommended by the OpenZeppelin reference implementation[6].

Addressed: STACK states the `StandardToken` contract is a placeholder for the STK Token already deployed on mainnet, thus the issue mentioned above is not relevant.

Locked Tokens inside WETH Contract L ✓ Acknowledged

The WETH contract can receive ERC-20 tokens. Numerous historic cases have shown that accidental ERC-20 transfers to token contracts occur frequently. Currently, such ERC-20 tokens would be locked and lost.

Acknowledged: STACK acknowledged to be aware of this.

[4] For details see the ERC20 standard: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#approve
[5] https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol#L73
[6] https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/ERC20.sol#L120-L157

`StandardToken` is not ERC20 compliant L ✓ Addressed

STACK's `StandardToken` does not meet the required specification. As per the ERC20 specification "Transfers of 0 values must be treated as normal transfers and fire the Transfer event"[7]. The current implementation only considers transfer that are strictly greater than 0.

```
10 function transfer(address _to, uint256 _value) public returns (bool success) {
11     if (balances[msg.sender] >= _value && _value > 0) {
```

contracts/StandardToken.sol

Addressed: STACK states the `StandardToken` contract is a placeholder for the STK Token already deployed on mainnet.

Special `transferFrom` in WETH L ✓ Fixed

The WETH contract contains the following code:

```
71   function transferFrom(address src, address dst, uint wad)
72   public
73   returns(bool) {
74     require(balances[src] >= wad);
75
76     if (src != msg.sender && allowance[src][msg.sender] != uint(-1)) {
77       require(allowance[src][msg.sender] >= wad);
78       allowance[src][msg.sender] -= wad;
79     }
```

WETH.sol

It seems that `uint(-1)` has a special value (representing infinity) as approval. This needs to be well-documented as it might lead to confusion. Additionally, the handling of `src == msg.sender` is non-standard and might lead to errors when not properly documented, as it does not decrease `allowance`.

Fixed: STACK removed this `transferFrom` function from `WETH.sol`. The purpose of the WETH contract is to take ETH, convert it to a usable entity - similar to an ERC20 function, then unwrap the WETH and send ETH to addresses. STACK has no use for a `transferFrom` function.

[7] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#transfer

Recommendations / Suggestions

☐ STACK should consider renaming the contract `Token.sol` to reflect that this contract is an interface (e.g. `IToken.sol`) and not an implementation. This will increase clarity as the actual token implementation is named very similar.

☑ Inside the test scripts several parameters are passed as inputs for the `MultiLibrary` contract. However, this contract does not take any constructor parameters. Hence, these arguments (which are currently incomplete) can be omitted.

☑ It is recommended to perform explicit error handling whenever `ecrecover` is used.

☑ For some of the functionalities, such as `StandardToken`, there are existing, well-tested contracts that could have been used.

- Under the previously mentioned assumptions that the user addresses differ from STACK's address this code in `close` is redundant: `require(signerAddress != msg.sender);`

☑ As `uint` variables cannot hold negative values the following check:

```
22        require(channels[addressOfToken].timeout_ <= uint(0));
```

MultiChannel.sol

can be restricted to the equals operator.

Post-audit comment: STACK has fixed some of the issues above and has is aware of all the implications of those points which were not addressed. Given this awareness, STACK has to perform no more code changes with regards to these recommendations.

Disclaimer

UPON REQUEST BY STACK, CHAINSECURITY LTD. AGREES MAKING THIS AUDIT REPORT PUBLIC. THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND CHAINSECURITY LTD. DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH CHAINSECURITY LTD..