CERTIK

Security Assessment

Agrifintech Finansal Teknolojiler A.Ş.

CertiK Verified on Mar 30th, 2023

CertiK Verified on Mar 30th, 2023

Agrifintech Finansal Teknolojiler A.Ş.

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	Ethereum (ETH)	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 03/30/2023	N/A

CODEBASE

https://github.com/incirLabs/agritoken-contracts

...View All

COMMITS

base: 8464d2d1adf5bc286f735c21d9c6371c4040ca3e

update: 6f8bd4e8dd47a730d3c7ff1d0eb5b582c548e44b

...View All

Vulnerability Summary

12	12	0	0	0	0	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined	Unresolved

2	Critical	2 Resolved	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
2	Major	2 Resolved	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
1	Medium	1 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
2	Minor	2 Resolved	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
5	Informational	5 Resolved	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS

AGRIFINTECH FINANSAL TEKNOLOJILER A.Ş.

Disclaimer

CODEBASE | AGRIFINTECH FINANSAL TEKNOLOJILER A.Ş.

Repository

https://github.com/incirLabs/agritoken-contracts

Commit

base: 8464d2d1adf5bc286f735c21d9c6371c4040ca3e
update: 6f8bd4e8dd47a730d3c7ff1d0eb5b582c548e44b

AUDIT SCOPE | AGRIFINTECH FINANSAL TEKNOLOJILER A.Ş.

4 files audited ● 3 files with Resolved findings ● 1 file without findings

ID	Repo	Commit	File	SHA256 Checksum
● MLB	incirLabs/agritoken-contracts	8464d2d	contracts/Manager.sol	3621f8d03c9b7089453e9c6fd062921580002 97eb968fe26e736ee7502871a8d
● MLU	incirLabs/agritoken-contracts	8464d2d	contracts/Mill.sol	3f70d428ef81af101bfbd2bccceba26e96667a aaa6553e10d850f9c8cb48d028
● TLB	incirLabs/agritoken-contracts	8464d2d	contracts/Trader.sol	aac45b2d826f05b7a04d95a6c90406379044f 606488cbfc0995e8ac92d2f9a2b
● WTL	incirLabs/agritoken-contracts	8464d2d	contracts/WheatToken.sol	e6a81d12b711ac0b712f08de44a7d6cf65d7cf a3d43bf047e3d26306df882273

APPROACH & METHODS

AGRIFINTECH FINANSAL TEKNOLOJILER A.Ş.

This report has been prepared for Agrifintech Finansal Teknolojiler A.Ş. to discover issues and vulnerabilities in the source code of the Agrifintech Finansal Teknolojiler A.Ş. project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

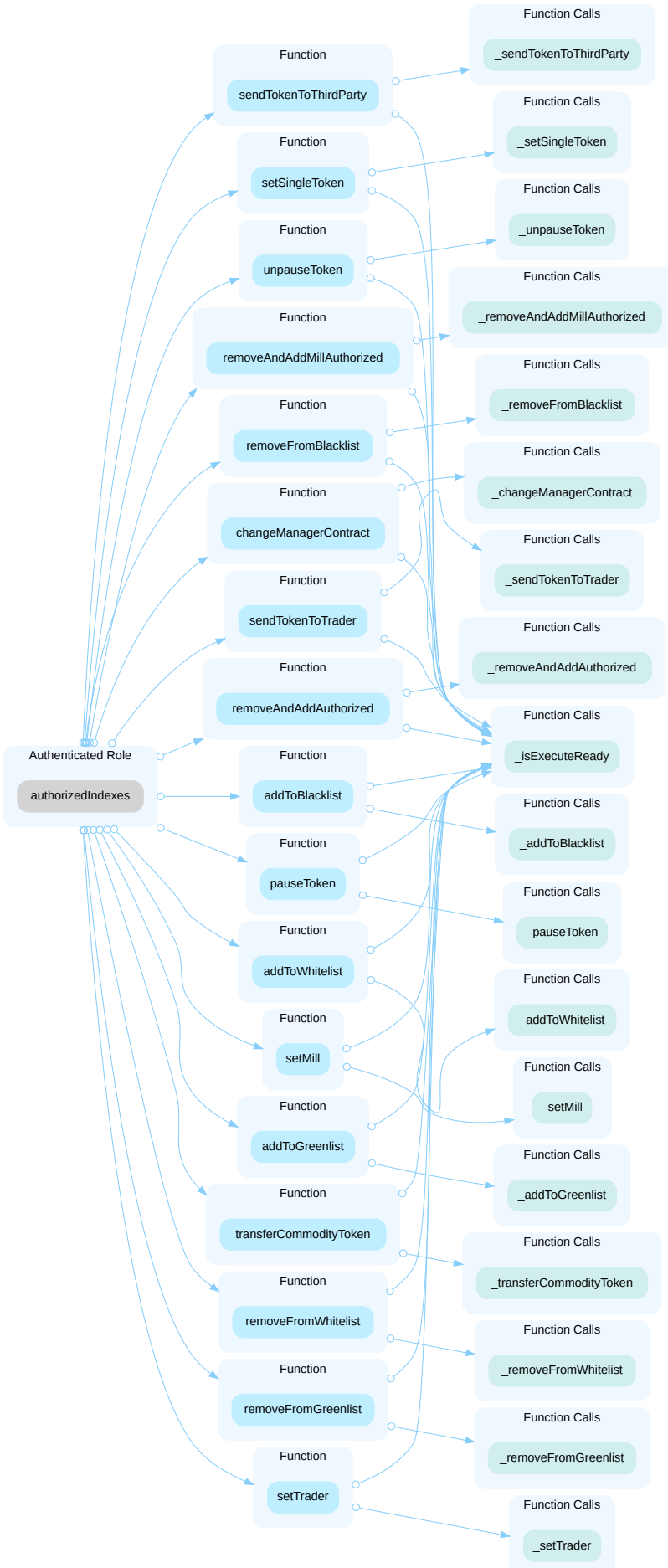
- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

DECENTRALIZATION EFFORTS

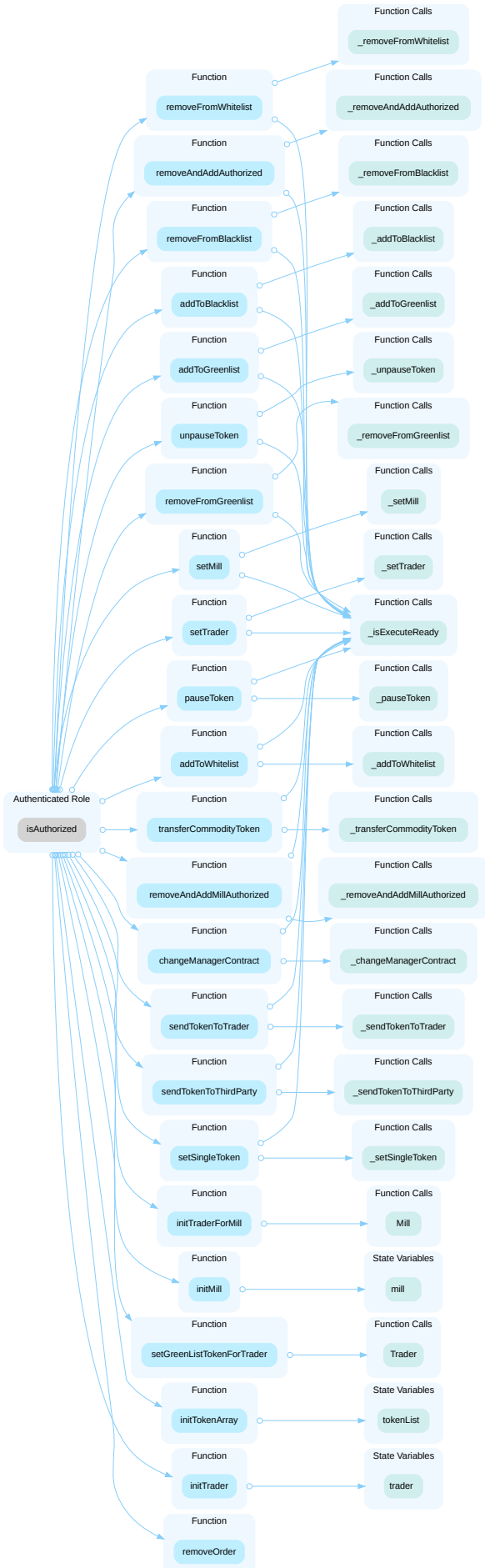
AGRIFINTECH FINANSAL TEKNOLOJILER A.Ş.

Description

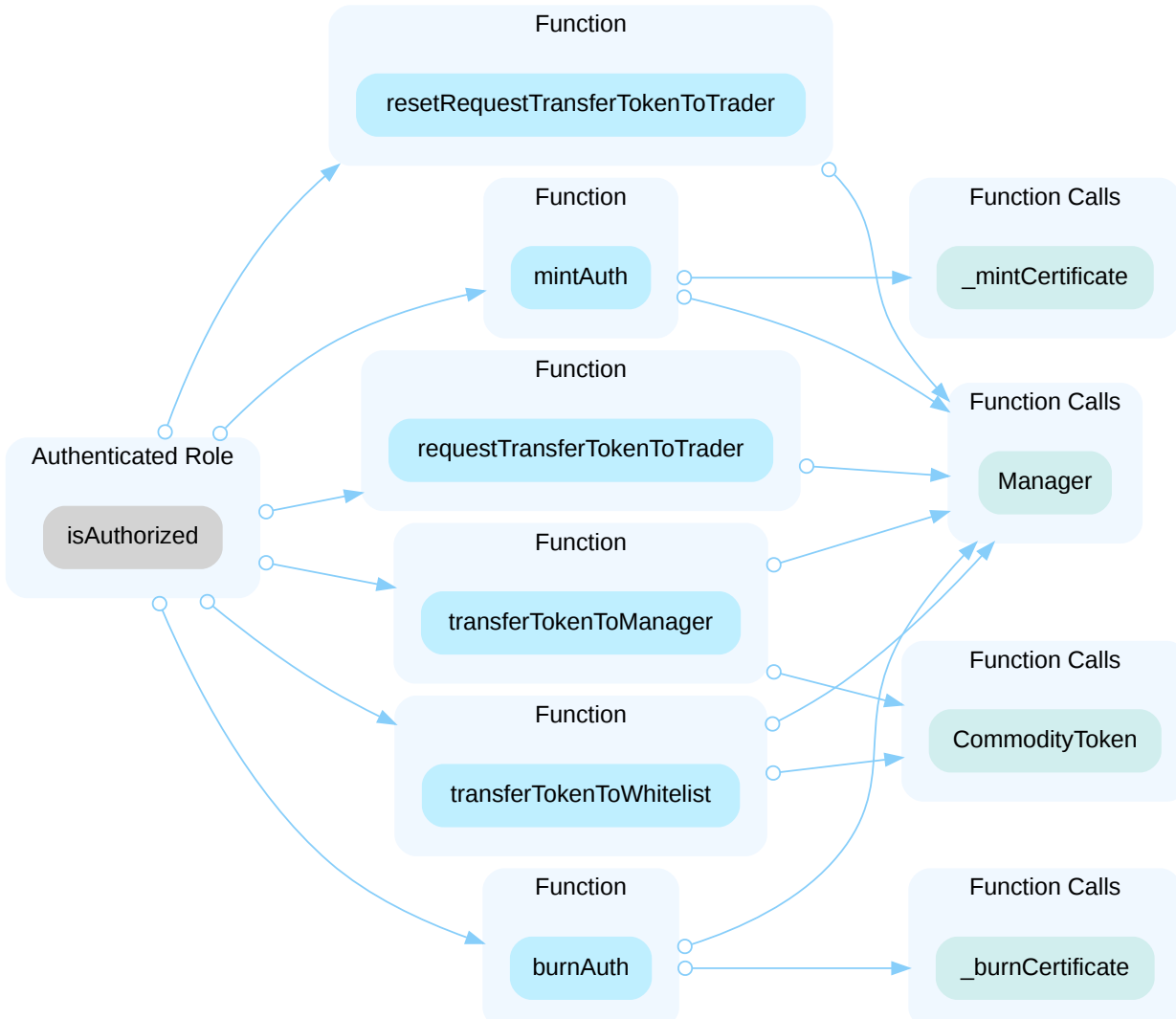
In the contract `Manager`, the role `authorizedIndexes` has authority over the functions shown in the diagram below. Any compromise to the `authorizedIndexes` account may allow the hacker to take advantage of this.


Authenticated Role
authorizedIndexes
Function
sendTokenToThirdParty
Function Calls
_sendTokenToThirdParty
Function
setSingleToken
Function Calls
_setSingleToken
Function
unpauseToken
Function Calls
_unpauseToken
Function
removeAndAddMillAuthorized
Function Calls
_removeAndAddMillAuthorized
Function
removeFromBlacklist
Function Calls
_removeFromBlacklist
Function
changeManagerContract
Function Calls
_changeManagerContract
Function
sendTokenToTrader
Function Calls
_sendTokenToTrader
Function
removeAndAddAuthorized
Function Calls
_removeAndAddAuthorized
Function Calls
_isExecuteReady
Function
addToBlacklist
Function Calls
_addToBlacklist
Function
pauseToken
Function Calls
_pauseToken
Function
addToWhitelist
Function Calls
_addToWhitelist
Function
setMill
Function Calls
_setMill
Function
addToGreenlist
Function Calls
_addToGreenlist
Function
transferCommodityToken
Function Calls
_transferCommodityToken
Function
removeFromWhitelist
Function Calls
_removeFromWhitelist
Function
removeFromGreenlist
Function Calls
_removeFromGreenlist
Function
setTrader
Function Calls
_setTrader

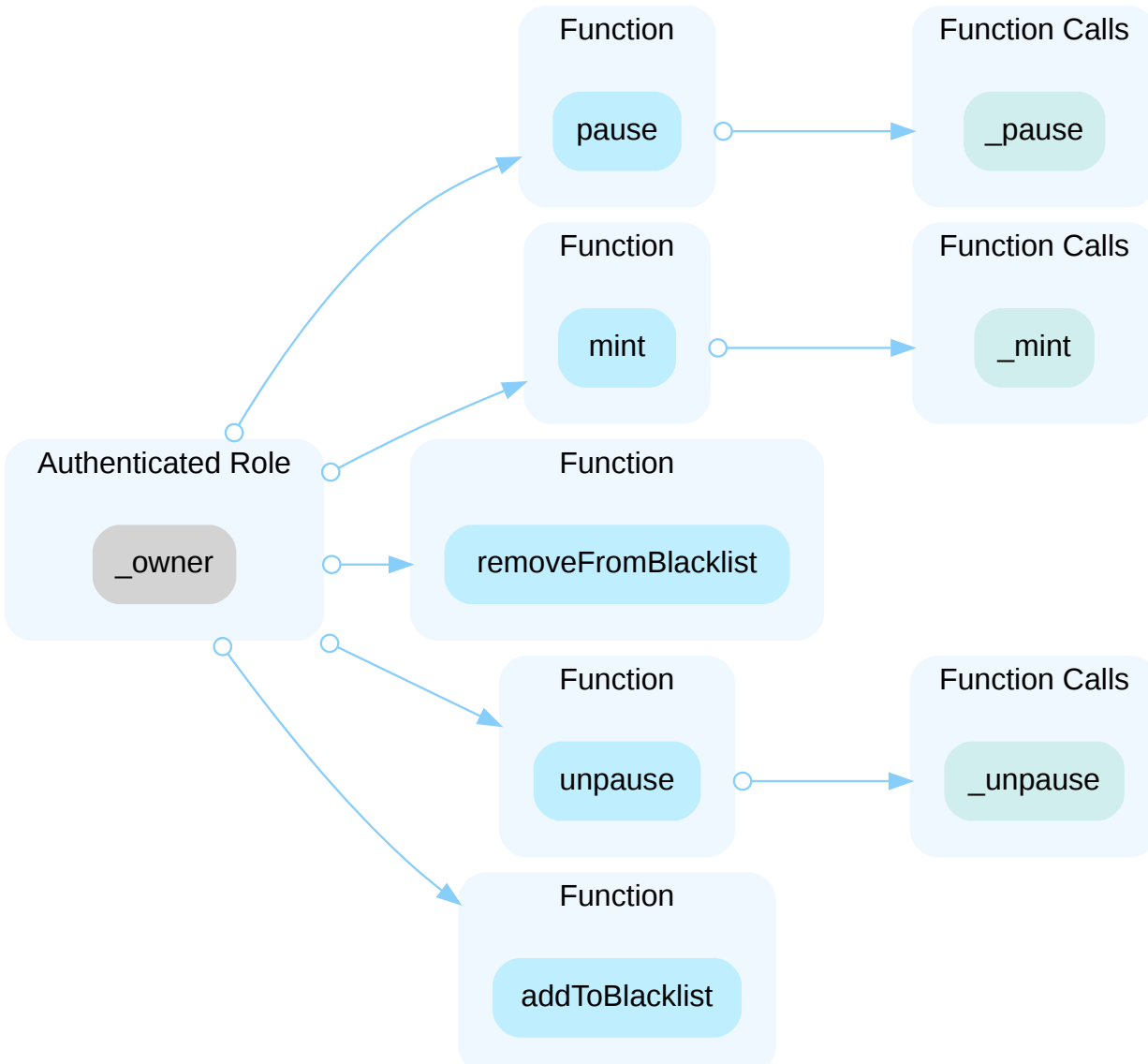
In the contract `Manager`, the role `isAuthorized` has authority over the functions shown in the diagram below. Any compromise to the `isAuthorized` account may allow the hacker to take advantage of this.


Function Calls
_removeFromWhitelist
Function
removeFromWhitelist
Function Calls
_removeAndAddAuthorized
Function
removeAndAddAuthorized
Function Calls
_removeFromBlacklist
Function
removeFromBlacklist
Function Calls
_addToBlacklist
Function
addToBlacklist
Function Calls
_addToGreenlist
Function
addToGreenlist
Function Calls
_unpauseToken
Function
unpauseToken
Function Calls
_removeFromGreenlist
Function
removeFromGreenlist
Function Calls
_setMill
Function
setMill
Function Calls
_setTrader
Function
setTrader
Function Calls
_isExecuteReady
Function
pauseToken
Function Calls
_pauseToken
Function
addToWhitelist
Function Calls
_addToWhitelist
Authenticated Role
isAuthorized
Function
transferCommodityToken
Function Calls
_transferCommodityToken
Function
removeAndAddMillAuthorized
Function Calls
_removeAndAddMillAuthorized
Function
changeManagerContract
Function Calls
_changeManagerContract
Function
sendTokenToTrader
Function Calls
_sendTokenToTrader
Function
sendTokenToThirdParty
Function Calls
_sendTokenToThirdParty
Function
setSingleToken
Function Calls
_setSingleToken
Function
initTraderForMill
Function Calls
Mill
Function
initMill
State Variables
mill
Function
setGreenListTokenForTrader
Function Calls
Trader
Function
initTokenArray
State Variables
tokenList
Function
initTrader
State Variables
trader
Function
removeOrder

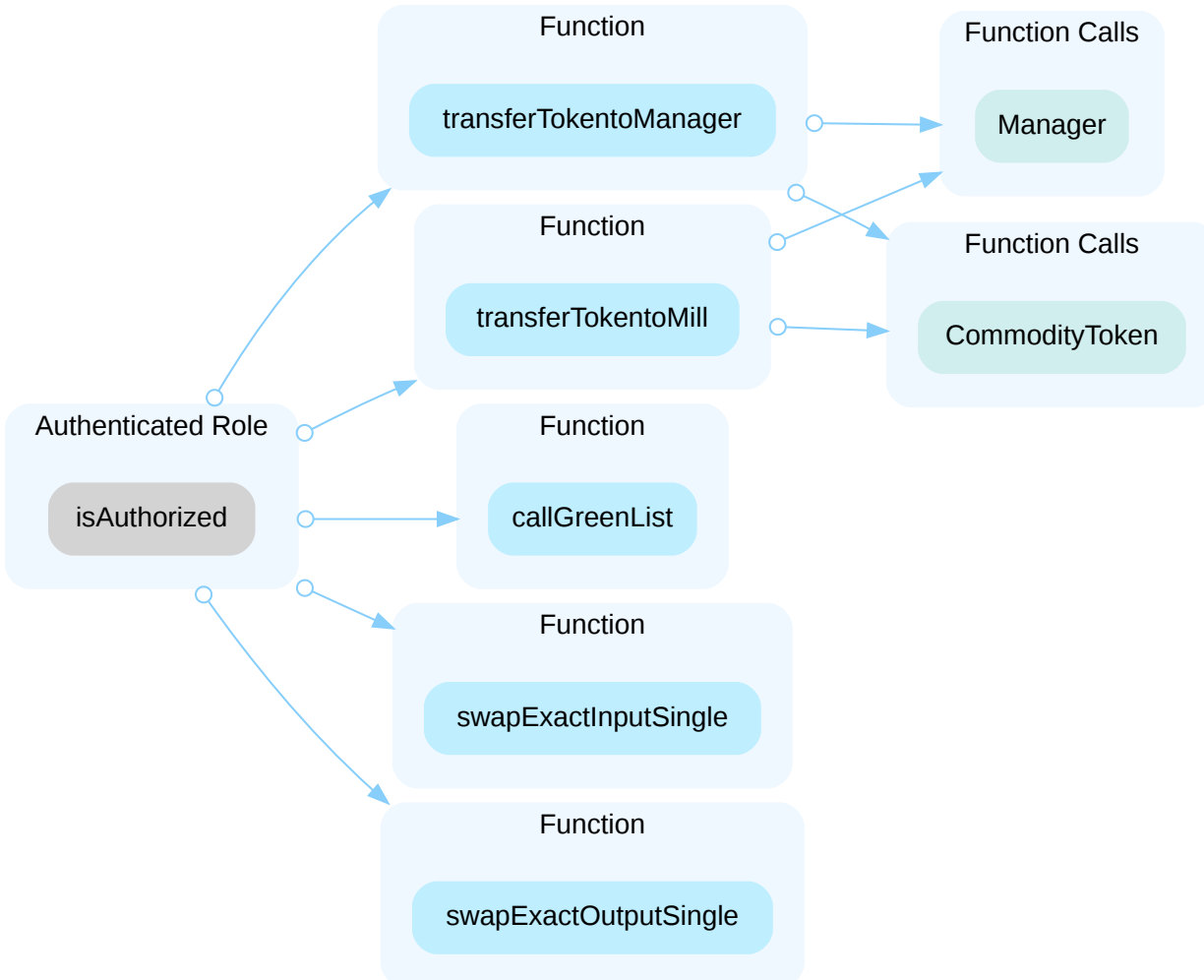
In the contract `Mill` , the role `isAuthorized` has authority over the functions shown in the diagram below. Any compromise to the `isAuthorized` account may allow the hacker to take advantage of this.



In the contract `WheatToken` , the role `_owner` has authority over the functions shown in the diagram below. Any compromise to the `_owner` account may allow the hacker to take advantage of this.



In the contract `Trader` the role `isAuthorized` has authority over the functions shown in the diagram below. Any compromise to the `isAuthorized` account may allow the hacker to take advantage of this.



Depending on which address got hacked, the malicious user could change the traders rendering other contracts unusable and create a large loss of funds. The malicious user could pause WheatToken or any ERC20 that is connected to the `Manager` contract. The malicious user could transfer tokens to an address of their choice, and they could blacklist users from interacting with the protocol. The manager contract has control over the `Trader`, `Mill`, and `WheatToken` contracts and can render them invalid.

If a malicious user gained access to any of the `Mill` contracts privileged roles, the user could create malicious certificates to create or destroy community funds. In addition, if a malicious user gained access to any of the trader contracts privileged roles they could swap for the maximum amount using `swapExactInputSingle()` or `swapExactOutputSingle()` as well as green-list contracts to then interact with using `callGreenList()`.

Recommendations

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets. Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Status/Alleviations

The client has mitigated this risk by requiring multiple approvals on almost all functions to ensure authorization levels are consistent. The Mill contract requires authorized accounts to mint or burn. Even if a hacker was able to get control over this contract, it can only send tokens to the trader account or to a manager account. All functions requiring authorization(s) are controlled by different manager addresses. Instead of using a single multi-sig, the client has opted for this approach which is similar by requiring multiple signatures for an action to be executed.

FINDINGS | AGRIFINTECH FINANSAL TEKNOLOJILER A.Ş.

12	2	2	1	2	5
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Agrifintech Finansal Teknolojiler A.Ş.. Through this audit, we have uncovered 12 issues ranging from different severity levels. Utilizing the techniques of Static Analysis & Manual Review to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
MLB-01	Incorrect Logic	Logical Issue	Critical	Resolved
TLB-01	Access Restriction On `swapExactOutputSingle()`	Logical Issue	Critical	Resolved
MLB-02	Wrong Logic In `removeAndAddAuthorized()`	Logical Issue	Major	Resolved
MLU-01	`burnAuth()` / `mintAuth()` Check Will Always Pass	Logical Issue	Major	Resolved
MLU-02	`burnAuth` Will Always Revert	Logical Issue	Medium	Resolved
COR-01	Check Always Passes	Logical Issue	Minor	Resolved
TLB-02	Third Party Dependencies	Volatile Code	Minor	Resolved
COR-02	Usage Of Hardhat's Console	Inconsistency, Coding Style	Informational	Resolved
COR-03	Typo	Inconsistency, Language Specific	Informational	Resolved
MLU-03	Missing Error Messages	Coding Style	Informational	Resolved
TLB-03	Unnecessary `abicoder v2` Pragma	Language Specific	Informational	Resolved

ID	Title	Category	Severity	Status
TLB-04	Function `removeFromGreenList()` Should Include `counter`	Logical Issue	Informational	Resolved

MLB-01 | INCORRECT LOGIC

Category	Severity	Location	Status
Logical Issue	Critical	contracts/Manager.sol (base): 100, 107~109	Resolved

Description

The order.length will always be six because of how `mapping` works. This means that only `isExcuteReady()` will only work if the `requiredConfirmationNumber` passed in is equal to 6. Rendering all the other functions inside this contract unusable if the `requiredConfirmationNumber` is less than 6.

The following for loop will always iterate to six.

```
109     for (uint8 i = 0; i < order[_orderhash].length; i++) {
110           confirmation++;
111     }
```

Recommendation

We recommend rethinking the logic of this function and determining if it is necessary to use an array. If the intent of this function is to verify that `msg.sender` has the correct authority to call the function, then we recommend checking them directly and reverting if they are not authorized. For example:

```
require(authorizedIndexes[msg.sender] == 6, "Do Not Have Correct Authorization");
            _changeManagerContract(_addr);
            emit ManagerContractChange(_addr);
```

Otherwise, we recommend changing the logic of the function so that it will work for any `requiredConfirmationNumber`.

Alleviation

`[CertiK]`: The client has resolved the issue in the following commit: a1aba658c58f7b99338d70f9c785b90f381919cb.

TLB-01 | ACCESS RESTRICTION ON `swapExactOutputSingle()`

Category	Severity	Location	Status
Logical Issue	Critical	contracts/Trader.sol (base): 156	Resolved

Description

The following function `swapExactOutputSingle()` lacks access restriction and any user can call it. Because the function calls the current addresses balance, a user could use this to get a favorable trade.

Recommendation

We recommend putting a modifier on this so only authorized users can call it.

Alleviation

`[CertiK]` : The client has resolved the issue in the following commit: bb34dcc9172e0fe028fc21c4a457d37a07acc8d0 .

MLB-02 | WRONG LOGIC IN `removeAndAddAuthorized()`

Category	Severity	Location	Status
Logical Issue	Major	contracts/Manager.sol (base): 141, 145	Resolved

Description

For and address to be authorized `authorizedIndexes[address]` must be nonzero. In `removedAndAddAuthorized()` it is checked that:

```
require(
            authorizedIndexes[_remove] == 0,
            "Address to get removed is not in the authorized list!"
        );
```

However, if `authorizedIndexes[_remove] == 0` that means that `_remove` was not authorized.

Similarly it is checked that:

```
require(
            authorizedIndexes[_add] != 0,
            "Address to get added is already authorized!"
        );
```

However, if `authorizedIndexes[_add] != 0` that means that `_add` is already authorized.

Recommendation

We recommend changing these checks to the following:

```
require(
            authorizedIndexes[_remove] != 0,
            "Address to get removed is not in the authorized list!"
        );
```

As `_remove` should be authorized.

```
require(
            authorizedIndexes[_add] == 0,
            "Address to get added is already authorized!"
        );
```

As `_add` should not be authorized.

Alleviation

`[CertiK]` : The client has resolved the issue in the following commit: 3b9217f3104b3818886d6a79584e21d522558df0.

MLU-01 | `burnAuth()` / `mintAuth()` CHECK WILL ALWAYS PASS

Category	Severity	Location	Status
Logical Issue	Major	contracts/Mill.sol (base): 101~104, 130~133	Resolved

Description

The following lines of code will always pass as `storeHash` is never used to update `certificate`. Only `certificateHash` is used to update `certificate`.

```
require(
            certificate[storeHash] == 0,
            "You have already approved this certificate!"
        );
```

```
require(
        certificate[storeHash] == 0,
        "This Certificate is already burnt or never existed!"
);
```

Recommendation

We recommend changing the code to use the `certificateHash` instead of the `storeHash` as this is what is used to update `certificate` in `_mintCertificate()` and `_burnCertificate()`. In addition, if this solution is used, then the check for `burnAuth` should be:

```
require(
        certificate[certificateHash] != 0,
        "This Certificate is already burnt or never existed!"
);
```

Alleviation

`[CertiK]`: The client has resolved the issue in the following commit: 7453346f220472367ca7af9859bb6fdd603a95f2.

MLU-02 | `burnAuth` WILL ALWAYS REVERT

Category	Severity	Location	Status
Logical Issue	Medium	contracts/Mill.sol (base): 101~104, 130~133	Resolved

Description

The following lines of code will always be true because `certificate[storeHash]` is never updated `certificate` is only updated for `certificateHash` :

```
require(
            certificate[storeHash] == 0,
            "You have already approved this certificate!"
        );
```

```
require(
            certificate[storeHash] == 0,
            "This Certificate is already burnt or never existed!"
        );
```

Recommendation

We recommend using two certificate hashes, for example `certificateMintHash` and `certificateMintHash` , and using these to check if a certificate for minting/burning has been requested and executed.

Alleviation

`[CertiK]` : The client has resolved the issue in the following commit: 7453346f220472367ca7af9859bb6fdd603a95f2.

COR-01 | CHECK ALWAYS PASSES

Category	Severity	Location	Status
Logical Issue	Minor	contracts/Manager.sol (base): 61~64; contracts/Mill.sol (base): 44~47	Resolved

Description

The check given in the constructor will always pass as `_authorized` is a fixed size array with 6 elements so its length will always be 6:

```
require(
            _authorized.length == 6,
            "There should be 6 adresses as authorized"
        );
```

Similarly for here `_authorized` is a fixed size array with 2 elements so its length will always be 2:

```
require(
            _authorized.length == 2,
            "There should be 2 adresses as authorized"
        );
```

Recommendation

We recommend instead checking that each of the addresses in the array is nonzero.

Alleviation

`[CertiK]: The client has resolved the issue in the following commit: 20036ae70c4fce060ff0e8f7fea168c6cee0e4f8.

TLB-02 | THIRD PARTY DEPENDENCIES

Category	Severity	Location	Status
Volatile Code	Minor	contracts/Trader.sol (base): 15~16, 21	Resolved

Description

The contract is serving as the underlying entity to interact with third party `AMM` or `greenlist` protocols. The scope of the audit treats 3rd party entities as black boxes and assume their functional correctness. However, in the real world, 3rd parties can be compromised and this may lead to lost or stolen assets. In addition, upgrades of 3rd parties can possibly create severe impacts, such as increasing fees of 3rd parties, migrating to new LP pools, etc.

Recommendation

We understand that the business logic of `AgriFintech` requires interaction with `UniswapV3` and `greenlist` protocols. We encourage the team to constantly monitor the statuses of 3rd parties to mitigate the side effects when unexpected activities are observed.

Alleviation

`[CertiK]`: The client acknowledged the issue and stated they will monitor these third parties heavily.

COR-02 | USAGE OF HARDHAT'S CONSOLE

Category	Severity	Location	Status
Inconsistency, Coding Style	Informational	contracts/Manager.sol (base): 8; contracts/Trader.sol (base): 11~12	Resolved

Description

The contract uses the console contract from Hardhat, which is meant to be used for testing purposes.

Recommendation

We recommend removing code that was used for testing purposes.

Alleviation

`[CertiK]` : The client has resolved the issue in the following commit: 732226ac96e9d04968c3eb534c75acbdbe77eef8.

COR-03 | TYPO

Category	Severity	Location	Status
Inconsistency, Language Specific	Informational	contracts/Manager.sol (base): 16, 57; contracts/Mill.sol (base): 11, 13, 79~80, 93~94, 174; contracts/Trader.sol (base): 129, 133	Resolved

Description

As all error messages are in english, we recommend that all comments be in english too to improve readability.

The following comment line says "pool fees transformation" instead of "pool transaction fees":

```
//pool fees transformation is 3000 = %0.3
```

The following says "In Token is not approved to this much of an amount!"", which can be clarified to "Input Token approval amount is too low!".

```
require(tokenGreenList[token1]>amountIn,"In Token is not approved to this much of an amount!");
```

The following has the typo `personal` instead of `personnel`.

```
mapping(bytes32 => bool[6]) public order; //every element of array indicates the authorized personal with same index
```

The following has the typo `authori` insead of `authorized`.

```
mapping(address => bool) public isAuthorized; //to Check if caller is one of the authori
```

Recommendation

We recommend rewriting all comments in english. In addition, we recommend fixing the typos mentioned above.

Alleviation

`[CertiK]`: The client has resolved the issue in the following commit: c39707b7943bd41defebc24cd0cef9ef9afd3b5b.

MLU-03 | MISSING ERROR MESSAGES

Category	Severity	Location	Status
Coding Style	Informational	contracts/Mill.sol (base): 196	Resolved

Description

require can be used to check for conditions and throw an exception if the condition is not met. It is better to provide a string message containing details about the error that will be passed back to the caller.

Recommendation

We recommend refactoring the linked code below:

```
require(toTraderRequest >= _amount, "Amount is too high");
```

Alleviation

`[CertiK]` : The client has resolved the issue in the following commit: b8995b7162017559c75eec06c8f9f836fe7e12e7.

TLB-03 | UNNECESSARY `abicoder v2` PRAGMA

Category	Severity	Location	Status
Language Specific	Informational	contracts/Trader.sol (base): 3	Resolved

Description

The linked statement explicitly declares the usage of `abicoder v2`. This feature is enabled by default in Solidity v0.8.0 and above. See Reference.

Recommendation

We recommend removing the linked statement.

Alleviation

`[CertiK]`: The client has resolved the issue in the following commit: d8ea7bcac7105762e8b1bb08a5754b46e746f042.

TLB-04 FUNCTION `removeFromGreenList()` SHOULD INCLUDE `counter`

Category	Severity	Location	Status
Logical Issue	Informational	contracts/Trader.sol (base): 66~72	Resolved

Description

In the current version, the counter will always go up when a address is added to the `greenlistIndex`. When an address is removed the counter is not lowered.

Recommendation

We recommend ensuring that this is intended and that the counter should not be lowered when an address is removed to balance it out.

Alleviation

`[CertiK]`: The client changed the finding in the following commit: 5d5457155134e7ae9d559d4df83f25782bcac735.

FORMAL VERIFICATION

AGRIFINTECH FINANSAL TEKNOLOJILER A.Ş.

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied automated formal verification (symbolic model checking) to prove that well-known functions in the smart contracts adhere to their expected behavior.

Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

Verification of ERC-20 Compliance

We verified properties of the public interface of those token contracts that implement the ERC-20 interface. This covers

- Functions `transfer` and `transferFrom` that are widely used for token transfers,
- functions `approve` and `allowance` that enable the owner of an account to delegate a certain subset of her tokens to another account (i.e. to grant an allowance), and
- the functions `balanceOf` and `totalSupply`, which are verified to correctly reflect the internal state of the contract.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
erc20-transfer-revert-zero	`transfer` Prevents Transfers to the Zero Address
erc20-transfer-correct-amount	`transfer` Transfers the Correct Amount in Non-self Transfers
erc20-transfer-correct-amount-self	`transfer` Transfers the Correct Amount in Self Transfers
erc20-transfer-change-state	`transfer` Has No Unexpected State Changes
erc20-transfer-exceed-balance	`transfer` Fails if Requested Amount Exceeds Available Balance
erc20-transfer-recipient-overflow	`transfer` Prevents Overflows in the Recipient's Balance
erc20-transfer-false	If `transfer` Returns `false`, the Contract State Is Not Changed
erc20-transferfrom-revert-from-zero	`transferFrom` Fails for Transfers From the Zero Address
erc20-transfer-never-return-false	`transfer` Never Returns `false`

Property Name	Title
erc20-transferfrom-revert-to-zero	`transferFrom` Fails for Transfers To the Zero Address
erc20-transferfrom-correct-amount	`transferFrom` Transfers the Correct Amount in Non-self Transfers
erc20-transferfrom-correct-amount-self	`transferFrom` Performs Self Transfers Correctly
erc20-transferfrom-correct-allowance	`transferFrom` Updated the Allowance Correctly
erc20-transferfrom-fail-exceed-balance	`transferFrom` Fails if the Requested Amount Exceeds the Available Balance
erc20-transferfrom-fail-exceed-allowance	`transferFrom` Fails if the Requested Amount Exceeds the Available Allowance
erc20-transferfrom-change-state	`transferFrom` Has No Unexpected State Changes
erc20-transferfrom-fail-recipient-overflow	`transferFrom` Prevents Overflows in the Recipient's Balance
erc20-transferfrom-false	If `transferFrom` Returns `false`, the Contract's State Is Unchanged
erc20-transferfrom-never-return-false	`transferFrom` Never Returns `false`
erc20-totalsupply-succeed-always	`totalSupply` Always Succeeds
erc20-totalsupply-change-state	`totalSupply` Does Not Change the Contract's State
erc20-balanceof-succeed-always	`balanceOf` Always Succeeds
erc20-totalsupply-correct-value	`totalSupply` Returns the Value of the Corresponding State Variable
erc20-balanceof-correct-value	`balanceOf` Returns the Correct Value
erc20-allowance-succeed-always	`allowance` Always Succeeds
erc20-balanceof-change-state	`balanceOf` Does Not Change the Contract's State
erc20-allowance-correct-value	`allowance` Returns Correct Value
erc20-allowance-change-state	`allowance` Does Not Change the Contract's State
erc20-approve-revert-zero	`approve` Prevents Approvals For the Zero Address
erc20-approve-succeed-normal	`approve` Succeeds for Admissible Inputs
erc20-approve-correct-amount	`approve` Updates the Approval Mapping Correctly

Property Name	Title
erc20-approve-change-state	`approve` Has No Unexpected State Changes
erc20-approve-false	If `approve` Returns `false`, the Contract's State Is Unchanged
erc20-approve-never-return-false	`approve` Never Returns `false`

Verification Results

For the following contracts, model checking established that each of the properties that were in scope of this audit (see scope) are valid:

Detailed Results For Contract WheatToken (contracts/WheatToken.sol) In Commit 8464d2d1adf5bc286f735c21d9c6371c4040ca3e

Verification of ERC-20 Compliance

Detailed results for function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-revert-zero	True	
erc20-transfer-correct-amount	True	
erc20-transfer-correct-amount-self	True	
erc20-transfer-change-state	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-recipient-overflow	True	
erc20-transfer-false	True	
erc20-transfer-never-return-false	True	

Detailed results for function `transferFrom`

Property Name	Final Result	Remarks
erc20-transferfrom-revert-from-zero	True	
erc20-transferfrom-revert-to-zero	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-amount-self	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-change-state	True	
erc20-transferfrom-fail-recipient-overflow	True	
erc20-transferfrom-false	True	
erc20-transferfrom-never-return-false	True	

Detailed results for function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-succeed-always	True	
erc20-totalsupply-change-state	True	
erc20-totalsupply-correct-value	True	

Detailed results for function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-succeed-always	True	
erc20-balanceof-correct-value	True	
erc20-balanceof-change-state	True	

Detailed results for function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-succeed-always	True	
erc20-allowance-correct-value	True	
erc20-allowance-change-state	True	

Detailed results for function `approve`

Property Name	Final Result	Remarks
erc20-approve-revert-zero	True	
erc20-approve-succeed-normal	True	
erc20-approve-correct-amount	True	
erc20-approve-change-state	True	
erc20-approve-false	True	
erc20-approve-never-return-false	True	

APPENDIX | AGRIFINTECH FINANSAL TEKNOLOJILER A.Ş.

Finding Categories

Categories	Description
Logical Issue	Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.
Language Specific	Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.
Coding Style	Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.
Inconsistency	Inconsistency findings refer to functions that should seemingly behave similarly yet contain different code, such as a constructor assignment imposing different require statements on the input variables than a setter function.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified using symbolic model checking. Each such contract was compiled into a mathematical model which reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

Technical Description

The model also formalizes a simplified execution environment of the Ethereum blockchain and a verification harness that performs the initialization of the contract and all possible interactions with the contract. Initially, the contract state is initialized non-deterministically (i.e. by arbitrary values) and over-approximates the reachable state space of the contract throughout any actual deployment on chain. All valid results thus carry over to the contract's behavior in arbitrary states after it has been deployed.

Assumptions and Simplifications

The following assumptions and simplifications apply to our model:

- Gas consumption is not taken into account, i.e. we assume that executions do not terminate prematurely because they run out of gas.
- The contract's state variables are non-deterministically initialized before invocation of any function. That ignores contract invariants and may lead to false positives. It is, however, a safe over-approximation.
- The verification engine reasons about unbounded integers. Machine arithmetic is modeled using modular arithmetic based on the bit-width of the underlying numeric Solidity type. This ensures that over- and underflow characteristics are faithfully represented.
- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for Property Specification

All properties are expressed in linear temporal logic (LTL). For that matter, we treat each invocation of and each return from a public or an external function as a discrete time step. Our analysis reasons about the contract's state upon entering and upon leaving public or external functions.

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates as atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `started(f, [cond])` Indicates an invocation of contract function `f` within a state satisfying formula `cond`.
- `willSucceed(f, [cond])` Indicates an invocation of contract function `f` within a state satisfying formula `cond` and considers only those executions that do not revert.
- `finished(f, [cond])` Indicates that execution returns from contract function `f` in a state satisfying formula `cond`. Here, formula `cond` may refer to the contract's state variables and to the value they had upon entering the function (using the `old` function).
- `reverted(f, [cond])` Indicates that execution of contract function `f` was interrupted by an exception in a contract state satisfying formula `cond`.

The verification performed in this audit operates on a harness that non-deterministically invokes a function of the contract's public or external interface. All formulas are analyzed w.r.t. the trace that corresponds to this function invocation.

Description of the Analyzed ERC-20 Properties

The specifications are designed such that they capture the desired and admissible behaviors of the ERC-20 functions `transfer`, `transferFrom`, `approve`, `allowance`, `balanceOf`, and `totalSupply`. In the following, we list those property specifications.

Properties related to function `transfer`

erc20-transfer-revert-zero

`transfer` Prevents Transfers to the Zero Address. Any call of the form `transfer(recipient, amount)` must fail if the recipient address is the zero address. Specification:

```
[](started(contract.transfer(to, value), to == address(0)) ==>
  <>(reverted(contract.transfer) || finished(contract.transfer(to, value), return
    == false)))
```

erc20-transfer-succeed-normal

`transfer` Succeeds on Admissible Non-self Transfers. All invocations of the form `transfer(recipient, amount)` must succeed and return `true` if

- the `recipient` address is not the zero address,
- `amount` does not exceed the balance of address `msg.sender`,
- transferring `amount` to the `recipient` address does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call. Specification:

```
[](started(contract.transfer(to, value), to != address(0) && to != msg.sender &&
    value >= 0 && value <= _balances[msg.sender] && _balances[to] + value <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[msg.sender] <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transfer(to, value), return == true)))
```

erc20-transfer-succeed-self

`transfer` Succeeds on Admissible Self Transfers. All self-transfers, i.e. invocations of the form `transfer(recipient, amount)` where the `recipient` address equals the address in `msg.sender` must succeed and return `true` if

- the value in `amount` does not exceed the balance of `msg.sender` and
- the supplied gas suffices to complete the call. Specification:

```
[](started(contract.transfer(to, value), to != address(0) && to == msg.sender &&
    value >= 0 && value <= _balances[msg.sender] && _balances[msg.sender] >= 0 &&
    _balances[msg.sender] <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transfer(to, value), return == true)))
```

erc20-transfer-correct-amount

`transfer` Transfers the Correct Amount in Non-self Transfers. All non-reverting invocations of `transfer(recipient, amount)` that return `true` must subtract the value in `amount` from the balance of `msg.sender` and add the same value to

the balance of the `recipient` address. Specification:

```
[](willSucceed(contract.transfer(to, value), to != msg.sender && _balances[to] >= 0
    && value >= 0 && _balances[to] + value <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[msg.sender] >= 0 && _balances[msg.sender] <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transfer(to, value), return == true ==>
      _balances[msg.sender] == old(_balances[msg.sender]) - value && _balances[to]
      == old(_balances[to]) + value)))
```

erc20-transfer-correct-amount-self

`transfer` Transfers the Correct Amount in Self Transfers. All non-reverting invocations of `transfer(recipient, amount)` that return `true` and where the `recipient` address equals `msg.sender` (i.e. self-transfers) must not change the balance of address `msg.sender`. Specification:

```
[](willSucceed(contract.transfer(to, value), to == msg.sender && _balances[to] >= 0
    && _balances[to] <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transfer(to, value), return == true ==> _balances[to] ==
      old(_balances[to]))))
```

erc20-transfer-change-state

`transfer` Has No Unexpected State Changes. All non-reverting invocations of `transfer(recipient, amount)` that return `true` must only modify the balance entries of the `msg.sender` and the `recipient` addresses. Specification:

```
[](willSucceed(contract.transfer(to, value), p1 != msg.sender && p1 != to) ==>
  <>(finished(contract.transfer(to, value), return == true ==> (_totalSupply ==
        old(_totalSupply) && _allowances == old(_allowances) && _balances[p1] ==
        old(_balances[p1]) && other_state_variables ==
        old(other_state_variables)))))
```

erc20-transfer-exceed-balance

`transfer` Fails if Requested Amount Exceeds Available Balance. Any transfer of an amount of tokens that exceeds the balance of `msg.sender` must fail. Specification:

```
[](started(contract.transfer(to, value), value > _balances[msg.sender] &&
    _balances[msg.sender] >= 0 && value <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(reverted(contract.transfer) || finished(contract.transfer(to, value), return
      == false)))
```

erc20-transfer-recipient-overflow

`transfer` Prevents Overflows in the Recipient's Balance. Any invocation of `transfer(recipient, amount)` must fail if it causes the balance of the `recipient` address to overflow. Specification:

```
[](started(contract.transfer(to, value), to != msg.sender && _balances[to] + value
    >= 0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[to] <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[msg.sender] <
    0x10000000000000000000000000000000000000000000000000000000000000000 && value >
    0 && value <= _balances[msg.sender]) ==> <>(reverted(contract.transfer) ||
    finished(contract.transfer(to, value), return == false) ||
    finished(contract.transfer(to, value), _balances[to] > old(_balances[to]) +
      value -
      0x10000000000000000000000000000000000000000000000000000000000000000)))
```

erc20-transfer-false

If `transfer` Returns `false`, the Contract State Is Not Changed. If the `transfer` function in contract `contract` fails by returning `false`, it must undo all state changes it incurred before returning to the caller. Specification:

```
[](willSucceed(contract.transfer(to, value)) ==> <>(finished(contract.transfer(to,
        value), return == false ==> (_balances == old(_balances) && _totalSupply ==
        old(_totalSupply) && _allowances == old(_allowances) &&
        other_state_variables == old(other_state_variables)))))
```

erc20-transfer-never-return-false

`transfer` Never Returns `false`. The transfer function must never return `false` to signal a failure. Specification:

```
[](!(finished(contract.transfer, return == false)))
```

Properties related to function `transferFrom`

erc20-transferfrom-revert-from-zero

`transferFrom` Fails for Transfers From the Zero Address. All calls of the form `transferFrom(from, dest, amount)` where the `from` address is zero, must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), from == address(0)) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom, return ==
      false)))
```

erc20-transferfrom-revert-to-zero

`transferFrom` Fails for Transfers To the Zero Address. All calls of the form `transferFrom(from, dest, amount)` where the `dest` address is zero, must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), to == address(0)) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom, return ==
      false)))
```

erc20-transferfrom-succeed-normal

`transferFrom` Succeeds on Admissible Non-self Transfers. All invocations of `transferFrom(from, dest, amount)` must succeed and return `true` if

- the value of `amount` does not exceed the balance of address `from`,
- the value of `amount` does not exceed the allowance of `msg.sender` for address `from`,
- transferring a value of `amount` to the address in `dest` does not lead to an overflow of the recipient's balance, and
- the supplied gas suffices to complete the call. Specification:

```
[](started(contract.transferFrom(from, to, value), from != address(0) && to !=
    address(0) && from != to && value <= _balances[from] && value <=
    _allowances[from][msg.sender] && _balances[to] + value <
    0x10000000000000000000000000000000000000000000000000000000000000000 && value >=
    0 && _balances[to] >= 0 && _balances[from] >= 0 && _balances[from] <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _allowances[from][msg.sender] >= 0 && _allowances[from][msg.sender] <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true)))
```

erc20-transferfrom-succeed-self

`transferFrom` Succeeds on Admissible Self Transfers. All invocations of `transferFrom(from, dest, amount)` where the `dest` address equals the `from` address (i.e. self-transfers) must succeed and return `true` if:

- The value of `amount` does not exceed the balance of address `from`,
- the value of `amount` does not exceed the allowance of `msg.sender` for address `from`, and
- the supplied gas suffices to complete the call. Specification:

```
[](started(contract.transferFrom(from, to, value), from != address(0) && from == to
    && value <= _balances[from] && value <= _allowances[from][msg.sender] && value
    >= 0 && _balances[from] <
    0x10000000000000000000000000000000000000000000000000000000000000000 &&
    _allowances[from][msg.sender] <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true)))
```

erc20-transferfrom-correct-amount

`transferFrom` Transfers the Correct Amount in Non-self Transfers. All invocations of `transferFrom(from, dest, amount)` that succeed and that return `true` subtract the value in `amount` from the balance of address `from` and add the same value to the balance of address `dest`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), from != to && value >= 0 &&
    _balances[from] >= 0 && _balances[from] <
    0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[to] + value <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true ==>
      _balances[from] == old(_balances[from]) - value && _balances[to] ==
      old(_balances[to] + value))))
```

erc20-transferfrom-correct-amount-self

`transferFrom` Performs Self Transfers Correctly. All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` and where the address in `from` equals the address in `dest` (i.e. self-transfers) do not change the balance entry of the `from` address (which equals `dest`). Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), from == to && value >= 0 &&
    value < 0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[from] >= 0 && _balances[from] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true ==>
      _balances[from] == old(_balances[from]))))
```

erc20-transferfrom-correct-allowance

`transferFrom` Updated the Allowance Correctly. All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` must decrease the allowance for address `msg.sender` over address `from` by the value in `amount`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, value), value >= 0 && value <
    0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[from] >= 0 && _balances[from] <
    0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[to] <
    0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _allowances[from][msg.sender] >= 0 && _allowances[from][msg.sender] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.transferFrom(from, to, value), return == true ==>
      ((_allowances[from][msg.sender] == old(_allowances[from][msg.sender]) -
      value) || (_allowances[from][msg.sender] ==
      old(_allowances[from][msg.sender]) && (from == msg.sender ||
        old(_allowances[from][msg.sender]) ==
        0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF))))))
```

erc20-transferfrom-change-state

`transferFrom` Has No Unexpected State Changes. All non-reverting invocations of `transferFrom(from, dest, amount)` that return `true` may only modify the following state variables:

- The balance entry for the address in `dest`,
- The balance entry for the address in `from`,
- The allowance for the address in `msg.sender` for the address in `from`. Specification:

```
[](willSucceed(contract.transferFrom(from, to, amount), p1 != from && p1 != to &&
    (p2 != from || p3 != msg.sender)) ==> <>(finished(contract.transferFrom(from,
      to, amount), return == true ==> (_totalSupply == old(_totalSupply) &&
      _balances[p1] == old(_balances[p1]) && _allowances[p2][p3] ==
      old(_allowances[p2][p3]) && other_state_variables ==
      old(other_state_variables)))))
```

erc20-transferfrom-fail-exceed-balance

`transferFrom` Fails if the Requested Amount Exceeds the Available Balance. Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the balance of address `from` must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), value > _balances[from] &&
    _balances[from] >= 0 && _balances[from] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom, return ==
      false)))
```

erc20-transferfrom-fail-exceed-allowance

`transferFrom` Fails if the Requested Amount Exceeds the Available Allowance. Any call of the form `transferFrom(from, dest, amount)` with a value for `amount` that exceeds the allowance of address `msg.sender` must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), msg.sender != from && value >
    _allowances[from][msg.sender] && _allowances[from][msg.sender] >= 0 && value <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom(from, to,
       value), return == false)))
```

erc20-transferfrom-fail-recipient-overflow

`transferFrom` Prevents Overflows in the Recipient's Balance. Any call of `transferFrom(from, dest, amount)` with a value in `amount` whose transfer would cause an overflow of the balance of address `dest` must fail. Specification:

```
[](started(contract.transferFrom(from, to, value), from != to && _balances[to] +
    value >= 0x1000000000000000000000000000000000000000000000000000000000000000 &&
    value < 0x1000000000000000000000000000000000000000000000000000000000000000 &&
    _balances[to] >= 0 && _balances[to] <
    0x1000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(reverted(contract.transferFrom) || finished(contract.transferFrom(from, to,
       value), return == false) || finished(contract.transferFrom(from, to,
       value), _balances[to] > old(_balances[to]) + value -
     0x1000000000000000000000000000000000000000000000000000000000000000)))
```

erc20-transferfrom-false

If `transferFrom` Returns `false`, the Contract's State Is Unchanged. If `transferFrom` returns `false` to signal a failure, it must undo all incurred state changes before returning to the caller. Specification:

```
[](willSucceed(contract.transferFrom(from, to, value)) ==>
  <>(finished(contract.transferFrom(from, to, value), return == false ==>
    (_balances == old(_balances) && _totalSupply == old(_totalSupply) &&
    _allowances == old(_allowances) && other_state_variables ==
    old(other_state_variables)))))
```

erc20-transferfrom-never-return-false

`transferFrom` Never Returns `false`. The `transferFrom` function must never return `false`. Specification:

```
[](!(finished(contract.transferFrom, return == false)))
```

Properties related to function `totalSupply`

erc20-totalsupply-succeed-always

`totalSupply` Always Succeeds. The function `totalSupply` must always succeeds, assuming that its execution does not run out of gas. Specification:

```
[](started(contract.totalSupply) ==> <>(finished(contract.totalSupply)))
```

erc20-totalsupply-correct-value

`totalSupply` Returns the Value of the Corresponding State Variable. The `totalSupply` function must return the value that is held in the corresponding state variable of contract contract. Specification:

```
[](willSucceed(contract.totalSupply) ==> <>(finished(contract.totalSupply, return
      == _totalSupply)))
```

erc20-totalsupply-change-state

`totalSupply` Does Not Change the Contract's State. The `totalSupply` function in contract contract must not change any state variables. Specification:

```
[](willSucceed(contract.totalSupply) ==> <>(finished(contract.totalSupply,
      _totalSupply == old(_totalSupply) && _balances == old(_balances) &&
      _allowances == old(_allowances) && other_state_variables ==
      old(other_state_variables))))
```

Properties related to function `balanceOf`

erc20-balanceof-succeed-always

`balanceOf` Always Succeeds. Function `balanceOf` must always succeed if it does not run out of gas. Specification:

```
[](started(contract.balanceOf) ==> <>(finished(contract.balanceOf)))
```

erc20-balanceof-correct-value

`balanceOf` Returns the Correct Value. Invocations of `balanceOf(owner)` must return the value that is held in the contract's balance mapping for address `owner`. Specification:

```
[](willSucceed(contract.balanceOf) ==> <>(finished(contract.balanceOf(owner),
      return == _balances[owner])))
```

erc20-balanceof-change-state

`balanceOf` Does Not Change the Contract's State. Function `balanceOf` must not change any of the contract's state variables. Specification:

```
[](willSucceed(contract.balanceOf) ==> <>(finished(contract.balanceOf(owner),
      _totalSupply == old(_totalSupply) && _balances == old(_balances) &&
      _allowances == old(_allowances) && other_state_variables ==
      old(other_state_variables))))
```

Properties related to function `allowance`

erc20-allowance-succeed-always

`allowance` Always Succeeds. Function `allowance` must always succeed, assuming that its execution does not run out of gas. Specification:

```
[](started(contract.allowance) ==> <>(finished(contract.allowance)))
```

erc20-allowance-correct-value

`allowance` Returns Correct Value. Invocations of `allowance(owner, spender)` must return the allowance that address `spender` has over tokens held by address `owner`. Specification:

```
[](willSucceed(contract.allowance(owner, spender)) ==>
  <>(finished(contract.allowance(owner, spender), return ==
    _allowances[owner][spender])))
```

erc20-allowance-change-state

`allowance` Does Not Change the Contract's State. Function `allowance` must not change any of the contract's state variables. Specification:

```
[](willSucceed(contract.allowance(owner, spender)) ==>
  <>(finished(contract.allowance(owner, spender), _totalSupply == old(_totalSupply)
    && _balances == old(_balances) && _allowances == old(_allowances) &&
    other_state_variables == old(other_state_variables))))
```

Properties related to function `approve`

erc20-approve-revert-zero

`approve` Prevents Approvals For the Zero Address. All calls of the form `approve(spender, amount)` must fail if the address in `spender` is the zero address. Specification:

```
[](started(contract.approve(spender, value), spender == address(0)) ==>
  <>(reverted(contract.approve) || finished(contract.approve(spender, value),
    return == false)))
```

erc20-approve-succeed-normal

`approve` Succeeds for Admissible Inputs. All calls of the form `approve(spender, amount)` must succeed, if

- the address in `spender` is not the zero address and
- the execution does not run out of gas. Specification:

```
[](started(contract.approve(spender, value), spender != address(0)) ==>
  <>(finished(contract.approve(spender, value), return == true)))
```

erc20-approve-correct-amount

`approve` Updates the Approval Mapping Correctly. All non-reverting calls of the form `approve(spender, amount)` that return `true` must correctly update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount`. Specification:

```
[](willSucceed(contract.approve(spender, value), spender != address(0) && value >=
    0 && value <
    0x10000000000000000000000000000000000000000000000000000000000000000) ==>
  <>(finished(contract.approve(spender, value), return == true ==>
      _allowances[msg.sender][spender] == value)))
```

erc20-approve-change-state

`approve` Has No Unexpected State Changes. All calls of the form `approve(spender, amount)` must only update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount` and incur no other state changes. Specification:

```
[](willSucceed(contract.approve(spender, value), spender != address(0) && (p1 !=
      msg.sender || p2 != spender)) ==> <>(finished(contract.approve(spender,
        value), return == true ==> _totalSupply == old(_totalSupply) && _balances
      == old(_balances) && _allowances[p1][p2] == old(_allowances[p1][p2]) &&
      other_state_variables == old(other_state_variables))))
```

erc20-approve-false

If `approve` Returns `false`, the Contract's State Is Unchanged. If function `approve` returns `false` to signal a failure, it must undo all state changes that it incurred before returning to the caller. Specification:

```
[](willSucceed(contract.approve(spender, value)) ==>
  <>(finished(contract.approve(spender, value), return == false ==> (_balances ==
      old(_balances) && _totalSupply == old(_totalSupply) && _allowances ==
      old(_allowances) && other_state_variables == old(other_state_variables)))))
```

erc20-approve-never-return-false

`approve` Never Returns `false`. The function `approve` must never returns `false`. Specification:

```
[](!(finished(contract.approve, return == false)))
```

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.