CONFIDENTIAL

FOR INTERNAL USE

Security Audit

of Polkadot Claims Smart Contracts

June 26, 2019

Produced for

W3F

by

CHAINSECURITY

Table Of Contents

Foreword

We would first and foremost like to thank WEB3 FOUNDATION for giving us the opportunity to audit their smart contracts. This document outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

This executive summary is for the internal private audit report. It is brief and focuses on discovered issues in the project which should be addressed before releasing. The executive summary of a public version of the audit report will describe the final state of the project holistically.

CHAINSECURITY audited the smart contracts which are going to be deployed on the public Ethereum chain. Audits of CHAINSECURITY use state-of-the-art tools for detection of generic vulnerabilities and verification of custom functional requirements. Additionally, a thorough manual code review by leading experts helps to ensure conformance with the latest security best practices.

During the audit CHAINSECURITY found a few minor issues which are explained in-depth in this report. On the non-technical side, the initially provided high-level documentation was not extensive. Communication with the WEB3 FOUNDATION team was fast and professional.

Audit Overview

Methodology of the Audit

CHAINSECURITY's methodology in performing the security audit consisted of four chronologically executed phases:

1. Understanding the existing documentation, purpose, and specifications of the smart contracts.
2. Executing automated tools to scan for common security vulnerabilities.
3. Manual analysis by one of our CHAINSECURITY experts covering both security and functional correctness (based on the provided documentation) of the smart contracts.
4. Formalizing security and correctness properties that capture the intended behavior of the smart contracts and checking these using analysis tools for Ethereum smart contracts.
5. Writing the report with checked properties, vulnerability findings, and potential exploits.

Scope of the Audit

Source code files received	June 25, 2019
Git commit	code provided as .zip
EVM version	PETERSBURG
Initial Compiler	SOLC compiler, version 0.5.9

The scope of the audit is limited to the following source code files.

File	SHA-256 checksum
contracts/Claims.sol	`709ec492d655faca220f2ae350994646f5e3463ba6b89be9afc40aa1c1330f94`

For these files the following categories of issues were considered:

In Scope	Issue Category	Description
✓	Security Issues	Code vulnerabilities exploitable by malicious transactions
✓	Trust Issues	Potential issues due to actors with excessive rights to critical functions
✓	Design Issues	Implementation and design choices that do not conform to best practices

Depth of the Audit

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

- Scanning the contracts listed above for common security issues using automated systems and manually inspecting the results.
- Manual audit of the contracts listed above for security issues.
- Specifying correctness properties for the contracts listed above and checking these both manually and using tools.

Terminology

For the purpose of this audit, ChainSecurity has adopted 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 categorise the findings into four distinct categories, depending on their severities:

- L Low: can be considered less important
- M Medium: should be fixed
- H High: we strongly recommend fixing it before release
- C Critical: needs to be fixed before release

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

	IMPACT		
LIKELIHOOD	High	Medium	Low
High	C	H	M
Medium	H	M	L
Low	M	L	L

During the audit, concerns might arise or tools might flag certain security issues. After carefully inspecting the potential security impact, we assign the following labels:

- ✓ No Issue : no security impact
- ✓ Fixed : the issue is addressed technically, for example by changing the source code
- ✓ Addressed : the issue is mitigated non-technically, for example by improving the user documentation and specification
- ✓ Acknowledged : the issue is acknowledged and it is decided to be ignored, for example due to conflicting requirements ot other trade-offs in the system

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 of what kind of issues were found during the audit.

Limitations

Security auditing cannot uncover all existing vulnerabilities; even an audit in which no vulnerabilities are found is not a guarantee of a secure smart contract. However, auditing enables discovery of 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 are completely unprotected against it. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. This is why we carry out a source code review aimed at determining 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.

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

System Overview

Web3 Foundation set up a smart contract, called Claims, to migrate the allocation of DOT from Ethereum accounts to Polkadot accounts. The Claims contract is linked to the already deployed token contract that contains user balances. Any user with a positive balance in the token contract can call `claim()` function which links the existing token balance to the specified Polkadot public key. Additionally, an index used as a short address is given to each Ethereum account.

The contract has an `owner` role, which can execute two privileged functions. The first function allows the Web3 Foundation to amend an Ethereum address. This is used as an emergency function in case an account has been lost or compromised and such an amendment can be changed by the `owner` as long as the amended account has not been claimed.

The second privileged function allows the Web3 Foundation to set a vesting flag (a boolean value) for specific accounts. The vesting is not part of the Ethereum smart contract. It is used and enforced inside Polkadot. Last but not least, a function called `assignIndices()` can be called by anyone and assigns the indices (mentioned before) to addresses. The final state of this contract will be used to initialize the genesis block for a Polkadot blockchain.

System Roles

In this section we outline the different roles and their permissions and purpose within the system.

Owner The owner deploys the `Claims` contract. It can call `amend()` to amend allocation holder address in extreme circumstances. Further calls to `amend()` update existing amendments. It can call `setVesting()` to set the vesting of unclaimed addresses. It can transfer the contract ownership to another address.

Allocation Holder Allocation holders of amendments can claim DOT token on Polkadot blockchain by specifying their Polkadot public key.

Claimer Claimers are able to change the state of the contract by associating a Polkadot public key to an allocation balance.

Trust Model

The owner is a trusted role in the `Claims` contract. Using the `amend` function the owner can redistribute funds arbitrarily. Furthermore, the owner can enable vesting for any account. The owner is trusted to use these functions correctly.

Assumptions on Frozen Token

The frozen token contract holds the current token balances and is used as a data source in the `Claims` contract. It has previously been audited and is outside the scope of this audit. For the correctness of the `Claims` we make the following security assumptions on the frozen token contract:

- there are no untrusted accounts with a positive token balance

In order to justify these assumptions, ChainSecurity investigated the state of the contract at the time of writing which is detailed below.

Current State of Frozen Token ChainSecurity inspected the state of the frozen token contract at block 8071600. At this time there are only two liquid accounts with positive token balances. We believe these accounts to be controlled by the Web3 Foundation. Hence, the necessary security assumptions are currently fulfilled.

Consequences of potential misbehavior If the frozen token would not adhere to the trust model above, the system could be attacked in certain ways. A liquid account can create empty claims. Many such empty claims could be used to spam the list of claims, however, this would likely not be more than an annoyance and provide no direct benefit for the attacker.

Best Practices in Web3 Foundation's project

ChainSecurity is determined to deliver the best results to ensure the security of a project. To enable us to do so, we are listing Hard Requirements which must be fulfilled to allow us to start the audit. Furthermore we are providing a list of proven best practices. Following them will make audits more meaningful by allowing efforts to be focused on subtle and project-specific issues rather than the fulfilment of general guidelines.

Hard Requirements

These requirements ensure that the Web3 Foundation's project can be audited by ChainSecurity.

- [x] **All files and software for the audit have been provided to ChainSecurity**
  The project needs to be complete. Code must be frozen and the relevant commit or files must have been sent to ChainSecurity. All third party code (like libraries) and third-party software (like the solidity compiler) must be exactly specified or made available. Third party code can be located in a folder separated from client code (and the separation needs to be clear) or included as dependencies. If dependencies are used, the version(s) need to be fixed.
- [x] The code must compile and the required compiler version must be specified. When using outdated versions with known issues, clear reasons for using these versions are being provided.
- [ ] There are migration/deployment scripts executable by ChainSecurity and their use is documented.
  Explanation: There are deployment scripts but the relevant part is commented out.
- [ ] The code is provided as a Git repository to allow reviewing of future code changes.
  Explanation: The code was provided as a zip file without any repository files.

Best Practices

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

- [x] There are no compiler warnings, or warnings are documented.
- [x] Code duplication is minimal, or justified and documented.
- [x] The output of the build process (including possible flattened files) is not committed to the Git repository.
- [x] The project only contains audit-related files, or, if this is not possible, a meaningful distinction is made between modules that have to be audited and modules that ChainSecurity should assume are correct and out-of-scope.
- [x] There is no dead code.
- [x] The code is well-documented.
- [ ] The high-level specification is thorough and enables a quick understanding of the project without any need to look at the code.
  Explanation: Interaction was needed to understand all details of the project but the communication was fast and professional.
- [ ] Both the code documentation and the high-level specification are up-to-date with respect to the code version ChainSecurity audits.
  Explanation: Some details have only been provided in the direct interaction. Especially, on the deployment process and some details on specific code parts.
- [x] Functions are grouped together according to either the Solidity guidelines[2], or to their functionality.

[2] `https://solidity.readthedocs.io/en/v0.4.24/style-guide.html#order-of-functions`

Smart Contract Test Suite

In this section, ChainSecurity comments on the smart contract test suite of Polkadot Claims. While the test suite is not a component of the audit, a good test suite is likely to result in better code.

Web3 Foundation provided 19 truffle test, covering the relevant parts of the smart contract.

Formal Functional Properties

ChainSecurity investigated selected functional requirements. Below, we list the considered requirements and indicate whether they have been successfully verified (marked with label ✓ Verified) or not (marked with label ✕ Does not hold).

Property syntax and semantics

The formalization of the properties uses the syntax of Solidity extended with temporal operators (such as `always` and `prev`) and additional logical connectives (such as implication, written with `==>`). For example, `always` quantifies over all contract states and `prev` refers to the previous state (before a transaction has executed). The expression `FUNCTION == Claims.claim(address,bytes32)` evaluates to `true` if the current transaction is a call to function `claim(address,bytes32)` of contract `Claims`. Finally, the expression `Claims.claim(address,bytes32)[0]` returns the first argument of function `Claims.claim(address,bytes32)`.

The properties are interpreted over a sequence of contract states, induced by executing a given sequence of transactions. A property is *verified* if it holds for any possible sequences of contract states.

Properties related to the Web3 Foundation contract

In the formalization of the properties, the length of the `claimed` array is always restricted to an upper bound of `10000`. To capture this, we use the idiom `always(always(Claims.claimed < 10000)==> prop)`. This property holds if `always(prop)` holds for any sequence of states where the array `Claims.claimed` never exceeded the length `10000`. This is necessary to verify the properties, due to the storage allocation scheme used in the Ethereum Virtual Machine (EVM). Namely, an unbounded array can theoretically overwrite any other variable in storage, resulting in violations of the considered properties. The upper bound of `10000` was chosen after consultation with Web3 Foundation.

Furthermore, in the properties below, `X` represents an arbitrary value (such as `0x123`).

2.1 A claimed Polkadot public key is immutable ✓ Verified

If the Polkadot public key has been set for any claim, then that public key must remain the same. Hence, it can only be changed if it previously had not been set.

```
always(
  always(Claims.claimed < 10000)
    ==> ((prev(Claims.claims[X].polkadot) == Claims.claims[X].polkadot)
          || (prev(Claims.claims[X].polkadot) == 0)));
```

2.2 Only the owner can modify the account vesting ✓ Verified

Each account has a vesting field. Any modification to this field can only be performed by the owner of the Claims contract.

```
always(
  always(Claims.claimed < 10000)
    ==> (Claims.claims[X].vested != prev(Claims.claims[X].vested))
      ==> (msg.sender == Claims.owner));
```

2.3 Successful claims require an allocation ✓ Verified

A claim associated with an Ethereum address is only possible if the FrozenToken balance of that Ethereum address is positive.

```
always(
  always(Claims.claimed < 10000)
     ==> (FUNCTION == Claims.claim(address,bytes32))
       ==> (FrozenToken.accounts[Claims.claim(address,bytes32)[0]].balance
             > 0));
```

2.4 The allocationIndicator is immutable ✓ Verified

The Claims contract contains a variable called allocationIndicator which points to the FrozenToken contract. This variable cannot be modified after deployment.

```
always(
  always(Claims.claimed < 10000)
    ==> (prev(Claims.allocationIndicator) == Claims.allocationIndicator));
```

2.5 Contract linking is correct ✓ Verified

The property above assures the immutability of the allocationIndicator, while this property assures the correctness of the variable. In particular it is always set to value provided in the deployment.

```
always(
  always(Claims.claimed < 10000)
    ==> (Claims.allocationIndicator == FrozenToken));
```

We also note that this property implies that the `allocationIndicator` variable is immutable.

2.6 Index immutable after initialization - Part 1 ✓ Verified

Every claim is associated with an index. After every transaction this index should either be unchanged or should have been previously uninitialized.

```
always(
  always(Claims.claimed < 10000)
    ==> ((prev(Claims.claims[X].index) == Claims.claims[X].index)
         || (prev(Claims.claims[X].index) == 0)));
```

2.7 Index immutable after initialization - Part 2 ✓ Verified

Analogously to the property above this property checks the immutability of the index based on the `hasIndex` field.

```
always(
  always(Claims.claimed < 10000)
    ==> ((prev(Claims.claims[X].index) == Claims.claims[X].index)
         || (prev(Claims.claims[X].hasIndex) == false)));
```

2.8 Amendments are restricted to the owner ✓ Verified

Any changes to the amendments are only possible if the caller was the owner of the Claims contract.

```
always(
  always(Claims.claimed < 10000)
    ==> (prev(Claims.amended[X]) != Claims.amended[X])
      ==> (msg.sender == Claims.owner));
```

2.9 Proper access control for claims ✓ Verified

The function `claim(address,bytes32)` can only be called if a user calls it for its own address (hence, its address matches the first argument of the function) or the caller is the amended address for the given allocation.

```
always(
  always(Claims.claimed < 10000)
   ==> (FUNCTION == Claims.claim(address,bytes32))
     ==> ((msg.sender == Claims.claim(address,bytes32)[0])
          || (msg.sender ==
              Claims.amended[Claims.claim(address,bytes32)[0]])));
```

2.10 Ownership can only be modified by the owner ✓ Verified

Only the current contract owner can set the new owner of the `Claims` contract.

```
always(
  always(Claims.claimed < 10000)
    ==> ((prev(Claims.owner) != Claims.owner)
      ==> (msg.sender == prev(Claims.owner))));
```

2.11 The ownership cannot be set to 0x0 ✕ Does not hold

As explained in this security issue, accidents in the function `setOwner` could significantly impact the contract. Therefore, sanity checks are often performed on the inputs of such functions. As this is not the case here, the owner address can take any value, including `0x0`.

```
always(
  always(Claims.claimed < 10000) ==> (Claims.owner != 0x0));
```

Security Issues

This section reports the security issues found during the audit.

Input verification and claim ownership scheme M

In contract `Owned`, WEB3 FOUNDATION uses function `setOwner(_new)` to set a new account as owner. The function does perform a sanity checks on the argument `_new`. Mistakes can happen and therefore, CHAINSECURITY suggest to ensure that the argument `_new` does not equal `address(0)`. Additionally, WEB3 FOUNDATION could consider to use a claim ownership scheme to ensure that the new address is an account under control by someone.

Likelihood: Low
Impact: High

Trust Issues

This section reports functionality that is not enforced by the smart contract and hence correctness relies on additional trust assumptions.

FrozenToken address not hardcoded L

The `FrozenToken` contract is already deployed on mainnet and hence, known. Therefore, WEB3 FOUNDATION could create more trust by hard coding the address in `Claims` contract, instead of providing it as constructor argument.

Design Issues

This section lists general recommendations about the design and style of WEB3 FOUNDATION's project. These recommendations highlight possible ways for WEB3 FOUNDATION to improve the code further.

Struct optimization in `Claim` possible L

WEB3 FOUNDATION uses the struct `Claim` in `Claims` contract.

```
struct Claim {
      bool    hasIndex;   // Has the index been set?
      uint    index;      // Index for short address.
      bytes32 polkadot;   // Polkadot public key.
      bool    vested;     // Is this allocation vested?
}
```

If the variables would be packed tightly (the two booleans together), WEB3 FOUNDATION could save one storage slot. This would lower the gas costs during contract execution.

Index `0` is both valid and invalid L

WEB3 FOUNDATION uses `claims.index == 0` and `claims.hasIndex == false` to check if the given index is not used. Only then, it is assigned to a user.

However, the first time either `assignIndices()` or `claim()` will be executed, the index `0` will be assigned to an Ethereum address. Hence, `0` is a valid index in this context. Such an ambiguity could lead to mistakes on the client side when processing the state of the smart contract.

Setup process can be interrupted L

WEB3 FOUNDATION needs to call `setVesting()` and `assignIndices()` shortly after deployment of the `Claims` contract and before any other account calls `claim()` or `assignIndices()`. Otherwise, the first 925 indices can not be assigned by WEB3 FOUNDATION to special addresses. Additionally, it is impossible to set `vested` to `true` for accounts that already called the `claim()` function.

Disregarded return value L

WEB3 FOUNDATION implemented the function `assignNextIndex()` which returns a boolean. This function is called from the following functions:

- `claim()`
- `assignIndices()`

In both of the above function calls the return value is not checked. CHAINSECURITY should evaluate if the return value is needed and if so, check it.

Unable to disable vesting L

The WEB3 FOUNDATION can call the `setVesting` function to set the vesting status for provided accounts. Once the vesting is set, it cannot be changed or reverted.

However, in case of any mistake the WEB3 FOUNDATION may want to revert the vesting status. Currently, it is not possible to revert the vesting status.

Recommendations / Suggestions

- ☐ WEB3 FOUNDATION sets two variables in the `constructor`.

  ```
  owner = _owner;
  allocationIndicator = FrozenToken(_allocations);
  ```

  There are no sanity checks for these two addresses. WEB3 FOUNDATION could consider to at least check for `address(0)` to avoid any mistakes when deploying.

- ☐ WEB3 FOUNDATION makes intense use of loops to handle the batch functions. It might be useful to set a boundary on the length of the input parameters, to avoid running a loop for a long time and in the end, run out of gas. At least, the WEB3 FOUNDATION needs to know the maximum length of the arrays it can pass into the functions. Exemplary, for `assignIndices()` the maximum array length will be roughly around 130. If too many elements are passed into the function, all gas will be consumed with, in the end, no state change at all.

- ☐ The `amend` function is called along with a list of addresses to be amended including the new addresses. However, when any of the addresses present in the list has been claimed before the whole transaction will revert. In such a case it would be non-trivial to identify which address caused the transaction to revert. Alternatively, events for failures could be emitted and valid amendments could be performed.

- ☐ WEB3 FOUNDATION sometimes specifies a return value and sometimes not. CHAINSECURITY does not know if the return value is needed in off-chain scripts. But if there is no special need, CHAINSECURITY suggest to use return values consistent throughout the code.

Open Questions

In this section, ChainSecurity describes the remaining doubts about Web3 Foundation's project and its specifications. These points are not issues per se, but can help Web3 Foundation specify and document the project further. They can also help in case another audit is undertaken in the future.

- There is no optimization flag in the Truffle configuration. Truffle compiles Solidity without optimization by default, which can lead to higher gas consumption. Why has this choice been made?