PUBLIC

Security Audit

of REN Smart Contracts

November 25, 2019

Produced for

Ren

by

CHAINSECURITY

Table Of Contents

Foreword

We would like to thank REN for choosing CHAINSECURITY to audit their smart contracts. This document outlines our methodology, limitations and results.

– ChainSecurity

Executive Summary

REN engaged CHAINSECURITY to perform a security audit of REN, an Ethereum-based smart contract system. The smart contracts of REN are used for certain features of the REN system. Namely, darknode registration, payments, and cross-chain token swap.

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 REN in addressing several security, trust and design issues of high, medium and low severity. The employed coding practices and partial documentation increased the complexity of the audit.

All reported issues have been addressed by REN. CHAINSECURITY has no further concerns regarding the audited smart contracts.

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 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. Preparing the report with the individual vulnerability findings and potential exploits.

Scope

Source code files received	November 5, 2019
Git commit	`a11a29e6987ab0d32fa4be7806215e1459dec9a2`
EVM version	PETERSBURG
Initial Compiler	SOLC compiler, version 0.5.12
Final code update received	November 25, 2019
Final commit	`a8a6cc3dcbd53b04a602a0bb522fcd46c60eb850`

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

In Scope	File	SHA-256 checksum
☑	DarknodePayment/DarknodePayment.sol	`d7cf38f93d859e37f2031e98d0f3b7fa13d9402924b4630b3d1aaa3f8ef22068`
☑	DarknodePayment/DarknodePaymentStore.sol	`0170427d0ae943202d086d583c19120f0cc95367e9d0a6b752a6b1e91ea82066`
☑	DarknodeRegistry/DarknodeRegistry.sol	`8c462e15b80071af2e22950d35d1036b3e9d56708cc0cc8b7a306c10caf7e753`
☑	DarknodeRegistry/DarknodeRegistryStore.sol	`ffb04679ca5333b4b22639cab6e579f7200143513719818fe1c8135cd7474a53`
☑	Shifter/ERC20Shifted.sol	`80ea237c6af42630bea8525c98848ad6f76ee90b550d23097e500a2c2f3334f2`
☑	Shifter/IShifter.sol	`f09815dbfc9bb5cbb0117ed20bc3282f27096e5657314891cbde0c2dc74ddbe8`
☑	Shifter/Shifter.sol	`34c46d5e6de9a850d916de958382d8a8316f57f8001b2afc350070d1efa3316d`
☑	Shifter/ShifterRegistry.sol	`2a67b6c5ca046169654d0ba46f47d1804d1d0f9221f6f3bdb9c6742e573e8268`
☑	libraries/CanReclaimTokens.sol	`45c5ddfac534ee6541a2aeaa4d7558c5f1f07f68a9f92403200c681948a91328`
☑	libraries/Claimable.sol	`cf19d856a726194e69a1c3841d45b7cf91bbe9a4936c2df468aea871f195c3ed`
☑	libraries/ERC20WithFees.sol	`4a3109da3b3caa47a9eb5bb227a85f669b186261dd854a9f2006e3d534db152a`
☑	libraries/LinkedList.sol	`d8ce39d6514af62234d98a04625c8b17ab704b18bdbcd4f6ae8f7c4435af194b`
☑	libraries/String.sol	`774ff898b6b63d26d0e1b7dae2b01e8fb0f6b3b8bf4e620fdcbf16e3a90b5026`

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

The security audit conducted by CHAINSECURITY was restricted to:

- Scanning the contracts listed above for generic security issues using automated systems and manually inspecting 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 from the likelihood and the impact calculated previously.

We categorise the findings, depending on their severities, into four distinct groups:

-

Low: can be considered less important
- 

Medium: should be fixed
- 

High: we strongly recommend fixing it before release
- 

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 labeled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is listed simply 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 a contract in which no vulnerabilities are found during the audit is not a guarantee of a secure smart contract. However, auditing enables the 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 others lack protection only in certain areas. This is why we carry out a source code review aimed at determining all issues that need to be fixed. Within the customer-determined timeframe, CHAINSECURITY has performed a security audit in order to discover as many vulnerabilities as possible.

System Overview

Ren

The goal of REN is to enable general-purpose privacy preserving dapps, where applications are run in secret and all data is encrypted. REN will initially focus on three features to enable private decentralized exchanges.

- Zero-knowledge transaction layer to settle trades in secret.
- Interoperability layer to allow zero-knowledge trustless swaps between blockchains.
- Dark Pool layer that provides a secret order matching engine.

To implement these high-level features REN uses several cryptographic algorithms.

- zkSNARKS
- Shamir Secret Sharing (SSS)
- RZL, a custom Secure Multiparty Computation (sMPC) algorithm

These three features can be combined to create a **Dark Pool**. This is a type of decentralized exchange (DEX) where over the counter (OTC) trades are possible but all data hidden. REN has created such a Dark Pool named RenEx[2]. RenEx supports `BTC`, `ZEC`, `BCH`, `USDT`, `ETH` and `DAI`. To enable cross-chain trades each non-Ethereum token has a special ERC20 token contract.

RenVM

REN is build on a decentralized virtual machine called **RenVM**. RenVM implements three features using the cryptographic algorithms. Combining these features enables building stateful applications in RenVM.

- Execution of zero-knowledge applications.
- Zero-knowledge data storage.
- Permission-based access to zero-knowledge data.

The RenVM is powered by a decentralized and trustless network of machines called **Darknodes**. The Darknodes contribute their CPU time and disk space in exchange for fees. The network is permissionless, anybody can join by putting down a bond of 100000 REN TOKENs. Misbehaving nodes can be punished by slashing their bond. The network is also Byzantine Fault Tolerant (BFT). The consensus algorithm used is called Hyperdrive, a modified version of the Tendermint consensus algorithm.

Contracts

REN makes use of Ethereum smart contracts for several low-level features.

- Darknode registration and slashing, implemented in the `DarknodeRegistry` contract.
- Darknode fee payments, implemented in the `DarknodePayment` contract.
- Cross-chain token swap, implemented in the `Shifter` contract.

The first two contracts above each have a store contract. This makes it possible to replace the logic contract without losing the data. The last contract has an accompanying `ShifterRegistry` contract which keeps track of all registered `Shifter` contracts.

[2] https://ren.exchange

Darknode Registry

Darknodes wanting to join the network can register in the Darknode Registry contract by depositing a 100000 REN bond. Besides registering, this contract also allows a Darknode operator to deregister a Darknode. After deregistration, anybody can call `refund` to refund a certain Darknode's bond to the Darknode owner. Once a previously registered Darknode becomes deregistered, it is allowed register again.

There is also a `slash` function that can only be called by the `DarknodeSlasher` contract. This function will deregister the Darknode and divide (part of) the bond between the challenger and `DarknodePaymentStore` contract.

Darknodes can be in one of six states and the state can only move forward.

1. Pending Registration
2. Registered
3. Pending Deregistration
4. Deregistered
5. Cooling
6. Refunded

The `DarknodeRegistry` partitions time into discrete intervals, called epochs. An epoch takes a certain amount of time (2 days on mainnet), and can be configured by the owner. Anybody can call the `epoch` function to move the contract to the next epoch at the right time.

The store contract is called `DarknodeRegistryStore` and is used to store:

- List of Darknode structs.
- The REN TOKEN bond of each Darknode.

Darknode Payment

The Darknode Payment contract takes care of Darknode fee payments. Darknodes earn fees over deposits during each payment cycle. Deposits can be in any registered ERC20 token or ETH, and anybody is allowed to deposit funds. Darknodes can claim their earned fees by calling `claim`. The `DarknodePayment` payment cycles are synced to the `DarknodeRegistry` epochs. Updating the `epoch` inside the `DarknodeRegistry` will automatically also update the payment cycle inside `DarknodePayment`.

The store contract is called `DarknodePaymentStore` and is used to store:

- Deposits of any ERC20 token and ETH.
- Claimed reward balance per token for each Darknode.

Shifter

Each token that is not on Ethereum has its own special ERC20 token contract. This custom ERC20 contract adds a mint and burn function which can only be executed by the owner. Each ERC20 token contract has a separate Shifter contract, which will be set as the token contract's owner. The `Shifter` contract contains two functions which can be called by anyone. However, the `shiftIn` function requires a signature of the `mintAuthority` account. The `mintAutority` can be configured by the `Shifter` contract owner.

- `shiftIn`, used to mint ERC20 tokens to any address, and sending a fee to `feeRecipient`.
- `shiftOut`, used to burn ERC20 token of caller, and sending a fee to `feeRecipient`.

Both of these shift functions will send a percentage of the shifted amount as a fee to the `feeRecipient`. The `ShifterRegistry` is used to register one `Shifter` per ERC20 token. This registry is not used in any of the other contracts. It only serves as an easy way to find out all the registered cross-chain ERC20 tokens, and their attached `Shifter` contract address. The deployment scripts contain three of such cross-chain ERC20 tokens.

- `zBTC`, used to shift in/out Bitcoin.
- `zZEC`, used to shift in/out ZCash.
- `zBCH`, used to shift in/out Bitcoin Cash.

System Roles

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

DarknodeRegistry

Deployer The deployer can set the initial parameters: version, minimum bond, minimum epoch interval, minimum pod size, REN TOKEN address, and `DarknodeRegistryStore` address. The deployer will also be set as the owner of the contract.

Owner The owner can transfer ownership and update the minimum bond, minimum epoch interval, minimum pod size, and `DarknodeSlasher`/`DarknodePayment` contract addresses. Furthermore, the owner can recover any accidentally sent ERC20 tokens or ETH, and blacklist tokens for recovery.

Darknode owner The Darknode owner can deregister their Darknode, but only if it is in the correct state.

Slasher The slasher can call the `slash` function to slash a misbehaving Darknode. This will also deregister the Darknode.

Anybody There are a number of functions which can be called by anybody.

- `register`, to register a Darknode.
- `epoch`, to move the contract to the next epoch, but only if the current epoch time has passed.
- `refund`, to remove a Darknode which is in state Refundable, sending its bond to the Darknode owner.

DarknodeRegistryStore

Deployer The deployer can set the version and REN TOKEN address. The deployer will also be set as the owner of the contract.

Owner The owner can transfer ownership and add/remove Darknodes. Furthermore, the owner can decrease the bond of a registered Darknode. The owner can also recover any accidentally sent ERC20 tokens or ETH, and blacklist tokens for recovery. After deployment the owner will transfer ownership to the `DarknodeRegistry` contract. Therefore, the above mentioned functions will only be called from functions inside `DarknodeRe gistry`.

DarknodePayment

Deployer The deployer can set the initial parameters: version, `DarknodeRegistry`/`DarknodePaymentSt ore` contract addresses, cycle payout percentage and the payment cycle changer. The deployer will also be set as the owner of the contract.

Owner The owner can transfer ownership and update the `DarknodeRegistry` address, cycle payout percentage and cycle changer. Furthermore, the owner can register/deregister tokens. After deployment the owner will set the `DarknodeRegistry` contract as cycle changer, and register some tokens.

Cycle changer The cycle changer can call `changeCycle` to update the payment cycle. This role is assigned to the `DarknodeRegistry` contract which will call this function inside `epoch`.

Anybody There are a number of functions which can be called by anybody.

- `deposit`, to deposit ERC20 tokens or ETH into the `DarknodePaymentStore`.
- fallback function, to deposit ETH into the `DarknodePaymentStore`.
- `forward`, to forward ERC20 tokens or ETH mistakenly sent to this contract to the `DarknodePay mentStore`.
- `withdraw`, to withdraw the Darknode balance of a particular ERC20 token or ETH to the Darknode owner.
- `withdrawMultiple`, to `withdraw` multiple ERC20 tokens and/or ETH in one call.
- `claim`, to add any earned tokens in the previous payment cycle for a particular Darknode to that Darknode's balances inside the `DarknodePaymentStore`.

DarknodePaymentStore

Deployer The deployer can set the initial parameter: version. The deployer will also be set as the owner of the contract.

Owner The owner can transfer ownership, increment Darknode token balances, and transfer Darknode tokens. After deployment the owner will transfer ownership to the `DarknodePayment` contract. Therefore, the above mentioned functions will only be called from functions inside `DarknodePayment`.

ShifterRegistry

Deployer The deployer can set no initial parameters. The deployer will also be set as the owner of the contract.

Owner The owner can transfer ownership and add/update/remove `Shifters` (together with the corresponding token address) to the registry. Furthermore, the owner can recover any accidentally sent ERC20 tokens or ETH.

Shifter

Deployer The deployer can set the initial parameters: minimum shift amount, ERC20 token address, mint authority, shift in/out fee, and the fee recipient.

Owner The owner can transfer ownership and update the minimum shift amount, mint authority, shift in/out fee, and the fee recipient. Furthermore, the owner can recover any accidentally sent ERC20 tokens or ETH, and blacklist tokens for recovery.

Mint Authority The mint authority needs to off-chain sign the data that is passed into the `shiftIn` function. The mint authority will be the RenVM. The private key for this account is managed through sMPC.

Fee Recipient The fee recipient will receive the fees from calls to `shiftIn`/`shiftOut`.

Anybody Anybody can call the `shiftIn` function to mint tokens. However, this function requires data signed by the mint authority. The `shiftOut` function can be called by anybody to burn their tokens.

Trust Model

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

Deployer The deployer is *trusted* to use the correct code during deployment and set the right parameters.

Owner The owner of each contract is *trusted* to call the right functions with valid parameters.

Slasher The slasher is *trusted* to perform valid slashes.

Mint Authority The mint authority will be the RenVM, which uses sMPC to generate signatures. It is *trusted* to call mint only when the RenVM produced a valid signature.

Darknode A Darknode is *semi-trusted*, it is one node in the RenVM network of Darknodes. Multiple Darknodes would have to become malicious before the RenVM outcome would become affected.

Darknode owner A Darknode owner is *untrusted* and assumed to be potentially malicious.

User A regular user is *untrusted* and assumed to be potentially malicious.

Best Practices in REN'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 REN'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[3], or to their functionality.

Smart Contract Test Suite

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

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.

[3] 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.

Anyone allowed to blacklist tokens for claiming H ✓ Fixed

The `DarknodeRegistryStore` contract inherits from `CanReclaimTokens`. This allows the owner of the `DarknodeRegistryStore` to claim ETH and other ERC20 tokens that are (accidentally) sent to this contract.

In the `CanReclaimTokens` contract the following function is defined to blacklist a token by address:

```
function blacklistRecoverableToken(address _token) public {
  recoverableTokensBlacklist[_token] = true;
}
```

Since the above function is `public` and has no access controls, anyone is allowed to call it and forever block the claiming of ETH or any ERC20 tokens. REN should consider adding access control to this function.

Likelihood: High
Impact: Medium

Fixed: REN solved the problem by allowing only the Owner to call `blacklistRecoverableToken`.

`recoverTokens` does not use `SafeERC20` L ✓ Fixed

The `recoverTokens` function can be used to transfer out any accidentally sent ERC20 tokens to the contract. It does this by calling `ERC20(_token).transfer()`. There exist tokens which do not revert on failure, but instead return `false`. Besides that there are also tokens that do not return `true` on success, but instead return nothing. REN should use `SafeERC20.safeTransfer` to prevent both of these cases.

Likelihood: Low
Impact: Low

Fixed: REN solved the problem by using `SafeERC20` for the `transfer` function call.

Deregistering token is immediate H ✓ Addressed

The `_claimDarknodeReward` function loops through all `registeredTokens` to claim a Darknode's rewards for that token in the previous cycle. If the owner deregisters a token, the token is immediately removed from `registeredTokens`. This means later calls after this cannot claim tokens for the deregistered token in the previous cycle.

```
/// @notice Removes a token from the list of supported tokens.
///         Deregistration is pending until next cycle.
```

According to the comment above the `deregisterToken` function, the deregistration would be pending until the next cycle. Because the deregistration is immediate this will exclude other Darknodes from claiming that token in this cycle. This puts the late claimers at a disadvantage to early claimers in a cycle where a token is deregistered. The owner could limit the impact of this issue by calling the `deregisterToken` function as soon as possible after or before an epoch change. Still, that is not a sound solution.

REN should make deregistrations pending until the next cycle, as the function comment states.

Likelihood: High
Impact: Medium

Addressed: REN acknowledged that they decided that an immediate deregistration was the best approach for a few reasons:

1. If there is an error in an ERC20 contract (or it is updated to contain some revert logic in one of its functions) this can prevent an epoch being able to trigger correctly. Under these circumstances some kind of mechanism to remove the token force-ably is needed.
2. Deregistration of a token can only be done by an owner. Right now, this is done by REN, but this will be moved to a Darknode-owned DAO in the future (alongside ownership of other contracts). Darknodes would have to collectively agree to deregister a token. The Darknodes are also the only ones affected by an immediate deregistration. Immediate deregistration loses rewards earned from that token so far in the current epoch, but no other effect is observed.
3. In the future, REN can also have the Darknode-owned DAO contract only able to deregister when it has been planned this will happen after some time period.

REN will document these future changes and responsibilities of the Darknode-owned DAO, and the effect that immediate deregistration has.

Floating pragma ✓ Fixed

REN uses a floating pragma solidity ^0.5.12. Contracts should be deployed with the same compiler version and flags that have been used during testing and the audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively[4].

Likelihood: Low
Impact: Low

Fixed: REN solved the problem by using specific solidity compiler version `0.5.12` in all contract files.

[4] https://github.com/SmartContractSecurity/SWC-registry/blob/b408709/entries/SWC-103.md

Trust Issues

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

`mintAuthority` can be set to address zero H ✓ Fixed

The owner of the `Shifter` contract can set the `mintAuthority` to address zero. This would allow any incorrect signature passed into `shiftIn` to be considered as a valid signature. Once `mintAuthority` is equal to address zero, the following function would always return `true` when an incorrect signature is provided.

```
function verifySignature(bytes32 _signedMessageHash, bytes memory _sig)
    public view returns (bool) {
  return mintAuthority == ECDSA.recover(_signedMessageHash, _sig);
}
```

The `mintAuthority` address is set in the `Shifter` constructor. There is no validation present to check that this address is not address zero. Furthermore, the owner can at any time call `updateMintAuthority` to set a new `mintAuthority`. This function does also not check for address zero.

CHAINSECURITY recommends either to not allow `mintAuthority` address to be set to address zero, or to revert the transaction when the `ECDSA.recover` function returns address zero.

Fixed: REN solved the problem by adding a check to ensure that the `mintAuthority` address cannot be set to address zero.

Design Issues

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

Not using modifier for role based access L ✓ Fixed

In the `DarknodePayment` contract the `changeCycle` function can only be called by the `cycleChanger` account. This is enforced by a `require` statement on the first line of `changeCycle`.

```
require(msg.sender == cycleChanger, "DarknodePayment: not cycle changer");
```

It is considered good practice to control the role based access using function modifiers. REN could consider using a modifier for this check.

Fixed: REN added a new modifier `onlyCycleChanger` and uses it in the `changeCycle` function.

Division by 2 leaves 1 wei if odd L ✓ Fixed

The `DarknodeRegistry.slash` function divides the `penalty` by 2, then sends this `reward` to both the challenger and `DarknodePaymentStore`. If the `penalty` is odd, dividing by 2 will result in 1 wei being leftover. REN could consider calculating both rewards such that they are exact.

Fixed: REN solved the problem by sending half the penalty to the `_challenger` address, and the remaining penalty to the `DarknodePaymentStore` contract.

`slash` function does not check for existence M ✓ Fixed

The `slash` function does not check if the Darknode exists. If it does not exist there will be zero REN send to both the challenger and `DarknodePaymentStore`. This is followed by the `LogDarknodeSlashed` event being emitted. This does not lead to tokens being incorrectly sent. Still, REN should consider adding a check to make sure the Darknode exists.

Fixed: REN solved the problem by adding a check in the `slash` function to ensure that the Darknode exists.

Code duplication in `Shifter` L ✓ Fixed

In the `ShifterRegistry` contract the following functions have the same code:

- `getShifters`: To get the registered shifters from LinkedList.
- `getShiftedTokens`: To get the registered shiftedTokens from LinkedList.

The above two function use the same code to generate the list of items from the LinkedList. It is considered bad practice to have code duplication. CHAINSECURITY recommends adding a generic function that can then be used by `getShifters` and `getShiftedTokens`.

Fixed: REN removed the code duplication by adding a new generic `elements` function inside the `LinkedList` contract.

`safeTransferFromWithFees` is non-ERC20 compliant M ✓ Acknowledged

The `safeTransferFromWithFees` function will use `SafeERC20.safeTransferFrom` to transfer tokens and return the amount of tokens transferred. The ERC20 standard `transferFrom` function does not include an optional fee on-top of the transfer amount. By expecting a token to do this, REN is expecting a token to be non-ERC20 compliant. If an ERC20 token wants to implement a `transferFromWithFees` it should be named differently than `transferFrom`.

Any ERC20 compliant token will only transfer the amount specified when calling `transferFrom`. In such cases, the `safeTransferFromWithFees` function is unnecessary. The function will call `Math.min` with two equal values. Furthermore, the input `value` will always be equal to the returned amount. Therefore, REN could simply do `SafeERC20.safeTransferFrom` in the `deposit` function.

Acknowledged: REN acknowledged that the `transferFromWithFees` function was introduced because they encountered tokens (DGX and TUSD) that can include an updatable fee at any time. These two tokens use an internal fee (someone sends N dollars, but the receiver actually gets $N - fee$). Unlike some other tokens that have external fees, and some even use things like demurrage.
The interface is not meant to be a replacement for ERC20s, but something that can wrap ERC20s to ensure that the caller of the `transfer` function can know the actual amounts transferred regardless of how fees are calculated (by checking balances before and after the transfer).

Using `string` as mapping key L ✓ Acknowledged

In the `ShifterRegistry` contract a mapping with `string` as key is defined as follows:

```
mapping (string=>address) private tokenBySymbol;
```

This is seen as code smell, as there could be two different strings which look the same using unicode characters, but are actually different.

Acknowledged: REN acknowledged that this trade-off was made to allow third-party contracts/programs to look up shifters by their token's symbol. Only the owner can register new tokens and the symbol is read directly from the token.

Unnecessary local variable L ✓ Fixed

In the `register` function the `bond` local variable is initialized with `minimumBond`:

```
// Use the current minimum bond as the darknode's bond.
uint256 bond = minimumBond;
```

However, the `bond` variable is not modified later on in the function. Hence, there is no need for this variable and REN could instead directly use `minimumBond`.

Fixed: REN solved the problem by removing the local variable `bond` from the function.

Darknode registration address zero allowed H ✓ Fixed

Anyone is allowed to register a Darknode with any address, including address zero. By allowing address zero to be registered, the links in the LinkedList can get messed up. However, since the LinkedList items are only retrieved by key, the application doesn't suffer from this.

If the LinkedList is empty, appending address zero has no effects besides `inList` being set to `true`. If on the other hand there already exist some Darknodes in the LinkedList, the list becomes split in a way. Any time address zero is appended to a non-empty LinkedList, it will update the address zero `previous` and `next` pointers to point to itself (address zero). The first `address` appended after address zero was added will have it's `previous` pointer set to address zero, instead of the existing last item. This comes down to the address zero entry being the `previous` and `next` of more than one item.

Although it doesn't have any negative effects on the application flow, CHAINSECURITY thinks this should not be possible. REN is advised to disallow the insertion of address zero in the LinkedList.

Fixed: REN solved the problem in the `LinkedList` contract by not allowing the zero address as element.

Removing last array item L ✓ Fixed

The `_deregisterToken` function removes the last element of an array by doing:

```
registeredTokens.length = registeredTokens.length.sub(1);
```

Solidity contains a `pop` function to remove the last element of an array. This has build-in underflow checking and costs less gas. Therefore, REN could consider using `array.pop()` to remove the last element of an array.

Fixed: REN solved the problem by using the `pop()` function on the `registeredTokens` array to remove the last element.

Depositing unregistered tokens possible M ✓ Fixed

Registering a token can only be performed by the owner and will be pending until the next cycle starts. Darknodes can only claim tokens that have been registered. However, depositing tokens is allowed for any token at any time by anyone. CHAINSECURITY thinks only allowing deposits for registered tokens would improve the design.

Fixed: REN solved the problem by allowing deposits of only registered tokens.

Recommendations / Suggestions

☐ A Darknode needs to be registered by calling `DarknodeRegistry.register` and passing the address of the Darknode. The caller of this function will be set as the Darknode owner. However, these is no check preventing these two addresses being the same. REN could add a check that prevents these two addresses being the same.

☑ The `Shifter` contract is not inheriting the `IShifter` interface.

☑ The `Claimable.transferOwnership` function allows transferring ownership to another address. There is no check that prevents the new owner being the same as the current owner. REN could consider adding such a check.

☑ In the `updateShifter` function the `currentShifter` variable is initialized. However, on the next line the variable is not used.

```
address currentShifter = shifterByToken[_tokenAddress];
require(shifterByToken[_tokenAddress] != address(0x0), "ShifterRegistry:
    token not registered");
```

REN could use the variable on the second line above.

☑ In the `removeShifter` function the mapping entry is updated like below:

```
shifterByToken[tokenAddress] = address(0x0);
tokenBySymbol[_symbol] = address(0x0);
```

REN could use `delete` to remove this mapping entry. REN uses `delete` in several other places in the code.

☑ In the `shiftIn` function an expression is unnecessarily surrounded by brackets:

```
uint256 absoluteFee = (_amount.mul(shiftInFee)).div(BIPS_DENOMINATOR);
```

The brackets are unnecessary since multiplication would be performed first, followed by the division.

The same applies to the `shiftOut` function:

```
uint256 absoluteFee = (_amount.mul(shiftOutFee)).div(BIPS_DENOMINATOR);
```

☑ There a couple of events inside the contracts which have arguments that are not indexed, but where it makes sense to index them. These events are:

`DarknodeRegistry`

- `LogSlasherUpdated`
- `LogDarknodePaymentUpdated`

`DarknodePayment`

- `LogCycleChangerChanged`
- `LogTokenRegistered`
- `LogTokenDeregistered`
- `LogDarknodeRegistryUpdated`

☑ The `fromBytes32` function performs the following cast on the first line:

```
bytes32 value = bytes32(uint256(_value));
```

However, this is unnecessary as `_value` is already of types `bytes32`.

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.

In this section, CHAINSECURITY mentions general issues which are relevant to REN's code, but do not require a fix. Additionally, in this section CHAINSECURITY clarifies or extends the information from the previous sections of this security report. This section, therefore, serves as a reminder to REN and to raise awareness of these issues among potential users.

Dependence on block time information

REN uses `block.timestamp` / `now` inside the `DarknodeRegistry` 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[5].

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.

There are multiple places throughout the contracts where REN does division before multiplication. For example inside the `slash` function. However, CHAINSECURITY expects these errors to have a negligible effect due to use of 18 decimals.

[5] https://consensys.github.io/smart-contract-best-practices/recommendations/#the-15-second-rule

Disclaimer

Contact:

ChainSecurity AG
Krähbühlstrasse 58
8044 Zurich, Switzerland
contact@chainsecurity.com