Token Conformance Audit Report

Oveit Token

Delivered: 2024-09-20 **Updated:** 2024-10-07

Prepared for Oveit Token, Inc. by Runtime Verification, Inc.

runtime verification

Summary

Update: On September 27th, the client presented a new OV8 ERC20 smart contract implementation. As we reviewed the updated code, we provided feedback and suggestions for possible optimizations to the Oveit Token team. Finally, on October 7th, the Oveit Token team presented the final version of the code in the codemelt-dev/oveit-token GitHub repository. As of this report update, the contract has not yet been deployed. The new smart contract addressed all the findings in their respective report sections. The ERC20 base contract has been updated to the latest version available in the OpenZeppelin repository and uses custom errors imported from the IERC20Errors interface. The updated version now uses the `Ownable2Step` module from OpenZeppelin, and the `Ownable` and `Pausable` modules have been updated to their latest version available from the OpenZeppelin repository. Finally, the pragma version has been bumped to `0.8.27`.

Runtime Verification, Inc. has audited the smart contract source code for the Oveit ERC20 OV8 token. The review was conducted from September 16 to 19, 2024.

Oveit Token engaged Runtime Verification to check the security of its ERC20 token project. Oveit is a technology company specializing in event management and ticketing solutions. It provides a platform that enables event organizers to manage registrations, sell tickets online, and control access seamlessly. The company is exploring using blockchain technology and NFTs to offer unique and secure transactions, reflecting its commitment to leveraging cutting-edge technologies in the event industry.

The identified issues are in the section Findings. Additional suggestions are included in the Informative Findings section.

Scope

The audited smart contract is `Oveit.sol`.

The review focused solely on the publicly available code deployed on the Ethereum chain at `0x49c953968a1c97fb71789f215c885d3ad23A3c30`, which was verified by Etherscan to match the bytecode deployed on-chain.

During this audit, we assumed that the address assigned to the `OWNER` role is trusted as long as it holds that role. Although we generally assume honesty and competence in such actors, our approach is focused on identifying areas where contracts can be improved to prevent unintended mistakes wherever possible.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation using techniques such as symbolic execution and fuzzing. Second, we developed a property test suite compatible with both Kontrol and Foundry to ensure the desired properties are holding. Thirdly, we carefully checked if the code was vulnerable to known security issues and attack vectors. Finally, we presented our findings to the Oveit team, providing feedback and suggested development practices and design improvements.

This report describes the **intended** behavior and invariants of the contracts under review and then outlines issues we have found, both in the intended behavior and in the ways the code differs from it. We also highlight lesser concerns, deviations from best practice, and any other weaknesses we encounter. Finally, we also give an overview of the important security properties we proved during the course of the review.

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that significant risks are inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information provided by you or is otherwise publicly available, there are likely additional unknown risks that otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carry substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims that impact a large quantity of funds.

This audit is a technical review of the smart contract code for the Oveit token, with the primary focus on verifying adherence to the ERC20 standard and identifying potential vulnerabilities that could be exploited by malicious actors. The audit aims to ensure that the code operates as intended and to safeguard stakeholders from potential misuse or loss of funds. This audit does not include any assessment of Oveit's financials, legal obligations, or other non-technical matters.

Oveit: Contract Description and Properties

This section provides a high-level overview of the Oveit project and outlines the key invariants we expect it to always respect at the end of a contract interaction.

Overview

According to the project website, the Oveit token lies at the intersection of the different components within the Oveit ecosystem and acts as the backbone of the ecosystem's internal transactions and rewards program. The Oveit token delivers the core functionalities of the ecosystem for both B2B consumers and end-consumers of live experiences. It is intended to be used as a form of payment within the Oveit ecosystem for loyalty programs and rewards.

Oveit is an ERC20 token with a few extra properties. The token follows the ERC20 token standard and implements its functionality through the given implementation.

The Oveit token defines a single Ownership role to restrict access to certain functionalities using an Ownable implementation.

The Oveit contract has an additional Pausable module, which provides the functionality to pause and unpause certain operations. The `pause` mechanism is integrated into the token transfer functions via the `_beforeTokenTransfer` hook using the `whenNotPaused` modifier. This means that any function that involves `_beforeTokenTransfer` will revert when the contract is paused. The public functions affected by this mechanism are:

- `function transfer(address to, uint256 amount) external returns (bool);`
- `function transferFrom(address from, address to, uint256 amount) external returns (bool);`
- `function mint(address to, uint256 amount) public onlyOwner;`
- `function burn(uint256 amount) public virtual;`
- `function burnFrom(address account, uint256 amount) public virtual;`

In addition, the Pausable mechanism does not prevent users from calling `approve` once the contract has been paused, allowing them to revoke approvals in an emergency.

On-Chain State

The OV8 token has been deployed on the main chain during transaction `0x7e968a9b74fd46e669bdaaeda12899ce00901fb372110f9b024931ece4f5b375` at the contract address `0x49c953968a1c97fb71789f215c885d3ad23A3c30` by the address `0x761A3b702e0d5A557e80C75F33293C8Cd9F98b17`. As of the audit date, the contract is unpaused, and the current owner is `0x761A3b702e0d5A557e80C75F33293C8Cd9F98b17`. The total supply stands at `100000000000000000` tokens, which are split between the

`0x761A3b702e0d5A557e80C75F33293C8Cd9F98b17` address that holds `99943572184900000` tokens and the `0xf1a94bea98d26adf4bdf7ed1e749baf45e997993` address that holds `56427815100000` tokens.

As security auditors, we must emphasize that as long as the contract's ownership is not renounced, the `OWNER` can mint new tokens, pause the contract, and be susceptible to private key attacks that could put the contract at risk.

Properties

The contract should always satisfy several key properties, with those specific to the Oveit contract highlighted here. These are not the only ones, but they are fundamental for the correctness of the protocol and, as such, deserve special attention. We have verified that the following properties hold using Kontrol v1.0.25 — our open-source tool for formally verifying Ethereum smart contracts. A full list of properties checked using fuzzing and/or formal verification is available in the Appendix.

1. The total supply should always equal the sum of all the balances, regardless of the state changes produced by the contract's functions.
2. The `pause` function should only be accessible to the `OWNER` and when the contract is not paused. It should prevent tokens from being minted, burned, and transferred. A `Paused` event is emitted, and the only change in the contract storage is in the `_paused` bool field.
3. The `unpause` function should only be accessible to the `OWNER` when the contract is paused. It should unlock all contract functionalities and emit an `Unpaused` event. The only change in the contract storage is in the `_paused` bool field.
4. The `mint` function should only be accessible to the `OWNER` when the contract is not paused. Then, it correctly increases the `totalSupply` and the `balance` of the token recipient provided that the token recipient is not the `0` address, no overflow occurs in both variables, and the contract is not `paused`. A `Transfer` event is emitted, and no other storage slots are modified. The `burn` function allows users to burn any amount of their tokens, provided their `balance` is sufficient, and the contract has not been paused. totalSupply and the sender's balance are updated accordingly. No other storage slots are modified. A `Transfer` event is emitted.
5. The `burnFrom` function allows any `spender` to burn an `account`'s tokens if and only if the `spender` has enough allowance provided by the `account`, the `account` has sufficient balance, both spender and `account` addresses are not the `0` address, and the contract is not in the `paused` state. A `Transfer` event is emitted, and the user balances, allowances, and `totalSupply` are correctly updated.
6. The `transfer` function allows users to transfer any amount of their tokens to any `receiver` (including the sender themselves), provided that the sender's `balance` is

sufficient, the `receiver` is not the `0` address, the contract has not been paused, and no overflow occurs during the transfer. `balance`s of sender and receiver are updated accordingly. A `Transfer` event is emitted, and the function returns `true`. No other storage slots are modified. If any of these conditions are violated, the function execution should revert.

7. The `approve` function allows any `owner` to approve any token amount to any `spender`, provided the `spender` is not the `0` address. The `spender`'s `allowance` is updated accordingly. Additionally, an `Approval` event is emitted, and the function returns `true`. No other storage slots are modified. If any of these conditions are violated, the function execution should revert.

8. The `transferFrom` function allows any `spender` to transfer any amount of `owner`'s tokens to any `receiver` (including the `spender` themselves), provided that the `owner`'s `balance` is sufficient, the `allowance` provided by the `owner` to the `spender` is sufficient, the `receiver` is not the `0` address, the contract has not been paused, and no overflow occurs during the transfer to the `receiver`, allowance of `allowance` is sufficient, the balance of `owner` and `receiver` is updated accordingly, and no other storage slots are modified. A `Transfer` event is emitted, and the function returns `true.` If any of these conditions are violated, the function execution should revert.

9. When called by the current `OWNER`, the `renounceOwnership` function sets the `PENDING_OWNER` and `OWNER` roles to the `0` address. The contract is not paused, and no other storage slots are modified.

10. When called by the current `OWNER`, the `transferOwnership` function sets the `PENDING_OWNER` role to the specified address if it is not the `0` address. The contract is not paused, and no other storage slots are modified.

11. The `increaseApproval` function allows any `owner` to increase the token `allowance` granted to any `spender` by a specified amount, provided that the `spender` is not the `0` address and the resulting `allowance` does not overflow. The `spender`'s `allowance` is updated accordingly. Additionally, an `Approval` event is emitted, and the function returns `true`. No other storage slots are modified. If any of these conditions are violated, the function execution should revert.

12. The `decreaseApproval` function allows any owner to decrease the token allowance granted to any spender by a specified amount, provided that the spender is not the `0` address and the current allowance is sufficient for the decrease. The spender's allowance is updated accordingly. Additionally, an `Approval` event is emitted, and the function returns `true`. No other storage slots are modified. If any of these conditions are violated, the function execution should revert.

Findings

A01: ERC20 approve function race condition vulnerability

[Severity: Medium | Difficulty: Low | Category: Design]

In the ERC20 token implementation, the `approve` function allows a token holder (the owner) to grant permission to a spender to withdraw up to a certain number of tokens from the owner's account. The typical approve function looks like this:

```
function approve(address spender, uint256 amount) public virtual override
returns (bool) {
       address owner = _msgSender();
       _approve(owner, spender, amount);
       return true;
}
```

The function is susceptible to a race condition that can lead to double-spending or unauthorized token transfers. The vulnerability arises when changing an existing non-zero allowance to a new value without setting it to zero.

Scenario

1. Owner A has `approved` Spender B to spend 100 tokens.
2. Owner A decides to increase the allowance to 200 tokens, and calls `approve(B, 200)`.
3. Spender B notices the pending transaction to change the allowance.
4. Before Owner A's new `approve` transaction is mined, Spender B uses the current allowance of 100 tokens by calling `transferFrom(A, B, 100)`.
5. Spender B's `transferFrom` transaction is mined first, transferring 100 tokens.
6. Owner A's `approve` transaction is mined next, resetting the allowance to 200 tokens.
7. Spender B now has an updated allowance of 200 tokens and can transfer 200 tokens, resulting in 300 tokens transferred instead of the intended 200 tokens.

Recommendation

To mitigate the race condition in the `approve` function, it is encouraged to use the `increaseAllowance` and `decreaseAllowance` functions already implemented in the codebase, given that the contract is already deployed on the chain and updating the `approve` function would require the contract to be re-deployed.

Additionally, the `approve` function can be changed to require that the allowance be set to zero before changing it to a new non-zero value. This forces a two-step process that reduces the risk of race conditions.

Status

The client acknowledged and addressed this issue. In commit [ddc23614adb23d14ee81edd4faed388b4edb87ef](), the OV8 token team updated the base ERC20 contract to the latest version available in the [OpenZeppelin GitHub repository](). The team retained the implementation of `increaseAllowance` and `decreaseAllowance` functions. Additionally, the client addressed the possible race condition by adding a check in the public `approve(address,uint256)` function to enforce a two-step approval process. This new check does not affect the `transferFrom` or `burnFrom` functions, which update the `_allowances` mapping internally. Finally, a new custom error, `ERC20WrongApprovalUsage,` has been added to the `IERC20Errors` interface to indicate an invalid approve call.

A02: renounceOwnership when the contract is paused

[Severity: Critical | Difficulty: High | Category: Security]

As the contract uses both `Ownable` and `Pausable` modules, there's a critical issue where the contract can become permanently locked if the owner renounces ownership while the contract is paused.

The contract becomes unusable for any function that requires the contract to be unpaused. Token holders cannot transfer or interact with their tokens.

Scenario

1. The owner calls the `pause()` function, which pauses the contract. While paused, all functions protected by the `whenNotPaused` modifier are disabled.
2. The owner calls `renounceOwnership`, setting the current owner to the `0` address. Since the contract doesn't have the `whenNotPaused` modifier, it can be called even when it is paused.
3. Since the `unpause()` function is restricted by the `onlyOwner` modifier, there is no longer an owner to call it.

The contract becomes unusable for any function that requires the contract to be unpaused. Consequently, token holders cannot transfer or interact with their tokens.

Recommendation

To mitigate this, the `renounceOwnership` function could be overridden in the main Oveit contract to include the `whenNotPaused` modifier. Given that the contract is already deployed and this scenario is quite specific, the owner can alternatively mitigate the risk by thoroughly verifying that the contract is not paused before proceeding to renounce ownership.

Status

The client acknowledged and addressed this issue in commit ddc23614adb23d14ee81edd4faed388b4edb87ef. The `renounceOwnership` function uses both `whenNotPaused` and `onlyOwner` modifiers in the new implementation.

Informative findings

B01: transferOwnership to invalid or non-existing addresses

[Severity: High | Difficulty: Low | Category: Security]

In smart contracts utilizing the Ownable pattern, the current owner can transfer the ownership to a new address using `transferOwnership`.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
       require(newOwner != address(0), "Ownable: new owner is the zero
address");
       _transferOwnership(newOwner);
       }

function _transferOwnership(address newOwner) internal virtual {
       address oldOwner = _owner;
       _owner = newOwner;
       emit OwnershipTransferred(oldOwner, newOwner);
       }
```

However, the transfer of ownership is made in a single-step operation in the `_owner = newOwner;` statement. While the current implementation prevents transferring ownership to the `0` address, it could be permanently lost if the function is called with a non-existent or mistyped address argument.

In addition, this function is available as long as the ownership is not renounced.

Recommendation

Implement a two-step ownership transfer process like OpenZeppelin's Ownable2Step, where the new owner must explicitly accept ownership. This ensures that the new `OWNER` address is valid.

Status

The client acknowledged and addressed this issue in commit ddc23614adb23d14ee81edd4faed388b4edb87ef. The new OV8 token now uses the `Ownable2Step` module, which has been bumped to the latest available version at the OpenZeppelin GitHub repository.

B02: Custom decimals could cause compatibility issues with external actors

[Severity: Low | Difficulty: Low | Category: Misc]

The Oveit contract has 8 decimals in the current implementation. This is not an issue by itself and adheres to the ERC20 standard. However, there is a risk of precision loss or incorrect display of tokens because other platforms integrating Oveit might incorrectly assume that the token uses a common value of 18 decimals.

Status

The client acknowledged and addressed this issue in ddc23614adb23d14ee81edd4faed388b4edb87ef. The new token implementation has a default value of 18 decimals.

Appendix: Properties Checked with Fuzzing and Symbolic Execution

This Appendix contains a table of all properties checked against the Oveit contract using fuzzing or symbolic execution. These have been extracted from the report generated by ERCx. Properties confirmed to pass are marked with a ✔️ symbol. Those confirmed to fail are indicated with a ❌ symbol. Properties not executed are marked with a ➖ symbol; please note that these were not tested due to the limited time available for the audit.

Property	Fuzz	SE
All functions and events required by ERC20 standard are present and conformant.	✔️	➖
For any sender that is the OWNER of the contract, if the contract has been paused, renounceOwnership should not change the OWNER address, since it would lead to the contract getting locked in the paused state.	➖	✔️
The approve function is designed to prevent race conditions or double-spend issues, ensuring that the spender cannot use the current allowance before it is properly updated.	➖	✔️
A tokenReceiver SHOULD NOT be able to call transferFrom of an amount more than her allowance from the tokenSender even if the tokenSender's balance is more than or equal to the said amount.	✔️	✔️
A tokenReceiver SHOULD NOT be able to call transferFrom for an amount greater than the tokenSender's balance, even if the tokenReceiver's allowance from the tokenSender exceeds the tokenSender's balance.	✔️	✔️
A user SHOULD NOT be able to call a transfer for an amount more than his balance.	✔️	✔️
A tokenReceiver SHOULD NOT be able to call transferFrom of any positive amount from a tokenSender if the tokenSender did not approve the tokenReceiver previously.	✔️	✔️
A msg.sender SHOULD NOT be able to call transferFrom of any positive amount from his/her own account to any tokenReceiver if the msg.sender did not approve him/herself prior to the call.	✔️	✔️
A successful approve call of zero or positive amount MUST emit the Approval event correctly.	✔️	✔️
After a tokenApprover approves a tokenApprovee some positive amount via an approve call, zero or any positive amount up to the said amount MUST be transferable by tokenApprovee via a transferFrom call, provided a sufficient	✔️	✔️

balance of tokenApprover.		
Zero or positive approved amount MUST be reflected in the allowance correctly.	✔	✔
A successful transfer call of a positive amount MUST emit the Transfer event correctly.	✔	✔
A successful transferFrom call of a positive amount MUST emit Transfer event correctly.	✔	✔
A successful transferFrom call of zero amount by any user other than the tokenSender MUST be possible, from and to the same account.	✔	✔
A successful transferFrom call of zero amount by any user other than the tokenSender MUST be possible, from and to different addresses.	✔	✔
A successful transferFrom of zero amount by any user other than the tokenSender, from and to the same address, MUST emit a Transfer event correctly.	✔	✔
A successful transferFrom call of zero amount by any user other than the tokenSender to the tokenSender MUST be possible.	✔	✔
A successful transferFrom call of zero amount by the tokenSender herself MUST be possible.	✔	✔
A successful transferFrom call of zero amount by the tokenSender herself to herself MUST emit a Transfer event correctly.	✔	✔
A successful transferFrom call of zero amount by the tokenSender herself to someone MUST emit a Transfer event correctly.	✔	✔
A successful transfer call of zero amount to another account MUST emit the Transfer event correctly.	✔	✔
A successful transfer call of zero amount to another account MUST be possible.	✔	✔
A successful transfer call of zero amount to self MUST emit the Transfer event correctly.	✔	✔
The zero address SHOULD NOT have any token from the contract.	✔	➖
Contract owner cannot control balance by using minting overflow (via mint(address,uint256).	✔	✔
A msg.sender SHOULD be able to retrieve his/her own balance.	✔	➖
A msg.sender SHOULD be able to retrieve the balance of an address different from his/hers.	✔	➖
Allowance should be updated correctly after burning via burnFrom(address,uint256).	✔	✔

User balance should be updated correctly after burning (via burnFrom(address,uint256) or burn(address,uint256)). The balance of the tokenApprover is decreased by the burned amount. The balance of the tokenApprovee (performing the burn) is left unchanged.	✔	✔
Total supply should be updated correctly after burning (via burnFrom(address,uint256) or burn(address,uint256)).	✔	✔
Allowance should not change after burning zero token (via burnFrom(address,uint256) or burn(address,uint256)).	✔	✔
User balance should not change after burning zero token (via burnFrom(address,uint256) or burn(address,uint256)).	✔	✔
Total supply should not be updated after burning zero token (via burnFrom(address,uint256) or burn(address,uint256)).	✔	✔
User balance should be updated correctly after burning (via burn(uint256) or burnToken(uint256)).	✔	✔
Total supply should be updated correctly after burning (via burn(uint256) or burnToken(uint256)).	✔	✔
User balance should not change when burning zero token (via burn(uint256) or burnToken(uint256)).	✔	✔
Total supply should not change after burning zero tokens (via burn(uint256) or burnToken(uint256)).	✔	✔
A successful call of approve of any amount to the zero address SHOULD NOT be allowed.	✔	✔
If consecutive calls of approve function of positive-to-positive amounts can be called, then the allowance is set to the right amount after the second call.	✔	✔
Consecutive calls of approve function of positive-to-zero amounts CAN be called and the allowance is set to the right amount after the second call.	✔	✔
Consecutive calls of approve function of zero-to-positive amounts CAN be called and the allowance is set to the right amount after the second call.	✔	✔
Consecutive calls of approve function of zero-to-zero amounts CAN be called and the allowance is set to the right amount after the second call.	✔	✔
A successful decreaseAllowance call MUST decrease the allowance of the tokenApprovee correctly.	✔	✔
A decreaseAllowance call will REVERT if there's not enough allowance to decrease.	✔	✔
The decreaseAllowance function DOES NOT ALLOW allowance to be double-spent, i.e., Alice CAN decrease her allowance for Bob by the correct amount even if Bob tries to front-run her by calling transferFrom call right before	✔	➖

her call.		
The transferFrom function DOES NOT have the potential to take fees.	✔	✔
The transferFrom function DOES NOT take fees at test execution time.	✔	✔
The transfer function DOES NOT have the potential to take fees.	✔	✔
The transfer function DOES NOT take fees at test execution time.	✔	✔
A successful increaseAllowance call MUST increase the allowance of the tokenApprovee correctly.	✔	✔
The increaseAllowance function DOES NOT ALLOW allowance to be double-spent, i.e., Alice CAN increase her allowance for Bob by the correct amount even if Bob tries to front-run her by calling transferFrom call right before her call.	✔	➖
Minting to the zero address should not be possible (via mint(address,uint256)).	✔	✔
User balance should be updated correctly after minting (via mint(address,uint256)).	✔	✔
Minting zero tokens (via mint(address,uint256)) should not change the total supply.	✔	✔
Total supply should be updated correctly after minting (via mint(address,uint256)).	✔	✔
Minting zero token (via mint(address,uint256)) should not change the balance of the account target of the mint.	✔	✔
Multiple calls of transferFrom SHOULD NOT be allowed once allowance reaches zero even if the tokenSender's balance is more than the allowance.	✔	➖
After a tokenApprover decreases the allowance of a tokenApprovee by some positive amount via a decreaseAllowance call, any amount up to the decreased allowance MUST be transferable by tokenApprovee, provided a sufficient balance of tokenApprover.	✔	➖
After a tokenApprover decreases the allowance of a tokenApprovee by some positive amount via a decreaseAllowance call, zero amount MUST be transferable by tokenApprovee.	✔	➖
A successful decreaseAllowance call of zero or positive amount MUST emit the Approval event correctly.	✔	✔
After a tokenApprover increases the allowance of a tokenApprovee by some positive amount via an increaseAllowance call, any amount up to the increased allowance MUST be transferable by tokenApprovee, provided a sufficient balance of tokenApprover.	✔	✔
After a tokenApprover increases the allowance of a tokenApprovee by some	✔	➖

positive amount via an increaseAllowance call, zero amount MUST be transferable by tokenApprovee, provided a sufficient balance of tokenApprover.		
A successful increaseAllowance call of zero or positive amount MUST emit the Approval event correctly.	✔️	✔️
Multiple transfer calls of positive amounts are ALLOWED given that the sum of the transferred amounts is less than or equal to the tokenSender's balance.	✔️	➖
Multiple transferFrom calls of positive amounts are ALLOWED given that the sum of the transferred amounts is less than or equal to the tokenSender's balance and approvals are given by the tokenSender.	✔️	➖
Self approval of zero or positive amount is ALLOWED and the allowance is correctly updated.	✔️	✔️
Self approval and call of transferFrom from its own account of positive amount is ALLOWED.	✔️	✔️
Self decreaseAllowance call of positive amount is ALLOWED.	✔️	✔️
Self decreaseAllowance call of zero or positive amount, followed by a transferFrom call of an amount up to the decreased allowance for herself is ALLOWED.	✔️	➖
Self increaseAllowance call of positive amount is ALLOWED.	✔️	✔️
Self increaseAllowance call, followed by a transferFrom call of some positive amount is ALLOWED.	✔️	➖
Self transfer call of positive amount is ALLOWED and SHOULD NOT modify the balance.	✔️	✔️
A tokenReceiver CAN call transferFrom of the tokenSender's total balance amount given that tokenSender has approved that.	✔️	✔️
A msg.sender CAN call transfer of her total balance amount to a tokenReceiver and the balances are modified accordingly.	✔️	✔️
A transferFrom call of zero or any positive amount to the zero address SHOULD revert.	✔️	✔️
A transfer call of zero or any positive amount to the zero address SHOULD revert.	✔️	✔️
The contract's totalSupply variable SHOULD NOT be altered after transfer is called.	✔️	✔️
The contract's totalSupply variable SHOULD NOT be altered after transferFrom is called.	✔️	✔️
The value of variable totalSupply, if it represents the sum of all tokens in the contract should not overflow when the sum of tokens changes via minting (via	✔️	✔️

mint(address,uint256) or mintToken(address,uint256) or issue(address,uint256)).		
Transfer should not be possible when token is paused.	✔	✔
A successful call of transfer DOES NOT update the balance of users who are neither the tokenSender nor the tokenReceiver.	✔	✔
TransferFrom should not be possible when token is paused.	✔	✔
A successful transferFrom of any positive amount MUST decrease the allowance of the tokenSender by the transferred amount.	✔	✔
A successful call of transferFrom DOES NOT update the balance of users who are neither the tokenSender nor the tokenReceiver.	✔	✔
An approve call of any positive amount SHOULD revert if the tokenSender is the zero address.	✔	✔
A transfer call of any positive amount SHOULD revert if the tokenSender is the zero address.	✔	✔
Multiple calls of transfer of zero amount are ALLOWED.	✔	✔
Multiple calls of transferFrom of zero amount are ALLOWED.	✔	✔
Self approval and call of transferFrom from its own account of zero amount is ALLOWED.	✔	✔
Self decreaseAllowance call of zero amount is ALLOWED.	✔	✔
Self increaseAllowance call of zero amount is ALLOWED.	✔	✔
Self increaseAllowance call, followed by a transferFrom call of zero amount is ALLOWED.	✔	➖
Self transfer call of zero amount is ALLOWED and SHOULD NOT modify the balance.	✔	✔
A tokenReceiver CAN call transferFrom of the tokenSender's total balance amount of zero.	✔	✔
A msg.sender CAN call transfer of her total balance of zero to a tokenReceiver and the balances are not modified.	✔	✔
Only OWNER can pause and unpause contract.	➖	✔
Only OWNER can renounce and transfer ownership (via renounceOwnership() and transferOwnership(address)).	➖	✔