PUBLIC

Security Audit

of Mt Pelerin Bridge Smart Contracts

September 30, 2019

Produced for

by

CHAINSECURITY

Table Of Contents

Foreword

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

– ChainSecurity

Executive Summary

MT PELERIN engaged CHAINSECURITY to perform a security audit of MT PELERIN BRIDGE, an Ethereum-based smart contract system. The MT PELERIN BRIDGE project offers rule-based enforcement mechanisms for ERC20 tokens, which can be used to guarantee custom compliance checks for token transfers. To make use of this system, the ERC20 token needs to inherit from the base ERC20 contracts provided by MT PELERIN. The system is open for anyone to join their ERC20 token. Furthermore, the MT PELERIN BRIDGE project contains a contract that allows the selling of tokens for ETH as well as a contract that allows voting on a set of resolutions.

CHAINSECURITY audited the smart contracts which are going to be deployed on the public Ethereum chain. The `Exchange.sol` contract was out-of-scope. The MT PELERIN BRIDGE's contract code was analyzed for generic security vulnerabilities using state-of-the-art tools. Additionally, a thorough manual audit of the contract was performed by CHAINSECURITY's experts to ensure compliance with the latest security standards and best practices.

During the audit, CHAINSECURITY was able to help MT PELERIN in addressing several security, trust and design issues of high, medium and low severity. All discovered issues were fixed, addressed or acknowledged and hence, no security concerns remain at this time.

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

Scope of the Audit

Source code files received	August 29, 2019
Git commit	`28aeb06e4dc782a33172ef730d4b966ac8b3122c`
EVM version	BYZANTIUM
Initial Compiler	SOLC compiler, version 0.5.2
Final code update received	September 25, 2019
Final commit	`053b51ef69298ca429b91ea47a4e43e679a979b7`

Depth of the Audit

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.

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

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

In Scope	File	SHA-256 checksum
☑	./access/Operator.sol	50a0e6f02332739ba28cc95598a8601116467cfe9c98daf9612bb42c8cfb10c7
☑	./interfaces/IAdministrable.sol	3ca173e561839cfd3f17c8c0cb4c2f29b87b82ae4610105438c921bd68d7e6e6
☑	./interfaces/IComplianceRegistry.sol	d522af35efaf96c25fccaa1f9b3aedf141c4d37150329f61ee0c333cd2e46da3
☑	./interfaces/IERC20Detailed.sol	225cc90da5247e34759f13f2a9f3f91d69c4caf2e0ba51ce9484d4b5ac250bb7
☑	./interfaces/IGovernable.sol	a2300b7a7c43fe60d7ae22fcef48d23c601bf8c618c6194c2b5225f440df37ba
☑	./interfaces/IMintable.sol	b2edfb284ae2ffd695e2ade54ef77d4524f979c983543822846ffe205b9851b1
☑	./interfaces/IOwnable.sol	32d0b398e204761b87a33b331606f7bd022727463de560f3ca665bea95bedfc8
☑	./interfaces/IPriceOracle.sol	e1e7e5e2ffe650165e73f3d1c946e600bacaea60ddd3e7fb9d7e96fee167fcfb
☑	./interfaces/IPriceable.sol	9f3b7f1bcb18a42f287642e3630c68a77107dd6982792ae4b04f31f3707ca807
☑	./interfaces/IProcessor.sol	51d4f7e341a838c3a46a9c6824eaaf4c20ec4a0f8b53365f09c14c6c8208abe7
☑	./interfaces/IRulable.sol	b1b959137b0d2fd29148a3a4786f612ef5db918bec777f74cf9e8eb41144910a
☑	./interfaces/IRule.sol	26f5a32929bb5965a757420b0eafcc6a67b579d4dae0700171cf1649deb91d7a
☑	./interfaces/IRuleEngine.sol	434732965dd92f1ef3b5e265fc1113e39e3249badfe9220188f849049c6b6f3b
☑	./interfaces/ISeizable.sol	12d1c5c6b563dd675b996d423d234ef71e5950522c0191e515c4e9a5858d1004
☑	./interfaces/ISuppliable.sol	9f2bff9fff7a6bc2368275d0200b50274b8306eeef3624345f8b0a0caf43ff63
☑	./operating/ComplianceRegistry.sol	0e162caf9a0fda6ad245683b99da6416f78cc25fc24502f5ec43312a90e0d47d
☑	./operating/PriceOracle.sol	7cf2c6b649b2da2ab80d448921c7bae28655428f65cd17b7aa013065a806833b
☑	./operating/Processor.sol	8aad139d3a11ea8081f1be6f31674d0f2a3a95f61c225077764b5805b7c0ca20
☑	./operating/RuleEngine.sol	6ad4af0cd999d62fad7a45eda22fd44779185eded38e66b53818646ba59f6a5b
☑	./rules/GlobalFreezeRule.sol	cd729dc02c561c1f2cbec3cf5bf9279592751fa4e5869788440f3d5ed085340d
☑	./rules/HardTransferLimitRule.sol	8d594d34871f83df73bb6ddf51ef0ace1674d85fa01993eaa698362a973fcc08
☑	./rules/MaxTransferRule.sol	324aa776f06e3d35c42ef07271477f5119edc7a56badbec716b11cb664959658
☑	./rules/MinTransferRule.sol	ece455c81047a97701b0f6f750abad687c65de33b5bf584c4bc1244e49f9ec6b
☑	./rules/SoftTransferLimitRule.sol	b51726cb700bf090f4afaee7f3f5a97fd381ed263e6519bd5fd27ff7e9de8288
☑	./rules/UserFreezeRule.sol	3b89898aefdc091abc423f903237e53754b10aa0b8ce0f67892935ec736e493f
☑	./rules/UserKycThresholdBothRule.sol	7aeffc8b5af04bfba57e599cc37bb2e3f420abb09fae2a358902747ab9c8593b
☑	./rules/UserKycThresholdFromRule.sol	53a04ac4302fa8fa7c3c744dac035bd9bc160bba9e877c1824b3a4d94040918a
☑	./rules/UserKycThresholdToRule.sol	b9902e79bdc5d816ba253c4cc0f5a7f93f48fbee2bcc327a87aa4b00110c9cd7
☑	./rules/UserValidRule.sol	69f1430e518703fdb535a5d99f84079aa8a022f88a979ea6562b7c96697261eb
☑	./rules/YesNoRule.sol	b4d869452d458a8de6047a8da2cfd4ac9d77e9d3b2164db212494efbf6eddd2a
☑	./rules/YesNoUpdateRule.sol	da21cca74643fb6d208bc5c16aa5b34564682af91913ca24931d98afc65f685a
☑	./rules/abstract/AbstractRule.sol	9064e4b0ebe4790f727d64e303d4c245cc7e8fce9dc302de28d5b4cd9853c057
☑	./sale/TokenSale.sol	bb2c1d45a3255c5e8daf83f2e6b09eb30c2ba6234ad9cd6e8f996b3e22552b54
☑	./token/BondBridgeToken.sol	d21e21cae1960544e6dfd9d4ce72fc023c4becc104e05e118d44d1f54d3898a4
☑	./token/BridgeToken.sol	aa2947b8c999f21adec049c15ea17c6a2037d198c6624cdbc2ead698c3d1cf6b
☑	./token/CoinBridgeToken.sol	efc4795a6842cce6a04fb94f1a087a54ff7f1f2c87a38c72a00042aee70cb1e3
☑	./token/ShareBridgeToken.sol	76155c488355e5463c74a4645efe808116f1067f7a2739594b42183d746b3a2d
☑	./token/abstract/BridgeERC20.sol	f894936aeab067d10cc6a1d0b0643bc0e88556899d1dc77ff37c5f3b10cd527d
☑	./token/abstract/SeizableBridgeERC20.sol	2ea4757c982cbc5f960d2eaae506ba2d8da6252508f541c6206de20cd07cadfb
☑	./voting/VotingSession.sol	0ddda5c0db23a828900721f21d66bd2e90a8b569cdf17eb76d508585db734692

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			
Medium			
Low			

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 simply 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 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.

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

MT PELERIN has implemented a Rule Engine along with a pre-defined set of Rule contracts. A third-party can create a `BridgeToken` and register it with the `Processor`. Then, the third party also specifies which rules the token behaviour should adhere to, from the pre-defined set of rules. In the future, MT PELERIN can add more such rules and these new rules might conflict with the current set of rules supported by the Rule Engine. Such conflicts or incorrectly implemented new rules could create unexpected or incorrect behavior of the system.

System Overview

The MT PELERIN BRIDGE smart contract system implements a way for ERC20 tokens to make use of pre-defined rules that apply checks to who can send/receive (how much) tokens. To be able to make use of this system an ERC20 token needs to inherit from a base ERC20 contract provided by MT PELERIN. This base ERC20 contract connects the token's ERC20 functions to the `Processor` contract. Instead of the tokens storing their state (balances/approvals), all of this information is stored inside the `Processor` contract.

When a token transfer is initiated, the call will be forwarded to the `Processor` contract, which will apply any rules that were enabled in the token. MT PELERIN has provided twelve rules. Each of these is a separate contract that needs to be separately deployed. Once deployed, a rule can be registered in the `RuleEngine` contract. When registered inside the `RuleEngine`, every token contract that registers with the `Processor` can enable these rules so that they are applied to the transfers of that token.

Any contract can register with the `Processor`, and all information regarding a token inside the `Processor` is stored under that token's address. Therefore, a token cannot touch the data of other tokens. This means token contracts are untrusted and can be implemented as its creator wishes.

There is also a token sale contract which is connected to a single token contract. The token sale contract allows anybody to purchase tokens using ETH. Furthermore, it allows the operator to add an investment of another "refCurrency" for any investor. This can be used to add FIAT currency investments for a specific investor.

There is also a `VotingSession` contract which can be used to let users vote. The operator of the contract can define any number of "Resolutions". Each of these resolutions can contain any number of "Proposals". The operator can register voters together with a weight for their votes. Registered voters can vote on each resolution by choosing the proposal they support. It is also possible for any registered voter to choose a delegate voter.

All of the contracts (including the tokens) are upgradable by making use of the OpenZeppelin upgradability contracts.

Extra Token Features

There are three base ERC20 contracts, each inheriting from the next: `BridgeToken` -> `SeizableBridgeERC20` -> `BridgeERC20`. All three ERC20 contracts provided by client make use of all three base ERC20 contracts by inheriting from the top one (`BridgeToken`). Besides the regular ERC20 token features these three base contracts add a number of extra features.

Set Administrators The owner can add/remove one or more Administrators.

Set Seizers An Administrator can add/remove one or more Seizers.

Set Suppliers An Administrator can add/remove one or more Suppliers.

Set Trusted Intermediaries An Administrator can update the list of trusted intermediaries.

Seize tokens A Seizer can seize tokens from any address.

Mint tokens A Supplier can mint tokens to any address.

Burn tokens A Supplier can burn tokens of any address.

Set Realm An Administrator can update the (`uint256`) realm of a token.

Set Rules An Administrator can update the list of rules to enable.

Set Price Oracle An Administrator can update the address of the price oracle contract to use.

Set Processor An Administrator can update the Processor to use.

Upgrade contract The upgradability proxy contract owner can upgrade the token contract.

Convert to other currency The `convertTo` function uses the set price oracle to convert tokens to another chosen currency

System Roles

There are three base ERC20 contracts, each inheriting from the next: `BridgeToken` -> `SeizableBridgeERC20` -> `BridgeERC20`. All three ERC20 contracts provided by client make use of all three base ERC20 contracts by inheriting from the top one (`BridgeToken`).

Administrators Can update the token realm, rules, trusted intermediaries, processor, seizers, suppliers.

Suppliers Can mint/burn token to/from any address.

Seizers Can seize tokens of any address of that `SeizableBridgeERC20` token.

Trusted Intermediaries Is set inside a token and is used to update compliance information inside the `ComplianceRegistry`.

- Can register new users, update user attributes, add/remove addresses to an existing user.
- Can approve/reject on-hold transfers (only applicable when `SoftTransferLimitRule` is used).

Proxy Owner The Proxy Owner can update the implementation of any upgradable contract (all contracts are upgradable).

Operators Is a separate contract inherited by all contracts who want to have Operators. It is set by the contract owner. There can be one or multiple operators. They have different powers depending on the contract.

Price Oracle Can update the prices of token pairs.

Processor Can set the `RuleEngine` contract.

RuleEngine Can update the enabled rules.

TokenSale Can empty the contract ETH balance, set the start/end time of the sale, set the max ether balance of the contract, and invest tokens in name of another account.

GlobalFreezeRule Can enable/disable the global freeze.

SoftTransferLimitRule Can execute the before and after transfer hooks. The `Processor` should be set as operator of this contract.

HardTransferLimitRule Can execute the after transfer hook. The `Processor` should be set as operator of this contract.

ComplianceRegistry Can call `updateTransfers` and `addOnHoldTransfer`. The `SoftTransferLimitRule` and `HardTransferLimitRule` contracts should be set as operator of this contract.

Trust Model

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

Proxy Admin The proxy admin is trusted to not update the token contract to a broken version.

Owner, Operators, Administrators, Suppliers, Seizers These roles are fully trusted to act honestly at all times.

Trusted Intermediaries The trusted intermediaries are trusted by the token contract in which they are defined. But since token contracts are untrusted, the trusted intermediaries are also untrusted from the perspective of the `ComplianceRegistry`.

User A regular user is not trusted and is assumed to be potentially malicious.

Best Practices in MT PELERIN'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 fulfillment of general guidelines.

Hard Requirements

These requirements ensure that the MT PELERIN'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 MT PELERIN BRIDGE. While the test suite is not a component of the audit, a good test suite is likely to result in better code.

The overall quality of the provided tests is good. However, the tests seem to only test the happy path in most cases. Furthermore, some Rule contracts do not have any tests.

[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.

Anyone can invest when not open  ✓ Fixed

The fallback function of `TokenSale.sol` can be called even when the contract is paused or closed.

```
function () external payable {
  require(msg.data.length == 0, "TS08");
  _invest(msg.sender, msg.value, 0);
  _rebalance();
}

function investEther() public isOpen whenNotPaused payable {
  _invest(msg.sender, msg.value, 0);
  _rebalance();
}
```

This breaks the whole concept of a token sale which has a start and end date, as ether investments can be initiated at any time. Since it's easier for users to send ether to a contract, than to a function inside a contract, it is likely most users will call the fallback function.

MT PELERIN should also apply the checks in the fallback function.

Likelihood: High
Impact: High

Fixed: MT PELERIN added the `isOpen` and `whenNotPaused` modifiers to the fallback function.

Token `name` argument not validated ✓ Fixed

In the `register` function inside the `Processor` contract, the token `name` is checked to ensure it has not already been registered.

```
function register(string calldata _name, string calldata _symbol, uint8
    _decimals) external {
  require(keccak256(abi.encodePacked(_tokens[msg.sender].name)) ==
      keccak256(""), "TR01");
  _tokens[msg.sender].name = _name;
  _tokens[msg.sender].symbol = _symbol;
  _tokens[msg.sender].decimals = _decimals;
}
```

However, it is not ensured that the `_name` function parameter is not an empty string. If `register` is called with empty string as `_name`, it will register the token having an empty string. If a token registers with an empty name, and later on calls register again but with a different decimals count, it could affect converting of tokens to an exchange rate.

MT PELERIN should prevent a token being registered with an empty name.

Likelihood: Low
Impact: Medium

Fixed: MT PELERIN added checks for empty `_name`/`_symbol`.

`transferFrom` does not decrease `allowance` H ✓ Fixed

The `transferFrom` function present in the `Processor` and `BridgeERC20` contracts does not decrease the `allowance` after a successful `transferFrom`. Therefore, if address `A` sets allowance for address `B` to spend value `S`. Address `B` can now keep calling `transferFrom` with value `S`, transferring out `S` tokens of `A` with each call, without lowering the `allowance`. Still, the victim `A` has to first approve allowance for attacker `B`, thereby lowering the likelihood as `A` likely does not know `B`.

MT PELERIN should decrease the allowance after a successful `transferFrom`.

Likelihood: Medium
Impact: High

Fixed: MT PELERIN now decreases the allowance after `transferFrom`.

Possible DoS using realm and untrusted token contract C ✓ Fixed

Anybody can deploy their own (token) contract and register as a token inside the `Processor`. These "token" contracts are untrusted and can thus be implemented in any way as long as they call the correct functions in `Processor`, and implement the correct token interfaces.

Inside the MT PELERIN smart contract system most data is stored under the token contract's address. Thereby, preventing one token contract from interfering with another. However, one piece of data that is shared among registered token contracts is the `realm`. This can be used by an attacker to DoS users of tokens that use at least one of: `SoftTransferLimitRule`, `HardTransferLimitRule`.

Each token contract has a `uint256 _realm`, which can be updated at any time by one of its Administrators using `setRealm`. The realm is used by the `HardTransferLimitRule` and `SoftTransferLimitRule` to apply monthly/yearly transfer caps. These rules must first be deployed and set inside the `RuleEngine` by its Operator. Once they exist, any token contract can request these rules to be applied to its transfer operations. Instead of using the token contract address, these rules use the token realm. The rules continuously retrieve the current token realm by calling `IGovernable(_token).realm()`.

An attacker could create a malicious token contract, enable the rule `SoftTransferLimitRule`, and register with an existing `Processor`. It would now be possible for the attacker contract to interfere with the applied limit rules of other tokens that are registered inside this `Processor`. The attacker contract can use the same realm as that of the victim token contract(s). Doing so will also store the transfer statistics of the attacker contract in the realm used by the victim token(s). This can be turned into a DoS of any chosen victim address inside a token contract in the realm. The attacker calculates how much tokens are needed to reach the monthly cap for a specific user. The attacker token contract transfers that amount of tokens to the victim address. The victim address will now temporarily be unable to do any more transfers, as he has reached his monthly cap inside this realm.

MT PELERIN should reconsider the usage of realms such that untrusted token contracts cannot interfere with an existing realm. Restricting who can register as a token inside `Processor` could also be considered.

Likelihood: High
Impact: High

Fixed: MT PELERIN updated `realm` to be an `address` and only allows the owner/administrator of the first contract in a realm to let another contract join the realm.

Trusted intermediaries allowed to transfer multiple times C ✓ Fixed

In the `ComplianceRegistry` contract the `processOnHoldTransfers` function is used by a trusted intermediary to either approve or reject an existing on-hold transfer. For this they need to pass in the index in the `transfers` array and either approve or reject in the `transferDecisions` array. The function will loop through the array of transfer indexes and decisions as shown below.

```
for (uint256 i = 0; i < transfers.length; i++) {
  onHoldTransfers[msg.sender][transfers[i]].decision = transferDecisions[i];
  if (transferDecisions[i] == TRANSFER_APPROVE) {
    _approveOnHoldTransfer(transfers[i]);
  } else {
```

```
    _rejectOnHoldTransfer(transfers[i]);
  }
}
```

The first line in the above loop updates the decision for the trusted intermediary and calls the correct function to either approve or reject this currently on-hold transfer. However, there is no check to ensure that the transfer at a certain index is currently on-hold or already approved/rejected. Hence, even when a transfer is already approved/rejected, it can again be approved/rejected, leading to another token transfer.

This issue is limited by the "cleanup" loop at the end of the `processOnHoldTransfers` function.

```
for (uint256 i = minBoundary; i <= maxBoundary; i++) {
  if (onHoldTransfers[msg.sender][i].decision != TRANSFER_ONHOLD) {
    minBoundary++;
  } else {
    break;
  }
}
onHoldMinBoundary[msg.sender] = minBoundary;
```

This loop will remove all "leading" transfers that are not in state "on-hold". If a transfer is not on-hold it is either approved, rejected or cancelled. If the above described issue is to be used by a trusted intermediary it is necessary to have a still on-hold transfer in front of the transfer that the trusted intermediary wants to approve/reject multiple times. Since in that case the cleanup loop will not remove the approved/rejected transfer from the array as there is still an on-hold transfer before it.

MT PELERIN should prevent already approved/rejected transfers from being approved/rejected again by a trusted intermediary.

Likelihood: High
Impact: High

Fixed: MT PELERIN added a check such that only on-hold transfers can be approved/rejected.

Floating Pragma ✓ Fixed

MT PELERIN uses a floating pragma solidity ^0.5.2. Contracts should be deployed with the same compiler version and flags that they have been used during testing and 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[3].

Likelihood: Low
Impact: Low

Fixed: MT PELERIN set pragma to static version `0.5.2`.

Unrestricted write to storage ✓ Acknowledged

In the contract `YesNoUpdateRule` there an after transfer hook, which increment the counter by one.

```
  function afterTransferHook(
    address /* _token */, address /* _from */, address /* _to */, uint256 /*
        _amount */, uint256 /* _param */)
    external returns (bool)
  {
    updateCount++;
    return true;
  }
```

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

This function is not restricted with any condition or access rights. Hence, anyone is allowed to call this function and can increment `updateCount`.

This variable is only used to keep track of the number of accepted transfers and is not used in any of the code. Therefore, the impact of this issue is low. This also means the likelihood is low as there is no good reason for an attacker to spend gas to increment this variable.

Likelihood: Low
Impact: Low

Acknowledged: MT PELERIN explained this file is only used for testing.

Theoretical integer overflow possible L ✓ Acknowledged

In the `YesNoUpdateRule.afterTransferHook` function the `updateCount` state variable is increased by one without checking for the overflow.

```
function afterTransferHook(
  address /* _token */, address /* _from */, address /* _to */, uint256 /*
      _amount */, uint256 /* _param */)
  external returns (bool)
{
  updateCount++;
  return true;
}
```

Hence, theoretically integer overflow is possible. However, for this to happen the `afterTransferHook` function needs to be called 2^256 times.

Likelihood: Low
Impact: Low

Acknowledged: MT PELERIN explained this file is only used for testing.

No return check H ✓ Fixed

There are some functions in the ERC20 standard API which return a `bool` to notify caller about the success/-failure of the function execution.

In `ComplianceRegistry` there are the `_approveOnHoldTransfer` and `_rejectOnHoldTransfer` functions which uses the ERC20 `transfer` function.

```
IERC20Detailed(transfer.token).transfer(transfer.to, transfer.amount);
...
IERC20Detailed(transfer.token).transfer(transfer.from, transfer.amount);
```

These functions do not verify that the return value must be `true` to continue execution of the function. Hence, it is possible that the `transfer` function failed and returned `false`, but the execution continuing as if it succeeded.

CHAINSECURITY recommends enclosing these function calls in a `require` statement. This ensures that the function execution only proceeds when the token transfer was successful. MT PELERIN can also use the `SafeERC20` contract from OpenZeppelin, which also deals with tokens that do not return `true` on success. Therefore, `SafeERC20` is safer than wrapping in `require`.

Likelihood: Medium
Impact: High

Fixed: MT PELERIN wrapped the ERC20 transfer calls in a `require`. CHAINSECURITY would like to note that there are still also some ERC20 tokens which do not return `true` on success, and that these would throw on success when wrapped in a `require`. To correctly handle all types of ERC20 tokens, OpenZeppelin `SafeERC20` should be used.

Trust Issues

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

Operator can change schedule at any time M ✓ Fixed

In contract `TokenSale` the below function allows an Operator to change the schedule and specify the start and end time to allow investments.

```
function setSchedule(uint256 _startAt, uint256 _endAt) public onlyOperator {
  require(_startAt < _endAt, "TS02");
  startAt = _startAt;
  endAt = _endAt;
}
```

There is no limit on when or how many times this function is called by the Operator. This could result in unexpected behavior. For example, if a TokenSale is currently "open", it could be set back to "closed" by updating `startAt` to a time in the future. In that case the `investEther` function cannot be called by potential investors until `startAt` has passed (again). Another example would be the Operator closing the token sale by updating `endAt` to a time in the past.

In the current implementation users need to trust the Operator to not perform such actions. MT PELERIN could consider not allowing an Operator to update `startAt` and `endAt` once a token sale has started.

Fixed: MT PELERIN added a modifier to only allow calling this function if the token sale did not yet start.

All contracts are upgradable H ✓ Acknowledged

All of the contracts of MT PELERIN BRIDGE are upgradable using OpenZeppelin. Furthermore, it's possible for untrusted tokens to make use of the MT PELERIN BRIDGE smart contract system. Because of this the upgradability of the smart contracts becomes a trust issue as all of the MT PELERIN BRIDGE contracts can be replaced at any time, possibly (negatively) affecting untrusted token contracts that are currently making use of the MT PELERIN BRIDGE smart contracts.

Acknowledged: MT PELERIN explained that upgradable contract brings flexibility for the type of services they will provide, and that they will mitigate the trust issues by having proper governance and policies.

Design Issues

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

No event emission in Operator role L ✓ Fixed

In the contract `Operator` the owner of the contract can add or remove Operators. When adding/removing an Operator, no event is emitted.

MT PELERIN could consider emitting an event on addition or removal of Operators.

Fixed: MT PELERIN added the `OperatorAdded` and `OperatorRemoved` events.

`maxEtherBalance` is not ether L ✓ Fixed

The `maxEtherBalance` variable inside `TokenSale.sol` is initialized to `100`. Since the `ether` part is missing, the current value means 100 wei. The `maxEtherBalance` variable is used in `_rebalance`, which sends wei out of the contract if it exceeds `maxEtherBalance`. Because the value is so low (100 wei), every wei invest call will trigger a wei `send` inside `_rebalance`. Still, it is possible to update the `maxEtherBalance` if the Operator calls `setMaxEtherBalance`.

Fixed: MT PELERIN added the `ether` unit keyword.

Could rebalance inside `setMaxEtherBalance` L ✓ Fixed

The `setMaxEtherBalance` function updates the `maxEtherBalance`. If it is updated to a lower value, than the current ether balance might exceed the new max. Therefore, MT PELERIN could consider calling `_rebalance` after updating the `maxEtherBalance`.

Fixed: MT PELERIN now calls `_rebalance` inside `setMaxEtherBalance`.

`_rebalance` not clearing the ETH balance M ✓ Fixed

The `_rebalance` function will check if the amount of wei in the contract exceeds `maxEtherBalance`. If it does, it will transfer all wei above the max to the `etherVault`. Besides `_rebalance`, there is also `withdrawEther`, which lets an Operator transfer the entire contract's wei balance to the `etherVault`.

In the current implementation `_rebalance` might keep transferring excess wei once the max has been reached. Once the max is reached, the amount of wei above the max will be transferred to the `etherVault`. This will lower the contract's wei balance again to the max. Therefore, every next investment will again make the max be exceeded, causing another wei transfer to the `etherVault`. If the Operator calls `withdrawEther`, its balance will be set to zero, and it would start over.

CHAINSECURITY thinks this design could be improved by transferring the entire (or a percentage of the) max allowed balance once the max is reached.

Note that we used wei in the above example, as the current implementation does not multiply `maxEtherBalance` by `10^18` and initially sets it to `100` instead of `100 ether`.

Fixed: MT PELERIN now transfers the entire balance to the `etherVault` when rebalancing.

Missing event emit in `setPrices` L ✓ Fixed

The price oracle emits a `PriceSet` event when `setPrice` is called. However, the batch version of that function does not emit a `PriceSet` event. MT PELERIN could consider also emitting the `PriceSet` event inside `setPrices`.

Fixed: MT PELERIN now also emits the event inside the batch function.

`getOnHoldTransfers` does multiple identical subtractions L ✓ Fixed

The `getOnHoldTransfers` view function computes `maxBoundary-minBoundary` five times. MT PELERIN should consider storing the result of `maxBoundary-minBoundary` in a variable and use that.

Fixed: MT PELERIN now does the subtraction once and saves the result in a variable.

Off-by-one `maxBoundary` L ✓ Fixed

The `addOnHoldTransfer` adds a new on-hold transfer to the mapping `onHoldTransfers`. The index of this on-hold transfer is computed as follows.

```
uint256 index = onHoldMaxBoundary[trustedIntermediary]++;
onHoldTransfers[trustedIntermediary][index] = OnHoldTransfer(
  token, TRANSFER_ONHOLD, from, to, amount
);
```

The different values of the index and max boundary will be:

transfer	index	onHoldMaxBoundary[trustedIntermediary]
#1	0	1
#2	1	2
#3	2	3
#4	3	4
#5	4	5

There are two `for` loops inside `processOnHoldTransfers` and `cancelOnHoldTransfers`. Both of these loops are defined as:

```
for (uint256 i = minBoundary; i <= maxBoundary; i++) {
```

Since the `maxBoundary` will be 1 more than the index of the last item, these loops should instead use `i < maxBoundary`. Nevertheless, this does not cause any issues as it is an index into a mapping, which can be any `uint256`. It also doesn't cause a problem in this loop. Still, MT PELERIN could fix the off-by-one.

Fixed: MT PELERIN replaced `<=` with `<`.

Code duplication for Hook functions L ✓ Fixed

In many of the Rule contracts the before and after transfer hooks are defined like below:

```
function beforeTransferHook(
  address /* _token */, address /* _from */, address /* _to */, uint256 /*
     _amount */, uint256 /* _param */)
  external returns (uint256, address, uint256)
{
  require(true == false, "RU02");
}

function afterTransferHook(
  address /* _token */, address /* _from */, address /* _to */, uint256 /*
     _amount */, uint256 /* _param */)
  external returns (bool)
{
  require(true == false, "RU02");
}
```

The same code is implemented in most of the Rule contracts. MT PELERIN could make a base contract to put these function in and let all Rule contracts inherit from this base contract. That way only contracts that want to override any of these functions need to implement these functions themselves. The base contract could also contain the validity constants, as those are also shared by all Rule contracts. This would reduce the code duplication and improve the overall structure of the Rule contracts.

Fixed: MT PELERIN implemented an `AbstractRule` contract which is inherited by all Rule contracts.

Unbounded loops M ✓ Fixed

The `processOnHoldTransfers` function present in the `ComplianceRegistry` contract has the following loop:

```
for (uint256 i = minBoundary; i <= maxBoundary; i++) {
  if (onHoldTransfers[msg.sender][i].decision != TRANSFER_ONHOLD) {
    minBoundary++;
  } else {
    break;
  }
}
```

When there are multiple on-hold transfers exist for a certain trusted intermediary, the loop might run through multiple iteration and could consume more gas than the block gas limit, resulting in an out-of-gas exception.

CHAINSECURITY would also like to note that with the upcoming Istanbul hard-fork multiple gas cost increases will take effect. Especially EIP-1884, which will quadruple the gas cost of `SLOAD`. This would likely mean all of the current loops will cost more gas and therefore the block gas limit will be reached sooner than pre-Istanbul.

Fixed: MT PELERIN made updating the boundaries optional in `processOnHoldTransfers`/`cancelOnHoldTransfers` and added a separate function to update boundaries in batches.

Missing check in `setVotingPeriod` M ✓ Fixed

The `setVotingPeriod` function lets an Operator update the vote start and end timestamp. There are currently no checks on the values of these two timestamp. CHAINSECURITY would expect some checks to make sure `votingStart` is above zero and that `votingStart` is below `votingEnd`.

Fixed: MT PELERIN added the checks.

`delegateVote` missing checks voter/delegate registered L ✓ Fixed

The `delegateVote` function allows a voter to delegate their vote to another address. There are currently no checks to make sure the caller of this function is a registered voter. Also, there is no check that the delegate address is a registered voter. MT PELERIN should consider adding such checks.

Fixed: MT PELERIN added an `isVoter` modifier.

Unnecessary return `bool` L ✓ Fixed

There are a number of functions which return a `bool`. All of these functions will throw on error. Therefore, they will never return `false`. However, some of these functions belong to an ERC20 token. Because the ERC20 token standard requires `true` to be returned on success, some functions need to `return true`. These functions are `transfer`/`transferFrom`/`approve` inside `BridgeERC20`. All of the other functions that return a `bool` would also work without returning a `bool`.

The functions for which the returning of a boolean can be removed are:

- `SeizableBridgeERC20.addSeizer`
- `SeizableBridgeERC20.removeSeizer`

- SeizableBridgeERC20.seize
- BridgeERC20.addAdministrator
- BridgeERC20.removeAdministrator
- BridgeToken.addSupplier
- BridgeToken.removeSupplier
- BridgeToken.mint
- BridgeToken.burn
- BridgeToken.setRules
- Processor.burn
- Processor.mint
- Processor.seize
- Processor.approve
- Processor.increaseApproval
- Processor.decreaseApproval
- ComplianceRegistry.addOnHoldTransfer
- ComplianceRegistry.updateTransfers
- ComplianceRegistry._updateTransfers

Fixed: MT PELERIN removed the unnecessary returning of booleans.

Incorrect return in `UserFreezeRule` H ✓ Fixed

The `UserFreezeRule` contract contains two internal functions to check if the sender and/or receiver are currently not frozen and therefore are allowed to send/receive. Both of these functions end with an almost identical ternary expression:

```
return
  (userAttributes[0] == FREEZE_DIRECTION_RECEIVE || userAttributes[0] ==
     FREEZE_DIRECTION_BOTH)
  && (
    // user.freeze_start <= now && user.freeze_end > now
    userAttributes[1] <= now && userAttributes[2] > now

      // user is currently frozen >>>
      ? (userAttributes[3] == FREEZE_INVERTED_NO)

      // user is currently not frozen >>>
      : (userAttributes[3] == FREEZE_INVERTED_YES)
  );
```

The problem lies in the last two checks. If a user is currently frozen (`now` is between start and end freeze time), and his freeze inversion is set to "no" (=0), than the last check `userAttributes[3] == FREEZE_INVERTED_NO` will return `true`. Which means the user is allowed to send even though he is frozen. It seems the `FREEZE_INVERTED_NO` and `FREEZE_INVERTED_YES` should be in each others place. CHAINSECURITY would also like to note that this rule contract is one of the few for which there are no tests, which likely would have uncovered this issue.

Any token which has this rule enabled, and whose users have the freeze direction key set, will get back incorrect results. Users that are frozen are allowed to send, while unfrozen users are not.

Fixed: MT PELERIN exchanged `FREEZE_INVERTED_NO` and `FREEZE_INVERTED_YES` and added tests.

`freezeAll` missing check L ✓ Fixed

The `freezeAll` function inside `GlobalFreezeRule.sol` allows an Operator to set a global freeze until a specific timestamp. There is however no check that the timestamp is in the future. MT PELERIN should add such a check.

Fixed: MT PELERIN added the check.

`GlobalFreezeRule` no unfreeze function L ✓ Fixed

The `GlobalFreezeRule` contract allows an Operator to enable a global freeze until a specific timestamp. There is however no function to disable the current global freeze. Still, it is possible to disable a freeze by calling `freezeAll` with a timestamp that has already passed. But that would again emit an event `GlobalFreeze(_until)`, which would not make it clear a global freeze was disabled/removed. MT PELERIN should consider adding an `unfreezeAll` function.

Fixed: MT PELERIN added an `unfreezeAll` function.

Pair encoding in `PriceOracle` might clash M ✓ Fixed

Inside `PriceOracle` the key at which the price of a pair of currencies is stored is defined as:

```
bytes memory _pair = abi.encodePacked(bytes(_currency1), bytes(_currency2));

/* Returns 32 first bytes of concatenated strings */
assembly { result := mload(add(_pair, 32)) }
```

Since not all tokens have three decimals, there might be clashes where a pair of tokens ends up having the same 32 bytes key as another token pair, as shown in the below table.

_currency1	_currency2	result
A	BCDE	ABCDE
AB	CDE	ABCDE
ABC	DE	ABCDE
ABCD	E	ABCDE

Although unlikely, and only the Operator being allowed to add a token pair, CHAINSECURITY would still advise MT PELERIN to mitigate this issue. MT PELERIN could for example use a nested mapping instead of concatenating the two token symbols.

Fixed: MT PELERIN now uses a mapping of mapping.

`validateTransferWithRules` first check should throw M ✓ Fixed

The `validateTransferWithRules` view function on the first line checks that the two rule arrays are of equal length. If they are not, the function returns `(false, 0, 0)`. Unlike all the other checks that are being done in this function, this first check does not return any meaningful information so that the caller knows the problem is that the two rule arrays have a different length. Also, the first rule will also have id zero. Therefore, returning `0, 0` for the different arrays error overlaps with the first rule's return value. MT PELERIN could instead throw with a descriptive error message if the first check fails.

Fixed: MT PELERIN now uses a `require` for the different-array-lengths check and added a new error code.

Limit transfer rules problems with other rules L ✓ Acknowledged

The SoftTransferLimitRule contract's beforeTransferHook function returns a different _to address than the original _to address. The address that is returned as _to is the address of ComplianceRegistry. This is necessary to put these transfers "on-hold" by transferring the tokens to ComplianceRegistry instead of the actual recipient. A trusted intermediary can later on approve such an on-hold transfer in which case the tokens will be send from the ComplianceRegistry to the actual recipient (the original _to).

The beforeTransferHook function of all token-enabled rules is called in a loop inside Processor.transferFrom.

```
/* Before transfer hook execution if needed */
for (i = 0; i < rules.length; i++) {
  if (ruleValid[i] == TRANSFER_VALID_WITH_BEFORE_HOOK) {
    (ruleValid[i], _to, _value) = rules[i].beforeTransferHook(msg.sender,
       _from, _to, _value, rulesParams[i]);
    require(ruleValid[i] > TRANSFER_INVALID, "RU03");
  }
}
```

The problem is that since the limit rules return the address of the ComplianceRegistry as _to, every next iteration of the loop (so every next applied rule) will use the address of the ComplianceRegistry as _to instead of the original _to. Therefore, rules that are executed after a limit transfer rule that implement a beforeTransferHook and/or afterTransferHook could give back incorrect results.

Currently only the SoftTransferLimitRule implements a beforeTransferHook. Furthermore, the only other contract which implements an afterTransferHook is the HardTransferLimitRule. CHAINSECURITY thinks it is unlikely that both the soft and hard transfer limit rules will be used by a token, as this would cause the issue that they both update the same stored transfer statistics.

Still, in case a new rule is added in the future, the above described issue might manifest itself if the new rule implements a beforeTransferHook and/or afterTransferHook.

Acknowledged: MT PELERIN explained that the functional documentation for users will clearly state that some rules are mutually exclusive. If needed, this mutual exclusiveness will be enforced in the frontend interfaces.

Approving on-hold transfers incorrectly calls updateTransfers H ✓ Fixed

An on-hold transfer will be approved inside ComplianceRegistry._approveOnHoldTransfer. This function will first call _updateTransfers to update the monthly/yearly transfer statistics of the **from** and to address. Afterwards it will execute:

```
IERC20Detailed(transfer.token).transfer(transfer.to, transfer.amount);
```

This will initiate a transfer of tokens from ComplianceRegistry to the actual recipient. The transfer will again go through the processor, and therefore the rules will be applied again. Therefore, the SoftTransferLimitRule will be executed again.

If the soft transfer limit is again exceeded a new on-hold transfer will be added. Which leads to incorrect results as _updateTransfers was already executed and incremented the monthly/yearly statistics of **from** and to, even though the transfer still didn't succeed.

If on the other hand the soft transfer limit is not exceeded the afterTransferHook of SoftTransferLimitRule is executed. This will again call ComplianceRegistry.updateTransfers. Therefore, updateTransfers is effectively executed twice. Since in the second transfer **from** is the ComplianceRegistry contract, the original **from** monthly/yearly transfer statistics is only updated once, and thus correct. However, the _to transfer statistics will be incremented twice.

CHAINSECURITY thinks this is not what MT PELERIN intended and advises to reevaluate the approval of on-hold transfers.

Fixed: MT PELERIN fixed the issue by "skipping" ComplianceRegistry._updateTransfers and the SoftTransferLimitRule when either **from** or to is the ComplianceRegistry.

`addOnHoldTransfer` missing return `true` L ✓ Fixed

In the `ComplianceRegistry` contract there is a function `addOnHoldTransfer`. This function has specified that it will return a `bool`. However, it does not have any `return` statement present. Hence, the return value will always be `false`.

Fixed: MT PELERIN removed the returning of a boolean from this function.

Recommendations / Suggestions

☑ As per the architecture diagram present at location `docs/assets/architecture.png`, the contract `BridgeToken` inherits from `IIssuable` contract. However, there is no such contract present in the project.

CHAINSECURITY recommends correcting the architecture diagram.

☑ In the `convertTo` function of the `BridgeERC20` contract the `_priceOracle` contract is used to fetch the latest price information. However, in the function there is no check to ensure that the price oracle contract address is initialized. If it is uninitialized, calling this function will result in an error with no explanation to why the transaction failed.

CHAINSECURITY recommends validating that the price oracle address is a non-zero address at the start of `convertTo`, and throwing a descriptive error message if it is not.

☑ The `_invest` function inside `TokenSale.sol` handles one of two cases. Either ETH is being invested, or the `_refCurrency` ERC20 token is being invested. Instead of having a `require` and an if/else inside this function to differentiate between these two cases, the function could simply be split into two: `_investEther`, `_investRefCurrency`. This would also save some gas as there is no more need for the if/else, and the `require` could be simplified.

☑ The `withdrawEther` function initializes a `balance` variable to the balance of the contract itself. Afterwards this variable is used once. Therefore, CHAINSECURITY sees no reason to define a variable in this case and recommends to directly use the value. This would also save some gas as there is no variable declared + initialized + loaded.

☑ Inside `ComplianceRegistry` the variable `MONTH` is initialized to `31 * 24 * 60 * 60`. MT PELERIN could make use of solidity time units[4]: `31 days`.

☑ Inside the rules contracts there is the following code: `require(true == false, "RU02");` For cases like these, there is `revert`. The same could be accomplished by doing `revert("RU02")`.

CHAINSECURITY recommends changing all such `require` statements with the above suggested `revert` statement.

☐ The `tokenizedSharePercentage` variable inside `ShareBridgeToken` is never used. Therefore, MT PELERIN could consider removing it.

☑ The `transfer` and `transferFrom` functions inside `BridgeERC20` return a boolean. However, the calls to the `Processor.transferFrom` in both of these functions will throw on error. It is therefore not possible to get back `false` for `success`. MT PELERIN could remove the if check and `return false`.

☑ In all of the Rule contracts the following constants are declared.

```
uint256 internal constant TRANSFER_INVALID = 0;
uint256 internal constant TRANSFER_VALID_WITH_NO_HOOK = 1;
uint256 internal constant TRANSFER_VALID_WITH_BEFORE_HOOK = 2;
uint256 internal constant TRANSFER_VALID_WITH_AFTER_HOOK = 3;
```

Most Rule contracts also define other constants which are not shared with any/some of the other Rule contracts.

MT PELERIN could maintain the common constants in a base Rule contract from which all Rule contracts inherit to avoid the duplication.

☑ In the `MaxTransferRule.isTransferValid` function the code comment is mentioned like below:

```
* @dev Validates a transfer if transfers are not globally frozen
```

CHAINSECURITY recommends correcting the code comment.

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

☑ Inside `VotingSession.sol` the modifier `whenVotingSessionOpen` is defined as:

```
require(votingEnd > 0 && votingStart <= now && votingEnd > now, "VO10");
```

If `votingEnd` is not above zero, the `votingEnd > now` check will fail. Therefore, the check `votingEnd > 0` is unnecessary and can be removed.

☐ There are multiple functions (`setPrices`, `registerVoters`, etc.) which take in one or multiple arrays. Although there are checks to make sure that all the arrays are of equal length, there is no check to make sure the length is above zero. Meaning calling these functions with an empty array will succeed, but no events will be emitted as nothing was added/updated.

MT PELERIN could consider adding such a check.

☑ In the `getOnHoldTransfers` function present in the `ComplianceRegistry` contract a variable is casted to an `address` like below:

```
token[length] = address(transfer.token);
```

Since `transfer.token` is already of type `address` the cast is unnecessary and can be removed.

☐ An Operator role is managed using an `Operator` contract. However, there are a few other roles:

- Administrator
- Supplier
- Seizer

The above roles are not defined in separate contracts. Instead, they are directly implemented in the contracts that need them.

MT PELERIN could consider also creating separate contracts for these roles and inherit from them when a contract needs any of these roles.

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 MT PELERIN'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 MT PELERIN and potential users.

Month number not accurate

The `_getMonth` function uses 31 days for 1 month. MT PELERIN has explained in a comment that they deliberately decided on this. The `_getMonth` function is will return the timestamp of the first day of the current 31-day month. This function is used to store monthly transfer statistics as used by the `SoftTransferLimitRule` and `HardTransferLimitRule` contracts.

```
function _getMonth(uint256 offset) internal view returns (uint256) {
  // solium-disable-next-line security/no-block-members
  uint256 _date = now - (offset * MONTH);
  return _date - (_date % MONTH);
}
```

Dependence on block time information

MT PELERIN uses `block.timestamp` / `now` inside the `ComplianceRegistry` 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].

Outdated compiler version

CHAINSECURITY could not find obvious issues with the compiler version MT PELERIN is using. MT PELERIN uses SOLC compiler, version 0.5.2. If MT PELERIN 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[6].

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.

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

Upgradable contracts

Upgradable contracts generally require additional trust from the user's perspective, because a central assumption about smart contracts (immutability) is lost. Furthermore, users need to trust MT PELERIN to correctly upgrade contracts, such that no funds get stuck or can be stolen. MT PELERIN uses the OpenZeppelin SDK for upgradability in all contracts. When using the upgradability pattern it is of utmost importance that the storage layout is only extended but not changed between upgrades. Although OpenZeppelin SDK contains a tool to automatically check this between upgrades, the tool's correctness is not complete and thus cannot be trusted 100%. Therefore, manually checking the storage layout between upgrades is highly recommended. What now follows is a non-extensive list of recommendations for writing upgradable contracts:

- do not change the type of variables
- do not change the order of variables
- do not change the internal order of structs
- do not delete any variables
- if possible, retain previous compiler settings
- place new variables after the last existing variable
- when adding a variable to a parent contract:
  1. decrement the OpenZeppelin SDK created `uint256[50]` array by 1
     - Note that the decrement amount depends on variable packing
  2. place the new variable after the last existing variable
- perform extensive testing on the upgrade

Furthermore, in order to retain compatibility with integrated applications and other smart contracts, the ABI should remain as stable as possible. Hence, `external` or `public` functions:

- should not be renamed
- parameter types should not be changed
- parameters should not be reordered
- return types and their size should not change (this doesn't influence the ABI, but can break compatibility due to `returndatasize` checks)

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.

Disclaimer

Contact:

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