PUBLIC

Security Audit

of Alethena Drag Along Smart Contracts

August 27, 2019

Produced for

by

Table Of Contents

Foreword

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

– ChainSecurity

Executive Summary

ALETHENA engaged CHAINSECURITY to perform a security audit of ALETHENA DRAG ALONG, an Ethereum-based smart contract system. The ALETHENA DRAG ALONG smart contracts facilitates the squeeze-out process of tokenized shares bound to a shareholder agreement. In addition, a functionality is implemented that allows claiming back token if the owner lost his private key.

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 checks of custom functional requirements. Additionally, a thorough manual code review by leading experts helps to ensure the highest security standards. During the audit, CHAINSECURITY was able to help ALETHENA in addressing several security, trust and design issues of high, medium and low severity. ALETHENA provided extensive documentation and professional and fast direct support. The code is well written but lacks consistency in some parts as well as basic style guide rules.

All reported issues have been addressed by ALETHENA. In particular, all major security and design issues have been eliminated with appropriate code fixes.

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 generic security vulnerabilities.
3. Manual analysis covering both functional (best effort based on the provided documentation) and security aspects of the smart contracts by one of our CHAINSECURITY experts.
4. Writing the report with the individual vulnerability findings and potential exploits.

Scope of the Audit

Source code files received	August 14, 2019
Git commit	`c2edf6e6ec2f35a071a192cdec3b0d6f18b7555c`
EVM version	BYZANTIUM
Initial Compiler	SOLC compiler, version 0.5.10
Final code update received	August 26, 2019
Final commit	`39efbabc9b6df7a47edbfe7f721e681dd4cbfaf7`

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

In Scope	File	SHA-256 checksum
☑	./ServiceHunterShares.sol	`ce89239f9346800eeaa68587e8f2a168d479348a990190a2220658db0f99c83d`
☑	./IERC20.sol	`5fdfe4a35fe549716b2bad348ee3c29dc35d57e9a31f3672f3998b5f53e5f526`
☑	./ERC20.sol	`b3b41559f998c01f4dec7993f541c8acfa3b31cc9ac0af8e65382c1f30c5a05b`
☑	./ERC20Claimable.sol	`8a9e1b4093b3c3f4d8606c7ed4c3d40dfec3086472d376eacef360acfd90a6f9`
☑	./ERC20Draggable.sol	`40a4633ec85cffe7650f928ca8fee94c25d4104195edb1380fb1605f0b69df7e`
☑	./Acquisition.sol	`dfc78922b159deda50755045ed77037165893ff843ce0c3655f57d6f73050e35`
☑	./Ownable.sol	`7686d1c77fb6799ca78784f943420ac0d4034393a8d9ee34053993516a1b4c65`
☑	./Pausable.sol	`ed86b492833fc341dbc50e3356e4730a3d0a7af30dcd79ab1548bf28a3991bee`
☑	./DraggableServiceHunterShares.sol	`fe5bb520962a6c3a5fef1ca323483a713f0f6013f5024a01b77035767ea6a9fa`
☑	./SafeMath.sol	`333a92c38641490c0d9dd35c3bf27dc0bb23521190f67fcdf9355bbaaa67ca11`

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 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, 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 or 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.

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

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.

System Overview

ALETHENA's project consist of two token contracts. The first one is an ERC-20 token which represents shares of a company. The second contract is a wrapper for share tokens which allows the acquisition of the company (through the share tokens). Both token are ERC-20 compliant. Acquisition terms are defined in the shareholder agreement and usually define the terms like the minimum quorum and contain so called "tag-along" and "drag-along" clauses. A drag-along clause allows a majority shareholder (defined by some quorum) to force the minority shareholders to participate in selling all company shares subject to the shareholder agreement, while the tag-along clause in turn guarantees the minority shareholders can always sell their shares if a majority shareholder sells theirs.

To allow the shareholder's agreement terms enforcement by the smart contract, ALETHENA implemented the corresponding functionality (1) to initiate an acquisition, (2) to vote, (3) to cancel or (4) complete an acquisition and finally (5) to migrate to the new shares and payout the old share holders.

Additionally, ALETHENA implemented a feature for both tokens that allows the tokens to be reclaimed by a new account in case the private key is lost.

Token Features Overview - `ServiceHunterShares`

Ownable The share token contract is ownable by one owner. There is a hardcoded address (so called master) which is the only account that can change the owner.

Mintable The share tokens are mintable by the owner.

Burnable Accounts can burn their funds.

Set shareamount The owner can set the total number of shares represented with the token.

Pausable The owner can pause all token transfers.

Numbering The owner can mint numbered shares. If numbered minting is done, the shares need to be tracked off-chain and the number needs to be set by the first-in first-out principle, taking all transfers from the original address into account. The only difference between numbered and usual minting is that each numbered mint emits an event with the corresponding share number.

Subregisters The owner can add addresses to a so called subregister to keep track of tokens held by other token contracts.

Announcements The owner can emit an even with an arbitrary string as message.

Invalidation The owner can call `declareInvalid` to declare a specific number of tokens of a specific account to be invalid. This is only done if a key is not recoverable anymore. The function only emits an event but does not burn, seize or freeze the invalidated tokens.

Claimable If a private key of an account gets lost a user can try to claim the tokens with a new account. Therefore, a collateral needs to be deposited and if no other address claims the tokens or does a transfer from the account, the tokens and collateral can be claimed with the new account. The owner can delete claims.

Token Features Overview - `DraggableServiceHunterShares`

Mintable The token is mintable but only by the contract itself. Minting is only done if tokens are wrapped by a user when calling `wrap()`.

Burnable The draggable tokens can be burned by the user.

Claimable If a private key of an account gets lost, a user can try to claim the tokens with a new account. To do this, a collateral needs to be deposited and if no other address claims the tokens or does a transfer from the account, the tokens and collateral can be claimed with the new account after a certain period has elapsed.

System Roles

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

Deployer The deployer can set the initial parameters to deploy the two contracts. The deployer by default becomes the `owner` when deploying the `ServiceHunterShares` contract.

Master The master is hardcoded in the `ServiceHunterShares` contract and the only power this address has is changing the owner address.

Owner Only the `ServiceHunterShares` contract has an owner. The owner has the powers mentioned above.

User Users can do what was mentioned in the features above.

Trust Model

Here, we present the trust assumptions for the roles in the system as provided by ALETHENA DRAG ALONG. Auditing the enforcement of these assumptions is outside the scope of the audit. Users of ALETHENA DRAG ALONG should keep in mind that they have to rely on ALETHENA DRAG ALONG to correctly implement and enforce these trust assumptions.

Master The master is fully trusted. Hence, it must be completely honest at all times and make no accidental mistakes such as typos.

Owner The owner is fully trusted. Hence, it must be completely honest at all times and make no accidental mistakes such as typos.

User A regular user is untrusted and assumed to be potentially malicious. Regular users are expected to only execute attacks which are economically beneficial for them.

Best Practices in ALETHENA'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 ALETHENA's project can be audited by CHAINSECURITY.

- ☑ **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.
- ☑ 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.
- ☑ The code is provided as a Git repository to allow reviewing of future code changes.

Best Practices

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

- ☑ There are no compiler warnings, or warnings are documented.
- ☑ Code duplication is minimal, or justified and documented.
- ☑ 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 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.
- ☑ There is no dead code.
- ☑ 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.
- ☑ Both the code documentation and the high-level specification are up-to-date with respect to the code version CHAINSECURITY audits.
- ☑ Functions are grouped together according to either the Solidity guidelines[2], or to their functionality.

Smart Contract Test Suite

In this section, CHAINSECURITY comments on the smart contract test suite of ALETHENA DRAG ALONG. While the test suite is not a component of the audit, a good test suite is likely to result in better code.

The tests are easy to run for CHAINSECURITY, using the documentation provided by the client. The tests are related to the migration scripts. This is good as it validates the migration scripts which are a crucial component. The tests seem extensive and appear to cover the intended use cases. The provided test cases also cover failure cases. The tests contain proper assertions to check for the correct error messages making sure that errors were handled as expected.

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

Security Issues

This section relates to our investigation into security issues. It is meant to highlight times when we found specific issues, but also mentions what vulnerability classes do not appear, if relevant.

Unchecked return value in contract calls  ✓ Fixed

`initiateAcquisition` in `ERC20Draggable` allows an potential buyer to file an offer for buying the company. To do this the potential buyer needs to pay an offer fee. This offer fee is transferred in

```
currency.transferFrom(buyer, offerFeeRecipient, offerFee);
```

The return value of this call to the `currency.transferFrom` call is not checked. The ERC20 standard does not define if a call shall return or throw. Therefore, there exist different implementations. If this call fails and returns `false` for various reasons, the code will still continue to be executed and the offer will be filed. There is no further check if the `offerFee` was ever received by `offerFeeRecipient`.

CHAINSECURITY notes that the intended functionality is to use the contracts with XCHF. This contract is safe to use. The issue arises if the ALETHENA's setup is used with other contracts which may not be implemented the same way.

To make the system compatible with all ERC-20 implementations, client needs to either use the `safeERC20` library for calls on ERC20 token[3]. Another approach would be use `require` statements and limit the compatible contracts in the documentation.

The same applies to all calls on ERC-20 `transfer` and `transferFrom` functions. Including the already existing `transfer` calls which are wrapped into requires are error prone and should be changed to `safeERC-20`.

Likelihood: High
Impact: High

Fixed: ALETHENA fixed the issue by adding `require` at each of the mentioned code lines. Nevertheless, this is not completely sufficient to serve all ERC-20 implementations. But ALETHENA added documentation which clearly state the requirements to use an ERC-20 contract in their system. Hence, not all ERC-20 tokens are compatible with the code but ALETHENA knows and explicitly mentions it in their docs.

[3] https://github.com/OpenZeppelin/openzeppelin-contracts/pull/1655

Trust Issues

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

`declareInvalid` does not enforce the underlying assumptions H ✓ Addressed

The `declareInvalid` function is used to invalidate tokens of a certain holder, and can only be executed by the contract owner. This feature is explained in the documentation as:

> The equity contract now also allows the owner to declare certain share tokens invalid (kraftlos). This is to be used for example after a court ruling or if for some reason certain tokens cannot be recovered through the standard claim process (e.g. tokens sent erroneously to the draggable contract). This should only be used if it is certain that the the tokens in question can never be accessed again as invalid tokens are not excluded from executing standard operations.

Smart contracts allow encoding and, thereby, enforcing agreed upon terms without needing a third party. This is one of the main selling points of converting traditional contracts into a smart contract. In the `ServiceHunterShares` contract the `declareInvalid` function allows "invalidating" any holder's tokens. According to the documentation and through communications with ALETHENA, it has become clear that the current implementation is such that no privileged contract role has the ability to touch any user's tokens. So instead of burning tokens that are invalidated, ALETHENA has chosen to simply leave them in the holders balance.

The current implementation rests on two assumptions:

- The holder balance contains at least the amount of tokens that are to be invalidated.
- The holder whose tokens are invalidated has definitely lost access to his account and thus cannot ever transfer his tokens again.

Instead of having to trust these assumptions, they could simply be encoded in the smart contract. First off, make sure the holder has at least the amount of tokens that are to be invalidated. Additionally, there are ways to also enforce the second assumption directly or indirectly. Directly by decreasing the holder balance by the amount of invalidated tokens. Indirectly, by freezing all tokens in this address (disallow all transfers, voting and privileged features). Although this means the owner can touch any holder's balance, the opposite, having to trust the two assumptions goes directly against why smart contracts are so useful to encode real-world contracts.

Addressed: ALETHENA explained the legal situation and the intention and decided to do the sanity check if the account balance at least have enough tokens to invalidate but decided to leave the remaining as it is.

Design Issues

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

Can try claiming of token contract itself M ✓ Addressed

There is no check to prevent claiming of the tokens from the contract address itself. Claims can only be deleted if the "lost address" calls `clearClaim`, or the "claim deleter" calls `deleteClaim`. Since the contract itself has no way to call `clearClaim`, only the "claim deleter" can prevent somebody from claiming all of the tokens belonging to the contract itself. ALETHENA could add a check in `declareLost` to prevent `lostAddress` from being the address of the contract itself. The `DraggableServiceHunterShares` contract explicitly disables the option to file claims for the draggable tokens. This contract is also not supposed to hold tokens. Therefore, `DraggableServiceHunterShares` is not part of this issue.

ALETHENA needs to be aware, that claiming tokens from the contract itself is right now the only chance to avoid locked tokens in the contract. ALETHENA might think about a solution to avoid this issue and the potential issue when fixing this one.

Addressed: ALETHENA addressed the problem by explaining that this is the intended behavior.

Possible underflow in `ServiceHunterShares.mintNumbered` M ✓ Fixed

The `mintNumbered` function inside `ServiceHunterShares` takes in a start and end number of the shares to mint, and can only be called by the contract owner. If the order of arguments is mixed due to a mistake, an underflow could happen. The following check in `mint` would throw an error as it is very likely that `totalShares` is not that high. Still, a check that start number is less than end number could be added, or `SafeMath.sub` could be used for the subtraction.

Fixed: ALETHENA solved the issue by using SafeMath for the calculation.

`clearClaim` perform claim exists check early M ✓ Fixed

The `clearClaim` function can be called directly, and is automatically executed each time `ERC20Claimable.transfer` is called. The function will first retrieve the collateral from the claim, followed by casting the claim's `currencyUsed` to `IERC20`. Only after these operations will it verify that this claim exists by checking that `collateral` is above zero. Since this function is executed on each call to `ERC20Claimable.transfer`, it will cause some overhead and extra gas cost. It is therefore advised to check that the claim exists before doing any other operations. ALETHENA should check that the claim exists on the first line and return if it is not. Thereby, the least amount of gas will be used when there is no claim.

Fixed: ALETHENA solved the problem by checking at the beginning of the function

Missing events in `Acquisition` M ✓ Fixed

There are no events defined/emitted in `Acquisition.sol`. Since this contract handles voting, CHAINSECURITY thinks it would make sense to emit events whenever a vote is placed/adjusted.

Fixed: ALETHENA solved the problem by emitting a new event called `VotesChanged`.

Fail early M ✓ Acknowledged

To save unnecessary and eventual gas expensive calculation, smart contracts should generally fail early. One example is the `_setClaimPeriod` function in `ERC20Claimable.sol`. This function converts the input number of days to seconds, before checking that the number of days does not exceed 90. CHAINSECURITY sees no reason why not to first do the check before converting to seconds. To save gas, ALETHENA should always fail early in their code.

Acknowledged: ALETHENA fixed some code parts which are called often but explained it does not really matter in the other functions as they are not executed often and the gas savings are therefore marginal.

Unnecessary `uint256` return from `clearClaim` and `resolveClaim` L ✓ Fixed

The `clearClaim` function returns zero if the claim does not exist. Otherwise, it will return the transferred collateral amount. Since it is not possible to get the return value of an on-chain transaction, and since this function is not called internally, returning a value from this function does not serve any purpose.

The same applies to the `resolveClaim` function. There is no reason to return a value from this function.

Fixed: ALETHENA solved the problem by removing the returns.

`preClaimPeriod` double meaning L ✓ Fixed

The `preClaimPeriod` variable is used to indicate how long a pre-claim should exist before it can be converted to a claim by calling `declareLost`, which makes total sense. However, it does have a second meaning, a pre-claim can only be converted to a claim if the current timestamp is less than the pre-claim timestamp plus `preClaimPeriod * 2`. ALETHENA could consider adding a separate variable for the second meaning, for example `preClaimPeriodEnd`.

Also the two checks have the same error message, they mean different things. Pre-claim period not yet reach and pre-claim period ended, respectively. ALETHENA could use different error messages for both of these.

Fixed: ALETHENA solved the problem by adding a separate variable called `preClaimPeriodEnd`

Unnecessary first transfer before burn L ✓ Addressed

The `burn` function in `ServiceHunterShares.sol` first transfers the tokens from the sender to the contract itself using its `_transfer` function. The `_transfer` function checks that the contract is not paused, before calling `_transfer` in `ERC20.sol`. After this the `ERC20._burn` function is called to burn these tokens which are now in possession of the `ServiceHunterShares` contract.

Addressed: ALETHENA addressed the problem by explaining that the share transfer need to happen before for legal reasons.

`ClaimParametersChanged` first parameter will always be `0` L ✓ Fixed

Inside the `_setClaimPeriod` function the event `_setClaimPeriod` is emitted. The first parameter of this event is set to `getCollateralRate(address(0))`, which will always return `0`. Hence, it is unclear why to first call `getCollateralRate` instead of simply putting in `0`.

Fixed: ALETHENA fixed the issue by splitting the event in two meaningful events called `CustomClaimCollateralChanged` and `ClaimPeriodChanged`.

Unused state variable `minOfferImprovement` L ✓ Fixed

ALETHENA defines a variable `minOfferImprovement` in `ERC20Draggable` which is not used anywhere else in the code. If not necessary, ALETHENA could consider removing the variable.

Fixed: ALETHENA solved the problem by using the variable in `ERC20Draggable.initiateAcquisition`

State variables can be declared `constant` L ✓ Fixed

ALETHENA defines state variables which can not be changed anywhere in the code. ALETHENA could consider adding the keyword `constant` to save gas.

Some examples are:

- In `ERC20Draggable`: `minHolding`, `minDragAlongQuote`
- In `Ownable`: `master`

Adding the keyword `constant` saves gas because the compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective constant expression (which might be computed to a single value by the optimizer)[4].

Fixed: ALETHENA solved the problem by declaring the respective variables constant.

Use enum instead of `-1`, `0`, and `1` L ✓ Fixed

Inside `Acquisition` ALETHENA uses `-1` to mean no, and `1` to mean yes, stored as a `uint8`. Using an enum would improve the readability of the contract by making it immediately clear what means "yes", what means "no", and what means "unset". The enum should have values: `Vote.NONE`, `Vote.YES`, `Vote.NO`.

Fixed: ALETHENA fixed the issue by using an enum.

Functions could be declared `external` L ✓ Fixed

The following functions are only used by external calls and have complex variables as arguments. Declaring these functions as `external` saves gas because the function data can directly be read from calldata instead of memory.

`ServiceHunterShares.sol`:

- `function announcement(string memory message)`
- `function declareInvalid(address holder, uint256 amount, string memory message)`

`Pausable.sol`:

- `function pause(bool _pause, string memory _message, address _newAddress, uint256 _fromBlock)`

Fixed: ALETHENA solved the problem by declaring the listed functions external and reading from `calldata`.

`vote` does not check if it differs from current vote L ✓ Fixed

The `vote` function in `Acquisition.sol` does not check that the vote differs from the already cast vote. Even though this has no negative consequences, ALETHENA could consider adding a check that the new vote differs from the current vote.

Fixed: ALETHENA fixed the problem by checking if the new vote differs from the previous.

Timestamp can be saved in `uint32` L ✓ Fixed

Inside `ERC20Claimable.sol` the timestamp could saved as a `uint32`. This would mean the `timestamp` + `currencyUsed` could share 1 storage slot. Thereby, one storage slot can be saved, leading to a gas cost reduction of 20k for creating a claim.

[4] https://solidity.readthedocs.io/en/v0.5.3/contracts.html#constant-state-variables%7D

Fixed: ALETHENA now uses `uint32` for `timestamp`.

Use solidity provided time units L ✓ Fixed

Solidity supports various time units[5] to improve the readability of time based values. ALETHENA could consider using these instead of lines such as:

```
uint256 public claimPeriod = 60 * 60 * 24 * 180; // Default of 180 days;
```

Fixed: ALETHENA solved the problem by using the time units provided by solidity.

[5] https://solidity.readthedocs.io/en/v0.5.11/units-and-global-variables.html#time-units

Recommendations / Suggestions

☑ The `validateClaim` function performs three checks using **`require`** statements. The last line contains another check but the result of this check is returned as a boolean. CHAINSECURITY does not see any reason why the last check's result is returned as a boolean instead of also being wrapped in a **`require`**. ALETHENA could consider also using a **`require`** for this last check.

☑ The `ERC20Claimable.sol` contract imports `./IMigratable.sol`, yet never uses it.

☑ The `recognizeSubRegister` and `removeSubRegister` functions could emit en event indicating the adding / removal of a subregister.

☑ `SafeMath.sol` is imported twice in `Acquisition.sol`. Although not causing any problems, ALETHENA could remove one of the two.

☐ The code is not very consistent style-wise. Take for example the `Acquisition` contract, in which a modifier is defined at the bottom of the contract. Also, the constants in `Acquisition` are in camelCase instead of uppercase. ALETHENA could consider following the official Solidity style guide[6] to improve the readability of the contracts.

☑ ALETHENA specifies a compiler version in their truffle config file but uses the floating pragma version `^0.5.10` in the solidity contracts. Contracts should be deployed with the same compiler version they have been tested with thoroughly. Locking the pragma in the contract, helps to ensure that it does not accidentally get deployed using another compiler version that might introduce bugs that affect the contract system negatively. ALETHENA should consider to lock the pragma version[7].

☐ The `wrap` function does two checks (`balanceOf + allowance`) before it calls `transferFrom` on the wrapped token. Since these two checks are also implicitly applied when calling `transferFrom`, ALETHENA could consider removing them.

☑ The following check inside `invertHoldings` will always be true, therefore ALETHENA could remove it.

```
require(totalSupply() == remaining, "Remaining token number incorrect");
```

☑ The last four lines in `invertHoldings` read:

```
newBacking.transferFrom(newOwner, address(this),
    conversionRate.mul(remaining));
wrapped = newBacking;
unwrapConversionFactor = conversionRate;
require(newBacking.balanceOf(address(this)) >=
    totalSupply().mul(unwrapConversionFactor), "Insufficient Backing");
```

Since `remaining` equals `totalSupply`, the last **`require`** should never fail if the `transferFrom` succeeded. Therefore, the last line would be more suited to be an **`assert`**, or to be removed completely.

☑ It makes no sense to file a claim for **`address`**`(0)`. Hence, ALETHENA could consider adding a sanity check and prevent this.

☑ It's possible to specify[8] the EVM version to use inside the `truffle.js` file. By default it's set to `Byzantium`, which is not the current one: `Petersburg`. ALETHENA could set the EVM version to `Petersburg` inside `truffle.js`, which would lower the gas cost of deploying/calling contracts by a small amount. Besides the gas cost savings there is no real benefit of using one over the other.

[6] https://solidity.readthedocs.io/en/v0.5.11/style-guide.html
[7] https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-103
[8] https://www.trufflesuite.com/docs/truffle/reference/configuration#solc

☑ Inside the `truffle.js` file the gas limit for the `development` network is set to `0x5e29d18`, which in base10 is `98.737.432`. This is unrealistically high as the current block gas limit is just over 8 million. ALETHENA should set this to a realistic value, for example 8 million. CHAINSECURITY ran the tests and migrations after setting the value to 8 million, they both still succeeded.

☐ ALETHENA has copy-pasted an older version of the OpenZeppelin `SafeMath` contract. The current version[9] of this contract provided by OpenZeppelin uses **`require`** statements instead of **`assert`**, and adds error messages to all**`require`** statements. The latest version of the other two used OpenZeppelin contracts (`Ownable`, `Pausable`) also use error messages in each **`require`**. ALETHENA could consider using this newer version of `SafeMath`, `Ownable`, and `Pausable`.

☑ The default value of an uninitialized variable is zero. Therefore, CHAINSECURITY sees no reason to initialize `customCollateralRate` to zero.

☑ ALETHENA uses non-breaking spaces (U+00A0) instead of regular spaces multiple times in the project. CHAINSECURITY advises to use common characters to not end up with unintended exceptions. An example where special characters end up problematic would be the Right-To-Left-Override character (U+202E)[10].

[9] `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v2.3.0/contracts/math/SafeMath.sol`
[10] `https://skylightcyber.com/2019/05/12/ethereum-smart-contracts-exploitation-using-right-to-left-override-character`

Addendum and general considerations

Blockchains and especially the Ethereum Blockchain might often behave differently from common software. There are many pitfalls which apply to all smart contracts on the Ethereum blockchain.

CHAINSECURITY mentions general issues in this section which are relevant for ALETHENA's code, but do not require a fix. Additionally, CHAINSECURITY mentions information in this section, to clarify or support the information in the security report. This section, therefore, serves as a reminder to create awareness for ALETHENA and potential users.

Dependence on block time information

ALETHENA uses `block.timestamp` / `now` inside the `ERC20Claimable` and the passthrough`Acquisition` contract. Although block time manipulation is considered hard to perform, a malicious miner is able to move forward block timestamps by around 15 seconds compared to the actual time. However, in the context of the project and given the required effort, this is not perceived as an issue[11].

Outdated compiler version

CHAINSECURITY could not find obvious issues with the compiler version ALETHENA is using. ALETHENA uses SOLC compiler, version 0.5.10. If ALETHENA is aware of the compiler's behavior and bugs, there might be good reasons for using an older compiler version. While the latest version does contain bug fixes, it might introduce new bugs.

CHAINSECURITY does, however, recommend to use the same compiler version homogeneously throughout the project and to use the compiler version for deployment that was used during testing. Furthermore, for any used version it is helpful to monitor the list of known bugs[12].

Forcing ETH into a smart contract

Regular ETH transfers to smart contracts can be blocked by those smart contracts. On the high-level this happens if the according solidity function is not marked as `payable`. However, on the EVM levels there exist different techniques to transfer ETH in unblockable ways, e.g. through `selfdestruct` in another contracts. Therefore, many contracts might theoretically observe "locked ETH", meaning that ETH cannot leave the smart contract any more. In most of these cases, it provides no advantage to the attacker and is therefore not classified as an issue.

Rounding Errors

(Unsigned) integer divisions generally suffer from rounding errors. The same holds true for divisions inside the EVM. Therefore, the results of arithmetic operations can be imprecise. The effects of these errors can be reduced by ordering arithmetic operations in a numerically stable manner. However, even then minor errors (e.g. in the order of one token wei) can occur.

[11] https://consensys.github.io/smart-contract-best-practices/recommendations/#the-15-second-rule
[12] https://solidity.readthedocs.io/en/develop/bugs.html

Disclaimer