PUBLIC

Code Assessment

of the SUPTB Smart Contracts

November 06, 2023

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Compound with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of SUPTB according to Scope to support you in forming an opinion on their security risks.

Compound implements an EIP-7246 (under review) compliant token SUPTB (Superstate short-term U.S. government bonds) and a permission list contract. It introduces a new feature: Encumbrance on top of ERC-20 to separate the ownership of tokens from the right to transfer them.

The most critical subjects covered in our audit are functional correctness, access control and standard compliance. Security regarding standard compliance is high. Security regarding access control has been improved since the first iteration of this report (see permission can be bypassed in transferFrom()). Additionally, a critical issue allowing users to spend encumbrance of other users in certain cases has been disclosed and fixed by Compound after the first iteration of this report: Encumbered balances can be transferred. Functional correctness is now extensive.

The general subjects covered are code complexity and quality of specification documentation. Some inconsistency has been identified in the specifications, see Incorrect specs, which was corrected.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	0
Low-Severity Findings	1
• Code Corrected	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the SUPTB repository based on the documentation files.

The following files were considered in scope:

1. src/interfaces/IERC7246.sol
2. src/PermissionList.sol
3. src/SUPTB.sol

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	07 September 2023	ecd1718746f329414e22aa238b1cb164875f6860	Initial Version
2	14 September 2023	cc1cc7410d954e3fa2cf93dd26f1e673a24d0e97	Second Version
3	19 September 2023	cd80ca6b28ed3e4868cbc1b6bc927d7136563efe	Third Version
4	20 October 2023	83e22292666753841a31b1f9498ca30209dfbb6e	Fourth Version
5	06 November 2023	96de27d108759c0bf359ba6251ed6935ee79939e	Fifth Version

For the solidity smart contracts, the compiler version `0.8.20` was chosen.

2.1.1 Excluded from scope

Any other files not explicitly mentioned in the scope section. In particular tests, scripts, external dependencies, and configuration files are not part of the audit scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Compound offers an EIP-7246 (under review) compliant token SUPTB (Superstate short-term U.S. government bonds) with a permission list contract. Both of them will be used in an upgradable way.

2.2.1 SUPTB

SUPTB is an ERC-20 compliant token with 6 decimals built on top of the EIP-7246 functionalities, introducing the concept of Encumbrance. Encumbrance separates the ownership of tokens from the right to transfer them. In contrast to allowance, the semantics of encumbrance is stronger and guarantees that a certain amount of tokens are available whenever needed. This enables token holders to commit to terms within a smart contract without transferring ownership of their tokens to an external address in a non-custodial approach.

The encumbrance mechanism is achieved by three main actions: `encumber()`, `encumberFrom()`, and `release()`.

1. `encumber()` - The `msg.sender` can lock a specific amount of tokens to a `taker`. The `taker` acquires the right to pull these tokens in the future by `transferFrom()`. Besides, it also prevents any other action from reducing the available balance of `msg.sender` below the encumbered amount.
2. `encumberFrom()` - By consuming an allowance, a `msg.sender` can encumber tokens on behalf of the owner to a taker.
3. `release()` - Encumbrance can, in comparison to allowance, not be revoked by the owner. Instead, the taker can voluntarily relinquish it and free the locked funds to the owner.

`transferFrom()` - Funds are transferred by consuming encumbrance before allowance.

A typical interaction flow between users, third party protocols, and EIP-7246 compliant tokens would be:

1. Users give allowance to protocols by `approve()`.
2. Protocols invoke `encumberFrom()` instead of `transferFrom()`.
3. When the protocol would ordinarily transfer tokens back to the user, it would call `release()`.

Given that a user's total balance consists of an idle amount and an encumbrance in this mechanism, the function `balanceOf()` loses its original semantics. Thus, a view function `availableBalanceOf()` is added to reveal the idle balance excluding the encumbrance. And it is checked in `transfer()`, `transferFrom()`, `encumber()`, and `encumberFrom()` to avoid spending any amount that is locked by the previous encumbrance. Above all, any protocols that interact with this token should query `availableBalanceOf()` to retrieve the idle balance instead of using `balanceOf()`.

This contract is governed by an `admin`, which has the privilege to call `mint()`, `burn()`, `pause()`, and `unpause()`. Specifically, the `admin` can only mint to addresses that are allowed by the `PermissionList` contract. However, it can burn tokens from any address. Once `pause()` is called, the main entrypoints are frozen and no token transfers can be made.

2.2.2 PermissionList

A PermissionList is provided to achieve permissioned token transfers. It has a `permissions` mapping, which stores several permission flags towards an address. The contact is governed by the `permissionAdmin` with the privilege to set the permission flags for users.

The permissionList will be checked upon `transfer()`, `transferFrom()`, `encumber()`, `encumberFrom()`, and `mint()`.

2.2.3 Changes in Version 4

SUPTB can now be paused in two different ways: The normal functionality (`pause()` / `unpause()`) pauses transfers and encumbrances between users. The newly introduced functionality (`accountingPause()` / `acountingUnpause()`) pauses minting and burning of the tokens. Additionally, the contract exposes `entityMaxBalance()` that indicates the maximum balance (not available balance) a single entity can hold (relative to the total supply. Note that the max entity balance is enforced **off-chain**. The `whenNotPaused` modifier on `release()` has been lifted, therefore, releasing encumbrance is no longer subject to the pause.

`PermissionList` has introduced a mapping `addressEntityIds`, which is used in conjunction with the `permissions` mapping to allow for editing the permissions of multiple addresses that belong to the same entity at once.

2.2.4 Roles and Trust Model

The token's `admin` and the `permissionAdmin` are assumed to never act maliciously. `permissionAdmin` can block transfers of any users and `admin` can `mint` or `burn` any amount of token to desired addresses or pause the whole contract.

Both contracts are deployed via proxy and their implementations can therefore change at any time.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	1

- Encumbered Balances Can Be Transferred Code Corrected

High-Severity Findings	1

- Permission Can Be Bypassed in transferFrom() Code Corrected

Medium-Severity Findings	0

Low-Severity Findings	1

- Unpaused Approve Code Corrected

Informational Findings	6

- No Caller in Burn Event Code Corrected
- Gas Optimizations Code Corrected
- Incorrect Specs Specification Changed
- Irregular Naming Code Corrected
- PausableUpgradeable Is Not Initialized Code Corrected
- Unreachable Branch Code Corrected

6.1 Encumbered Balances Can Be Transferred

Correctness Critical Version 1 Code Corrected

CS-COMPSPT-007

`transferFrom()` allows a user to spend tokens encumbered to them from a given `src` address and then, if there are still more tokens to be transferred, spend additional allowance of the same `src`:

```
uint256 excessAmount = amount - encumberedToTaker;

_releaseEncumbrance(src, msg.sender, encumberedToTaker);

if (availableBalanceOf(src) < excessAmount) revert InsufficientAvailableBalance();

_spendAllowance(src, msg.sender, excessAmount);
```

In this case, the encumbrance is first released and then the available balance of the `src` is checked. This is problematic because `availableBalanceOf()` relies on the `encumberedBalanceOf` variable which has just been decreased in `_releaseEncumbrance()`. The result is that if `src` has given encumbrance to more than one address (and additional allowance to the `msg.sender` that is greater than the available balance of `src`), the `msg.sender` is able to spend its allowance on an amount that is encumbered to someone else.

Note: This issue has been disclosed by Compound after the first intermediate report has been released.

Code corrected:

The `availableBalanceOf()` check is now performed before the encumbrance of the `msg.sender` is released. The `excessAmount` can no longer be higher than the actual available balance.

6.2 Permission Can Be Bypassed in `transferFrom()`

Correctness | High | Version 1 | Code Corrected

CS-COMPSPT-008

`transferFrom()` checks the permissions of the `src` address only when there is nothing encumbered to the `msg.sender`:

```
uint256 encumberedToTaker = encumbrances[src][msg.sender];
if (encumberedToTaker == 0 && !permissionList.getPermission(src).isAllowed) {
    revert InsufficientPermissions();
}
```

Users could encumber 1 wei of the token to themselves and would then still be able to transfer all of their balance to another (permissioned) address by approving it first and then using `transferFrom()`. This is possible because the check above does not make sure that the `amount` being transferred is less or equal to `encumberedToTaker` before disabling the permission check.

Code corrected:

The check has been corrected to revert in case the encumbrance is insufficient and the transfer permission check failed. As a result, bypassing the permission check with a small non-zero encumbrance is no longer possible.

6.3 Unpaused Approve

Design | Low | Version 1 | Code Corrected

CS-COMPSPT-006

In case the admin pauses the contract, `permit()` will also be paused so that users cannot update their allowance by signatures. However, users can still use `approve()`, `increaseAllowance()`, and `decreaseAllowance()` to update the allowance, as these functions are not overridden with the `whenNotPaused` modifier.

Code corrected:

The modifier `whenNotPaused` has been removed from `permit()` since users will not be able to transfer in a paused contract anyways.

6.4 Gas Optimizations

Informational | Version 1 | Code Corrected

CS-COMPSPT-001

The following code parts can be further optimized to improve gas efficiency:

1. In case of using allowance in `transferFrom()`, the check towards available funds and the call of `_spendAllowance()` can be conducted before the call to `_releaseEncumbrance()` to reduce gas waste for reverting transactions.
2. Some subtractions can be done in an unchecked scope in case the order has already been checked before. For example:

```
if (amount > encumberedToTaker) {
    uint256 excessAmount = amount - encumberedToTaker;
    ...
}
```

3. `DOMAIN_SEPARATOR` can be cached as an `immutable` and only recomputed in runtime in case there is a fork.
4. Since permissions are often retrieved for two different addresses in one call, a bulk version of `PermissionList.getPermission()` could be implemented.

Code corrected:

The code has been corrected to check available funds first, see the fix to Encumbered balances can be transferred.

The computation of `excessAmount` has been wrapped into an unchecked scope since the order is already checked before.

6.5 Incorrect Specs

Informational | Version 1 | Specification Changed

CS-COMPSPT-002

A doc comment of the `transfer()` function states:

```
Includes extra functionality to burn tokens if ``dst`` is the contract address
```

This is not true as burning occurs when `dst == address(0)`.

Furthermore, a doc comment of the function `release()` states:

```
Spends all of the encumbrance if ``amount`` is greater than ``owner``'s
```

This is also not true as the function reverts in that case.

Specifications changed:

The specifications have been changed to reflect the actual function behaviors.

6.6 Irregular Naming

Informational | Version 1 | Code Corrected

CS-COMPSPT-009

The function `PermissionList.setPermissionAtIndex()` is defined as `internal` but the function name does not start with an underscore. This is in contrast to other functions in the contract such as `_requireAuthorized()`.

Code corrected:

The function has been renamed to `_setPermissionAtIndex()`.

6.7 No Caller in Burn Event

Informational | Version 1 | Code Corrected

CS-COMPSPT-003

The `Burn` event contains a `burner` address, while the `Mint` event contains a `minter` address. In case of the `Mint` event, the `minter` address is set to the address that performs the minting. In the `Burn` event however, `burner` is set to the address from which tokens are burned (equivalent to the `dst` address in the `Mint` event). The address that performs the burning is not emitted at all.

Code corrected:

`Burn` event has been updated to contain a `from` address where tokens are burned, and the `burner` now represents the address that performs the burning.

6.8 PausableUpgradeable Is Not Initialized

Informational | Version 1 | Code Corrected

CS-COMPSPT-004

`__Pausable_init()` is not invoked in `initialize()`. Though this is not necessarily required (because the function simply initializes a storage variable to the default value), it might be considered bad practice.

Code corrected:

The function `initialize()` has been updated to explicitly initialize `PausableUpgradeable` by calling `__Pausable_init()`.

6.9 Unreachable Branch

Informational | Version 1 | Code Corrected

CS-COMPSPT-005

`permit()` checks the returned boolean flag of `isValidSignature()` and reverts in case it is `false`. However, `isValidSignature()` always returns true in case it does not revert, thus the `else` branch in `permit()` can never be reached.

Code corrected:

The unreachable branch has been removed.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Encumbrance Can Bypass permissionList and Burning

Note Version 1

`PermissionList` can only block transfers of idle (i.e., unencumbered) funds. The `admin` can also only burn idle funds of a user. If a user encumbers all the funds to itself (any address is possible as takers are not subject to permissions), future un-whitelisting and burning attempts will fail.

The user won't be able to encumber any funds to another address though (contrary to the issue permission can be bypassed in transferFrom()).

7.2 `balanceOf()` Semantics

Note Version 1

EIP-7246 changes the semantics of the ERC-20 `balanceOf()` function. While one would typically expect from an ERC-20 token that the return value of the `balanceOf()` function is the full balance a user is able to spend, this is no longer true for EIP-7246 tokens.

This might be problematic for third-party integrations that expect the token to behave like a regular ERC-20 token.