PUBLIC

Security Audit

of the Polkadot Claims Smart Contract

July 26, 2019

Produced for

by

Table Of Contents

Foreword . . . **1**

Executive Summary . . . **1**

Audit Overview . . . **2**

1. Methodology . . . 2
2. Scope . . . 2
3. Audit Tasks . . . 2
4. Terminology . . . 3
5. Limitations . . . 3

System Overview . . . **4**

1. System Roles . . . 4
2. Trust Model . . . 4
3. Assumptions about the Frozen Token contract . . . 4

Best Practices in the POLKADOT CLAIMS contract . . . **5**

1. Hard Requirements . . . 5
2. Best Practices . . . 5
3. Smart Contract Test Suite . . . 6

Formal Functional Properties . . . **7**

1. Property syntax and semantics . . . 7
2. Properties related to the WEB3 FOUNDATION contract . . . 7
   - 2.1 A claimed Polkadot public key is immutable ✓ Verified . . . 7
   - 2.2 Only the owner can modify the account vesting ✓ Verified . . . 7
   - 2.3 Successful claims require an allocation ✓ Verified . . . 7
   - 2.4 The allocationIndicator is immutable ✓ Verified . . . 8
   - 2.5 Contract linking is correct ✓ Verified . . . 8
   - 2.6 Index indicator immutable after initialization ✓ Verified . . . 8

2.7 Index immutable after initialization ✓ Verified . . . 8

2.8 Amendments are restricted to the owner ✓ Verified . . . 8

2.9 Proper access control for claims ✓ Verified . . . 8

2.10 Ownership can only be modified by the owner ✓ Verified . . . 9

2.11 Set up period safety of indices ✓ Verified . . . 9

2.12 Set up period safety of claims ✓ Verified . . . 9

Security Issues . . . 10

1. Input verification and claim ownership scheme M ✓ Acknowledged . . . 10

Trust Issues . . . 11

1. Frozen Token address not hardcoded L ✓ Acknowledged . . . 11

Design Issues . . . 12

1. Struct optimization in `Claim` possible L ✓ Fixed . . . 12

2. Index `0` is both valid and invalid L ✓ Acknowledged . . . 12

3. Setup process can be interrupted L ✓ Acknowledged . . . 12

4. Disregarded return value L ✓ Fixed . . . 12

5. Unable to disable vesting L ✓ Acknowledged . . . 13

6. Vesting amount not logged into event L ✓ Fixed . . . 13

7. Zero vesting amount is allowed L ✓ Fixed . . . 13

8. Inconsistent amount in the `Vested` event M ✓ Fixed . . . 13

9. `amend` does not check balances anymore L ✓ Fixed . . . 13

Recommendations / Suggestions . . . 14

Disclaimer . . . 15

Foreword

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

– ChainSecurity

Executive Summary

The WEB3 FOUNDATION engaged CHAINSECURITY to perform a security audit of the POLKADOT CLAIMS smart contract. The contract will allow holders of the DOT allocation indicator token to claim their balances of DOTs to a Polkadot public key ahead of Polkadot genesis. Additionally, the WEB3 FOUNDATION can set a parameter to indicate vesting and amend addresses in emergency cases.

CHAINSECURITY audited the smart contract which will be deployed on the public Ethereum chain. To guarantee that the POLKADOT CLAIMS contract is secure and functionally correct, ChainSecurity formally verified the contract's code with respect to its intended specification using its state-of-the-art tool for verification of functional requirements. Additionally, CHAINSECURITY's experts conducted a thorough code review to ensure that the contract conforms to the latest security best practices.

All reported issues have been addressed or acknowledged by the WEB3 FOUNDATION.

Audit Overview

Methodology

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. Preparing the report with checked properties, vulnerability findings, and potential exploits.

Scope

Source code files received	June 25, 2019
Git commit	`code provided as .zip`
EVM version	Petersburg
Initial Compiler	solc compiler, version 0.5.9
Final code update received	July 25, 2019
Final commit	`code provided as .zip`
Final Compiler	solc compiler, version 0.5.9

The repository is public and available at https://github.com/w3f/polkadot-claims.
The scope of the audit is limited to the following source code files.

File	SHA-256 checksum
./Claims.sol	`f3035d1c3586846a15ab9a280eb49896c58522765561a8eb210ef4d7e8dcad22`

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

Audit Tasks

The security audit conducted by ChainSecurity consisted of the following tasks:

- Formalizing functional requirements pertaining to the immutability of the state after the initialization, access-control requirements, and the safety of the contract set-up period.
- Formally verifying the correctness of the Polkadot Claims contract with respect to the formalized properties.
- Analyzing the Polkadot Claims contract for generic security vulnerabilities.
- A thorough manual audit of the Polkadot Claims contract for compliance with best security practices.

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

The 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 vesting and/or non-vesting Polkadot public keys. Additionally, an index used as a short address is given to each Ethereum account. The deployer also specifies a setup phase to delay that user can claim or assign indices. This ensures that the initial state of the contract can be set up by the admin before allowing the public to write to the storage.

The contract has an `owner` role, which can execute three 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 contract's `owner` as long as the amended account has not been claimed. The second privileged function allows the WEB3 FOUNDATION to set a vesting amount or increase a vesting amount for specific accounts. The vesting is not part of the Ethereum smart contract. It is used and enforced inside Polkadot.The third admin function is able to freeze claiming forever.

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

Allocation Holder Allocation holders or the amendment accounts 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 about the Frozen Token contract

The Frozen Token contract holds the current token balances of users and is used as a data source in the POLKADOT CLAIMS contract. It has previously been audited and is outside the scope of this audit.

For the correctness of the POLKADOT CLAIMS contract, we assume that the Frozen Token contract has no no untrusted accounts with a positive token balance To justify this assumption, 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 are controlled by the WEB3 FOUNDATION. Hence, the necessary security assumption is 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. For example, a liquid account could create empty claims, which could be used to spam the list of claims. However, this would not benefit a potential attacker and is therefore unlikely.

Best Practices in the POLKADOT CLAIMS contract

We list Hard Requirements which must be fulfilled in order for us to start the audit. Furthermore, we provide a list of best practices, the following of which allows the audit efforts to be focused on project-specific issues.

Hard Requirements

The fulfillment of the requirements below ensures that CHAINSECURITY can start the audit.

- [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

These requirements are complimentary to the hard requirements and can further enhance the efficiency of the audit.

- [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 test suite of the Polkadot Claims contract. While the test suite is not a component of the audit, a thorough test suite indicates that the code likely contains fewer correctness issues.

The Web3 Foundation provided 19 truffle tests, covering the relevant parts of the Polkadot Claims 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].pubKey) == Claims.claims[X].pubKey)
         || (prev(Claims.claims[X].pubKey) == 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 Frozen Token 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 Frozen Token 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 indicator immutable after initialization ✓ Verified

Every claim is associated with an index. The `hasIndex` field indicates whether the index has been initialized. Hence, the `hasIndex` field should not be resettable.

```
always(
  always(Claims.claimed < 10000)
   ==> ((prev(Claims.claims[0x123].hasIndex) == Claims.claims[0x123].hasIndex)
        || (prev(Claims.claims[0x123].hasIndex) == false)));
```

2.7 Index immutable after initialization ✓ 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 Set up period safety of indices ✓ Verified

Only the current contract owner can call the function `assignIndices()` during the setup period.

```
always(
  always(Claims.claimed < 10000)
    ==> (((block.number < Claims.endSetUpDelay) &&
            (FUNCTION == Claims.assignIndices(address[])))
          ==> (msg.sender == prev(Claims.owner))
    )
);
```

2.12 Set up period safety of claims ✓ Verified

The function `claim()` can only be called after the setup period.

```
always(
  always(Claims.claimed < 10000)
    ==> ((FUNCTION == Claims.claim(address,bytes32))
      ==> (block.number >= Claims.endSetUpDelay)));
```

Security Issues

This section reports the security issues found during the audit.

Input verification and claim ownership scheme ✓ Acknowledged

Function `setOwner(_new)` in contract `Owned` is used to set a new account as owner. This function does not validate the argument `_new`. As mistakes can happen, CHAINSECURITY suggests the WEB3 FOUNDATION to ensure that the argument `_new` does not equal `address(0)`. Additionally, the WEB3 FOUNDATION could consider using a claim ownership scheme to ensure that the new address is controlled by an active user.

Likelihood: Low
Impact: High

Acknowledged: The WEB3 FOUNDATION acknowledged that no change will be applied in the interest of keeping the contract as minimal as possible. The WEB3 FOUNDATION will take special care in case the function becomes necessary.

Trust Issues

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

Frozen Token address not hardcoded ✓ Acknowledged

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

Acknowledged: The WEB3 FOUNDATION acknowledged that the address is not hardcoded in order to facilitate testing and deployment on test networks at different addresses. The WEB3 FOUNDATION will consider hardcoding before mainnet deployment.

Design Issues

This section lists general recommendations about the design and style of the POLKADOT CLAIMS contract. These recommendations highlight possible ways for the WEB3 FOUNDATION to improve the code further.

Struct optimization in `Claim` possible ✓ Fixed

The 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), the WEB3 FOUNDATION could save one storage slot. This would lower the gas costs during contract execution.

Fixed: The WEB3 FOUNDATION fixed the problem. The struct was updated to a version that cannot be optimized further.

Index `0` is both valid and invalid ✓ Acknowledged

The 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 execution of either `assignIndices()` or `claim()` will assign the index `0` 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.

Acknowledged: The WEB3 FOUNDATION acknowledged that the use of index `0` is what necessitated the addition of the `hasIndex` boolean.

Setup process can be interrupted ✓ Acknowledged

The 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 the WEB3 FOUNDATION to special addresses. Additionally, it is impossible to set `vested` to `true` for accounts that already called the `claim()` function.

Acknowledged: The WEB3 FOUNDATION acknowledged that they will have scripts to do the entire setup. This provides a buffer during which no one can interrupt the setup of the initial state of the contract and force a re-deployment by the WEB3 FOUNDATION.

Disregarded return value ✓ Fixed

The 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. The WEB3 FOUNDATION should evaluate if the return value is needed and if so, check it.

Fixed: The WEB3 FOUNDATION fixed the problem by modifying the `claim()` and `assignIndices()` function declarations. These functions no longer return any value.

Unable to disable vesting ✓ Acknowledged

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.

Acknowledged: The WEB3 FOUNDATION acknowledged that this is intended behavior. The WEB3 FOUNDATION do not want to disable vesting. Since vesting is part of the setup procedure, in the case of mistakes the WEB3 FOUNDATION can deploy a new contract.

Vesting amount not logged into event ✓ Fixed

The `setVesting()` function takes two arrays as arguments. The first argument is an array with Ethereum addresses and the second array contains the corresponding vesting amounts.

Once the vesting of an Ethereum address is updated, the contract emits `Vested` event. However, the current event only logs the Ethereum address. It could also include the vesting amount corresponding to that Ethereum address.

Fixed: In the latest version, the WEB3 FOUNDATION logs the vesting amount.

Zero vesting amount is allowed ✓ Fixed

In the `setVesting()` function the owner is allowed to send `0` (zero) as a value in the `_vestingAmts` array. Doing this would emit the `Vested` event for the corresponding address even though there is no vested amount. The owner could additionally call `setVesting()` again for the same address. CHAINSECURITY recommends adding a check to ensure that the vesting amount is not `0` (zero).

Fixed: The WEB3 FOUNDATION does not allow zero vesting anymore.

Inconsistent amount in the `Vested` event ✓ Fixed

The WEB3 FOUNDATION emits a `Vested` event in `setVesting` and `increaseVesting`. In `setVesting` the second parameter for this event `_vestingAmts[i]` is the total vesting amount. In `increaseVesting` this parameter is the additional amount (delta). These result in inconsistent and unreliable events.

Fixed: The WEB3 FOUNDATION introduced a new event called `VestedIncreased`.

`amend` does not check balances anymore ✓ Fixed

After a code update, the WEB3 FOUNDATION did not check if an address has a DOT allocation in the amend function.

Fixed: The WEB3 FOUNDATION added the check again.

Recommendations / Suggestions

- [x] The WEB3 FOUNDATION sets two variables in the `constructor`.

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

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

- [ ] The 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.

- [x] The 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.

- [ ] The POLKADOT CLAIMS contract initialization process is protected using `endSetUpDelay` (delay in block number), which allows only the owner of the contract to set the indices and set vesting. The WEB3 FOUNDATION puts an unnecessary time restriction to his own process. In case the WEB3 FOUNDATION is unable to complete it, the WEB3 FOUNDATION have to re-start the process or redeploy.

  CHAINSECURITY suggests having this initialization stage protected using a boolean variable and resetting it after finishing the initialization. This would give more control over the setup and could be easier and more efficient to implement.

- [ ] The WEB3 FOUNDATION added an increase vesting function to mitigate front-running and allowed to increase the vesting amount. In case of mistakes there is no way to decrease vesting apart from calling setVesting (if the account did not claim already), which is vulnerable to front-running. The WEB3 FOUNDATION can consider adding a decreaseVesting function.

Disclaimer

UPON REQUEST BY THE WEB3 FOUNDATION, CHAINSECURITY LTD. AGREES TO MAKE 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..