Exactly EscrowedExa Audit

OPENZEPPELIN SECURITY | OCTOBER 27, 2023

Security Audits

Table of Contents

- Table of Contents
- Summary
- Scope
- System Overview
- Privileged Roles
- Trust Assumptions
- Critical Severity
  - Unrestricted Minting of esEXA Tokens
- Low Severity
  - Missing Docstrings
  - User May Commit To Vesting Terms They Did Not Anticipate
  - Lack of Input Validation
  - Incorrect Initialization of ERC20Permit
- Notes & Additional Information
  - Missing Named Parameters in Mapping
  - Lack of Security Contact
  - Unused Named Return Variable
  - Typographical Error
  - Improper Use of assert
  - Code Is Not Fully Consistent With Solidity Style Guide

Summary

Type
DeFi
Timeline
From 2023-10-02
To 2023-10-03
Languages
Solidity

Total Issues
11 (5 resolved)
Critical Severity Issues
1 (1 resolved)
High Severity Issues
0 (0 resolved)
Medium Severity Issues
0 (0 resolved)
Low Severity Issues
4 (3 resolved)
Notes & Additional Information
6 (1 resolved)

Scope

We audited the Exactly protocol repository at commit 1dfca20.

In scope was the following contract:

```
contracts
└── periphery
    └── EscrowedExa.sol
```

System Overview

The `EscrowedEXA` contract is an ERC-20 token contract that allows users to mint `esEXA` tokens in exchange for `EXA` tokens. The `esEXA` tokens are not transferrable and `esEXA` holders have the option to redeem these tokens for the underlying `EXA` tokens, subject to a vesting period configured by the protocol administrators. To initiate the vesting process, users must

Vesting is orchestrated through an external protocol called Sablier, which establishes a dedicated stream for the user's vesting, linearly releasing `EXA` tokens to the user over time. Users retain the ability to cancel the vesting at their discretion, triggering the withdrawal of all vested `EXA` tokens from Sablier. Any unvested tokens are returned to the `EscrowedEXA` contract, where corresponding `esEXA` tokens are minted and provided to the user.

This mechanism guarantees that the exclusive path for the users to retrieve locked `EXA` tokens from the `EscrowedEXA` contract is by adhering to the stipulated vesting schedule.

Privileged Roles

The `EscrowedEXA` contract implements the following privileged roles:

- The `DEFAULT_ADMIN_ROLE` can change the vesting period and modify the reserve ratio.
- The `TRANSFERRER_ROLE` is the only role that allows transferring `esEXA` tokens.
- The `REDEEMER_ROLE` has the authority to instantly exchange `esEXA` tokens for `EXA` tokens, bypassing the need for the vesting process.

Trust Assumptions

- The `EscrowedEXA` contract heavily relies on Sablier protocol to allow users to redeem the locked `EXA` tokens. The Sablier protocol is trusted to behave according to its specification.
- The holders of `DEFAULT_ADMIN_ROLE` and `REDEEMER_ROLE` are expected to be non-malicious and act in the protocol's best interest.

Unrestricted Minting of `esEXA` Tokens

The `EscrowedEXA` contract does not verify the origin or the asset of the Sablier stream expected to be canceled. Cancellation can happen through `EscrowedEXA`'s `cancel` function or directly through the Sablier protocol. Depending on what was set as `stream.sender` and `stream.recipient`, the `onStreamCanceled` hook will be called by Sablier's smart contracts. This leads to a scenario where an attacker can create a vesting stream directly through the Sablier protocol and then force its processing via the `EscrowedEXA` contract, resulting in the minting of `esEXA` tokens to the attacker, without the corresponding `EXA` tokens reaching the `EscrowedEXA` contract.

There are two exploitation scenarios that start with the attacker deploying a bogus ERC-20 token.

In the first scenario, a malicious stream is created with the attacker set as the sender and the `EscrowedEXA` contract set as the recipient. Upon the attacker canceling the stream directly through Sablier, the `onStreamCanceled` function is invoked, allowing the attacker to mint `esEXA` tokens. This scenario is also possible if the roles are reversed, with the attacker set as the recipient and the `EscrowedEXA` contract set as the sender.

The second scenario involves creating a stream with the `EscrowedEXA` contract set as the sender and the attacker as the recipient. Calling the `cancel` function of the `EscrowedEXA` contract results in the minting of `esEXA` tokens.

Consider adding a mapping that identifies streams created by the `EscrowedEXA` contract. Upon cancellation, verify the `streamId` in the `cancel` function and the `onStreamCanceled` hook, ensuring that only the streams created by the `EscrowedEXA` contract are processed successfully.

Update: *Resolved in pull request #672 at commit 0182437.*

Low Severity

Missing Docstrings

Within `EscrowedEXA.sol`, there are several parts that do not have docstrings. For instance:

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in pull request #676 at commit 9a29592.*

User May Commit To Vesting Terms They Did Not Anticipate

The `EscrowedEXA` contract enables users to initiate the vesting process through the `vest` function, allowing them to retrieve `EXA` tokens. To begin this process, users are required to provide a reserve amount of `EXA` tokens, which subsequently triggers the creation of a Sablier stream for the specified vesting period. It is important to note that the `reserveRatio` and `vestingPeriod` parameters can be modified by an administrative account at any time.

The issue arises from the uncertainty faced by users when initiating the vesting process, as they cannot be certain of whether the `reserveRatio` or `vestingPeriod` will remain the same when their transaction is included in a block. If the administrator modifies the `reserveRatio` or `vestingPeriod` parameters and their transaction is executed before the user's vesting, the user may inadvertently commit to vesting parameters that they did not anticipate.

Consider adding `expectedReserveRatio` and `expectedVestingPeriod` parameters to the `vest` function. If the values of these parameters differ from the actual `reserveRatio` and `vestingPeriod`, the transaction should revert, ensuring that users are not bound to vesting parameters they did not expect.

Update: *Resolved in pull request #673 at commit ef251d3.*

Lack of Input Validation

The `EscrowedEXA` contract implements two administrative functions that miss input validation:

- The `setVestingPeriod` function updates the vesting period. Consider adding a check to restrict it to reasonable values.

Update: *Acknowledged, not resolved. The Exactly team stated:*

> *We believe it is difficult to see what is a valid range for it beforehand which will not leave us too limited in the future. At the same time, the check could also bring a false sentiment of safety.*

Incorrect Initialization of `ERC20Permit`

In the OpenZeppelin Upgradeable contracts, unchained functions are used to mitigate potential double initialization problems. However, because of the simple inheritance structure in the `EscrowedEXA` contract, double initialization is not a concern in the current implementation.

Using `__ERC20Permit_init_unchained` instead of `__ERC20Permit_init` is not necessary, and leaves the `EIP712Upgradeable` contract's name and version uninitialized. This is not consistent with the EIP-712 domain separator specification, and will lead to issues when integrating with signatures that fully comply with EIP-712.

Consider using the `__ERC20Permit_init` function in order to correctly initialize the EIP-712 domain separator.

Update: *Resolved in pull request #674 at commit 1725765.*

Notes & Additional Information

Missing Named Parameters in Mapping

Since Solidity 0.8.18, developers can utilize named parameters in mappings. This means mappings can take the form of `mapping(KeyType KeyName? => ValueType ValueName?)`. This updated syntax provides a more transparent representation of the mapping's purpose.

Consider adding named parameters to the `reserves` mapping in the `EscrowedEXA.sol` contract to improve the readability and maintainability of the codebase.

Update: *Acknowledged, will resolve. The Exactly team stated:*

Lack of Security Contact

Providing a specific security contact, such as an email or ENS, within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice proves beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. Additionally, if the contract incorporates third-party libraries and a bug surfaces in these, it becomes easier for the creators of those libraries to make contact, inform the code owners about the problem, and provide mitigation instructions.

The `EscrowedEXA` contract does not have a security contact.

Consider adding a NatSpec comment with a security contact, on top of the contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the Openzeppelin Wizard and the ethereum-lists.

Update: *Acknowledged, not resolved. The Exactly team stated:*

> *We believe our channels and bug bounty programs are public enough if someone wants to reach us.*

Unused Named Return Variable

Named return variables are a way to declare variables that are meant to be used within a function's body for the purpose of being returned as the function's output. They are an alternative to explicit in-line `return` statements.

In `EscrowedEXA` contract, the `streamId` return variable for the `vest` function is unused.

Consider either using or removing any unused named return variables.

Update: *Acknowledged, not resolved. The Exactly team stated:*

> *This is a style choice that we believe makes the function more readable.*

Typographical Error

Update: *Resolved in pull request #675 at commit 61efaf9. The Exactly team stated:*

> *We removed the internal function where the NatSpec had a typographical error.*

Improper Use of `assert`

In the `EscrowedEXA` contract, there are several instances where `assert` is used to validate certain conditions: lines 68, 78, 88, 134, 149 and 170.

Generally, `assert` is used to test system invariants, while `require` is used to check return values or validate inputs. Moreover, if a transaction reverts because of `assert`, no leftover gas is returned to the caller, potentially resulting in a negative user experience.

Consider replacing the above occurrences of `assert` with `require`.

Update: *Acknowledged, not resolved. The Exactly team stated:*

> *This is not true, after solidity 0.8.* `assert` *no longer consumes all caller gas. It is also a style choice, and we are being consistent with the rest of the contracts in the protocol.*

Code Is Not Fully Consistent With Solidity Style Guide

There are several occurrences where the Solidity style guide is not followed, which makes the code more error-prone and difficult to read:

- The `withdrawMax` function is internal and its name should start with an underscore `_`.
- The order of functions within the contract should start with the `constructor`, followed by `external`, `public`, `internal`, and `private` functions at the end.
- Consider adjusting the layout of the `EscrowedEXA.sol` file by declaring the `ISablierV2LockupLinear` interface and defining top-level structs before the `EscrowedEXA` contract implementation.

To increase the overall readability of the codebase, consider following the Solidity style guide.

Update: *Acknowledged, not resolved. The Exactly team stated:*

A critical vulnerability was identified early on in this audit and was immediately disclosed to the Exactly team. Communication with the Exactly team was smooth and efficient. They provided us with timely answers to our questions and diligently started working on the findings as soon as they were sent over.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

Threat Monitoring
Incident Response
Operation and Automation

Zero Knowledge Proof Practice

Blog

Company

About us
Jobs
Blog

Contracts Library

Docs