CERTIK

Security Assessment

Zeebu

CertiK Assessed on Jul 2nd, 2023

CertiK Assessed on Jul 2nd, 2023

Zeebu

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

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	Binance Smart Chain (BSC)	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Solidity	Delivered on 07/02/2023	N/A

CODEBASE

Deployed on BSC Testnet

View All in Codebase Page

COMMITS

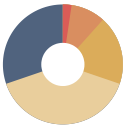
Invoice: 0x1de2057d81aa91e3c9e65f7127d202f65e8f767d

usdRate: 0x84ac006caf090ba005d9146e23fe3bb7d6d9edfd

MyWallet: 0xf157d36e184d01a96700cafd7043649d674817f9

View All in Codebase Page

Vulnerability Summary



43	34	0	2	7	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined

1	Critical	1 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.
4	Major	1 Resolved, 3 Acknowledged	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.
8	Medium	5 Resolved, 1 Partially Resolved, 2 Acknowledged	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
17	Minor	16 Resolved, 1 Partially 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.
13	Informational	11 Resolved, 2 Acknowledged	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 | ZEEBU

Disclaimer

CODEBASE | ZEEBU

Repository

Deployed on BSC Testnet

Commit

Invoice: 0x1de2057d81aa91e3c9e65f7127d202f65e8f767d
usdRate: 0x84ac006caf090ba005d9146e23fe3bb7d6d9edfd
MyWallet: 0xf157d36e184d01a96700cafd7043649d674817f9

updates: private codebase

AUDIT SCOPE | ZEEBU

4 files audited ● 4 files with Acknowledged findings

ID	File	SHA256 Checksum
● USD	usdRate.sol	60ed177a926d8e2adb79a63dc3e90f8199c39a713f10fda293306187fb68961f
● MWC	MyWallet.sol	7e083804f51e706b04d5c8a16a0b9b78b57649af158fac4f95b8aeedc0b747d2
● ERC	ERC20MinterPauser.sol	c252cafc168e321b5de89362a59598333c412cafd164da8f0993a45df9441221
● INV	Invoice.sol	c100eac5e4959578851df2eefd3ad48b6769b524940fd77a9f43499a82a66ecd

APPROACH & METHODS | ZEEBU

This report has been prepared for Zeebu to discover issues and vulnerabilities in the source code of the Zeebu 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.

DEPENDENCIES | ZEEBU

Third Party Dependencies

The protocol is serving as the underlying entity to interact with third party protocols. The third parties that the contracts interact with are:

- ERC20 tokens in `MyWallet.sol` ;
- In `Invoice.sol` :
    - merchantPaymentToken
    - merchantCreditToken
    - merchantRewardToken
    - customerRewardToken
    - stackRewardToken
    - earnToken
    - stackerPool
    - earningPool

The scope of the audit treats third party entities as black boxes and assumes their functional correctness. However, in the real world, third parties can be compromised and this may lead to lost or stolen assets. In addition, upgrades of third parties can possibly create severe impacts, such as increasing fees of third parties, migrating to new LP pools, etc.

Out Of Scope Dependencies

The protocol is serving as the underlying entity to interact with out-of-scope dependencies. The out-of-scope dependencies that the contracts interact with are:

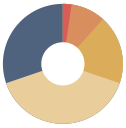
- there is an assumption that `walletAddress` refers to an in-scope `MyWallet` contract instance and that `usdRateContract` refers to in-scope `usdRate` contract. Use of other contracts for these respective variables is outside the scope of the audit.
- The dependency on signatures provided for valid messages in the `Invoice` and `MyWallet` contracts. It is expected that the signers check customer/merchant addresses are correct, verify the `amount` and `fee` makes sense, and that payload is not already used.
- The method for constructing the `payload` to identify a message uniquely in `Invoice`

The scope of the audit treats out-of-scope dependencies as black boxes and assumes their functional correctness.

Recommendations

We recommend constantly monitoring the third parties involved to mitigate any side effects that may occur when unexpected changes are introduced. Additionally, we recommend all out-of-scope dependencies are carefully vetted to ensure they function as intended.

FINDINGS | ZEEBU



43	1	4	8	17	13
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Zeebu. Through this audit, we have uncovered 43 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
TES-01	Incompatibility With `onERC20Receive()` Token Implementation	Logical Issue	Critical	Resolved
0X1-01	Potential Reentrancy Attack (Sending Ether Or Token)	Volatile Code	Major	Resolved
INV-06	Function `pay()` Does Not Check `tokenValue` Sent By `msg.sender`	Logical Issue	Major	Acknowledged
MWC-22	**Centralization Risks In MyWallet.Sol**	**Centralization**	**Major**	**Acknowledged**
TES-11	**Centralization Related Risks**	**Centralization**	**Major**	**Acknowledged**
INV-07	No Upper Limit For Fees And Commission	Logical Issue	Medium	Resolved
INV-08	Some Valid Signatures May Be Used Without A Corresponding Transfer Of Tokens	Logical Issue	Medium	Resolved
INV-09	No Check That `signer2` And `signer3` Are `withdrawSigner` S	Logical Issue	Medium	Resolved
INV-13	Use Of `getZeebuToZusdRate()`	Incorrect Calculation, Inconsistency	Medium	Acknowledged
INV-19	Potential Replay Attack	Logical Issue	Medium	Partially Resolved

ID	Title	Category	Severity	Status
MWC-21	Lack Of Access Control	Logical Issue	Medium	Resolved
USD-02	Potentially Dangerous Fixed Conversion Rates	Logical Issue	Medium	Acknowledged
ZCK-01	Malleability Of `abi.encodePacked`	Volatile Code	Medium	Resolved
ERC-02	Purpose Of Holding Native Tokens In `ERC20MinterPauser`	Inconsistency	Minor	Resolved
FFZ-01	Potential Reentrancy Attack (Incrementing State)	Concurrency	Minor	Resolved
INV-03	`walletAddress` Is Not Initialized On Deployment Of The Contract	Logical Issue	Minor	Resolved
INV-10	Unused Value	Inconsistency	Minor	Resolved
INV-11	Potential For Unintended Sent Ether	Logical Issue	Minor	Resolved
MWC-04	Conflict In Use Of `onlySigner()` Modifier	Logical Issue	Minor	Resolved
MWC-05	SafeMath Not Used	Incorrect Calculation	Minor	Resolved
MWC-06	Insufficient Checks Before Attempting Transfer	Logical Issue	Minor	Resolved
MWC-07	Purpose Of Included `msg.data` In `ForwarderDeposited` Event	Logical Issue, Access Control	Minor	Resolved
MWC-08	Purpose Of Flushing Functions	Coding Style	Minor	Resolved
MWC-09	Purpose Of `payable` Casting On `transferMultiSigEther()`	Logical Issue	Minor	Resolved
MWC-11	Outdated Solidity Version	Coding Issue	Minor	Resolved

ID	Title	Category	Severity	Status
TES-02	Unprotected `onERC20Receive()` Function	Logical Issue	Minor	Resolved
TES-06	Lack Of Input Validation	Volatile Code	Minor	Resolved
TES-15	Potential Reentrancy Attack (Out-Of-Order Events)	Volatile Code	Minor	Resolved
USD-01	Variables Not Initialized On Deploy	Logical Issue, Volatile Code	Minor	Resolved
ZCP-01	Potential Reentrancy Attack (Out-Of-Order Events)	Concurrency	Minor	Partially Resolved
ERC-01	Use Case Of `onERC20Receive()`	Logical Issue	Informational	Resolved
ERC-03	Shadowing Local Variable	Coding Style	Informational	Resolved
INV-01	Lack Of Documentation On Context Of Contract	Coding Style	Informational	Resolved
INV-12	Purpose Of `merchant` Vs. `customer` Addresses	Logical Issue	Informational	Acknowledged
INV-14	Calculations In `getOriginalAmount()`	Incorrect Calculation, Logical Issue	Informational	Acknowledged
INV-15	`validateAddress()` Does Not Guarantee `customer` And `merchant` Are Forwarder Contracts	Logical Issue	Informational	Resolved
INV-16	Hidden Assumption On Decimals Of Tokens Used	Incorrect Calculation, Logical Issue	Informational	Resolved
MWC-13	Consider Use Of SafeERC20 For Transfers	Volatile Code	Informational	Resolved
TES-07	Missing Error Messages	Coding Style	Informational	Resolved

ID	Title	Category	Severity	Status
TES-08	Use Of `abi.encodePacked()`	Volatile Code	Informational	Resolved
TES-09	Missing Check For `v` And `s`	Logical Issue	Informational	Resolved
TES-12	Missing Emit Events	Coding Style	Informational	Resolved
TES-13	Typos	Coding Style	Informational	Resolved

TES-01 INCOMPATIBILITY WITH `onERC20Receive()` TOKEN IMPLEMENTATION

Category	Severity	Location	Status
Logical Issue	Critical	ERC20MinterPauser.sol (Invoice_Base): 2310~2322; MyWallet.sol (My Wallet_Base): 72~73, 82~83	Resolved

Description

`ERC20MinterPauser` tokens can get stuck in the `ForwarderContract`.

Scenario

1. Contract `Invoice` transfers `ERC20MinterPauser` tokens to a `ForwarderContract` instance which has a `parentAddress` equivalent to the set `walletAddress` in the `Invoice` contract. Since the `ForwarderContract` has the `onERC20Receive()` function implemented, this transfer will successfully execute.
2. The `parentAddress` of the `ForwarderContract` then interacts with function `flushDeposit()` to transfer the tokens from the `ForwarderContract` to the `parentAddress`. The `parentAddress` is assumed to be an instance of the `MyWallet` contract.
3. Since `to.code.length` is positive for an instance of `MyWallet`, the `ERC20MinterPauser` contract will `try` to call function `onERC20Receive()` in `MyWallet`, but this function is not implemented.
4. As a result, none of the tokens that are meant to be sent to `parentAddress` can be transferred out of the `ForwarderContract`.

Recommendation

We recommend implementing `onERC20Receive()` in the `MyWallet` contract since it is expected that the contract should be transferred `ERC20MinterPauser` tokens. Otherwise, we recommend considering the removal of the `onERC20Receive()` function from the `ERC20MinterPauser` contract.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

0X1-01 POTENTIAL REENTRANCY ATTACK (SENDING ETHER OR TOKEN)

Category	Severity	Location	Status
Volatile Code	Major	ERC20MinterPauser.sol (Invoice_Base): 136, 339; Invoice.sol (Invoice_Base): 336, 338, 362, 404, 526	Resolved

Description

A reentrancy attack can occur when the contract creates a function that makes an external call to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

External call(s)

```
336         processTCommision(msg.sender,customer,merchant,tokenValue,amount,fee);
```

- This function call executes the following external call(s).
- In `SafeERC20._callOptionalReturn`,
  - `returndata = address(token).functionCall(data,"SafeERC20: low-level call failed")`
- In `Address.functionCallWithValue`,
  - `(success,returndata) = target.call{value: value}(data)`
- In `Invoice.transferToken`,
  - `token.safeTransfer(user,amount)`
- In `Invoice.processTCommision`,
  - `token.safeTransferFrom(msgSender,address(this),tAmount)`
- In `Invoice.processTCommision`,
  - `erc20.burn(burnAmount)`

State variables written after the call(s)

```
338         isPaid[payload] = true;
```

Since `ERC20MinterPauser` transfers include a hook that makes an external call to the recipient, if any of the recipient addresses transferred these tokens are malicious (such as the signed `customer` and `merchant` addresses), then the set up allows for reentering before `isPaid` is updated, allowing replay of the same signature until all corresponding tokens are drained from the contract.

Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts or applying OpenZeppelin ReentrancyGuard library - `nonReentrant` modifier for the aforementioned functions to prevent reentrancy attack.

Alleviation

`[CertiK]` : The team made changes resolving the finding.

INV-06 FUNCTION `pay()` DOES NOT CHECK `tokenValue` SENT BY `msg.sender`

Category	Severity	Location	Status
Logical Issue	Major	Invoice.sol (Invoice_Base): 322~323, 323~324, 328~329, 335~336, 362~363, 373~374, 381~382, 389~390, 397~398, 404~405	Acknowledged

Description

Function `pay()` includes both an `amount` and a `tokenValue` as input. This function is also `payable`. The `amount` is used in the signature verification step, while `tokenValue` can be any value specified by the `msg.sender`.

In the internal call to `processTCommision()`, the `tokenValue` is transferred from the `msg.sender` into the contract, while the `amount` is used to calculate an amount of tokens to transfer from the contract to the `merchant`, `customer`, `stackerPool`, and `earningPool`. Moreover, a portion of the `merchantPaymentToken` is burned if `burnAmount` is positive.

There is no check on the `tokenValue` specified nor is there a check that the `msg.sender` included a `msg.value`.

If the `msg.sender` is meant to transfer an amount of ERC20 tokens or native tokens that corresponds to transferring out a proportional amount of the other ERC20 tokens, this is not enforced by the function logic. Whatever amount the contract currently holds of `merchantPaymentToken` may be used in the burn, rather than coming from the `msg.sender`.

Recommendation

We recommend correcting the function logic of `pay()` and `processTCommision()` to require the `msg.sender` to transfer an adequate amount of tokens to compensate the subsequent transfers to `merchant`, `customer`, `stackerPool`, and `earningPool` respectively.

If this is not the intended implementation of the contract, we recommend providing documentation on what is intended so that the implementation can be confirmed.

Alleviation

`[CertiK]`: The team updated function `pay()` to include a check to `validateTokenAmount()` which is a new function that compares calculations with `amount` to `tokenValue`. The `payable` attribute has also been removed from the function.

However, the current implementation of `validateTokenAmount()` seems to include errors in its conversion calculations, meaning that the validation is likely to fail even for legitimate values.

As an example, say that `tokenValue` is `10 * (10**18)`, `amount` is `10 * (10**18)` and `rate` is 1 (for simplicity, we assume the exchange rate is 1 to 1).

Then `amountInUsdFormated` would be calculated to be `(10 * (10 ** 18)) / (10 ** 18) / (10 ** 16)` which is 0, while `amount` would be `10 * (10 ** 18) / 10 ** 16` which is `10 * (10 ** 2)`. The values are not equivalent, and

therefore the function would return false. It appears the calculation is made with the expectation of a specific value for `rate`, while the `rate` can be updated, and may be different dependent upon the type of payment being processed. We recommend updating the logic fix the issue outlined above.

If the example provided above does not take into account information about the protocol, please provide further information to clarify.

In response to the above, team states the formula is working as intended citing the following calculation:

```
amountInUsdFormated = ((10**18 * (10 * (10**18))) / 10**18)/10**16 = 10 * (10**2)
```

```
amount = (10 * (10**18))/10**16 = 10 * (10**2)
```

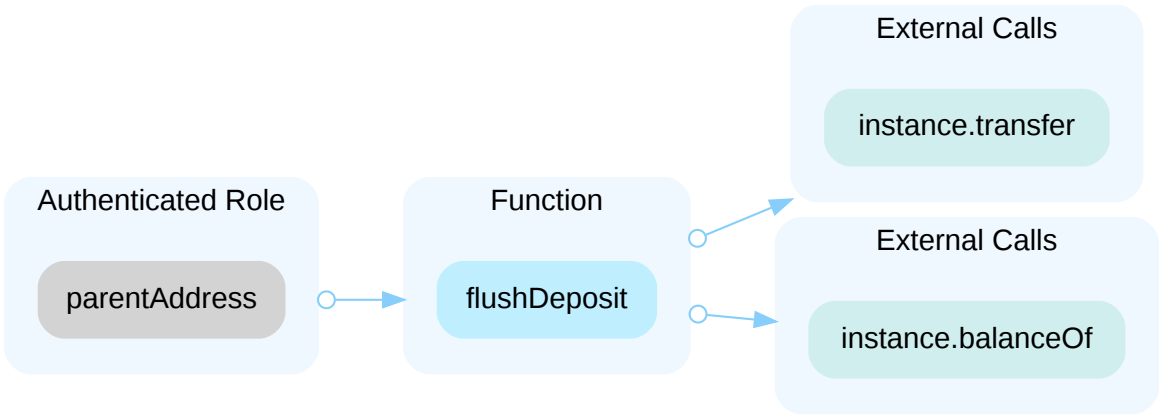
The order of operations in the calculation provided do not appear to match the implementation in the code. The implementation cannot be confirmed without proof that the formula will validate the correct token amount has been included in every case.

MWC-22 | CENTRALIZATION RISKS IN MYWALLET.SOL

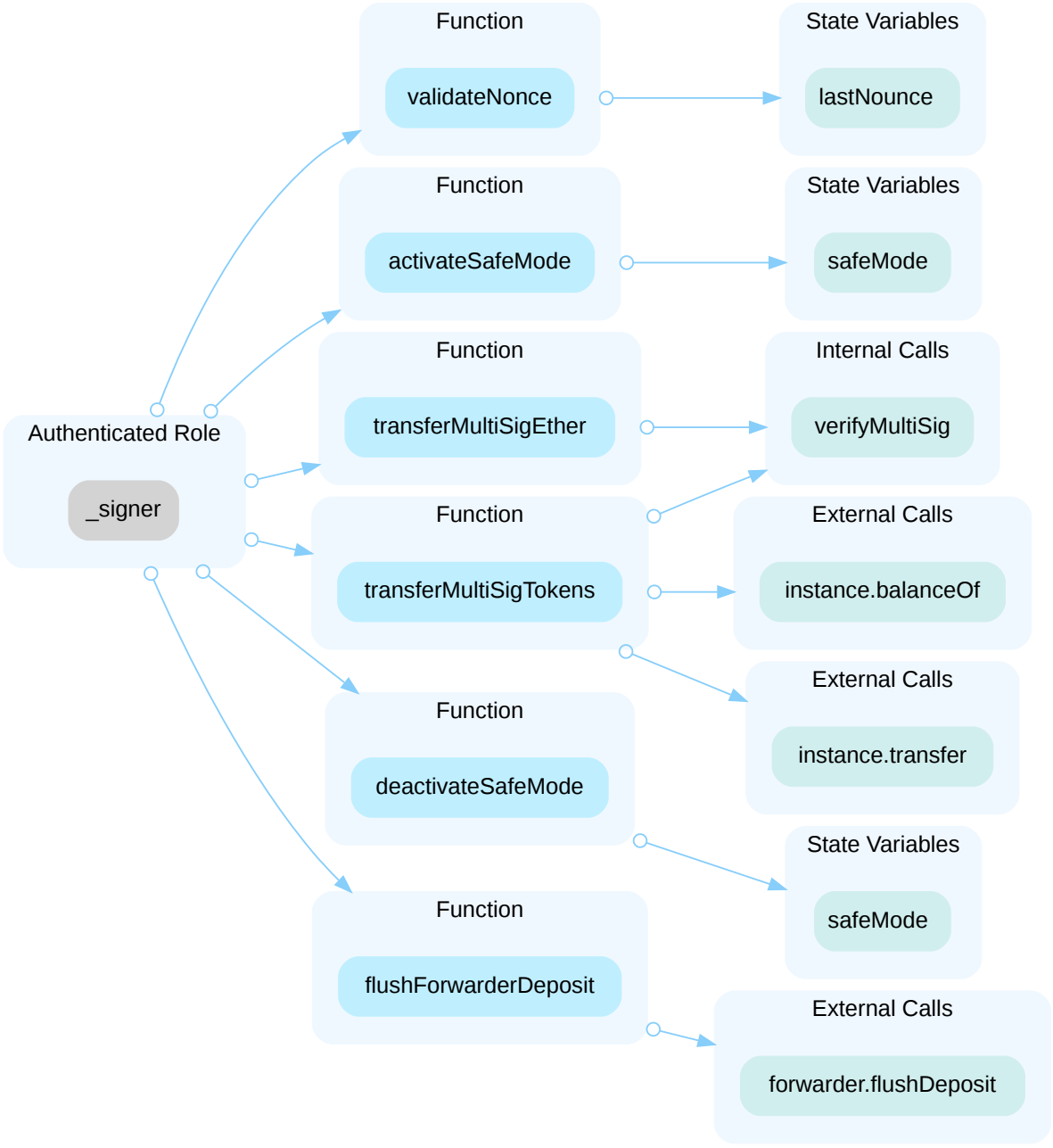
Category	Severity	Location	Status
Centralization	Major	MyWallet.sol (MyWallet_Base): 68, 160, 169, 199, 233, 262, 297	Acknowledged

Description

In the contract `ForwarderContract` the role `parentAddress` has authority over the functions shown in the diagram below. Any compromise to the `parentAddress` account may allow the hacker to take advantage of this authority and interact with any input address via external call to a `transfer()` function. This can allow for transferring the entire balance of the contract to the `parentAddress`, or, depending on the implementation of the contract address used as input, may interact with the contract in unexpected ways.



In the contract `MyWallet` the role `_signer` has authority over the functions shown in the diagram below. Any compromise to the `_signer` account may allow the hacker to take advantage of this authority and call `deactivateSafeMode()`, allowing native tokens or ERC20 tokens to be sent to any address. Additionally, a hacker may take advantage of the current signature implementation and use the same signature to call functions `activateSafeMode()`, `deactivateSafeMode()`, `flushForwarderDeposit()` and `flushForwarderFund()` more than once, in unintended scenarios.



Recommendation

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.

Alleviation

`[CertiK]` : The team acknowledges the finding. We note that the team has made efforts to include signature verification that requires the approval of at least two signers for each privileged function.

To mitigate the finding, all issues with signature replay must be resolved, and a timelock with latency of at least 24 hours should be implemented for interaction with each privileged function.

`[CertiK]` : The team states that they recommend their clients store keys securely and cite that two signers are required for the centralized functions. They opt not to use a time lock because they feel it will impact end user transaction processing.

TES-11 | CENTRALIZATION RELATED RISKS

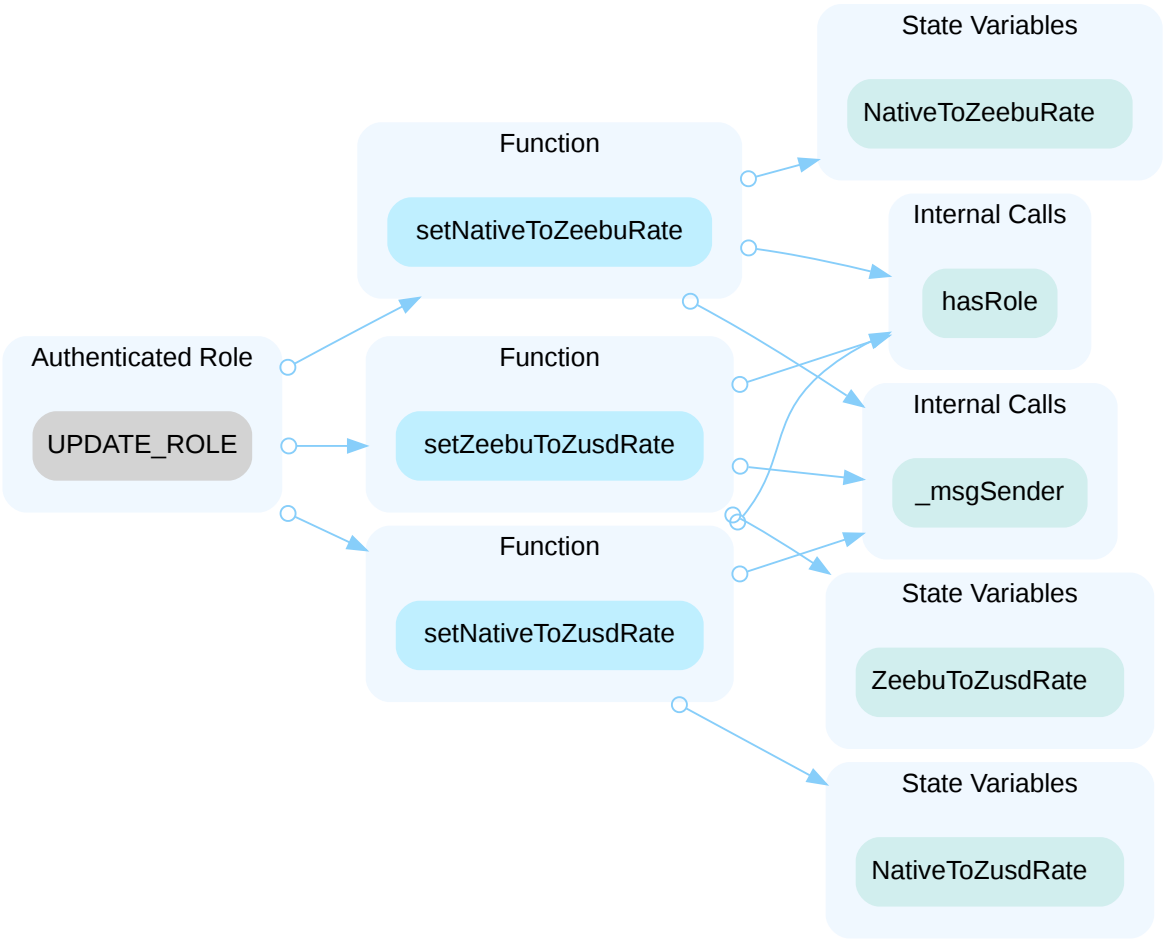
Category	Severity	Location	Status
Centralization	Major	ERC20MinterPauser.sol (Invoice_Base): 1494, 1509, 1529, 2334, 2350, 2364, 2378; Invoice.sol (Invoice_Base): 408, 432, 436, 441, 447, 453, 459, 466, 473, 480, 489, 498, 507; usdRate.sol (usdRate_Base): 688, 750, 762, 774	Acknowledged

Description

In the contract `usdRate` the role `DEFAULT_ADMIN_ROLE` has authority over the functions

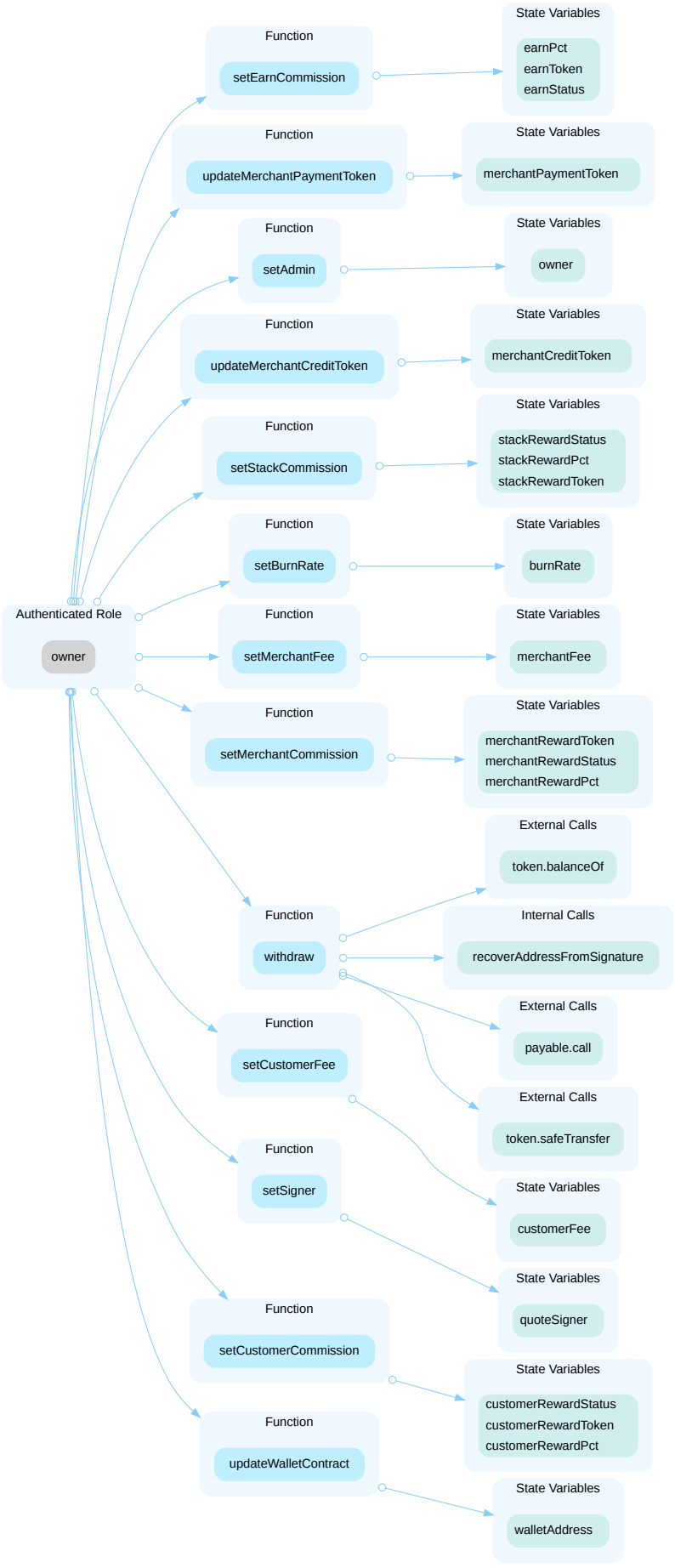
- `grantRole()`;
- `revokeRole()`; Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow the hacker to take advantage of this authority and grant `UPDATE_ROLE` to addresses they control.

In the contract `usdRate` the role `UPDATE_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `UPDATE_ROLE` account may allow the hacker to take advantage of this authority and update the corresponding conversion rates, potentially allowing for exploitative arbitrage in any protocols relying on these values.



In the contract `Invoice` 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 authority and

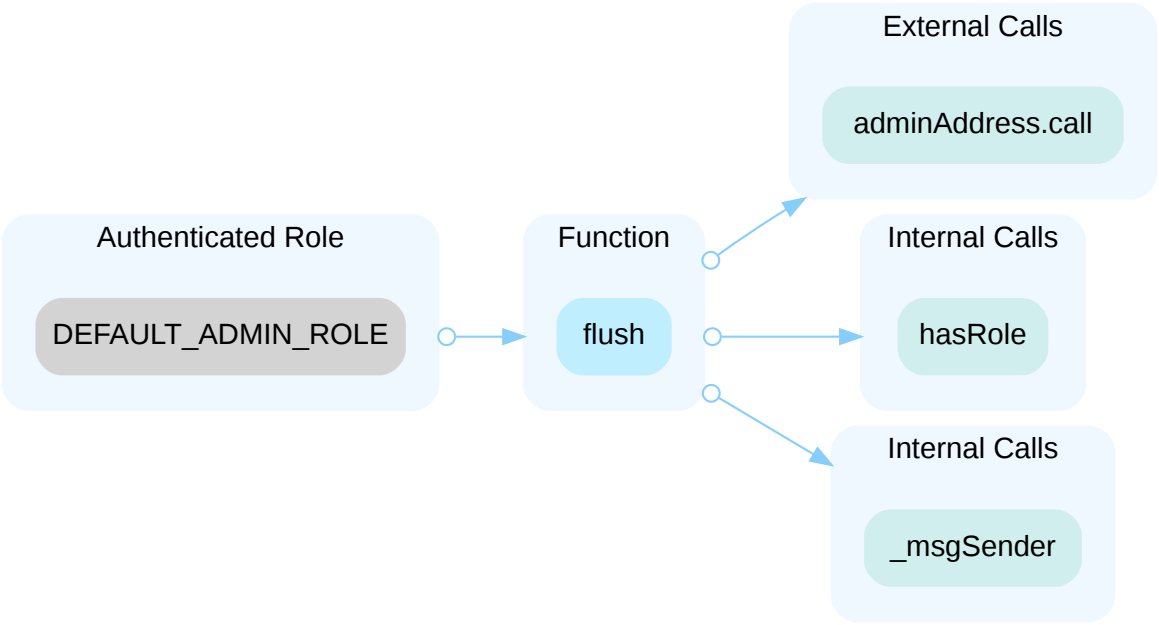
- withdraw all tokens from the contract using only their own signature and two fake signatures;
- update the wallet address so that all funds are redirected to their own account or owned contracts;
- reset the signer and admin addresses to addresses they control;
- update commissions and fees so that a maximum amount of value can be taken out of the Invoice contract with one signed message;


Function
setEarnCommission
State Variables
earnPct
earnToken
earnStatus
Function
updateMerchantPaymentToken
State Variables
merchantPaymentToken
Function
setAdmin
State Variables
owner
Function
updateMerchantCreditToken
State Variables
merchantCreditToken
Function
setStackCommission
State Variables
stackRewardStatus
stackRewardPct
stackRewardToken
Function
setBurnRate
State Variables
burnRate
Authenticated Role
owner
Function
setMerchantFee
State Variables
merchantFee
Function
setMerchantCommission
State Variables
merchantRewardToken
merchantRewardStatus
merchantRewardPct
External Calls
token.balanceOf
Function
withdraw
Internal Calls
recoverAddressFromSignature
External Calls
payable.call
External Calls
token.safeTransfer
Function
setCustomerFee
Function
setSigner
State Variables
customerFee
State Variables
quoteSigner
Function
setCustomerCommission
State Variables
customerRewardStatus
customerRewardToken
customerRewardPct
Function
updateWalletContract
State Variables
walletAddress

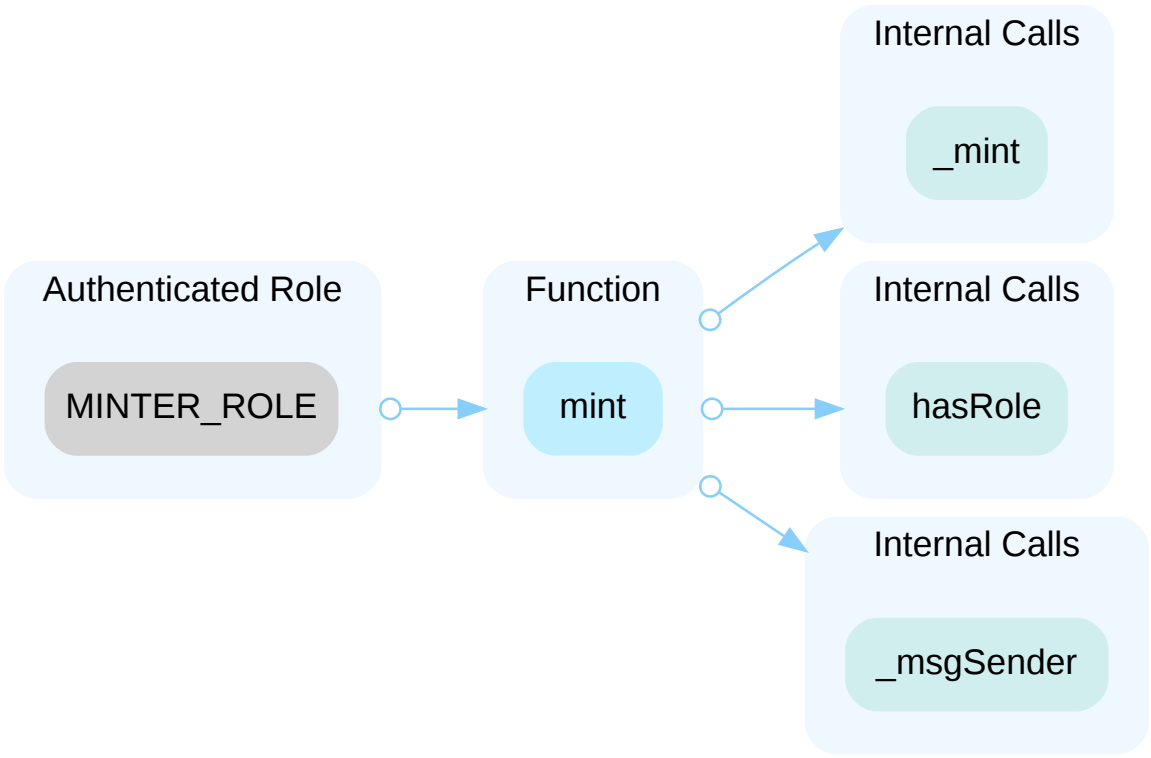
In the contract `ERC20PresetMinterPauser` the role `DEFAULT_ADMIN_ROLE` has authority over the functions

- `grantRole()` ;
- `revokeRole()` ; Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow the hacker to take advantage of this authority and grant any role to addresses they control.

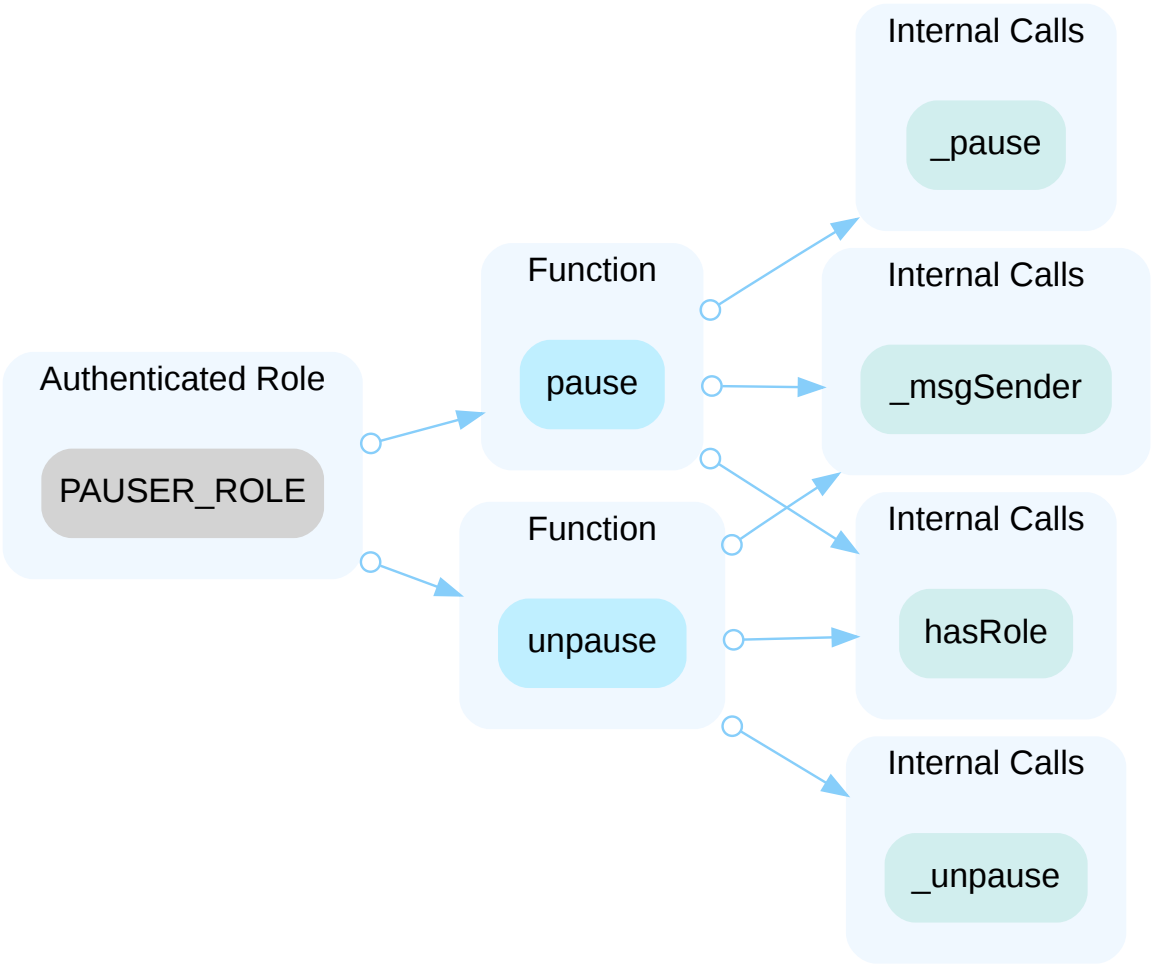
In the contract `ERC20PresetMinterPauser` the role `DEFAULT_ADMIN_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `DEFAULT_ADMIN_ROLE` account may allow the hacker to take advantage of this authority and remove all native tokens from the contract, sending them to `adminAddress` .



In the contract `ERC20PresetMinterPauser` the role `MINTER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `MINTER_ROLE` account may allow the hacker to take advantage of this authority and mint as many tokens as they want, to any address.



In the contract `ERC20PresetMinterPauser` the role `PAUSER_ROLE` has authority over the functions shown in the diagram below. Any compromise to the `PAUSER_ROLE` account may allow the hacker to take advantage of this authority and pause the contract. We note, however, that the `whenNotPaused` modifier is only used on function `pause()`.



Recommendation

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.

Alleviation

`[CertiK]`:

- `ERC20MinterPauser`: The team acknowledges the finding and makes no change at this time;
- `Invoice`: The following efforts are in place for alleviating centralization risk
  - requirement of at least 2 distinct signers' approval for execution of all privileged functions;
  - inclusion of timelock on most privileged actions;

  To mitigate the finding for `Invoice.sol`, we recommend that
  - all privileged functions include a timelock with latency of *at least* 24 hours. Currently, function `withdraw()` does not include a delay and all other privileged functions can have a `operationDelay` of any nonnegative value.
- `usdRate`: The following efforts are in place for alleviating centralization risk
  - two roles are required to call and successfully execute a privileged function
  - inclusion of timelock on privileged actions;

  To mitigate the finding for `usdRate.sol`, we recommend that

- all privileged functions include a timelock with latency of *at least* 24 hours. Currently, all privileged functions can have a `operationDelay` of any nonnegative value.
- all privileged functions require the approval of at least 2 distinct signers. Currently, State variables `DEFAULT_ADMIN_ROLE` and `adminSigner` can be set to the same address. Even with the requirement that `_adminSigner` is not `msg.sender` in the `constructor`, the `DEFAULT_ADMIN_ROLE` can always be updated to the `adminSigner` address at any time through the functionality in the inherited `AccessControl` contract.

INV-07 | NO UPPER LIMIT FOR FEES AND COMMISSION

Category	Severity	Location	Status
Logical Issue	Medium	Invoice.sol (Invoice_Base): 461, 468, 475~476, 483~484, 492~493, 501~502, 510~511	Resolved

Description

There are no upper boundaries for the following functions:

- setMerchantFee
- setCustomerFee
- setBurnRate
- setMerchantCommission
- setCustomerCommission
- setStackCommission
- setEarnCommission

It is possible to set the total fee and commission rate up to any arbitrary amount, exceeding the `amount` specified in the signed message. Moreover, some values act as flat fees, while others act as percentages, making it impossible to ensure that the `amount` used in the signed message covers all fees and commissions.

Recommendation

We recommend adding reasonable boundaries for the fees and commissions. Furthermore, we recommend against the use of flat fees since the `amount` for the signed message may be lesser than this value.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

INV-08 SOME VALID SIGNATURES MAY BE USED WITHOUT A CORRESPONDING TRANSFER OF TOKENS

Category	Severity	Location	Status
Logical Issue	Medium	Invoice.sol (Invoice_Base): 329~330	Resolved

Description

In the case where `merchantPaymentToken` is still set as `address(0)`, a valid signature can be recorded as used without any corresponding tokens being transferred.

In such a case, the message cannot be used again, and an event is still emitted that makes it appear as if a payment was made, where the `msg.sender` may include a `tokenValue` input of any number.

If the protocol relies on the status of `isPaid` or the `tokenValue` used, this may be misleading.

Recommendation

We recommend reverting if the payment cannot be processed by `processTCommision()`.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

INV-09 NO CHECK THAT `signer2` AND `signer3` ARE `withdrawSigner` S

Category	Severity	Location	Status
Logical Issue	Medium	Invoice.sol (Invoice_Base): 410~411, 414~415, 417~418	Resolved

Description

Function `withdraw()` uses three signatures to verify the `operationHash` of a withdrawal of native tokens or ERC20 tokens. However, there is no check that the recovered `signer2` and `signer3` addresses are valid `withdrawSigner` addresses for the contract. This being the case, the `owner` can sign the message, and provide two signatures for any other two addresses, even both recovering `address(0)`, and the logic of `withdraw()` will allow the withdrawal.

Recommendation

We recommend ensuring that `sign2` and `sign3` recover addresses within array `withdrawSigner`.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

INV-13 | USE OF `getZeebuToZusdRate()`

Category	Severity	Location	Status
Incorrect Calculation, Inconsistency	Medium	Invoice.sol (Invoice_Base): 344~345, 351~352, 355~356, 371~372, 379~380, 387~388, 395~396, 556~557, 560~561	Acknowledged

Description

Use of function `getZeebuToZusdRate()` as the `zUsdRate` implies it is a value used in the conversion from Zeebu to a USD value. However, the naming and logic in function `getOriginalAmount()` implies that the `originalAmount` is already an amount in USD.

With both these aspects in mind, it appears incorrect that in the function call `getExtraReward()`, the `extraAmt` is divided by the `rate`.

Please clarify whether `getZeebuToZusdRate()` is supposed to return a value used in converting from a USD value to some other denomination.

Recommendation

We recommend clarifying whether `getZeebuToZusdRate()` is supposed to return a value used in converting from a USD value to some other denomination.

Alleviation

`[ZeebuMobile]`: "getZeebuToZusdRate() used for reward distribution process by calculating using received rate, where originalAmount is base calculation process"

`[CertiK]`: Please clarify what the `tokenValue` transferred into the contract is in a stable coin denomination pegged to the US dollar and is being converted to another denomination, *or* if the `tokenValue` transferred is in a denomination that is to be converted to a stable coin value.

INV-19 | POTENTIAL REPLAY ATTACK

Category	Severity	Location	Status
Logical Issue	Medium	Invoice.sol (Invoice_Base): 252~269, 300~317; MyWallet.sol (MyWallet_Base): 234~235, 263~264; projects/Zeebu/wallet/ZEEBUWallet.sol (update2): 381~386	Partially Resolved

Description

`MyWallet.sol`

The signature verification process in contract `MyWallet` does not include the following information in the message signed:

- chain Id
- contract address
- function name

If this contract is ever deployed more than once using any of the same signer addresses, either on the same or different chain, then it may be possible to replay messages in an unintended setting.

Additionally, even though strings "ERC20" and "ETHER" are used to distinguish between which function the message is intended for, if this formatting is ever used in other projects, it may be ambiguous enough to allow for message to be replayed in other projects the team has set up.

`Invoice.sol`

Likewise, the signature verification process in contract `Invoice` excludes the following information:

- chain Id
- contract address
- processing deadline
- function name

Recommendation

We recommend including information such as chain Id, function name, nonce, and contract address in signed messages. Many projects do this through the use of a "domain separator". More information on this can be see in **EIP-712**. Additionally, consider inheriting OpenZeppelin's EIP712 contract to handle the inclusion of this information: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/EIP712.sol

Alleviation

`[CertiK]`: The team made changes partially resolving the finding.

The following functions are still vulnerable to replay attack and need to be modified in order to fully resolve the finding:

`ZeebuWallet.sol`

- For functions `activateSafeMode()` and `deactivateSafeMode()`, one valid signature can be used in either function. To prevent this, the function and signature should also include a information such as a selector or a new network id *that distinguishes between the two functions*.

The team notes that signature replay may not be possible because the functions cited are privileged. The finding concerns the potential vulnerability in the event that the admin accounts are compromised.

MWC-21 | LACK OF ACCESS CONTROL

Category	Severity	Location	Status
Logical Issue	Medium	MyWallet.sol (MyWallet_Base): 81~82, 89~90, 179~180	Resolved

Description

`ForwarderContract`

- Function `flushAmountToken()` allows anyone to transfer any ERC20 tokens from the `ForwarderContract` to the `parentAddress` at any time. This function allows external interactions with user-provided addresses.
- Function `flush()` allows anyone to transfer native tokens from the `ForwarderContract` to the `parentAddress` at any time.

`MyWallet`

- Function `generateForwarder()` allows anyone to create a new `ForwarderContract` instance which has the `MyWallet` address as the `parentAddress`. This may allow for an unnecessary amount of `ForwarderContract` instances being created and in some instances, may cause a denial of service.

Recommendation

We recommend restricting access to the functions above.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

USD-02 | POTENTIALLY DANGEROUS FIXED CONVERSION RATES

Category	Severity	Location	Status
Logical Issue	Medium	usdRate.sol (usdRate_Base): 739~741	Acknowledged

Description

Contract `usdRate` appears to be a reference contract for set fixed values `NativeToZeebuRate`, `NativeToZusdRate` and `ZeebuToZusdRate`.

The relative worth of native tokens and stable coins is constantly fluctuating and may allow users to profit through bad arbitrage if these recorded values are intended to be used to determine conversion rates for swaps.

Recommendation

We recommend providing more information on how the above described issue is prevented, if it is, or handling the issue described otherwise.

Alleviation

`[CertiK]`: The team acknowledges the finding and opts to provide no further information at this time.

The team states the rate will be used for reward distribution and that the value will be updated when changed.

Please confirm if the following assumptions are correct:

1. The tokens that rely on the fixed conversion rates `NativeToZeebuRate`, `NativeToZusdRate` and `ZeebuToZusdRate` are tokens which will not be available in any automated market maker through an LP token pair
2. All tokens which rely on the fixed conversion rates above are tokens which include the `onERC20Receive` hook in their transfer logic so that the tokens cannot be used effectively in AMMs if someone independently creates a pair with the tokens

If these two items are confirmed, the finding can be resolved, since the `ERC20MinterPauser` transfer hook logic prevents the token from being effectively transferred in AMM exchanges.

ZCK-01 | MALLEABILITY OF `abi.encodePacked`

Category	Severity	Location	Status
Volatile Code	Medium	wallet/WalletFactory.sol (update1): 18, 18, 20; wallet/ZEEBUWallet.sol (update1): 186, 193~202, 239, 240, 254~262	Resolved

Description

Encoding through `abi.encodePacked()` is used with dynamically-sized inputs controlled by the user. This type of encoding is susceptible to malleability when used with strings, bytes, or other dynamically-sized types. As such, it is not a good choice for signature verification with the input used, since users can change the input and potentially get the same bytes output from `abi.encodePacked`.

```
20      bytes32 finalSalt = keccak256(abi.encodePacked(allowedSigners,_allowedToken,
 salt));
```

- `abi.encodePacked` is called with dynamically-sized arguments: `allowedSigners`, `_allowedToken`

```
18    function createWallet(address[] calldata allowedSigners,address[] calldata
 _allowedToken, bytes32 salt) external {
```

- `allowedSigners` is a dynamically-sized external parameter.

```
18    function createWallet(address[] calldata allowedSigners,address[] calldata
 _allowedToken, bytes32 salt) external {
```

- `_allowedToken` is a dynamically-sized external parameter.

```
193         abi.encodePacked(
194           getChainId(),
195           address(this),
196           getNetworkId(),
197           toAddress,
198           value,
199           data,
200           expireTime,
201           sequenceId
202         )
```

- `abi.encodePacked` is called with dynamically-sized arguments: `getNetworkId()`, `data`

```
186     bytes calldata data,
```

- `data` is a dynamically-sized external parameter.

```
254       abi.encodePacked(
255         getChainId(),
256         address(this),
257         getBatchNetworkId(),
258         recipients,
259         values,
260         expireTime,
261         sequenceId
262       )
```

- `abi.encodePacked` is called with dynamically-sized arguments: `getBatchNetworkId()`, `recipients`, `values`

```
239     address[] calldata recipients,
```

- `recipients` is a dynamically-sized external parameter.

```
240     uint256[] calldata values,
```

- `values` is a dynamically-sized external parameter.

Recommendation

We recommend the use of `abi.encode` instead of `abi.encodePacked`.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

ERC-02 PURPOSE OF HOLDING NATIVE TOKENS IN `ERC20MinterPauser`

Category	Severity	Location	Status
Inconsistency	Minor	ERC20MinterPauser.sol (Invoice_Base): 2325~2332, 2334~2335	Resolved

Description

Low-level functions `receive()` and `fallback()` are present in the `ERC20MinterPauser` contract, allowing the contract to hold native tokens.

There does not appear to be any need for these functions, as the rest of the contract is independent from their inclusion. Their inclusion may cause confusion, since if a low-level call is made to the contract with the mistaken intention of calling a function which does not exist in the contract, the `fallback()` function will capture the `msg.data` without reverting, potentially causing users to lose any included `msg.value`.

Recommendation

We recommend providing information on the intended purpose of including these functions.

Alleviation

`[ZeebuMobile]`: "its for take charge from end user during transfer at implementation but later on not needs, but function remain in contract, if transferred native coin, it can be return by admin/owner as it has functionality to flush"

FFZ-01 POTENTIAL REENTRANCY ATTACK (INCREMENTING STATE)

Category	Severity	Location	Status
Concurrency	Minor	wallet/ForwarderFactory.sol (update1): 31~34, 35	Resolved

Description

A reentrancy attack can occur when the contract creates a function that makes an external call to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

This finding is considered minor because the state variable is only incremented or decremented. So, the effect of out-of-order increments may be unobservable after transaction. However, the reentrancy vulnerability may still cause other issues in the middle of transaction.

External call(s)

```
31      Forwarder(clone).init(
32        parent,
33        _allowedToken
34      );
```

State variables written after the call(s)

```
35      forwarderCount[parent] += 1;
```

Note: this finding is separate from other reentrancy findings to distinguish the newly added locations. In the next iteration of the report, the finding will be merged with the relevant finding.

Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts or applying OpenZeppelin ReentrancyGuard library - `nonReentrant` modifier for the aforementioned functions to prevent reentrancy attack.

Alleviation

`[CertiK]` : The team made changes resolving the finding.

INV-03 `walletAddress` IS NOT INITIALIZED ON DEPLOYMENT OF THE CONTRACT

Category	Severity	Location	Status
Logical Issue	Minor	Invoice.sol (Invoice_Base): 181~182	Resolved

Description

Address `walletAddress` is not initialized at the time of deploying the `Invoice` contract. This address is used in verifying the legitimacy of a payment, and leaving as `address(0)` may allow unintended destinations to be set.

Recommendation

We recommend setting `walletAddress` upon deployment of the `Invoice` contract.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

INV-10 | UNUSED VALUE

Category	Severity	Location	Status
Inconsistency	Minor	Invoice.sol (Invoice_Base): 542, 545, 548, 551	Resolved

Description

```
542             tokenDecimal = getTokenDecimal(merchantRewardToken);
```

- The variable `tokenDecimal` is never used after this assignment.

```
545             tokenDecimal = getTokenDecimal(customerRewardToken);
```

- The variable `tokenDecimal` is never used after this assignment.

```
548             tokenDecimal = getTokenDecimal(stackRewardToken);
```

- The variable `tokenDecimal` is never used after this assignment.

```
551             tokenDecimal = getTokenDecimal(earnToken);
```

- The variable `tokenDecimal` is never used after this assignment.

Recommendation

We recommend reviewing those unused values for omission of their usage or considering to remove their definitions.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

INV-11 | POTENTIAL FOR UNINTENDED SENT ETHER

Category	Severity	Location	Status
Logical Issue	Minor	Invoice.sol (Invoice_Base): 319~320	Resolved

Description

The contract `Invoice` includes a payable function `pay()` without any verification of the amount sent. If a user mistakenly sends native tokens, there is no way for the caller to be refunded.

Recommendation

We recommend adding in logic that checks whether a `msg.value` should be included in the function call, and, if not, reverting.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

MWC-04 | CONFLICT IN USE OF `onlySigner()` MODIFIER

Category	Severity	Location	Status
Logical Issue	Minor	MyWallet.sol (MyWallet_Base): 160~161, 169~170, 199~200	Resolved

Description

The following functions can immediately be called by any valid signer address in `MyWallet` at any time:

- `activateSafeMode()`;
- `deactivateSafeMode()`;
- `flushForwarderDeposit()`;

If there is any disagreement in the use of these functions, the state of the contract may be volatile. For instance, if one signer believes the contract should be in safemode while another thinks it should not, the latter signer can call `deactivateSafeMode()` after the first has called `activateSafeMode()`, leaving potential for unexpected updates.

Recommendation

We recommend requiring updates to the functions above also require 2 out of 3 signers to agree with the action before it is executed, similar to functions `transferMultiSigEther()` and `transferMultiSigTokens()`. See the finding "Centralization Risks in `MyWallet.sol`" for more information

Alleviation

`[CertiK]`: The team made changes resolving the finding.

MWC-05 | SAFEMATH NOT USED

Category	Severity	Location	Status
Incorrect Calculation	Minor	MyWallet.sol (MyWallet_Base): 181~182, 209~210, 298~299	Resolved

Description

OpenZeppelin's `SafeMath` library is not used in the following functions, making it possible for overflow/underflow to occur and lead to inaccurate calculations.

- In function `generateForwarder()`, the `forwarderCount` can theoretically overflow if it is called enough times;
- In function `getNonce()`, the return value can overflow if `lastNounce` is $2^{256} - 1$;
- In function `validateNonce()`, `lastNounce+1000` may overflow, restricting the validity of the input `nounce` and potentially disrupting the protocol's ability to validate signatures.

Recommendation

We recommend either upgrading to compiler version 0.8.0 or above so that overflow/underflow check is incorporated, or else we recommend using OpenZeppelin's SafeMath library for all relevant mathematical operations.

Reference: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/SafeMath.sol

Alleviation

`[CertiK]`: The team made changes resolving this finding.

MWC-06 INSUFFICIENT CHECKS BEFORE ATTEMPTING TRANSFER

Category	Severity	Location	Status
Logical Issue	Minor	MyWallet.sol (MyWallet_Base): 236~237, 266~267	Resolved

Description

Function `transferMultiSigEther()` does not check that `address(this)` has at least the specified input `value` of native tokens before attempting to transfer the amount. If the contract does not hold this amount, the function will revert without a specific error message.

Function `transferMultiSigTokens()` only checks that `balanceOf(address(this))` is positive before attempting to transfer a `value` amount of tokens. If the contract does not hold at least this balance, then the transfer will revert.

Recommendation

We recommend adding a check in each function above that the contract holds at least the `value` meant to be transferred and reverting with a specific error message in that case.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

MWC-07 PURPOSE OF INCLUDED `msg.data` IN `ForwarderDeposited` EVENT

Category	Severity	Location	Status
Logical Issue, Access Control	Minor	MyWallet.sol (MyWallet_Base): 56~57, 138~139	Resolved

Description

When a low-level call is made to a `ForwarderContract` or `MyWallet` contract, an event, `ForwarderDeposited` (respectively, `Deposited), is emitted which includes any` msg.data` that was also sent.

Recommendation

We recommend providing information on what `msg.data` is expected to be sent by users interacting via low-level call with these contracts and the intention behind emitting this information in the event.

Alleviation

`[ZeebuMobile]` : "Future implementaion aspect is consider , if we take message data with some values and from event validate and process function"

MWC-08 | PURPOSE OF FLUSHING FUNCTIONS

Category	Severity	Location	Status
Coding Style	Minor	MyWallet.sol (MyWallet_Base): 68~74, 81~83, 199~202	Resolved

Description

The `Forwarder` contract includes both a protected method (function `flushDeposit()`) and unprotected method (`flushAmountDeposit()`) for performing the same action.

If the action should be protected, then including an unprotected function poses a security risk.

If anyone can perform the action, there is no need to include a privileged version of the function.

Recommendation

We recommend stating the intended purpose behind including both a protected method (function `flushDeposit()`) and unprotected method (`flushAmountDeposit()`) for performing the same action.

Alleviation

`[ZeebuMobile]` : Flushing is for centralize fund in wallet at single place , as its wallet functionality

MWC-09 PURPOSE OF `payable` CASTING ON `transferMultiSigEther()`

Category	Severity	Location	Status
Logical Issue	Minor	MyWallet.sol (MyWallet_Base): 233~234	Resolved

Description

Function `transferMultiSigEther()` includes a `payable` casting, allowing the signer calling the function to include native tokens in the call. The function caller also specifies an input `value` which is not checked against the `msg.value` the signer may include in the call, or the balance of the contract itself.

Please provide information on whether it is intended for the signer calling `transferMultiSigEther()` to contribute to the `value` that is being transferred from the `MyWallet` contract to the input `toAddress` account.

Recommendation

If the signer calling `transferMultiSigEther()` is expected to include `msg.value` in the amount `value` used as input in the function call, then we recommend adding a check in the function logic that this is the case.

If the signer is not expected to include a `msg.value` with the function call, we recommend removing the `payable` casting from the function.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

MWC-11 | OUTDATED SOLIDITY VERSION

Category	Severity	Location	Status
Coding Issue	Minor	MyWallet.sol (MyWallet_Base): 1~2	Resolved

Description

File `MyWallet.sol` uses Solidity version 0.5.12, which is a version introduced several years ago. Outdated versions may contain security issues and miss the new features and security enhancements included in the recent solidity version.

Recommendation

We recommend using a recent version of solidity, at least version 0.8.0 or higher, which has the built-in Safemath protection.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

TES-02 | UNPROTECTED `onERC20Receive()` FUNCTION

Category	Severity	Location	Status
Logical Issue	Minor	Invoice.sol (Invoice_Base): 587~590; MyWallet.sol (MyWallet_Base): 44~47	Resolved

Description

Function `onERC20Receive()` is a function called by the `ERC20MinterPauser.sol` contract which is supposed to signify that a receiving contract can properly handle a transfer of the token. In the `ForwarderContract` and `Invoice` contract, its implementation is unprotected, meaning anyone can call it using any input as for parameters `from`, `amount`, and `msgSender`. Since this function emits a `ReceivedTokens` event, this may transmit misleading information. If any other systems rely on information emitted by the event, this lack of control can be exploited.

Recommendation

We recommend making the `msgSender` parameter used in the `ReceivedTokens` event emitted based on the actual `msg.sender` calling the function by including this in the implementation logic of `onERC20Receive()`, rather than leaving it as an input parameter. Similarly, address `to` does not need to be an input parameter, since in the event, the destination address should always be `address(this)`.

Additionally, we recommend considering restricting the access to this function in `ForwarderContract` and `Invoice` to be only the tokens expected to call the function.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

TES-06 | LACK OF INPUT VALIDATION

Category	Severity	Location	Status
Volatile Code	Minor	Invoice.sol (Invoice_Base): 239~240, 240~241, 241~242, 242~243, 244~245, 408, 432~433, 436~437, 441~442, 447~448, 453~454, 480~481, 489~490, 498~499, 507~508; MyWallet.sol (MyWallet_Base): 128	Resolved

Description

`MyWallet.sol`

- In the `constructor` there is no check that the addresses in array `allowedSigners` are all distinct. If all addresses are listed as the same, then function `verifyMultiSig()` will always revert since `otherSigner` cannot be distinct from `msg.sender`;
- In the `constructor`, there is no check that `address(0)` is not in the `allowedSigners` array. If it is included, then invalid signatures may be processed as valid, since `ecrecover()` returns `address(0)` when an improper signature is used;

`Invoice.sol`

- In the `constructor` there is no check that the addresses used to set `quoteSigner`, `usdRateContract`, `stackerPool`, `earningPool` and `withdrawSigner` are nonzero addresses
  - If `withdrawSigner` contains `address(0)` this may allow invalid signatures to be accepted, since `ecrecover()` returns `address(0)` on failure;
- In the `constructor` there is no check that `_withdrawSigner` contains two distinct addresses. Function `withdraw()` allows for the possibility of both addresses being the same;
- In function `withdraw()`, the `expiration` used to create a signed message is not used to verify the timestamp does not exceed this value;
- In function `setSigner()` there is no check that `newQuoteSigner` is not `address(0)`. If set as `address(0)`, this may allow invalid signatures to be used with the protocol;
- In function `setAdmin()` there is no check that `newAdmin` is not `address(0)`. If set as `address(0)`, this may allow invalid signatures to be used with the protocol;
- In function `updateWalletContract()` there is no check that `walletAdd` is not `address(0)`;
- In function `updateMerchantCreditToken()` there is no check that `tokenAdd` is not `address(0)`;
- In function `updateMerchantPaymentToken()` there is no check that `tokenAdd` is not `address(0)`;
- In function `setMerchantCommission()`, if `status` is updated to `true`, then it should be checked that `tokenAdd` is not `address(0)`;

- In function `setCustomerCommission()`, if `status` is updated to `true`, then it should be checked that `tokenAdd` is not `address(0)`;
- In function `setStackCommission()`, if `status` is updated to `true`, then it should be checked that `tokenAdd` is not `address(0)`;
- In function `setEarnCommission()`, if `status` is updated to `true`, then it should be checked that `tokenAdd` is not `address(0)`;

Recommendation

We recommend including the specified input validation above.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

TES-15 POTENTIAL REENTRANCY ATTACK (OUT-OF-ORDER EVENTS)

Category	Severity	Location	Status
Volatile Code	Minor	ERC20MinterPauser.sol (Invoice_Base): 136, 339; Invoice.sol (Invoice_Base): 336, 339, 362, 365, 366, 372, 373, 380, 381, 388, 389, 396, 404, 526; MyWallet.sol (MyWallet_Base): 54, 56, 72, 73, 236, 238, 267, 268	Resolved

Description

A reentrancy attack can occur when the contract creates a function that makes an external call to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

This finding is considered minor because the reentrancy only causes out-of-order events.

External call(s)

```
336         processTCommision(msg.sender,customer,merchant,tokenValue,amount,fee);
```

- This function call executes the following external call(s).
- In `SafeERC20._callOptionalReturn`,
  - `returndata = address(token).functionCall(data,"SafeERC20: low-level call failed")`
- In `Address.functionCallWithValue`,
  - `(success,returndata) = target.call{value: value}(data)`
- In `Invoice.transferToken`,
  - `token.safeTransfer(user,amount)`
- In `Invoice.processTCommision`,
  - `token.safeTransferFrom(msgSender,address(this),tAmount)`
- In `Invoice.processTCommision`,
  - `erc20.burn(burnAmount)`

Events emitted after the call(s)

```
339         emit PaymentAccepted(hash,customer,merchant,tokenValue,amount,fee,payload
);
```

External call(s)

```
362         token.safeTransferFrom(msgSender, address(this), tAmount);
```

```
365         transferToken(merchantCreditToken,merchant,paymentInUsd);
```

- This function call executes the following external call(s).
- In `SafeERC20._callOptionalReturn`,
  - `returndata = address(token).functionCall(data,"SafeERC20: low-level call failed")`
- In `Address.functionCallWithValue`,
  - `(success,returndata) = target.call{value: value}(data)`
- In `Invoice.transferToken`,
  - `token.safeTransfer(user,amount)`

Events emitted after the call(s)

```
366         emit MerchantPaid(merchant,paymentInUsd,feeInUsd);
```

```
372             emit merchantrewarded(merchant,originalAmount,extraRewardAmount,
zUsdRate,merchantRewardPct);
```

External call(s)

```
362         token.safeTransferFrom(msgSender, address(this), tAmount);
```

```
365         transferToken(merchantCreditToken,merchant,paymentInUsd);
```

- This function call executes the following external call(s).
- In `SafeERC20._callOptionalReturn`,

- `returndata = address(token).functionCall(data,"SafeERC20: low-level call failed")`

- In `Address.functionCallWithValue`,
  - `(success,returndata) = target.call{value: value}(data)`
- In `Invoice.transferToken`,
  - `token.safeTransfer(user,amount)`

```
373             transferToken(merchantRewardToken,merchant,extraRewardAmount);
```

Events emitted after the call(s)

```
380             emit customerrewarded(customer,originalAmount,rewardAmount,zUsdRate
,customerRewardPct);
```

External call(s)

```
362         token.safeTransferFrom(msgSender, address(this), tAmount);
```

```
365         transferToken(merchantCreditToken,merchant,paymentInUsd);
```

- This function call executes the following external call(s).
- In `SafeERC20._callOptionalReturn`,
  - `returndata = address(token).functionCall(data,"SafeERC20: low-level call failed")`
- In `Address.functionCallWithValue`,
  - `(success,returndata) = target.call{value: value}(data)`
- In `Invoice.transferToken`,
  - `token.safeTransfer(user,amount)`

```
373             transferToken(merchantRewardToken,merchant,extraRewardAmount);
```

```
381             transferToken(customerRewardToken,customer,rewardAmount);
```

Events emitted after the call(s)

```
388             emit stackerrewarded(stackerPool,originalAmount,stackRewardAmount,
zUsdRate,stackRewardPct);
```

External call(s)

```
362         token.safeTransferFrom(msgSender, address(this), tAmount);
```

```
365         transferToken(merchantCreditToken,merchant,paymentInUsd);
```

- This function call executes the following external call(s).
- In `SafeERC20._callOptionalReturn`,
  - `returndata = address(token).functionCall(data,"SafeERC20: low-level call failed")`
- In `Address.functionCallWithValue`,
  - `(success,returndata) = target.call{value: value}(data)`
- In `Invoice.transferToken`,
  - `token.safeTransfer(user,amount)`

```
373             transferToken(merchantRewardToken,merchant,extraRewardAmount);
```

```
381             transferToken(customerRewardToken,customer,rewardAmount);
```

```
389             transferToken(stackRewardToken,stackerPool,stackRewardAmount);
```

Events emitted after the call(s)

```
396             emit earnTransfer(earningPool,originalAmount,earnAmount,zUsdRate,
earnPct);
```

External call(s)

```
54          (bool sent,) = parentAddress.call.value(msg.value)("");
```

Events emitted after the call(s)

```
56          emit ForwarderDeposited(msg.sender, msg.value, msg.data);
```

External call(s)

```
72          require(instance.transfer(parentAddress, forwarderBalance));
```

Events emitted after the call(s)

```
73          emit TokensFlushed(address(this), forwarderBalance,
 _tokenContractAddress);
```

External call(s)

```
236          (bool sent,) = toAddress.call.value(value)("");
```

Events emitted after the call(s)

```
238          emit Transacted(msg.sender, otherSigner, operationHash, toAddress,
value, data);
```

External call(s)

```
267          require(instance.transfer(toAddress, value));
```

Events emitted after the call(s)

```
268          emit TokensTransfer(tokenContractAddress, value);
```

Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts or applying OpenZeppelin ReentrancyGuard library - `nonReentrant` modifier for the aforementioned functions to prevent reentrancy attack.

Alleviation

`[CertiK]` : The team made changes resolving the finding.

USD-01 | VARIABLES NOT INITIALIZED ON DEPLOY

Category	Severity	Location	Status
Logical Issue, Volatile Code	Minor	usdRate.sol (usdRate_Base): 739~740, 740~741, 741~742	Resolved

Description

`uint` variables `NativeToZeebuRate`, `NativeToZusdRate`, and `ZeebuToZusdRate` are 0 on deployment of the `usdRate` contract. If these values are not updated before the getter functions are referenced in other contracts, such as `Invoice`, then this may result in unintended conversion rates or reverts.

Recommendation

We recommend setting the variables referenced above upon deployment of the `usdRate` contract.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

ZCP-01 POTENTIAL REENTRANCY ATTACK (OUT-OF-ORDER EVENTS)

Category	Severity	Location	Status
Concurrency	Minor	ERC20MinterPauser.sol (update1): 2339, 2341; wallet/ForwarderFactory.sol (update1): 31~32, 36~39	Partially Resolved

Description

A reentrancy attack can occur when the contract creates a function that makes an external call to another untrusted contract before resolving any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the external call resolved the effects.

This finding is considered minor because the reentrancy only causes out-of-order events.

External call(s)

```
2339          (bool success, ) = adminAddress.call{value:value }("");
```

Events emitted after the call(s)

```
2341          emit DepositedToAdmin(msg.sender, value);
```

External call(s)

```
31              Forwarder(clone).init(
32        parent,
33        _allowedToken
34      );
```

Events emitted after the call(s)

```
36              emit ForwarderCreated(
37        clone,
38        parent
39      );
```

Recommendation

We recommend using the Checks-Effects-Interactions Pattern to avoid the risk of calling unknown contracts or applying OpenZeppelin ReentrancyGuard library - `nonReentrant` modifier for the aforementioned functions to prevent reentrancy attack.

Alleviation

`[CertiK]`: The team partially resolved the finding by adding a lock to the `createForwarder()` function in contract `ForwarderFactory`.

Function `flush()` in contract `ERC20MinterPauser` is unresolved. once this case is correctly handled, the finding can be resolved.

ERC-01 | USE CASE OF `onERC20Receive()`

Category	Severity	Location	Status
Logical Issue	Informational	ERC20MinterPauser.sol (Invoice_Base): 2310~2322	Resolved

Description

The contract `ERC20MinterPauser` modifies the function `_afterTokenTransfer()` to include a call to external function `onERC20Receive()` in the token recipient address `to`, when the recipient is a contract.

This can limit the use-case of the token, since any contract expecting to receive these tokens must include an implementation of the `onERC20Receive()` function. Moreover, this function is not part of a common EIP and other contracts may not know or be able to implement this function. For example, exchanges may not be able to handle the token because they do not include this function.

On the other hand, the check to this external function allows for the opportunity of reentrancy. Any protocol which transfers this token to a destination contract address must consider the possibility that the destination contract address may have modified `onERC20Receive()` to include malicious logic to perform reentrancy and correctly set up function logic to guard against this possibility.

Recommendation

We recommend considering whether the `onERC20Receive()` check is necessary for the correct functioning of the token and protocol, and if so, ensuring that the two points above are always handled.

Alleviation

`[CertiK]`: The team provides information that their `Invoice` contract acts as a payment processor. With this information, it appears intentional that not all protocols will be able to handle the `ERC20MinterPauser` token. However, we encourage the team to continue to consider the potential issues outlined above.

ERC-03 | SHADOWING LOCAL VARIABLE

Category	Severity	Location	Status
Coding Style	Informational	ERC20MinterPauser.sol (Invoice_Base): 2300, 2300	Resolved

Description

A local variable is shadowing another component defined elsewhere.

```
2300      constructor(string memory name, string memory symbol) ERC20(name, symbol)
{
```

- Local variable `name` in `ERC20PresetMinterPauser.constructor` shadows:
  - Function `name` in `ERC20`
  - Function `name` in `IERC20Metadata`

```
2300      constructor(string memory name, string memory symbol) ERC20(name, symbol)
{
```

- Local variable `symbol` in `ERC20PresetMinterPauser.constructor` shadows:
  - Function `symbol` in `ERC20`
  - Function `symbol` in `IERC20Metadata`

Recommendation

We recommend removing or renaming the local variable that shadows another definition.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

INV-01 | LACK OF DOCUMENTATION ON CONTEXT OF CONTRACT

Category	Severity	Location	Status
Coding Style	Informational	Invoice.sol (Invoice_Base): 174~175, 193~194, 194~195, 197~198, 202~203, 207~208, 212~213	Resolved

Description

The context in which a contract is used is an important aspect of security. A contract may be safely used in an isolated environment, but become insecure when composed with other contracts.

Recommendation

We recommend providing documentation on the context in which contract `Invoice` is used so that its interactions with other contracts can be considered. Namely, we recommend providing more information on the intended use of this contract, as well as any relevant information regarding the contracts intended to be used for the following addresses in the contract:

- `stackerPool`;
- `earningPool`;
- `merchantRewardToken`;
- `customerRewardToken`;
- `stackRewardToken`;
- `earnToken`;

Alleviation

`[CertiK]`: The team provides the following information below.

`[ZeebuMobile]`: We can say it is a payment processor, who accept crypto token as payment currency. Here the concept is as the Invoice payment process by user there will some reward process define which will execute during payment process & credit to specific addresses.

- stackerPool : it will be address where people staking the token, on which invoice contract will share reward on each payment transactions based on payment amount and configured percentage.
- earningPool : it will be address where company make profits on each payment transaction based on payment amount and configured percentage. There will be some fee taken from both sender and receiver.
- merchantRewardToken : its token contract address which will be consume during payment process to credit reward at receiver address.
- customerRewardToken : its token contract address which will be consume during payment process to credit reward at sender address.

- stackRewardToken : its token contract address which will be consume during payment process to credit reward at stackerPool address.
- earnToken : its token contract address which will be consume during payment process to credit reward at earningPool address.

INV-12 | PURPOSE OF `merchant` VS. `customer` ADDRESSES

Category	Severity	Location	Status
Logical Issue	Informational	Invoice.sol (Invoice_Base): 365~366, 369~374, 377~382, 603~611	Acknowledged

Description

Function `validateAddresses()` confirms that `merchant` and `customer` are expected to be two `ForwarderContract` instances which have the same `walletAddress` as their `parentAddress`. That being the case, any funds sent to either `ForwarderContract` are expected to eventually be transferred to the same `MyWallet` contract instance.

Please elaborate on the intended purpose of including both a `merchant` and a `customer` in a signed message. If both `customerRewardStatus` and `merchantRewardStatus` are set to true in the `Invoice` contract, the `walletAddress` of the contract will be sent both `customerRewardToken`, `merchantCreditToken`, and `merchantRewardToken`.

It seems the same `ForwarderContract` address could be used to send all tokens to the same location.

Recommendation

We recommend providing information on how a `merchant` versus a `customer` transfer is intended to be distributed if all funds eventually go to the same `parentAddress`.

Alleviation

`[ZeebuMobile]`: "Purpose is based on requirement, only forwarder belong to wallet only can process txns and received rewards."

`[CertiK]`: While it is understood that the Forwarder and Wallet contracts can receive rewards, all Forwarder contracts will send the value to the same `parentAddress` Wallet contract, where it may at that point be more difficult to distinguish which merchant and customer addresses are intended to receive the funds sent, and what amount should be sent.

INV-14 | CALCULATIONS IN `getOriginalAmount()`

Category	Severity	Location	Status
Incorrect Calculation, Logical Issue	Informational	Invoice.sol (Invoice_Base): 561~571	Acknowledged

Description

The `fee` and `uamount` used as input in `pay()` must be verified by a signer. Within the `Invoice` contract, function `getOriginalAmount()` requires that the input `fee` is the same as `feeInUsd` which is:

```
feeInUsd = originalAmount * customerFee / 100;
feeInUsd = feeInUsd.div(oneToken);
```

where `originalAmount` is the following:

```
originalAmount = uAmount.sub(fee);
```

This creates an atypical dependency between `fee` and `uamount`, where the signed message will only be accepted if the following relationship holds:

$$u = \left(\frac{f \cdot 100 \cdot 10^{18}}{c}\right) + f$$

where u represents `uamount`, f represents `fee` and c represents `customerFee`. This implies that for fixed `customerFee` and a given `fee`, only one `uamount` will be correct. Moreover, truncation by division must be considered in specifying the correct values of `fee` and `uamount`.

Please specify whether this relationship is intended. If so, please provide information on what the calculation is intended to represent.

Recommendation

We recommend specifying whether this relationship is intended. If so, please provide information on what the calculation is intended to represent.

Alleviation

`[ZeebuMobile]`: getOriginalAmount() , its to get amount excluding fee, also here validating fee received in input and fee configured in contract requirement.

`[CertiK]`: It is understood that the intended purpose of `getOriginalAmount()` is to get the amount excluding the fee and also to valid that the fee received as input matches the fee configured in the contract. It is now also understood that the division by `oneToken` is to account for additional precision factored into the `customerFee` value.

However, the logic in `getOriginalAmount()` uses the `originalAmount` in determining the `feeInUsd` *instead* of using the `uAmount` to determine the `feeInUsd` which we believe to be the correct value to be used.

Additionally, note that if `customerFee` is positive while the `fee` used as input was 0, then the check that the fees are the same will not be executed. If it is the intention to check that the fee approved by the input `signature` is the same as the `customerFee`, then this should be checked regardless of the `fee` input value.

INV-15 `validateAddress()` DOES NOT GUARANTEE `customer` AND `merchant` ARE FORWARDER CONTRACTS

Category	Severity	Location	Status
Logical Issue	Informational	Invoice.sol (Invoice_Base): 603~611	Resolved

Description

The function `validateAddress()` wraps addresses `merchant` and `customer` in the interface `walletContract` which declares function `getParentOfAddress()`. The function calls `getParentOfAddress()` and checks that the return address `userWalletIs` is the same as the defined `walletAddress` in the `Invoice` contract.

This only confirms that the `merchant` and `customer` addresses implement this function and does not guarantee that the respective addresses are actually `ForwarderContract` instances that were deployed by the `walletAddress`.

Recommendation

We recommend that the signers take the above information into consideration when signing messages and ensure they do not use the return address of `getParentOfAddress()` as a sufficient check that `merchant` and `customer` addresses are Forwarder contracts deployed by the `walletAddress`.

Alleviation

`[CertiK]`: The team acknowledges the finding. Since there is no action item regarding a change in the codebase, the finding is considered resolved, and it is assumed that the team will take the above information into consideration.

INV-16 | HIDDEN ASSUMPTION ON DECIMALS OF TOKENS USED

Category	Severity	Location	Status
Incorrect Calculation, Logical Issue	Informational	Invoice.sol (Invoice_Base): 561~562, 569~570, 576~577, 581~582	Resolved

Description

The calculation checked in `getOriginalAmount()` and used in `getMerchantFee()` both include the assumption that the token represented by `uAmount` has 18 decimals. If this is not the case in all situations of use, the calculation may be incorrect.

Recommendation

We recommend ensuring that tokens represented by `uAmount` include 18 decimals.

Alleviation

`[CertiK]`: The team acknowledges the finding. Since there is no action item regarding a change in the codebase in the case where all tokens used include 18 decimals, the finding is considered resolved, and it assumed that the team will take the above information into consideration.

MWC-13 | CONSIDER USE OF SAFEERC20 FOR TRANSFERS

Category	Severity	Location	Status
Volatile Code	Informational	MyWallet.sol (MyWallet_Base): 72~73, 82~83	Resolved

Description

Some ERC-20 tokens do not return a bool on successful transfer. If the cited contract functions are expected to handle a variety of ERC20 implementations, consider using OpenZeppelin's SafeERC20 library to handle all transfer logic, rather than checking the success of a bool in the contract logic itself.

Recommendation

Since some ERC-20 tokens return no values and others return a `bool` value, they should be handled with care. We recommend using OpenZeppelin's `SafeERC20.sol` implementation to interact with the `transfer()` and `transferFrom()` functions of external ERC-20 tokens. The OpenZeppelin implementation checks for the existence of a return value and reverts if `false` is returned, making it compatible with all ERC-20 token implementations.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

TES-07 | MISSING ERROR MESSAGES

Category	Severity	Location	Status
Coding Style	Informational	Invoice.sol (Invoice_Base): 243~244, 285~286; MyWallet.sol (MyWallet_Base): 32~33, 71~72, 72~73, 117~118, 129~130, 161~162, 170~171, 266~267, 267~268, 278~279	Resolved

Description

The `require()` statements should provide a description of the condition not met. Excluding an error message can make it difficult to troubleshoot the issue causing a revert.

Recommendation

We recommend adding the condition description for better code readability and usage. As an example:

```
function add(uint256 a, uint256 b) internal pure returns (uint256 c) {
    c = a + b;
    require(c >= a, "SafeMath: addition overflow");
}
```

Alleviation

`[CertiK]`: The team made changes resolving the finding.

TES-08 | USE OF `abi.encodePacked()`

Category	Severity	Location	Status
Volatile Code	Informational	Invoice.sol (Invoice_Base): 264~265, 279~280, 310~311, 411~412; MyWallet.sol (MyWallet_Base): 218~219, 234~235, 247~248, 263~264	Resolved

Description

Encoding through `abi.encodePacked()` with dynamically-sized inputs can cause hash collision in some instances. This type of encoding is susceptible to malleability when used with consecutive strings, bytes, or other dynamically-sized types.

While in this instance, hash collision cannot be accomplished, we recommend against using `abi.encodePacked` for hashing messages.

Recommendation

We recommend using an alternative to `abi.encodePacked` such as `abi.encode`.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

TES-09 | MISSING CHECK FOR `v` AND `s`

Category	Severity	Location	Status
Logical Issue	Informational	Invoice.sol (Invoice_Base): 287~298; MyWallet.sol (MyWallet_Base): 280~290	Resolved

Description

The following description is adapted from OpenZeppelin's ECDSA file:

EIP-2 still allows signature malleability for `ecrecover()`. Appendix F in the Ethereum Yellow paper, defines the valid range for s in (311): `0 < s < secp256k1n ÷ 2 + 1` and for the recovery identifier (312): `v ∈ {0,1}`. This should not be confused with the input for `ecrecover()` where `v ∈ {27,28}`. (See doc) However, these values can be obtained by taking `27 + "recovery identifier"`, so that they will also yield a unique signature and are often the `v` values returned from signatures. (For example `web3.eth.accounts.sign()`)

If your library generates malleable signatures, such as `s`-values in the upper range, calculate a new `s`-value with `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141 - s1` and flip `v` from `27` to `28` or vice versa. If your library also generates signatures with `0/1` for `v` instead `27/28`, add `27` to `v` so that `ecrecover()` accepts these signatures as well.

Recommendation

We recommend adding the following checks or to consider the example in `ECDSA.sol` from the OpenZeppelin library.

```
require(uint256(s) <=
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0, "ECDSA: invalid
signature 's' value");
require(uint8(v) == 27 || uint8(v) == 28, "ECDSA: invalid signature 'v' value");
```

Alleviation

`[CertiK]`: The team made changes resolving the finding.

TES-12 | MISSING EMIT EVENTS

Category	Severity	Location	Status
Coding Style	Informational	ERC20MinterPauser.sol (Invoice_Base): 2334~2335; Invoice.sol (Invoice_Base): 408~409, 432~433, 436~437; usdRate.sol (usdRate_Base): 750~751, 762~763, 774~775; MyWallet.sol (MyWallet_Base): 81~82, 199~200	Resolved

Description

Functions that update state variables should emit relevant events as notifications.

Recommendation

We recommend adding events for state-changing actions, and emitting them in their relevant functions.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

TES-13 | TYPOS

Category	Severity	Location	Status
Coding Style	Informational	Invoice.sol (Invoice_Base): 336~337, 343~344; MyWallet.sol (MyWallet_Base): 103~104, 205~206, 206~207, 209~210, 217~218, 218~219, 230~231, 233~234, 234~235, 235~236, 246~247, 247~248, 253~254, 259~260, 262~263, 263~264, 264~265, 295~296, 297~298, 298~299, 299~300, 308~309, 311~312, 319~320	Resolved

Description

`MyWallet.sol`

- The word "nonce" is written as "nounce" in the corresponding citations;
- The word "irrevocably" is incorrectly used in the comments of function `activateSafeMode()` and `deactivateSafeMode()`, since each action can be undone by the other;

`Invoice.sol`

- In function `processTCommision()`, the word "commission" is spelled incorrectly;

Recommendation

We recommend correcting the typos above.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

OPTIMIZATIONS | ZEEBU

ID	Title	Category	Severity	Status
INV-17	Tautology	Gas Optimization	Optimization	Resolved
INV-18	Unnecessary Use Of SafeMath	Gas Optimization	Optimization	Resolved
MWC-20	Unnecessary Use Of `onlySigner` Modifier	Gas Optimization	Optimization	Resolved
TES-03	User-Defined Getters	Gas Optimization	Optimization	Resolved
TES-04	Variables That Could Be Declared As Immutable	Gas Optimization	Optimization	Resolved
TES-14	Inefficient Memory Parameter	Gas Optimization	Optimization	Resolved

INV-17 | TAUTOLOGY

Category	Severity	Location	Status
Gas Optimization	Optimization	Invoice.sol (Invoice_Base): 460, 467, 474	Resolved

Description

Comparisons that are always true are unnecessary as written.

```
460         require(pct >= 0,"Invalid percentage");
```

```
467         require(pct >= 0,"Invalid percentage");
```

```
474         require(pct >= 0,"Invalid percentage");
```

In the cases above, the requirement does not ensure anything about the input `pct` used, since `pct` is of type `uint` and so necessarily is at least value 0.

Recommendation

We recommend fixing the incorrect comparison by updating the requirement to a stricter inequality.

Alleviation

`[CertiK]` : The team removed the requirement. which is equivalent to what was previously written, since all `pct` values can still be positive or equal to 0. However, gas is saved by not performing the check, and from context it appears that the intention is to allow a `pct` of 0 in some cases.

INV-18 | UNNECESSARY USE OF SAFEMATH

Category	Severity	Location	Status
Gas Optimization	Optimization	Invoice.sol (Invoice_Base): 19, 358, 556, 565, 569, 581	Resolved

Description

The `SafeMath` library is used unnecessarily. With Solidity compiler versions 0.8.0 or newer, arithmetic operations will automatically revert in case of integer overflow or underflow.

```
19  library SafeMath {
```

- An implementation of `SafeMath` library is found.

```
176     using SafeMath for uint;
```

- `SafeMath` library is used for `uint256` type in `Invoice` contract.

```
358         paymentInUsd = originalAmount.sub(feeInUsd);
```

- `SafeMath.sub` is called in `processTCommision` function of `Invoice` contract.

Note: Only a sample of 2 `SafeMath` library usage in this contract (out of 6) are shown above.

Recommendation

We advise removing the usage of `SafeMath` library and using the built-in arithmetic operations provided by the Solidity programming language.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

MWC-20 | UNNECESSARY USE OF `onlySigner` MODIFIER

Category	Severity	Location	Status
Gas Optimization	Optimization	MyWallet.sol (MyWallet_Base): 297	Resolved

Description

Function `validateNonce()` includes modifier `onlySigner`. Since this function is only called through functions `transferMultiSigTokens()` and `transferMultiSigEther()` which also include the modifier `onlySigner`, the inclusion of the modifier on the private function is not necessary.

Recommendation

We recommend removing the modifier `onlySigner` in the function `validateNonce()`.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

TES-03 | USER-DEFINED GETTERS

Category	Severity	Location	Status
Gas Optimization	Optimization	Invoice.sol (Invoice_Base): 170~171, 605~606; MyWallet.sol (MyWallet_Base): 22~23, 60~62	Resolved

Description

The linked function `getParentOfAddress()` is equivalent to the compiler-generated getter function for the respective variable `parentAddress`.

Recommendation

We recommend removing the linked function `getParentOfAddress()` and instead relying on the compiler-generated public getter function for variable `parentAddress()`.

Alleviation

`[CertiK]`: The team made changes resolving this finding.

TES-04 | VARIABLES THAT COULD BE DECLARED AS IMMUTABLE

Category	Severity	Location	Status
Gas Optimization	Optimization	Invoice.sol (Invoice_Base): 179~180, 180~181; MyWallet.sol (MyWallet_Base): 22	Resolved

Description

The linked variables assigned in the constructor can be declared as `immutable`. Immutable state variables can be assigned during contract creation but will remain constant throughout the lifetime of a deployed contract. A big advantage of immutable variables is that reading them is significantly cheaper than reading from regular state variables since they will not be stored in storage.

Recommendation

We recommend declaring these variables as immutable. Please note that the `immutable` keyword only works in Solidity version `v0.6.5` and up.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

TES-14 | INEFFICIENT MEMORY PARAMETER

Category	Severity	Location	Status
Gas Optimization	Optimization	Invoice.sol (Invoice_Base): 259, 277, 327, 408, 408, 408; MyWallet.sol (MyWallet_Base): 217, 233, 233, 262	Resolved

Description

One or more parameters with `memory` data location are never modified in their functions and those functions are never called internally within the contract. Thus, their data location can be changed to `calldata` to avoid the gas consumption copying from calldata to memory.

```
217     function generateEtherHash(address toAddress, uint value, bytes memory data
, uint expireTime, uint nounce)public pure returns (bytes32){
```

`generateEtherHash` has memory location parameters: `data`.

```
233     function transferMultiSigEther(address payable toAddress, uint value, bytes
memory data, uint expireTime, uint nounce, bytes memory signature) public payable
onlySigner {
```

`transferMultiSigEther` has memory location parameters: `data`, `signature`.

```
262     function transferMultiSigTokens(address toAddress, uint value, address
tokenContractAddress, uint expireTime, uint nounce, bytes memory signature) public
onlySigner {
```

`transferMultiSigTokens` has memory location parameters: `signature`.

One or more parameters with `memory` data location are never modified in their functions and those functions are never called internally within the contract. Thus, their data location can be changed to `calldata` to avoid the gas consumption copying from calldata to memory.

```
252     function isValidPayment(
```

`isValidPayment` has memory location parameters: `signature`.

```
271     function recoverQuoteSigner(
```

`recoverQuoteSigner` has memory location parameters: `signature`.

```
319     function pay(
```

`pay` has memory location parameters: `signature`.

```
408     function withdraw(uint amount,uint expiration, address tokenContract,bytes
memory sign1,bytes memory sign2,bytes memory sign3) public isAdmin {
```

`withdraw` has memory location parameters: `sign1`, `sign2`, `sign3`.

Recommendation

We recommend changing the parameter's data location to `calldata` to save gas.

- For Solidity versions prior to 0.6.9, since public functions are not allowed to have calldata parameters, the function visibility also needs to be changed to `external`.
- For Solidity versions prior to 0.5.0, since parameter data location is implicit, changing the function visibility to `external` will change the parameter's data location to calldata as well.

Alleviation

`[CertiK]`: The team made changes resolving the finding.

FORMAL VERIFICATION | ZEEBU

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-succeed-normal	`transfer` Succeeds on Admissible Non-self Transfers
erc20-transfer-correct-amount	`transfer` Transfers the Correct Amount in Non-self Transfers
erc20-transfer-succeed-self	`transfer` Succeeds on Admissible 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-transfer-never-return-false	`transfer` Never Returns `false`

Property Name	Title
erc20-transferfrom-revert-from-zero	`transferFrom` Fails for Transfers From the Zero Address
erc20-transferfrom-revert-to-zero	`transferFrom` Fails for Transfers To the Zero Address
erc20-transferfrom-succeed-normal	`transferFrom` Succeeds on Admissible Non-self Transfers
erc20-transferfrom-succeed-self	`transferFrom` Succeeds on Admissible Self Transfers
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-fail-exceed-balance	`transferFrom` Fails if the Requested Amount Exceeds the Available Balance
erc20-transferfrom-change-state	`transferFrom` Has No Unexpected State Changes
erc20-transferfrom-correct-allowance	`transferFrom` Updated the Allowance Correctly
erc20-transferfrom-fail-exceed-allowance	`transferFrom` Fails if the Requested Amount Exceeds the Available Allowance
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-correct-value	`totalSupply` Returns the Value of the Corresponding State Variable
erc20-totalsupply-change-state	`totalSupply` Does Not Change the Contract's State
erc20-transferfrom-fail-recipient-overflow	`transferFrom` Prevents Overflows in the Recipient's Balance
erc20-balanceof-succeed-always	`balanceOf` Always Succeeds
erc20-balanceof-correct-value	`balanceOf` Returns the Correct Value
erc20-balanceof-change-state	`balanceOf` Does Not Change the Contract's State
erc20-allowance-succeed-always	`allowance` Always Succeeds
erc20-allowance-correct-value	`allowance` Returns Correct Value
erc20-allowance-change-state	`allowance` Does Not Change the Contract's State

Property Name	Title
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
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

In the remainder of this section, we list all contracts where model checking of at least one property was not successful. There are several reasons why this could happen:

- Model checking reports a counterexample that violates the property. Depending on the counterexample,this occurs if
  - The specification of the property is too generic and does not accurately capture the intended behavior of the smart contract. In that case, the counterexample does not indicate a problem in the underlying smart contract. We report such instances as being "inapplicable".
  - The property is applicable to the smart contract. In that case, the counterexample showcases a problem in the smart contract and a correspond finding is reported separately in the Findings section of this report. In the following tables, we report such instances as "invalid". The distinction between spurious and actual counterexamples is done manually by the auditors.
- The model checking result is inconclusive. Such a result does not indicate a problem in the underlying smart contract. An inconclusive result may occur if
  - The model checking engine fails to construct a proof. This can happen if the logical deductions necessary are beyond the capabilities of the automated reasoning tool. It is a technical limitation of all proof engines and cannot be avoided in general.
  - The model checking engine runs out of time or memory and did not produce a result. This can happen if automatic abstraction techniques are ineffective or of the state space is too big.

Detailed Results For Contract ERC20 (ERC20MinterPauser.sol) In Commit 0x1de2057d81aa91e3c9e65f7127d202f65e8f767d

Verification of ERC-20 Compliance

Detailed results for function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-revert-zero	True	
erc20-transfer-succeed-normal	True	
erc20-transfer-correct-amount	True	
erc20-transfer-succeed-self	True	
erc20-transfer-correct-amount-self	True	
erc20-transfer-change-state	True	
erc20-transfer-exceed-balance	True	
erc20-transfer-recipient-overflow	False	Context not considered
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-succeed-normal	True	
erc20-transferfrom-succeed-self	True	
erc20-transferfrom-correct-amount	True	
erc20-transferfrom-correct-amount-self	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-change-state	True	
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-false	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-fail-recipient-overflow	False	Context not considered

Detailed results for function `totalSupply`

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

Detailed Results For Contract ERC20PresetMinterPauser (ERC20MinterPauser.sol) In Commit 0x1de2057d81aa91e3c9e65f7127d202f65e8f767d

Verification of ERC-20 Compliance

Detailed results for function `transfer`

Property Name	Final Result	Remarks
erc20-transfer-revert-zero	True	
erc20-transfer-correct-amount-self	True	
erc20-transfer-correct-amount	True	
erc20-transfer-succeed-normal	False	Intended behavior
erc20-transfer-succeed-self	False	Intended behavior
erc20-transfer-exceed-balance	True	
erc20-transfer-change-state	True	
erc20-transfer-false	True	
erc20-transfer-never-return-false	True	
erc20-transfer-recipient-overflow	False	Context not considered

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-succeed-self	False	Intended behavior
erc20-transferfrom-succeed-normal	False	Intended behavior
erc20-transferfrom-correct-allowance	True	
erc20-transferfrom-change-state	True	
erc20-transferfrom-fail-exceed-balance	True	
erc20-transferfrom-fail-exceed-allowance	True	
erc20-transferfrom-false	True	
erc20-transferfrom-never-return-false	True	
erc20-transferfrom-fail-recipient-overflow	False	Context not considered

Detailed results for function `totalSupply`

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

Detailed results for function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-succeed-always	True	
erc20-balanceof-change-state	True	
erc20-balanceof-correct-value	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-false	True	
erc20-approve-never-return-false	True	
erc20-approve-change-state	True	

APPENDIX | ZEEBU

Finding Categories

Categories	Description
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.
Concurrency	Concurrency findings are about issues that cause unexpected or unsafe interleaving of code executions.
Access Control	Access Control findings are about security vulnerabilities that make protected assets unsafe.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.

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

Technical description

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.

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 of those functions. 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 as operations on the congruence classes arising from the bit-width of the underlying numeric 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 an ERC-20 token 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 definitions

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 steps. 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 to reason about the validity of 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 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 for ERC-20 function `transfer`

erc20-transfer-revert-zero

Function `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)))
```

erc20-transfer-succeed-normal

Function `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 <= type(uint256).max && _balances[to] >= 0
          && _balances[msg.sender] <= type(uint256).max)
          ==> <>(finished(contract.transfer(to, value), return)))
```

erc20-transfer-succeed-self

Function `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] <= type(uint256).max)
          ==> <>(finished(contract.transfer(to, value), return)))
```

erc20-transfer-correct-amount

Function `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 <= type(uint256).max
   && _balances[msg.sender] >= 0 && _balances[msg.sender] <= type(uint256).max)
      ==> <>(finished(contract.transfer(to, value), return
             ==> _balances[msg.sender] == old(_balances[msg.sender]) - value
                 && _balances[to] == old(_balances[to]) + value)))
```

erc20-transfer-correct-amount-self

Function `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] <= type(uint256).max)
     ==> <>(finished(contract.transfer(to, value), return
         ==> _balances[to] == old(_balances[to]))))
```

erc20-transfer-change-state

Function `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
      ==> (_totalSupply == old(_totalSupply) && _allowances == old(_allowances)
           && _balances[p1] == old(_balances[p1])  ))))
```

erc20-transfer-exceed-balance

Function `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 <= type(uint256).max)
   ==> <>(reverted(contract.transfer) || finished(contract.transfer(to, value),
          !return)))
```

erc20-transfer-recipient-overflow

Function `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 > type(uint256).max
   && _balances[to] >= 0 && _balances[to] <= type(uint256).max
   && _balances[msg.sender] <= type(uint256).max
   && value > 0 && value <= _balances[msg.sender])
   ==> <>(reverted(contract.transfer) || finished(contract.transfer(to, value),
          !return) || finished(contract.transfer(to, value), _balances[to]
                      > old(_balances[to]) + value - type(uint256).max - 1)))
```

erc20-transfer-false

If Function `transfer` Returns `false`, the Contract State Has Not Been 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]
   ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
                                    && _allowances == old(_allowances)  ))))
```

erc20-transfer-never-return-false

Function `transfe` Never Returns `false`.

The transfer function must never return `false` to signal a failure.

Specification:

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

Properties for ERC-20 function `transferFrom`

erc20-transferfrom-revert-from-zero

Function `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)))
```

erc20-transferfrom-revert-to-zero

Function `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)))
```

erc20-transferfrom-succeed-normal

Function `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 <= type(uint256).max
   && value >= 0 && _balances[to] >= 0 && _balances[from] >= 0
   && _balances[from] <= type(uint256).max
   && _allowances[from][msg.sender] >= 0
   && _allowances[from][msg.sender] <= type(uint256).max)
      ==> <>(finished(contract.transferFrom(from, to, value), return)))
```

erc20-transferfrom-succeed-self

Function `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] <= type(uint256).max
   && _allowances[from][msg.sender] <= type(uint256).max)
     ==> <>(finished(contract.transferFrom(from, to, value), return)))
```

erc20-transferfrom-correct-amount

Function `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] <= type(uint256).max
&& _balances[to] >= 0 && _balances[to] + value <= type(uint256).max)
   ==> <>(finished(contract.transferFrom(from, to, value), return
          ==> _balances[from] == old(_balances[from]) - value
              && _balances[to] == old(_balances[to] + value))))
```

erc20-transferfrom-correct-amount-self

Function `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 <= type(uint256).max && _balances[from] >= 0
   && _balances[from] <= type(uint256).max)
      ==> <>(finished(contract.transferFrom(from, to, value), return
             ==> _balances[from] == old(_balances[from]))))
```

erc20-transferfrom-correct-allowance

Function `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 <= type(uint256).max && _balances[from] >= 0
   && _balances[from] <= type(uint256).max && _balances[to] >= 0
   && _balances[to] <= type(uint256).max && _allowances[from][msg.sender] >= 0
   && _allowances[from][msg.sender] <= type(uint256).max)
      ==> <>(finished(contract.transferFrom(from, to, value), return
          ==> ((_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])
                               == type(uint256).max))))))
```

erc20-transferfrom-change-state

Function `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
     ==> (_totalSupply == old(_totalSupply) && _balances[p1] == old(_balances[p1])
         && _allowances[p2][p3] == old(_allowances[p2][p3])  ))))
```

erc20-transferfrom-fail-exceed-balance

Function `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] <= type(uint256).max)
       ==> <>(reverted(contract.transferFrom)
              || finished(contract.transferFrom, !return)))
```

erc20-transferfrom-fail-exceed-allowance

Function `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), value > _allowances[from]
[msg.sender]
     && _allowances[from][msg.sender] >= 0 && value <= type(uint256).max)
     ==> <>(reverted(contract.transferFrom)
           || finished(contract.transferFrom(from, to, value), !return)
           || finished(contract.transferFrom(from, to, value), return
              && (msg.sender == from
                  || _allowances[from][msg.sender] == type(uint256).max))))
```

erc20-transferfrom-fail-recipient-overflow

Function `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 > type(uint256).max && value <= type(uint256).max
     && _balances[to] >= 0 && _balances[to] <= type(uint256).max)
        ==> <>(reverted(contract.transferFrom)
            || finished(contract.transferFrom(from, to, value), !return)
            || finished(contract.transferFrom(from, to, value), _balances[to]
                > old(_balances[to]) + value - type(uint256).max - 1)))
```

erc20-transferfrom-false

If Function `transferFrom` Returns `false`, the Contract's State Has Not Been Changed.

If `transferFrom` returns `false` to signal a failure, it must undo all incurred state changes before returning to the caller.

Specification:

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

erc20-transferfrom-never-return-false

Function `transferFrom` Never Returns `false`.

The `transferFrom` function must never return `false`.

Specification:

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

Properties related to function `totalSupply`

erc20-totalsupply-succeed-always

Function `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

Function `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

Function `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)  )))
```

Properties related to function `balanceOf`

erc20-balanceof-succeed-always

Function `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

Function `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

Function `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)  )))
```

Properties related to function `allowance`

erc20-allowance-succeed-always

Function `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

Function `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

Function `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)  )))
```

Properties related to function `approve`

erc20-approve-revert-zero

Function `approve` Prevents Giving 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)))
```

erc20-approve-succeed-normal

Function `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)))
```

erc20-approve-correct-amount

Function `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 <= type(uint256).max)
       ==> <>(finished(contract.approve(spender, value), return
              ==> _allowances[msg.sender][spender] == value)))
```

erc20-approve-change-state

Function `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
              ==> _totalSupply == old(_totalSupply) && _balances == old(_balances)
                  && _allowances[p1][p2] == old(_allowances[p1][p2])  )))
```

erc20-approve-false

If Function `approve` Returns `false`, the Contract's State Has Not Been Changed.

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
          ==> (_balances == old(_balances) && _totalSupply == old(_totalSupply)
               && _allowances == old(_allowances)  ))))
```

erc20-approve-never-return-false

Function `approve` Never Returns `false`.

The function `approve` must never returns `false`.

Specification:

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

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.