Security Review Report
NM-0068 eCredits

(Jan 22, 2023)

Contents

1 Executive Summary 2

2 Audited Files 3

3 Summary of Issues 3

4 System Overview 4
- 4.1 EVaultCampaigns . . . 5
- 4.2 EVaultCampaign . . . 5
- 4.3 EVaultCampaignBudget . . . 5
- 4.4 Roles . . . 5

5 Risk Rating Methodology 6

6 Issues 7
- 6.1 [High] Rewards are computed based on single deposits, instead of considering the whole deposited amount . . . 7
- 6.2 [High] Rewards are the same for different values deposited due to truncation errors . . . 7
- 6.3 [Medium] Adding new campaigns may be blocked by other users . . . 8
- 6.4 [Medium] freezingThresholdLimit and maxFreezableAmountPerAddress must be validated . . . 8
- 6.5 [Low] Claiming may be too costly to execute . . . 9
- 6.6 [Low] Freezing small amounts is possible when budget contract has zero balance . . . 9
- 6.7 [Low] Function calculateFreezableAmountForBudgetLeft(...) performs division before multiplication . . . 10
- 6.8 [Low] Function getFreezableAmount(...) reverts due to overflow . . . 11
- 6.9 [Low] Funds may be locked in the protocol . . . 11
- 6.10 [Low] Multiple transfers can be avoided (risk of out-of-gas failures) . . . 12
- 6.11 [Low] Not using the Checks-Effects-Interactions pattern . . . 12
- 6.12 [Low] Payable functions not validating msg.value can grow storage unlimitedly . . . 13
- 6.13 [Low] User can bypass the function EVaultCampaigns.fund() . . . 14
- 6.14 [Info] Address may freeze amount above maxFreezableAmountPerAddress . . . 15
- 6.15 [Info] Application does not keep track of funders . . . 15
- 6.16 [Info] Events are missing indexed fields . . . 15
- 6.17 [Info] Function calculateRewardsForSeconds(...) can be slightly optimized . . . 15
- 6.18 [Info] Not checking subscriptionNftAddress for address(0x0) . . . 16
- 6.19 [Info] Old campaigns can be funded . . . 16
- 6.20 [Info] Storage accessed inside loops . . . 17
- 6.21 [Info] Unlocked Pragma . . . 17
- 6.22 [Info] Unnecessary use of a mapping for storing the campaigns in EVaultCampaigns . . . 18
- 6.23 [Info] EVaultCampaign.sol imports itself . . . 18
- 6.24 [Best Practice] Inline comments not following the NatSpec format . . . 19
- 6.25 [Best Practices] Avoid using strings with length greater than 32 bytes . . . 19
- 6.26 [Best Practices] Emitted events related to the Vault don't mention the claimer / sender . . . 19
- 6.27 [Best Practices] Functions reading the same storage variable twice . . . 20
- 6.28 [Best Practices] Functions that should have external visibility . . . 20
- 6.29 [Best Practices] Interaction between contracts without using interfaces . . . 21
- 6.30 [Best Practices] Use the local variable activeCampaignAddress instead accessing the storage twice . . . 21
- 6.31 [Best Practices] bool in storage is more expensive than uint256 . . . 21

7 Documentation Evaluation 22

8 Test Suite Evaluation 23
- 8.1 Contracts Compilation Output . . . 23
- 8.2 Tests Output . . . 23
- 8.3 Code Coverage . . . 29
- 8.4 Slither . . . 29

9 About Nethermind 30

1 Executive Summary

This document presents the security review performed by Nethermind in the eCredits code. This protocol allows users to freeze funds for a period of 12 months for earning interests. Standard users receive an annual interest rate of 2%. Users with Gold subscription receive an annual interest rate of 3%, while users having a Platinum subscription receive 6%, Any user can freeze several times, and freezes are only accepted when there is enough budget for rewarding the users.

The source code consists of 318 lines of Solidity with code coverage of 89%. The audit was performed with manual analysis of the codebase, use of automated tools, followed by the deployment and operation of the contracts. The audit team has not received a formal documentation suite, but a set of use cases for conducting the audit. The client was responsive and guided us through the audit.

The most important issues are related to the way rewards are computed, the possibility of bypassing the main contract and funding old campaigns (which can be solved by improving the access control), missing validation for some input parameters, payable functions not checking `msg.value` for being greater than zero, storage access inside loops, costly loops, and the lack of a formal use case for closing the campaign, preventing new funds to be added to a closed campaign.

During the audit, we point out 31 points of attention. The client has fixed 26 points of attention, while 5 points of attention were acknowledged. The acknowledged points do not bring any risk to the protocol and they are just minor optimizations in the code. The distribution of issues is summarized in Fig. 1. **This document is organized as follows.** Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the protocol overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided for this audit. Section 8 presents the compilation, tests, coverage and automated tests. Section 9 concludes the document.

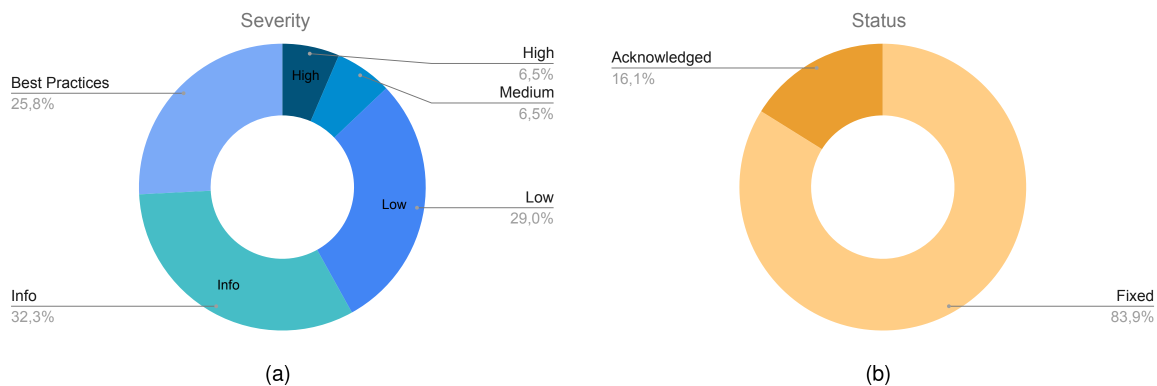

Fig. 1: Distribution of issues: Critical (0), **High** (2), **Medium** (2), **Low** (9), **Undetermined** (0), **Informational** (10), **Best Practices** (8). **Distribution of status: Fixed** (26), **Acknowledged** (5), **Mitigated** (0), **Unresolved** (0).

Summary of the Audit

Audit Type	Security Review
Initial Report	Jan. 3, 2023
Response from Client	Jan. 10, 2023
Final Report	Jan. 22, 2023
Methods	Manual Review, Automated Analysis
Repository	eCredits by Cryptix
Commit Hash (Initial Audit)	00271bf3a046c872d79196ab19593b7b2dcb7149
Commit Hash (Final Audit)	7107cc5ad957d5491bdb862820653b84cb051b39
Documentation	Low
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	contracts/EVaultCampaignBudget.sol	24	9	37.5%	9	42
2	contracts/EVaultCampaigns.sol	83	45	54.2%	24	152
3	contracts/EVaultCampaign.sol	211	52	24.6%	56	319
	Total	**318**	**106**	**33.3%**	**89**	**513**

3 Summary of Issues

	Finding	Severity	Update
1	Rewards are computed based on single deposits, instead of considering the whole deposited amount	High	Fixed
2	Rewards are the same for different values deposited due to truncation errors	High	Fixed
3	Adding new campaigns may be blocked by other users	Medium	Fixed
4	`freezingThresholdLimit` and `maxFreezableAmountPerAddress` must be validated	Medium	Fixed
5	Claiming may be too costly to execute	Low	Acknowledged
6	Freezing small amounts is possible when budget contract has zero balance	Low	Fixed
7	Function `calculateFreezableAmountForBudgetLeft(...)` performs division before multiplication	Low	Fixed
8	Function `getFreezableAmount(...)` reverts due to overflow	Low	Fixed
9	Funds may be locked in the protocol	Low	Acknowledged
10	Multiple transfers can be avoided (risk of out-of-gas failures)	Low	Fixed
11	Not using the Checks-Effects-Interactions pattern	Low	Fixed
12	Payable functions not validating `msg.value` can grow storage unlimitedly	Low	Fixed
13	User can bypass the function `EVaultCampaigns.fund()`	Low	Fixed
14	Address may freeze amount above `maxFreezableAmountPerAddress`	Info	Acknowledged
15	Application does not keep track of funders	Info	Fixed
16	Events are missing indexed fields	Info	Fixed
17	Function `calculateRewardsForSeconds(...)` can be slightly optimized	Info	Fixed
18	Not checking `subscriptionNftAddress` for `address(0x0)`	Info	Fixed
19	Old campaigns can be funded	Info	Fixed
20	Storage accessed inside loops	Info	Fixed
21	Unlocked Pragma	Info	Fixed
22	Unnecessary use of a mapping for storing the `campaigns` in `EVaultCampaigns`	Info	Fixed
23	`EVaultCampaign.sol` imports itself	Info	Fixed
24	Avoid using strings with length greater than 32 bytes	Best Practices	Fixed
25	Emitted events related to the Vault don't mention the `claimer` / `sender`	Best Practices	Acknowledged
26	Functions reading the same storage variable twice	Best Practices	Fixed
27	Functions that should have external visibility	Best Practices	Fixed
28	Inline comments not following the NatSpec format	Best Practices	Fixed
29	Interaction between contracts without using interfaces	Best Practices	Fixed
30	Use the local variable `activeCampaignAddress` instead accessing the storage twice	Best Practices	Fixed
31	`bool` in storage is more expensive than `uint256`	Best Practices	Acknowledged

4 System Overview

The protocol is composed of four contracts summarized in Fig. 2. The contract **SubscriptionNFT** is responsible for storing the category of subscription. Users having a Gold or Platinum subscription will have an `ERC1155` token stored in this contract. This is a simple and straightforward contract that is out of the scope of this audit. The contract **EVaultCampaigns** works like a factory, managing the active campaign, instantiating a new one when necessary, and also providing the entry point and interface for external interactions. As we show along the report, some functions provided by this contract can be bypassed, which can be easily solved by adding access control to the underlying contracts.

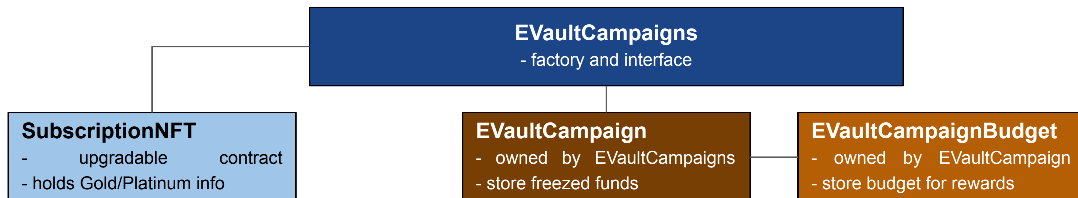

Fig. 2: Contracts composing the protocol

The contract **EVaultCampaign** is owned by the factory contract and receives in its constructor the address of the `SubscriptionNFT` contract, the `freezingThresholdLimit` and the `maxFreezableAmountPerAddress`. The `freezingThresholdLimit` is a reserve of funds used to guarantee the existence of funds for paying all the rewards after 12 months, while the `maxFreezableAmountPerAddress` imposes a limit on the amount that each address can freeze for earning interests. This contract holds the investments made by users.

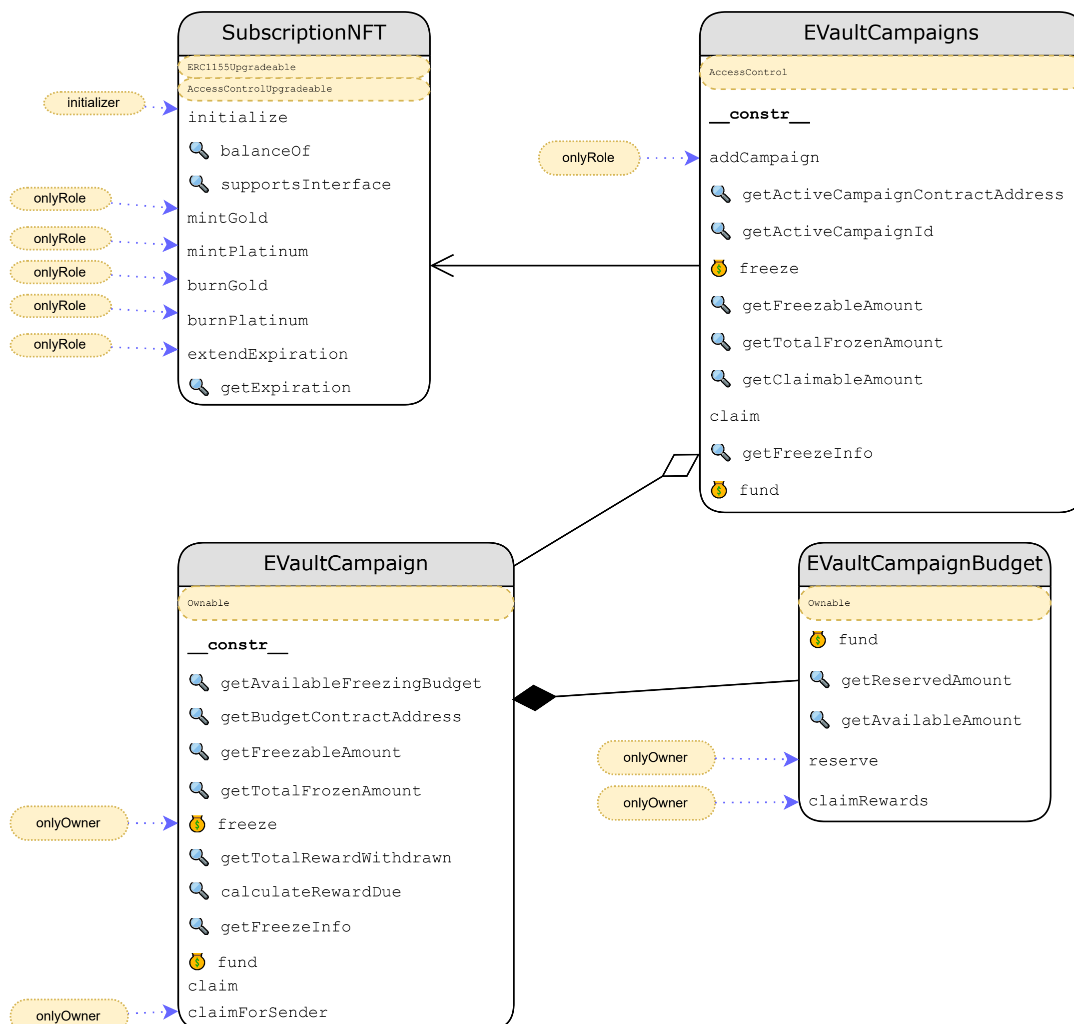

Fig. 3: Structural Diagram of the Contracts

However, the reward funds are stored in a separate contract named **EVaultCampaignBudget**. This contract is instantiated during the construction of the contract `EVaultCampaign`. Each campaign is associated with one single budget contract, and each budget contract is also associated with just one campaign contract. The budget contract stores the funds that are available for paying rewards. When any user freezes a given amount in the contract `EVaultCampaigns`, that operation propagates to the active campaign. The active campaign checks the availability of funds in the budget contract. In case funds are available, the budget contract reserves the amount required for paying the reward to the user and updates the available balance. New investments (freezes) are only accepted when the budget contract has sufficient available funds for paying the rewards. The contracts are described below in accordance with the structural diagram presented in Fig. 3.

4.1 `EVaultCampaigns`

This contract is a wrapper for the wallet orchestrating calls to all existent campaigns and also responsible for adding new campaigns. Campaigns don't have a particular end date, meaning as long as funds are available they can be used for freezing. When a campaign is getting closer to its end, it means that funds have been drained close to total budget. Then, the owner decides whether they send additional funds to the budget or let the campaign end (with the last available freezing) and create a new one.

4.2 `EVaultCampaign`

The contracts `EVaultCampaign` and `EVaultCampaignBudget` consist of the logic for storing assets and computing rewards based on amounts that users freeze and claim. `EVaultCampaign` is a contract that users and owner use to get several information, such as compute the freezable amount for a particular claimer, all freezings of a particular claimer within the campaign, and get total reward withdrawn by a claimer. Only the owner can call `freeze()` and `claimForSender()` functions. The freezings are locked in the campaign contract and they apply two strategies to know if all funds have been transferred from the campaign contract to users: a) campaign contract balance which should decrease down to 0 over time; b) by parsing the emitted events.

4.3 `EVaultCampaignBudget`

The `EVaultCampaignBudget` contract holds the budget funds and acts as the source for the rewards given by a campaign. The budget contract only transfers funds upon user claim. This is computed proportionally to the freezing period held by the user. However, there are situations where some funds can remain available in the contract after the 12 months: a) the users do not claim their accumulated rewards, which means they are marked as reserved but are finally not withdrawn; b) if nobody wants to freeze anymore, which essentially means that the budget will not be reduced (in terms of reserved amount) and therefore the available funds still remain there.

4.4 Roles

The `DEFAULT_ADMIN_ROLE` is the address who has deployed the contract `EVaultCampaigns`. The role can:

- Set new roles in the AccessManager contract.

The `CAMPAIGN_CREATOR_ROLE` is set by the `DEFAULT_ADMIN_ROLE` during the deployment of the contract `EVaultCampaigns`. The role can:

- Create a new campaign in `EVaultCampaigns.addCampaign(...)`;

The `CAMPAIGN_OWNER` is the contract `EVaultCampaigns`. The role can:

- Freeze amounts into the campaign by calling `EVaultCampaign.freeze(address sender)`;
- Claim for a sender by calling the function `EVaultCampaign.claimForSender(address claimer)`.

The `BUDGET_OWNER` is the contract `EVaultCampaign`. The role can:

- Reserve funds in the budget contract by calling the function `EVaultCampaignBudget.reserve(uint256 amount)`;
- Claim rewards for any user by calling the function `EVaultCampaignBudget.claimRewards(address payable claimer, uint256 amount)`.

The protocol provides the following public functionalities.

- `EVaultCampaigns.freeze()`
- `EVaultCampaigns.claim()`
- `EVaultCampaigns.fund()`
- `EVaultCampaign.fund()`
- `EVaultCampaign.claim()`
- `EVaultCampaignBudget.fund()`

In addition to these features, the contract also provides `getters` which are not listed above.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to: Data/state integrity, loss of availability, financial loss, reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] Rewards are computed based on single deposits, instead of considering the whole deposited amount

File(s): contracts/EVaultCampaigns.sol, contracts/EVaultCampaign.sol,contracts/EVaultCampaignBudget.sol

Description: Let's assume Bob is a regular user who will receive an interest rate of 2%. Bob makes the following freezes in sequential blocks into a single campaign:

```
Bob freezes 40 Wei in Block 1;
Bob freezes 40 Wei in Block 2;
Bob freezes 40 Wei in Block 3;
...
Bob freezes 40 Wei in Block 10;
```

Along the same day, Bob has deposited 400 Wei using 10 different transactions. Then, Bob waits for the period of 12 months for claiming the rewards. Since Bob has freezed 400 Wei, he expects to receive 8 Wei as rewards. However, since interests are computed using individual deposits, and there is a truncation, Bob will receive 0 Wei as a total reward for his 400 Wei freezed during 12 months, as presented below.

```
Bob's Rewards for Block 1 = 2% of 40 Wei = 0.8, which will be rounded down to 0 Wei;
Bob's Rewards for Block 2 = 2% of 40 Wei = 0.8, which will be rounded down to 0 Wei;
Bob's Rewards for Block 3 = 2% of 40 Wei = 0.8, which will be rounded down to 0 Wei;
...
Bob's Rewards for Block 10 = 2% of 40 Wei = 0.8, which will be rounded down to 0 Wei.
```

A better technical solution would be to compute the interests considering the whole amount freezed up to this moment, which would allow Bob to receive the proper reward for his freezed investment. Although we are presenting an extreme example, this situation can happen even when users freeze amounts that are not multiples of 50 when considering the interest rate of 2%. Another example is presented below. Alice freezes two values:

```
Alice freezes 170 Wei in Block 1;
Alice freezes 230 Wei in Block 2;
```

Since Alices has also freezed 400 Wei, she expects to receive 8 Wei as rewards. However, what happens is described below:

```
Alice's Rewards for Block 1 = 2% of 170 Wei = 3.4, which will be rounded down to 3 Wei;
Alice's Rewards for Block 2 = 2% of 230 Wei = 4.6, which will be rounded down to 4 Wei.
```

Instead of 8 Wei, Alice will get only 7 Wei as rewards. However, if she delays 20 Wei from Block 1 to Block 2, she would freeze as follows:

```
Alice freezes 150 Wei in Block 1;
Alice freezes 250 Wei in Block 2.
```

In this setup, Alice would get 8 Wei of rewards, as show below:

```
Alice's Rewards for Block 1 = 2% of 150 Wei = 3, which won't be rounded;
Alice's Rewards for Block 2 = 2% of 250 Wei = 5, which won't be rounded.
```

Recommendation(s): Compute the interests considering the whole amount freezed up to each moment, and not using the individual amounts freezed at each block. Modifying this functionality will also require reviewing the `reserve` value stored in the budget contract. Another approach is to ensure the amounts to be multiples of 100.

Status: Fixed

Update from the client: Fixed by accepting only amounts which are multiple of 100 Wei in the `freeze()` function. Wallet UI validation will be updated in order to reflect this. Fixed in commit hash 7107cc5ad957d5491bdb862820653b84cb051b39.

6.2 [High] Rewards are the same for different values deposited due to truncation errors

File(s): contracts/EVaultCampaigns.sol, contracts/EVaultCampaign.sol,contracts/EVaultCampaignBudget.sol

Description: According to the specification, by default users are rewarded with 2% in ECS. Users with Gold subscription are rewarded with 3% in ECS. Users with Platinum subscription are rewarded with 6% in ECS (interest rates considering 12 months of investment).

Let's consider the scenario where users are rewarded with 2% in ECS. If the user invests 50 ECS, after 12 months he/she will be rewarded with 1 ECS. This is the same value obtained by another user who invested 99 ECS. The values are the same due to rounding errors.

In the scenario of 2% of reward, every 50 units provide 1 unit of interest. Users with a Gold subscription are rewarded 3%. So, every 33.33333...3 units lead to an interest of 1. Platinum users are rewarded with 6%. So, every 16.66...6 units lead to an interest of 1 unit. A straightforward way of addressing this issue is requiring every investment to be a multiple of 100. By doing so, all the three interest rates can be properly computed.

Recommendation(s): The smart contract must assist the users when depositing. If depositing more does not change the rewards, the users must be alerted. One straightforward solution is to require deposited amounts to be multiple of 100. Other approaches can be proposed as well. Also update the user facing documentation.

Status: Fixed

Update from the client: Fixed by accepting only amounts which are multiple of 100 Wei in the `freeze()` function. Wallet UI validation will be updated in order to reflect this. Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.3 [Medium] Adding new campaigns may be blocked by other users

File(s): `contracts/EVaultCampaigns.sol`

Description: The new campaign may be added only if the current active campaign has less than `1 ether` of funds for rewards. Since anyone can fund a campaign, the possibility of finishing the campaign and creating a new one is not ensured by the protocol. This may lead to an infinite run of one active campaign.

Recommendation(s): Consider adding a mechanism that would safely allow the owner to start a new campaign.

Status: Fixed

Update from the client: Fixed by adding the function `addCampaignWithForce()`.

Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.4 [Medium] `freezingThresholdLimit` and `maxFreezableAmountPerAddress` must be validated

File(s): `contracts/EVaultCampaigns.sol`, `contracts/EVaultCampaign.sol`

Description: The `EVaultCampaign.constructor(...)` is called by `EVaultCampaigns.addCampaign(...)`, which is a protected function only callable by the `CAMPAIGN_CREATOR_ROLE`. Since the protocol is ultimately managed by humans, writing a defensive code is useful to promptly identify human errors. By inspecting the code, we notice that `EVaultCampaign.constructor(...)` does not perform any validation on the input parameters `_freezingThresholdLimit` and `_maxFreezableAmount`. For instance, the `_freezingThresholdLimit` and `_maxFreezableAmount` can be checked for a validity range. The code with audit comments is reproduced below.

```
constructor(address _subscriptionNftAddress, uint256 _freezingThresholdLimit, uint256 _maxFreezableAmount) {
    ///////////////////////////////////////////////////////////////////////////////
    // @audit "_freezingThresholdLimit" not validated
    // @audit "_maxFreezableAmount" not validated.
    ///////////////////////////////////////////////////////////////////////////////
    subscriptionNFT = SubscriptionNFT(_subscriptionNftAddress);
    freezingThresholdLimit = _freezingThresholdLimit;
    maxFreezableAmountPerAddress = _maxFreezableAmount;
    budget = new EVaultCampaignBudget();
}
```

These parameters play a key role in the application, and the protocol can be useless if wrong values are accepted for these parameters. The function `EVaultCampaigns.addCampaign(...)` uses the input parameters `maxFreezableAmount` and `freezingThresholdLimit` to create a campaign, as reproduced below.

```
function addCampaign(uint256 freezingThresholdLimit, uint256 maxFreezableAmount) external {
    ...
    address campaignAddress = createCampaign(freezingThresholdLimit, maxFreezableAmount);
    ...
}
```

Once the campaign is created by instantiating the contract `EVaultCampaign`, `maxFreezableAmount` and `freezingThresholdLimit` can not be changed, since those variables are declared as `immutable` in `EVaultCampaign.sol`, as reproduced below.

```
uint256 private immutable freezingThresholdLimit;
uint256 private immutable maxFreezableAmountPerAddress;
```

When `maxFreezableAmountPerAddress` is zero and the campaign is funded, the functions `freeze(...)` and `getFreezableAmount(...)` will revert due to overflow, as presented in the code snippets below:

```
function freeze(address sender) public payable onlyOwner {
    ////////////////////////////////////////////////////////////
    // @audit The operation "maxFreezableAmountPerAddress - getTotalFrozenAmount(sender)"
    // overflows if "maxFreezableAmountPerAddress" is zero.
    ////////////////////////////////////////////////////////////
    require (maxFreezableAmountPerAddress - getTotalFrozenAmount(sender) >= msg.value, 'freeze: msg.value of sender
↪   above max freezable amount');
    ...
}
```

```
function getFreezableAmount(address claimer) public view returns (uint256) {
    ////////////////////////////////////////////////////////////
    // @audit The operation "maxFreezableAmountPerAddress - getTotalFrozenAmount(claimer)"
    // overflows if "maxFreezableAmountPerAddress" is zero.
    ////////////////////////////////////////////////////////////
    uint256 claimerFreezable = maxFreezableAmountPerAddress - getTotalFrozenAmount(claimer);
    ...
    return claimerFreezable - freezingThresholdLimit;
}
```

As the users can never freeze, the `budget.claimRewards(...)` call will never be executed and assets in `EVaultCampaignBudget` will be locked permanently. Furthermore, the function `getAvailableFreezingBudget(...)` will always return the `notReserved` amount when `freezingThresholdLimit` is zero. This threshold is used to ensure that all users can receive the reward at the end of the campaign. The code snippet is reproduced below.

```
function getAvailableFreezingBudget() public view returns (uint256) {
    uint256 notReserved = budget.getAvailableAmount();
    if (notReserved < freezingThresholdLimit)
        return 0;

    ////////////////////////////////////////////////////////////
    // @audit If "freezingThresholdLimit" is zero, the function
    // always returns "notReserved".
    ////////////////////////////////////////////////////////////
    return notReserved - freezingThresholdLimit;
}
```

Recommendation(s): Ensure that `maxFreezableAmount` and `freezingThresholdLimit` are within reasonable intervals and they are greater than zero.

Status: Fixed

Update from the client: Fixed by adding basic validation. Fixed in commit hash 7107cc5ad957d5491bdb862820653b84cb051b39.

6.5 [Low] Claiming may be too costly to execute

File(s): `contracts/EVaultCampaign.sol`

Description: To claim rewards, user can call two functions: `EVaultCampaigns.claim(...)` which iterates over all `FrozenAmount` objects in all existing campaigns, or `EVaultCampaign.claim(...)` which computes reward due from one campaign. If there are sufficient campaigns and `FrozenAmount` objects, the cost of claiming funds in function `EVaultCampaigns.claim(...)` can be high. In such a situation, the user would have to call `claim(...)` on each campaign. This is not ideal, since the wallet works only with the active campaign, and calling previous campaigns requires interacting with the smart contract outside of the user interface.

Recommendation(s): To optimize the process of claiming rewards, consider allowing the user to easily claim rewards from previous campaigns through the user interface, so it would not require technical knowledge on the user side. You can also consider adding a pagination scheme in case the user has too many entries in the `FrozenAmount` structure.

Status: Acknowledged

Update from the client: That's a valid and known issue. For the sake of simplicity it was decided to offer one "claim-for-everything" approach with potentially increased gas costs. The required changes would be too big to implement in tine. Additionally as the expected count of campaigns is rather low and the gas costs in the network are cheap it is a tradeoff we accept.

6.6 [Low] Freezing small amounts is possible when budget contract has zero balance

File(s): `contracts/EVaultCampaign.sol`

Description: The function `freeze(...)` computes the value that must be reserved in the budget contract (`EVaultCampaignBudget`). When `msg.value` is small, the function `calculateBudgetToReserve(...)` will return zero due to truncation. Then, the code checks if `toReserve` is smaller than or equal to the balance of the budget contract in the line reproduced below.

```
require(toReserve <= budget.getAvailableAmount(), 'freeze: no budget left anymore');
```

Since `toReserve` and `budget.getAvailableAmount()` have the same value (zero), the `require(...)` will pass, and the user will be able to freeze an amount without having any reserves in the budget contract. The whole function is reproduced below.

```
function freeze(address sender) public payable onlyOwner {
    require (maxFreezableAmountPerAddress - getTotalFrozenAmount(sender) >= msg.value, 'freeze: msg.value of sender
↪   above max freezable amount');

    uint256 rate; uint256 subscription;
    (rate, subscription) = determineRate(sender);

    /////////////////////////////////////////////////////////////////////////////////////////////////
    // @audit When "msg.value" is small, "toReserve" receives zero.
    /////////////////////////////////////////////////////////////////////////////////////////////////
    uint256 toReserve = calculateBudgetToReserve(msg.value, rate);

    /////////////////////////////////////////////////////////////////////////////////////////////////
    // @audit When "toReserve" is zero and the budget contract has zero balance,
    // the "require" clause is evaluated as "true".
    /////////////////////////////////////////////////////////////////////////////////////////////////
    require(toReserve <= budget.getAvailableAmount(), 'freeze: no budget left anymore');
    budget.reserve(toReserve);

    FrozenAmount[] storage frozenSenderAmounts = frozenAmounts[sender];

    if (frozenSenderAmounts.length > 0) {
        require(frozenSenderAmounts[frozenSenderAmounts.length - 1].blockNumber != getBlockNumber(), 'freeze: sender
↪   already did a freezing in same block');
    }

    FrozenAmount memory fa = FrozenAmount(msg.value, getBlockNumber(), getBlockTimestamp(), rate, 0, false);
    frozenSenderAmounts.push(fa);

    emit VaultLock(fa.blockNumber, msg.value, rate, subscription);
}
```

Recommendation(s): There are some options. a) if `toReserve` is zero, revert the operation; b) set a minimal amount for freezes; c) change the `require(...)` clause to check if `toReserve` is smaller than `budget.getAvailableAmount()`. This option is not the best one, since at least 1 wei will be locked forever.

Status: Fixed

Update from the client: Fixed with option (a). Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.7 [Low] Function `calculateFreezableAmountForBudgetLeft(...)` performs division before multiplication

File(s): `contracts/EVaultCampaign.sol`

Description: The function `getFreezableAmount(...)` performs division before the multiplication. Solidity integer division might truncate. As a result, performing multiplication before division can sometimes avoid loss of precision. The function with audit comments is reproduced below.

```
function calculateFreezableAmountForBudgetLeft(uint256 budgetLeft, uint256 rate) private pure returns (uint256) {
    ////////////////////////////////////////////////////////////////
    // @audit Performing multiplication on the result of a division.
    ////////////////////////////////////////////////////////////////
   uint256 amount = budgetLeft * 100 * WEI_PRECISION / ONE_PERCENT_ATTO_WEI_REWARD_PER_SECOND / SECONDS_PER_YEAR / rate;
    return amount * 1 ether;
}
```

Recommendation(s): Perform multiplication before division.

Status: Fixed

Update from the client: Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.8 [Low] Function `getFreezableAmount(...)` reverts due to overflow

File(s): `contracts/EVaultCampaign.sol`

Description: The function `getFreezableAmount(...)` returns the amount that can be freezed by the user. However, whenever the user has `claimerFreezable` less than `freezingThresholdLimit` the function reverts. The reversion is due to an overflow in the last line of the function (reproduced below).

```
function getFreezableAmount(address claimer) public view returns (uint256) {
    ////////////////////////////////////////////////////////////////////////
    // @audit Function can have external visibility
    ////////////////////////////////////////////////////////////////////////
    uint256 claimerFreezable = maxFreezableAmountPerAddress - getTotalFrozenAmount(claimer);
    uint256 budgetLeft = getAvailableFreezingBudget();
    if (budgetLeft == 0)
        return 0;

    (uint256 rate, ) = determineRate(claimer);
    uint256 toReserve = calculateBudgetToReserve(claimerFreezable, rate);

    if (toReserve > budgetLeft) {
        // we cannot reserve the full claimable amount -> lets inverse calculate with the rest
        claimerFreezable = calculateFreezableAmountForBudgetLeft(budgetLeft, rate);
        return claimerFreezable;    // freezingThresholdLimit already considered in budgetLeft, no need to subtract!
    }

    ////////////////////////////////////////////////////////////////////////
    // @audit Function reverts due to overflow when user has freezed the
    // whole amount. Remember that Solidity 0.8.* uses SafeMath.
    ////////////////////////////////////////////////////////////////////////
    return claimerFreezable - freezingThresholdLimit;
}
```

Whenever `claimerFreezable` is between `0` and `freezingThresholdLimit-1`, the subtraction `claimerFreezable - freezingThresholdLimit` will overflow and the function will revert.

Recommendation(s): Mitigate the underflow error by validating that `claimerFreezable` is greater than or equal to `freezingThresholdLimit`. Otherwise return `0`.

Status: Fixed

Update from the client: Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.9 [Low] Funds may be locked in the protocol

File(s): `contracts/EVaultCampaigns.sol`, `contracts/EVaultCampaign.sol`,`contracts/EVaultCampaignBudget.sol`

Description: Users may freeze only in the active campaign. A new campaign may be created when the current active campaign has less than `1 ether`. If a new campaign is created when the current active campaign still has freezable funds (< `1 ether`), those funds would be locked in the campaign forever.

After some clarification from the client, there is no withdrawal function in the budget contract on purpose in order to guarantee that nobody has access to the budget funds except the users freezing funds. The only way for the protocol to get the funds out again is to freeze on our own funds for the remaining budget and claim it one year later (or earlier for partial access).

Recommendation(s): We are okay with the clarification provided by the client. In order to contribute, we present some comments below that maybe can help the application. When creating a new campaign, the first step could be accumulating the balance of all users into a new vault contract (invested amount plus rewards). Once these funds are transferred to the vault contract, remaining funds in the active campaign can be transferred to a new campaign that is about to be created. By doing so, users will withdraw the funds from all finished campaigns in one single transaction from the vault contract. The simplified flow for creating a campaign is presented below.

```
if the current campaign has budget < 1 ether then
    create and activate the new campaign;
end if
```

In summary, the decision of creating a new campaign is made based on the available balance of the current campaign, and the code does not engage in special actions for the campaign that is being terminated. Below we present another possible flow.

```
if the current campaign has budget < 1 ether then
    transfer the rewards of each user from the current campaign to the balance of the user in the Vault contract;
   transfer the freezed values from each user from the current campaign to the balance of the user in the Vault contract;
    create the new campaign;
    transfer the remaining budget in EVaultCampaignBudget from the current campaign to the new campaign;
    finish the current campaign avoiding new values to be freezed or funded into this campaign;
    formally activate the new campaign;
end if
```

Status: Acknowledged

Update from the client: Valid point and the suggestion is really appreciated. Having this issue in mind we have also created a half-shaped prototype version to replace tracking of FrozenAmount objects with a contract deployment per freezing() and just store list of contract addresses per freezer/campaign in order to get rid of homebrew bookkeeping logic and let EVM track the state of the funds. Due to the project timeline we have to postpone such an improved version to V2 of the protocol.

6.10 [Low] Multiple transfers can be avoided (risk of out-of-gas failures)

File(s): `contracts/EVaultCampaign.sol`

Description: The functions `claim(...)` and `claimForSender(...)` call `calculateRewardDueAndUpdateStorage(...)` to calculate the reward claimable for all frozen amounts and pay the unlocked amounts from the campaign. This function calls `transfer(...)` within a loop. There are different reasons why sending assets to another address may fail. One reason for failure is running out of gas, e.g., many transfers made within a single function call. Another example that results in failure is if the receiver is not able to receive funds via transfer. So, every new attempt to transfer `toUnlock` amount to the claimer will result in an exception, thus, making it impossible to pay unlocked amounts from the campaign. To minimize the damage caused by these failures, it would be a better choice to isolate external calls into its own transaction, initiated by the claimer. If possible, avoid combining multiple transfers in a single transaction. The code is reproduced below.

```
function calculateRewardDueAndUpdateStorage(address claimer) private returns (uint256) {
    ...
    for (uint i; i < frozenSenderAmounts.length; i++) {
        FrozenAmount storage fa = frozenSenderAmounts[i];
        (uint256 rewardWei, uint256 toUnlock) = calculateRewardWei(fa);
        totalDue += rewardWei;
        if (toUnlock > 0) {
            fa.returned = true;
            ////////////////////////////////////////////
            // @audit Multiple transfers
            ////////////////////////////////////////////
            payable(claimer).transfer(toUnlock);
            emit VaultReturn(fa.blockNumber, toUnlock);
        }
        ...
    }
    ...
}
```

Recommendation(s): There are some options to address this issue: a) all the values can be transferred to the user in a single transaction; b) let the claimer "pull" the amounts, instead of "pushing" the values into the claimer. The protocol can store the value able to be claimed from all campaigns into a single storage variable for each user. For more information on the attack discussed here, please check this reference.

Status: Fixed

Update from the client: As the receiver is always the same single address (the claimer who can only claim his very own funds to his same address) the risk of having single transfers failing is neglectable (only a matter of gas, which is anyway pre-estimated by wallet via eth_estimateGas call). Nevertheless it is valid that multiple transfers in a loop result in increased gas consumption which will be fixed by doing only one single transfer call with summed up total amount after the loop while still emitting multiple `VaultReturn` events to track individual returns. Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.11 [Low] Not using the Checks-Effects-Interactions pattern

File(s): `contracts/EVaultCampaign.sol`

Description: The `calculateRewardDueAndUpdateStorage(...)` and `freeze(...)` functions are not following the checks-effects-interactions pattern. This pattern indicates that the function must perform all the checks in the first place. After checking, the function must do all state updates. Only after the updates, the function must interact with external contracts.

We can notice that the function `calculateRewardDueAndUpdateStorage(...)` applies state changes after the function `transfer(...)`. The code is reproduced below.

```
function calculateRewardDueAndUpdateStorage(address claimer) private returns (uint256) {
    FrozenAmount[] storage frozenSenderAmounts = frozenAmounts[claimer];
    ...
    for (uint i; i < frozenSenderAmounts.length; i++) {
    ...
        if (toUnlock > 0) {
            fa.returned = true;
            //////////////////////////////////////////////////////
            // @audit Interacting with external contracts
            // and making state updates after.
            //////////////////////////////////////////////////////
            payable(claimer).transfer(toUnlock);
            emit VaultReturn(fa.blockNumber, toUnlock);
        }

        uint256 rewardNew = rewardWei - fa.withdrawn;
        totalWithdrawnByClaimer[claimer] += rewardNew;
        fa.withdrawn = rewardWei;
        emit VaultReward(fa.blockNumber, rewardNew);
    }

    return totalDue - claimerWithdrawn;
}
```

Similarly, the function `freeze(...)` applies state changes after calling the external functions `getAvailableAmount()` and `reserve(...)`. The code is reproduced below.

```
function freeze(address sender) public payable onlyOwner {
    ...
    uint256 toReserve = calculateBudgetToReserve(msg.value, rate);

    require(toReserve <= budget.getAvailableAmount(), 'freeze: no budget left anymore');
    budget.reserve(toReserve);

    FrozenAmount[] storage frozenSenderAmounts = frozenAmounts[sender];

    if (frozenSenderAmounts.length > 0) {
        require(frozenSenderAmounts[frozenSenderAmounts.length - 1].blockNumber != getBlockNumber(), 'freeze: sender
↪   already did a freezing in same block');
    }

    FrozenAmount memory fa = FrozenAmount(msg.value, getBlockNumber(), getBlockTimestamp(), rate, 0, false);
    frozenSenderAmounts.push(fa);

    emit VaultLock(fa.blockNumber, msg.value, rate, subscription);
}
```

Since the contract is using `transfer(...)`, the contract will be safe by adopting the checks-effects-interactions pattern. However, if you change `transfer(...)` to `call(...)`, the contract becomes highly vulnerable to reentrancy attacks.

Recommendation(s): Always apply the checks-effects-interactions pattern when writing smart contracts. Keep using `transfer(...)`. Discuss the possibility of adding the `nonReentrant` modifier.

Status: Fixed

Update from the client: `calculateRewardDueAndUpdateStorage()` fixed by changes from the issue **Multiple transfers can be avoided (risk of out-of-gas failures)**. Function `freeze()` is fixed by moving contract calls to the end of function.

Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.12 [Low] Payable functions not validating `msg.value` can grow storage unlimitedly

File(s): `contracts/EVaultCampaign.sol`, `contracts/EVaultCampaigns.sol`

Description: The payable functions below do not validate `msg.value` for being greater than zero.

```
EVaultCampaign.freeze(...);
EVaultCampaigns.freeze(...);
EVaultCampaign.fund();
EVaultCampaigns.fund();
```

The lack of validation in the function `EVaultCampaigns.freeze(...)` is not desired, since new entries will be created in the storage structure `EVaultCampaign.frozenAmounts` that can grow indefinitely. The code of `EVaultCampaigns.freeze()` is presented below.

```
function EVaultCampaigns.freeze() public payable {
    EVaultCampaign activeCampaign = getActiveCampaign();
    return activeCampaign.freeze{value: msg.value}(msg.sender);
}
```

As we can notice, the function `EVaultCampaigns.freeze()` calls `EVaultCampaign.freeze()`, whose code is reproduced below and we can note the command `frozenSenderAmounts.push(fa)`.

```
function freeze(address sender) public payable onlyOwner {
    require (maxFreezableAmountPerAddress - getTotalFrozenAmount(sender) >= msg.value, 'freeze: msg.value of sender
↪   above max freezable amount');

    uint256 rate; uint256 subscription;
    (rate, subscription) = determineRate(sender);

    uint256 toReserve = calculateBudgetToReserve(msg.value, rate);

    require(toReserve <= budget.getAvailableAmount(), 'freeze: no budget left anymore');
    budget.reserve(toReserve);

    FrozenAmount[] storage frozenSenderAmounts = frozenAmounts[sender];

    if (frozenSenderAmounts.length > 0) {
        require(frozenSenderAmounts[frozenSenderAmounts.length - 1].blockNumber != getBlockNumber(), 'freeze: sender
↪   already did a freezing in same block');
    }

     FrozenAmount memory fa = FrozenAmount(msg.value, getBlockNumber(), getBlockTimestamp(), rate, 0, false);
    ///////////////////////////////////////////////////////////////////////////////////////////////////////
    // @audit New items are added to the storage without validating "msg.value" for being greater than zero.
    ////////////////////////////////////////////////////////////////////////////////////////////////////////
    frozenSenderAmounts.push(fa);

    emit VaultLock(fa.blockNumber, msg.value, rate, subscription);
}
```

Recommendation(s): In payable functions, always check if `msg.value` is greater than zero. Revert otherwise.

Status: Fixed

Update from the client: Fixed in commit hash 7107cc5ad957d5491bdb862820653b84cb051b39.

6.13 [Low] User can bypass the function `EVaultCampaigns.fund()`

File(s): contracts/EVaultCampaigns.sol, contracts/EVaultCampaign.sol,contracts/EVaultCampaignBudget.sol

Description: The fund functionality is not respecting the specification, which defines that funding must occur only through the function `EVaultCampaigns.fund()`. The paragraph 1.1.4 of the specification is reproduced below.

```
1.1.4 Anybody is allowed to extend Allocated Budget of the Campaign. Additional funding always goes to the latest
↪   created campaign even if it has not started yet (to fulfill 1.1.3 and prevent extension of the old campaign after
↪   the new one is already created/scheduled).
```

However, the contracts `EVaultCampaign` and `EVaultCampaignBudget` have a public function `fund()`, and the user can bypass the contract `EVaultCampaigns`. The functions `EVaultCampaign.fund()` and `EVaultCampaignBudget.fund()` are reproduced below.

```
// EVaultCampaign
function EVaultCampaign.fund() public payable {
    budget.fund{value: msg.value}();
}
```

```
function EVaultCampaignBudget.fund() public payable {
    emit CampaignFunded(msg.value);
}
```

Recommendation(s): Add access control to functions `EVaultCampaign.fund()` and `EVaultCampaignBudget.fund()`.

Status: Fixed

Update from the client: Fixed by adding onlyOwner modifier to EVaultCampaign.fund() and EVaultCampaignBudget.fund(). Fixed in commit hash 7107cc5ad957d5491bdb862820653b84cb051b39.

6.14 [Info] Address may freeze amount above maxFreezableAmountPerAddress

File(s): contracts/EVaultCampaigns.sol

Description: The check for available amount to freeze for each address is done by subtracting the value of getTotalFrozenAmount(claimer) from the maxFreezableAmountPerAddress. However, the function getTotalFrozenAmount(...) sums frozen amounts that have not been claimed yet. Thus, any address can freeze the maximum amount, claim it, and then freeze again exceeding maxFreezableAmountPerAddress.

Recommendation(s): This is a business decision. In case one address cannot freeze amounts above maxFreezableAmountPerAddress, keep track of all the deposits. On the other hand, to circumvent this restriction, the user can create another address.

Status: Acknowledged

Update from the client: Known limitation of the current max freezing amount protection logic. As stated it could be circumvented anyway if someone does not use our mobile wallet but instead interacts with the contract directly with another address.

6.15 [Info] Application does not keep track of funders

File(s): contracts/EVaultCampaigns.sol, contracts/EVaultCampaign.sol,contracts/EVaultCampaignBudget.sol

Description: The contracts are not keeping track of the funders, being impossible to return funds when the campaign is over. Also notice that the three contracts have a public function fund(), which does not add clarity on how the functionality should be used.

Recommendation(s): Clearly specify the use case related to funders. If necessary, keep track of funders. Also enforce funding to be made using the function EVaultCampaigns.fund(), and restrict the access to EVaultCampaign.fund() and EVaultCampaignBudget.fund() to onlyOwner.

Status: Fixed

Update from the client: Funders are tracked off-chain by processing and parsing all transactions to the contracts from the backend. In case funds are not claimed theoretically we can notify the users as we have firebase push Ids. Funding should be available for everybody.

Update from Nethermind: We agree and we consider this point of attention as fixed.

6.16 [Info] Events are missing indexed fields

File(s): contracts/EVaultCampaign.sol contracts/EVaultCampaigns.sol

Description: Index event parameters for logged events allow you to search them more quickly by using the indexed fields as filters. Up to three fields can receive the attribute indexed. It's possible to filter for particular values of indexed fields in the user interface. However, it's important to remember that each index parameter costs extra gas during emission.

Recommendation(s): Consider using indexed fields to make them accessible quicker to tools that parse events.

Status: Fixed

Update from the client: Indexed fields omitted to save gas as those events are only parsed by backend.

Update from Nethermind: We agree. Since this decision does not impose any risk to the protocol, we consider this point of attention as resolved (fixed).

6.17 [Info] Function calculateRewardsForSeconds(...) can be slightly optimized

File(s): contracts/EVaultCampaign.sol

Description: The function calculateRewardsForSeconds(...) can be optimized by joining all divisions into one single operation. The current code is reproduced below, and it requires 23,757 gas units.

```
function calculateRewardsForSeconds(uint256 amount, uint256 rate, uint256 sec) private pure returns (uint256) {
    uint256 rewardDueAttoWei = amount * ONE_PERCENT_ATTO_WEI_REWARD_PER_SECOND * sec * rate / 100 / WEI_PRECISION;
    return rewardDueAttoWei / 1 ether;
}
```

The proposed version is presented below, and it consumes 23,425 gas units (saving 332 gas units).

```
function opt_calculateRewardsForSeconds(uint256 amount, uint256 rate, uint256 sec) private pure returns (uint256) {
    return (amount * ONE_PERCENT_ATTO_WEI_REWARD_PER_SECOND * sec * rate ) / 1e24;
}
```

Recommendation(s): Evaluate the possibility of adopting the optimized function.

Status: Fixed

Update from the client: Gas is not critical here. Keeping the old version in favor of readability.

Update from Nethermind: We agree and we consider this point of attention as resolved (`fixed`).

6.18 [Info] Not checking `subscriptionNftAddress` for `address(0x0)`

File(s): `contracts/EVaultCampaigns.sol`

Description: The immutable storage variable `EVaultCampaigns.subscriptionNftAddress` is not checked for `address(0x0)` in the constructor of `EVaultCampaigns`. The code is reproduced below.

```
address internal immutable subscriptionNftAddress;

constructor (address _subscriptionNftAddress) {
    subscriptionNftAddress = _subscriptionNftAddress;

    _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
    _setRoleAdmin(CAMPAIGN_CREATOR_ROLE, DEFAULT_ADMIN_ROLE);
}
```

Recommendation(s): Ensure that `_subscriptionNftAddress` is different from `address(0x0)`. An interesting approach is to change the type of the variable `subscriptionNftAddress` from `address` to `SubscriptionNFT` as presented below.

```
- address         internal immutable subscriptionNftAddress;
+ SubscriptionNFT internal immutable subscriptionNftAddress;
```

Status: Fixed

Update from the client: Address validation added. Address type still used as the address is just passed through to `EVaultCampaign()` contract constructor. Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.19 [Info] Old campaigns can be funded

File(s): `contracts/EVaultCampaigns.sol`, `contracts/EVaultCampaign.sol`,`contracts/EVaultCampaignBudget.sol`

Description: The new campaign may be added only if the current active campaign has balance smaller than `1 ether`. However, anyone can fund old campaigns by calling the functions `EVaultCampaignBudget.fund()`, `EVaultCampaign.fund()`, `EVaultCampaign.fund()`. Funding old campaigns is not expected by the protocol. The functions are reproduced below.

```
function EVaultCampaigns.fund() public payable {
    EVaultCampaign activeCampaign = getActiveCampaign();
    activeCampaign.fund{value: msg.value}();
}
```

```
function EVaultCampaign.fund() public payable {
    budget.fund{value: msg.value}();
}
```

```
function EVaultCampaignBudget.fund() public payable {
    emit CampaignFunded(msg.value);
}
```

Recommendation(s): Ensure that only the newest campaign can be funded.

Status: Fixed

Update from the client: This is known behavior. If anyone is generous enough it's up to him to fund old campaigns (whose funds will never be used as `EVaultCampaigns` redirects all calls to current active campaign contract only). The only problem could be that someone might keep the campaign alive by constantly funding it so the `EVaultCampaigns.addCampaign()` will not allow the creation of a new campaign as there are still funds on the old one (fixed as part of the next issue). Fixed by adding `onlyOwner` modifier to `EVaultCampaign.fund()` and `EVaultCampaignBudget.fund()` essentially allowing only `EVaultCampaigns` to fund the latest campaign. Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.20 [Info] Storage accessed inside loops

File(s): `contracts/EVaultCampaign.sol`

Description: The functions `getTotalFrozenAmount(...)`, `calculateRewardDue(...)` and `getFreezeInfo(...)` access the struct `FrozenAmount` in a loop directly from the storage. Accessing storage variables in loops is expensive. Instead, we can access the storage variable from the memory. Note that those are view functions. The functions are reproduced below.

```
function getTotalFrozenAmount(address claimer) public view returns (uint256 totalFrozen) {
    FrozenAmount[] storage freezingsByClaimer = frozenAmounts[claimer];

    for (uint i; i < freezingsByClaimer.length; i++) {
        if (!freezingsByClaimer[i].returned) {
            totalFrozen += freezingsByClaimer[i].amount;
        }
    }
}
```

```
function calculateRewardDue(address claimer) public view returns (uint256) {
    FrozenAmount[] storage frozenSenderAmounts = frozenAmounts[claimer];
    uint256 claimerWithdrawn = totalWithdrawnByClaimer[claimer];

    uint256 totalDue = 0;
    for (uint i; i < frozenSenderAmounts.length; i++) {
        FrozenAmount storage fa = frozenSenderAmounts[i];
        (uint256 rewardWei, uint256 toUnlock) = calculateRewardWei(fa);
        totalDue += rewardWei + toUnlock;   // here we add both rewards and unlocked amount to indicate gross amount to
↪   wallet
    }

    return totalDue - claimerWithdrawn;
}
```

```
function getFreezeInfo(address claimer, uint256 blockNumber)
    external view returns (uint256 rate, uint256 rewardDue, uint256 withdrawn) {
    FrozenAmount[] storage frozenSenderAmounts = frozenAmounts[claimer];

    for (uint i; i < frozenSenderAmounts.length; i++) {
        FrozenAmount storage fa = frozenSenderAmounts[i];
        if (fa.blockNumber == blockNumber) {
            (rewardDue, ) = calculateRewardWei(fa);
            withdrawn = fa.withdrawn;
            rate = fa.rate;
            break;
        }
    }
}
```

Recommendation(s): Replace the identifier `storage` per `memory` as shown below.

```
function getTotalFrozenAmount(...)
- FrozenAmount[] storage freezingsByClaimer = frozenAmounts[claimer];
+ FrozenAmount[] memory  freezingsByClaimer = frozenAmounts[claimer];
```

```
function calculateRewardDue(...)
- FrozenAmount[] storage frozenSenderAmounts = frozenAmounts[claimer];
+ FrozenAmount[] memory  frozenSenderAmounts = frozenAmounts[claimer];
```

```
function getFreezeInfo(...)
- FrozenAmount[] storage frozenSenderAmounts = frozenAmounts[claimer];
+ FrozenAmount[] memory  frozenSenderAmounts = frozenAmounts[claimer];
```

Status: Fixed

Update from the client: Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.21 [Info] Unlocked Pragma

File(s): **File(s)**: `contracts/EVaultCampaigns.sol`, `contracts/EVaultCampaign.sol`,`contracts/EVaultCampaignBudget.sol`

Description: The contracts use unlocked pragma `^0.8.2`. A better approach is to stick to a single Solidity version. `solc` frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. Also notice that some versions are more stable than others.

Recommendation(s): Lock the pragma to version `0.8.16` and use this version for deploying the contracts. For more information, please check this reference.

Status: Fixed

Update from the client: Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.22 [Info] Unnecessary use of a mapping for storing the `campaigns` in `EVaultCampaigns`

File(s): `contracts/EVaultCampaigns.sol`

Description: The contract `EVaultCampaigns` uses a mapping for storing the campaigns. Mappings are hash tables that use hash functions to generate new values according to a mathematical hashing algorithm, and they are widely used when we need to reduce the search time from a) `O(n)` when the elements are unsorted or b) `O(log(n))` when the elements are sorted down to `O(1)`. However, the campaigns are indexed by integers in a sequential way, and there's no need to have the extra overhead imposed by mappings/hash tables. Instead of using a mapping, you can rely on an array of campaigns. The code snippet is reproduced below.

```
///////////////////////////////////////////////////////////////////////////////////////////////
// @audit Declaring the “campaigns” as a mapping.
// You can use an array.
///////////////////////////////////////////////////////////////////////////////////////////////
mapping(uint => address) private campaigns;


function addCampaign(...) external onlyRole(CAMPAIGN_CREATOR_ROLE) {
    ////////////////////////////////////////////////////
    // @audit "campaigns" is accessed as an array.
    ////////////////////////////////////////////////////
    address activeCampaignAddress = campaigns[activeCampaignId];
    ...
    address campaignAddress = createCampaign(freezingThresholdLimit, maxFreezableAmount);

    ////////////////////////////////////////////////////
    // @audit "campaigns" is accessed as an array.
    ////////////////////////////////////////////////////
    uint256 campaignId = activaCampaignAddress == address(0) ? 0 : activeCampaignId + 1;    // first campaign already
↪   created?

    ////////////////////////////////////////////////////
    // @audit Here is where you would "push" a new
    // campaign into the array.
    ////////////////////////////////////////////////////
    campaigns[campaignId] = campaignAddress;
    activeCampaignId = campaignId;
    ...
}
```

Recommendation(s): Instead of using a mapping for `campaigns`, consider adopting an array.

Status: Fixed

Update from the client: Refactored using array. Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.23 [Info] `EVaultCampaign.sol` imports itself

File(s): `contracts/EVaultCampaign.sol`

Description: The contract `EVaultCampaign.sol` imports itself, as reproduced in the code snippet below.

```
pragma solidity ^0.8.2;

import '@openzeppelin/contracts/access/Ownable.sol';
////////////////////////////////////////////////////////////
// @audit "EVaultCampaign.sol" is importing itself
////////////////////////////////////////////////////////////
import './EVaultCampaign.sol';
import './EVaultCampaignBudget.sol';
import './SubscriptionNFT.sol';
```

Recommendation(s): Do not import `EVaultCampaign.sol` inside `EVaultCampaign.sol`.

Status: Fixed

Update from the client: Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.24 [Best Practice] Inline comments not following the NatSpec format

File(s): **File(s)**: `contracts/EVaultCampaigns.sol`, `contracts/EVaultCampaign.sol`,`contracts/EVaultCampaignBudget.sol`

Description: All the contracts in scope don't follow the NatSpec standard.

Recommendation(s): It is recommended that Solidity contracts are fully annotated using NatSpec.

Status: Fixed

Update from the client: Fixed. Comments added to all contracts and external functions.

Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.25 [Best Practices] Avoid using strings with length greater than 32 bytes

File(s): `contracts/EVaultCampaigns.sol`

Description: The `require(...)` string "addCampaign: there is another campaign still active" has a length of 51 bytes.

Recommendation(s): Consider reducing the length of this string up to 32 bytes.

Status: Fixed

Update from the client: Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.26 [Best Practices] Emitted events related to the Vault don't mention the `claimer` / `sender`

File(s): `contracts/EVaultCampaign.sol`

Description: Emitting events is a good practice and helps to improve protocol monitoring. Events related to the vault are linked to a smart contract (the active campaign) but they don't mention the sender or claimer of funds.

Recommendation(s): Modify the events below to add the concerned sender/claimer of funds.

```
 /* event fired on successful claiming per frozen amount rewarded by the claim */
event VaultReward(
    address user
    uint256 frozenAmountId,
    uint256 amount,
    uint256 rate,
    uint256 subscription
);
```

```
/* event fired on successful claiming for each frozen amount older than one year (and returned by claim) */
event VaultReturn(
    address user
    uint256 frozenAmountId,
    uint256 amount
);
```

```
/* event fired on successful freezing */
event VaultLock(
    address user
    uint256 frozenAmountId,
    uint256 amount,
    uint256 rate,
    uint256 subscription
);
```

Status: Acknowledged

Update from the client: Claimer is always the sender of the transaction so not needed for implementation. Actually We had it there in the beginning, but then we removed it to shrink event data. If it's considered best practice we will add it to new events next time - changing event signatures now would mean a bunch of changes in front-end and backend-end which we would like to avoid.

6.27 [Best Practices] Functions reading the same storage variable twice

File(s): contracts/EVaultCampaigns.sol

Description: The functions getClaimableAmount(...), claim(...), getFreezeInfo(...) can use a local variable to hold the value of activeCampaignId when performing loops for saving gas. The code snippets of these functions are reproduced below.

```
function getClaimableAmount(...) ... {
    ///////////////////////////////////////////////////////
    // @audit Avoid using storage variables inside loops.
    ///////////////////////////////////////////////////////
    for (uint i; i <= activeCampaignId; i++) {
        ...
    }
}
```

```
function claim() external {
    ///////////////////////////////////////////////////////
    // @audit Avoid using storage variables inside loops.
    ///////////////////////////////////////////////////////
    for (uint i; i <= activeCampaignId; i++) {
        ...
    }
}
```

```
function getFreezeInfo(...) ... returns (uint256 rate, uint256 rewardDue, uint256 withdrawn) {
    ///////////////////////////////////////////////////////
    // @audit Avoid using storage variables inside loops.
    ///////////////////////////////////////////////////////
   for (uint i; i <= activeCampaignId; i++) {
       ...
    }
}
```

Recommendation(s): Create a local variable to store the contents of the storage variable activeCampaignId. Use the local variable inside the loops as shown below.

```
- for (uint i; i <= activeCampaignId; i++)
+ uint256 local = activeCampaignId;
+ for (uint i; i <= local; i++)
```

Status: Fixed

Update from the client: Fixed in commit hash 7107cc5ad957d5491bdb862820653b84cb051b39.

6.28 [Best Practices] Functions that should have external visibility

File(s): contracts/EVaultCampaigns.sol, contracts/EVaultCampaign.sol, contracts/EVaultCampaignBudget.sol

Description: The functions below can have external visibility since they are called outside of the contract. This can also result in gas savings.

```
- EVaultCampaign.getFreezableAmount(address)
- EVaultCampaign.freeze(address)
- EVaultCampaign.getTotalRewardWithdrawn(address)
- EVaultCampaign.calculateRewardDue(address)
- EVaultCampaign.fund()
- EVaultCampaign.claim()
- EVaultCampaign.claimForSender(address)
- EVaultCampaignBudget.fund()
- EVaultCampaigns.freeze()
- EVaultCampaigns.fund()
```

Recommendation(s): Make these functions external.

Status: Fixed

Update from the client: Fixed in commit hash 7107cc5ad957d5491bdb862820653b84cb051b39.

6.29 [Best Practices] Interaction between contracts without using interfaces

File(s): `contracts/EVaultCampaign.sol`

Description: The contract `EVaultCampaign.sol` uses functionalities implemented in the contract `SubscriptionNFT.sol` by importing the whole file. The recommended approach is to define interfaces that should be imported when modules interact.

Recommendation(s): Create an interface for the contract `SubscriptionNFT.sol` and make all access considering the interface.

Status: Fixed

Update from the client: Fixed using `@openzeppelin/contracts/token/ERC1155/IERC1155.sol`.

Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.30 [Best Practices] Use the local variable `activeCampaignAddress` instead accessing the storage twice

File(s): `contracts/EVaultCampaigns.sol`

Description: The function `addCampaign(...)` reads the value of the storage variable `campaigns[activeCampaignId]` twice. The function can be optimized by using the memory variable `activeCampaignAddress` directly. The code snippet with audit comments is reproduced below.

```
function addCampaign(...) external onlyRole(CAMPAIGN_CREATOR_ROLE) {
    address activeCampaignAddress = campaigns[activeCampaignId];
    if (activeCampaignAddress != address(0)) {
        //////////////////////////////////////////////////////////////
        // @audit No need to read the storage again.
        // The content is already saved in "activeCampaignAddress".
    //////////////////////////////////////////////////////////////
        EVaultCampaign activeCampaign = EVaultCampaign(campaigns[activeCampaignId]);
        ...
    }
    ...
}
```

Recommendation(s): Use the `activeCampaignAddress` instead of making a second access to the storage as reproduced below.

```
- EVaultCampaign activeCampaign = EVaultCampaign(campaigns[activeCampaignId]);
+ EVaultCampaign activeCampaign = EVaultCampaign(activeCampaignAddress);
```

Status: Fixed

Update from the client: `activeCampaignAddress` not needed anymore due to fix of **Unnecessary use of a mapping for storing the `campaigns` in `EVaultCampaigns`**. Fixed in commit hash `7107cc5ad957d5491bdb862820653b84cb051b39`.

6.31 [Best Practices] `bool` in storage is more expensive than `uint256`

File(s): `contracts/EVaultCampaign.sol`

Description: Booleans in storage are more expensive than `uint256`, since an extra `SLOAD` operation is required to read the slot content, replace the bit taken up by the boolean, and then write it back. More information can be found here. We reproduce the struct `FrozenAmount` below.

```
struct FrozenAmount {
    uint256 amount;
    uint256 blockNumber;
    uint256 timestamp;
    uint256 rate;
    uint256 withdrawn;
    bool    returned;
}
```

Recommendation(s): Use `uint256` instead of `bool` for the field `returned` as shown below.

```
- bool returned;
+ uint256 returned;
```

Status: Acknowledged

Update from the client: Omitted in favor of readability.

Update from Nethermind: We agree in favoring readability.

7 Documentation Evaluation

The client has provided the high level description of the system and the most important use cases, as reproduced below.

```
1. Consumers and Businesses are allowed to freeze funds on eVault. The amount specified by the user (Frozen amount) is
↪  sent to the smart contract where it is locked for the next 12 months.

1.1. Amounts are frozen in scope of Campaign (not related to marketing campaigns in the wallet). Campaign has Start Time
↪  and Allocated Budget.

1.1.1. Campaign is configured via Backoffice and later created on blockchain.

1.1.2. Campaign cannot be cancelled as it is created on blockchain.

1.1.3. There can't be two Campaigns running in parallel.

1.1.4. Anybody is allowed to extend Allocated Budget of the Campaign. Additional funding always goes to the latest
↪  created campaign even if it hasn't started yet. (to fulfil 1.1.3 and prevent extension of the old campaign after the
↪  new one is already created/scheduled).

1.2. Users are allowed to freeze ECS from their reference addresses as long as the smart contract allows it as many
↪  times as they wish.

1.2.1. By default users are rewarded with 2% in ECS. Users with Gold subscription are rewarded with 3% in ECS. Users
↪  with Platinum subscription are rewarded with 6% in ECS.

1.2.2. Purchasing of Gold or Platinum subscription does not change the reward of the existing Frozen Amounts.

1.2.3. Users with Gold subscription are allowed to vault 1 hour before Campaign start. Users with Platinum - 2 hours
↪  before Campaign starts.

1.2.4. Upon each Frozen Amount the Total Reward is calculated according to the current subscription of the user (if
↪  there is Golden or Platinum NFT on the freezing address). Smart contract controls if the Total Reward exceeds the
↪  available Allocated Budget of the Campaign.

1.3. Users are allowed to claim Due Rewards. Due Reward is calculated by the smart contract ad-hoc and grows with each
↪  blockchain block.

1.3.1. As the frozen amount is ready to be unlocked it can be claimed together with the remaining reward.

Note: ECS is the equivalent of ETH (native currency in our network)
```

8 Test Suite Evaluation

8.1 Contracts Compilation Output

```
$ npx hardhat compile
Compiled 27 Solidity files successfully
```

8.2 Tests Output

```
$ npx hardhat test
  eVault
Campaigns address: 0xfc28330f1ecE0ef2371B724E0D19c1EE60B728b2
Campaign 0 address: 0x446C4263D7332de0A8cb1eC8FF7Ca35f4CF74410
BudgetContract address of campaign 0: 0xd1a99f07df31e3b09C7390Aa819225E71e3f2fE6
Campaigns address: 0x4370ED4d3A3fFb5be6B5ac7a9f0B781f24B092fA
     Should revert funds() and freeze() calls in case no active campaign created (141ms)
Campaigns address: 0x484F2eAccf8FC314e450e9d4EA1fa2CCf073e895
Campaign 0 address: 0x087323E10B52b8a4Ac6634225fEaeC248bb2b4eE
BudgetContract address of campaign 0: 0x8046A759948A904c2EAb70286ab45448A6Fe4648
     Should have campaign contract as owner of budget contract
Campaigns address: 0xaF960E333cbcabF806EB97dbaD153aF9eF915eF9
Campaign 0 address: 0x230b0b40E06F87d1A2E8d19BeDe2094f533100C4
BudgetContract address of campaign 0: 0xC26Ff310C7c2A4822969B4B9a6CaCb8Db7dbcD5B
Budget balance of campaign: 2
     Should allow everybody to fund the budget contract
Campaigns address: 0x12FE8Fa6a3b6364164B52496689D8028B5198368
Campaign 0 address: 0xB698d887A2f54a7980Cba094319195CF8dbE78F0
BudgetContract address of campaign 0: 0xFc7dcBf4c71257A05bAB883086E5306a58B67538
     Should return 0 for getFreezableAmount() before funding
Campaigns address: 0x52E043C9F012d2F74dcd53A2D8C9B3FF80bA842d
Campaign 0 address: 0xDecc49b556d425B6b7FC4d0dFa1E4E807655a8cD
BudgetContract address of campaign 0: 0x806b18C3F7BFb554804E5d945d33017a6d3f4d14
Budget balance: 1000000
     Should return correct amount for getFreezableAmount() after funding
Campaigns address: 0x3b70BD17857a43144B4Be40fb66640CFf9B7C5c6
Campaign 0 address: 0x2398757E8f8beb36c543b1c19F75AD693f11C29b
BudgetContract address of campaign 0: 0xdEC4b743b16A5DbeCbDF34668A1718B61F20F65B
Budget balance: 1000000
Sending 100000 to freeze to campaign...
Frozen funds (campaign): 100000
FreezableAmount (user1): 380000
New budget balance (campaign): 1000000
AvailableBudget (campaign): 978000
Current BlockTimestamp: 1672327491. Forwarding...New BlockTimestamp: 1703863491
rewardDue user1: 102000
     Should allow everbody to freeze (112ms)
Campaigns address: 0x5130A40bf6A13c35d8A7524ff01aaCFd608c3F11
Campaign 0 address: 0xA2ac8A63787E44C99E86789fa4b4B4CF55849b3b
BudgetContract address of campaign 0: 0xfF2c520B25c387cA643738B70B8B24683672361d
Budget balance: 1000000
FreezableAmount (user1): 380000
New budget balance (campaign): 1000000
AvailableBudget (campaign): 978000
Current BlockTimestamp: 1672327498. Forwarding...New BlockTimestamp: 1703863498
rewardDue user1: 102000
     Should allow everbody to freeze via campaigns wrapper contract (103ms)
Campaigns address: 0xfC716b024a84585FaA0999fCd4B17638AE719805
Campaign 0 address: 0xdCd44545fa47f5AeA9A0e905EabD9aEc9143B2D4
BudgetContract address of campaign 0: 0xd43Dfb8c378e6A365a5e0c123353a09ED45e8a3d
Budget balance: 1000000
Sending 100000 to freeze to campaign...
Sending 100000 to freeze to campaign...
Sending 100000 to freeze to campaign...
Frozen funds (campaign): 300000
FreezableAmount (standardUser): 380000
FreezableAmount (goldUser): 380000
FreezableAmount (platinumUser): 380000
```

```
AvailableBudget (campaign): 969000
Current BlockTimestamp: 1672327509. Forwarding...New BlockTimestamp: 1703863509
rewardDue standardUser: 102000
rewardDue goldUser: 103000
rewardDue platinumUser: 106000     Should calculate different rate for SubscriptionNFT holders (275ms)
Campaigns address: 0x74E1798336d5A3094D5dac54A333631174e96C81
Campaign 0 address: 0x3bb5A38cA231CcA594bbEce22ce3CAaF57a2649c
BudgetContract address of campaign 0: 0x17313E39CADa6291307612B3F7B09f270Ee87B27
Budget balance: 1000000
Sending MaxFreezableAmount (500000) to freeze to campaign...
     Should NOT allow an address to freeze more than MaxFreezableAmount (63ms)
Campaigns address: 0x30908509F8B498Dc2A65CBcEAf6e8418E75c3901
Campaign 0 address: 0x174E450D5ff4825507f96B070AA6838d126b5a35
BudgetContract address of campaign 0: 0xAe5905706097797052121 4e38e3919d869e68e61
Budget balance: 1000000
Freezable campaign: 480000000000000000000000
     Should consider MaxFreezableAmount correctly in getFreezableAmount() call
Campaigns address: 0x1A19699907673C22b92A84A1d0053f18E5B5d1F6
Campaign 0 address: 0x3e5F76aE4b7740A3dB7dE823b762f759E8e63CEE
BudgetContract address of campaign 0: 0x33f886ef7BCD5dDB76d83fc1CdD1bA9c2D6d7872
Budget balance: 500000
Freezable campaign: 480000000000000000000000
     Should consider FreezingThresholdLimit correctly in getFreezableAmount() call
Campaigns address: 0xB0e47ffdeD65D6FF8A078b3B0A5b50Bf52451287
Campaign 0 address: 0x3d20d51b29Dc06f7A416C294fe3864a7d34d5bcc
BudgetContract address of campaign 0: 0x334fa8E19E36F66c2F40235974Bd017A66546968
Budget balance: 1000000
Sending 100000 to freeze to campaign...
Frozen funds (campaign): 100000
FreezableAmount (user1): 380000
New budget balance (campaign): 1000000
AvailableBudget (campaign): 978000
Current BlockTimestamp: 1672327533. Forwarding...New BlockTimestamp: 1703863533
rewardDue user1: 102000
Current BlockTimestamp: 1703863533. Forwarding...New BlockTimestamp: 1735399533
rewardDue user1: 102000
     Should not produce additional yield after 1 year (119ms)
Campaigns address: 0xcf761682be08844792E0565b31468Ac9979Bc25A
Campaign 0 address: 0x393Db85B803b3a615f8cC38084522F578a784A6b
BudgetContract address of campaign 0: 0x19C339Bbb9ddB25dAB4488EB7257E89bde218eB2
Budget balance: 30000
AvailableBudget (campaign): 10000
Pure Budget: 30000
Sending 100000 to freeze for user1 to campaign...
AvailableBudget (campaign): 7999.999999999999
freezable: 399999
Pure Budget: 28000
Sending 100000 to freeze for user2 to campaign...
AvailableBudget (campaign): 5999.999999999998
Pure Budget: 26000
Sending 100000 to freeze for user3 to campaign...
AvailableBudget (campaign): 3999.9999999999973
Pure Budget: 23999.999999999996
Sending 100000 to freeze for user4 to campaign...
AvailableBudget (campaign): 1999.9999999999966
Pure Budget: 21999.999999999996
Sending 100000 to freeze for user5 to campaign...
AvailableBudget (campaign): 0
Pure Budget: 19999.999999999996
Sending 100000 to freeze for user6 to campaign (should succeed even availableBudget == 0 due to freezingThresholdLimit
↪ buffer)...
AvailableBudget (campaign): 0
Pure Budget: 17999.999999999996
Pure Budget: 1999.9999999999882
Frozen funds (campaign): 1400000
AvailableBudget (campaign): 0
Add additional 30000 to campaign to allow user6 to freeze too!
AvailableBudget (campaign): 11999.999999999987
Pure Budget: 31999.99999999999
     Should consider FreezingThresholdLimit and NOT allow freeze if no budget left (653ms)
```

```
Campaigns address: 0x9C4710178FDDbC261fd22232C96488bf7d1A319c
Campaign 0 address: 0x7f20eC3CdF31970635227671198E6EE6031bC820
BudgetContract address of campaign 0: 0xcaee103a2ceBbF5BA9395dEa319119F8efb9Aa87
Budget balance: 100000
Freezing on block 106
Freezing on block 107
Freezing on block 108
Freezing on block 109
Freezing on block 110
Freezing on block 111
Freezing on block 112
Freezing on block 113
Freezing on block 114
Freezing on block 115
Freezing on block 116
Freezing on block 117
Freezing on block 118
Freezing on block 119
Freezing on block 120
Freezing on block 121
Freezing on block 122
Freezing on block 123
Freezing on block 124
Freezing on block 125
Freezing on block 126
Freezing on block 127
Freezing on block 128
Freezing on block 129
Freezing on block 130
Freezing on block 131
Freezing on block 132
Freezing on block 133
Freezing on block 134
Freezing on block 135
Freezing on block 136
Freezing on block 137
Freezing on block 138
Freezing on block 139
Freezing on block 140
Freezing on block 141
Freezing on block 142
Freezing on block 143
Freezing on block 144
Freezing on block 145
Freezing on block 146
Freezing on block 147
Freezing on block 148
Freezing on block 149
Freezing on block 150
Freezing on block 151
Freezing on block 152
Freezing on block 153
Freezing on block 154
Freezing on block 155
Frozen funds (campaign): 50000
Claimable before: 776889903602209
Current BlockTimestamp: 1672327612. Forwarding...New BlockTimestamp: 1703863612
Claimable after 1 year: 51000000000000000425600
     Should emit all events successfully for many freezings upon claim() (4088ms)
Campaigns address: 0x0146E03CEEdd8295e52ffCee2E254fbE33CD7E83
Campaign 0 address: 0xeC10bD45c2497b1eF5ffA70117e42Ab724af51d4
BudgetContract address of campaign 0: 0xFC9175679efaC2345dCdc9ccAe41de3cfb3ae654
Budget balance: 1000000
Sending 100000 to freeze to campaign1...
Frozen funds (campaign1): 100000
New budget balance (campaign1): 1000000
AvailableBudget (campaign1): 978000
Total eligible rewards for user per year: 20000000000000000000000
Current BlockTimestamp: 1672327620. Forwarding...New BlockTimestamp: 1672327621
rewardDue user1 (1 SECOND): 634195839675 29

Current BlockTimestamp: 1672327621. Forwarding...New BlockTimestamp: 1672327625
rewardDue user1 (1 BLOCK): 317097919837646
```

```
Current BlockTimestamp: 1672327625. Forwarding...New BlockTimestamp: 1672327680
rewardDue user1 (1 MIN): 3805175038051752

Current BlockTimestamp: 1672327680. Forwarding...New BlockTimestamp: 1672331220
rewardDue user1 (1 HOUR): 228310502283105120

Current BlockTimestamp: 1672331220. Forwarding...New BlockTimestamp: 1672414020
rewardDue user1 (1 DAY): 5479452054794522880

Current BlockTimestamp: 1672414020. Forwarding...New BlockTimestamp: 1672932420
rewardDue user1 (1 WEEK): 38356164383561660160

Current BlockTimestamp: 1672932420. Forwarding...New BlockTimestamp: 1703863620
rewardDue user1 (1 YEAR): 102000000000000000851200

Current BlockTimestamp: 1703863620. Forwarding...New BlockTimestamp: 1735399620
rewardDue user1 (2 YEARS): 102000000000000000851200

     Should correctly calculate rewards for different freezing periods (240ms)
Campaigns address: 0x1DFa4C05aB0f6C19DC2E99f608dADde5BBFF772F
Campaign 0 address: 0xd1F84ad362fA8bA489b5255F93CEef18d730D52D
BudgetContract address of campaign 0: 0xf8517E9929223dAd1861554962e487fBC17f8aa7
Budget balance: 1000000
Sending 100000 to freeze to campaign1...
Frozen funds (campaign1): 100000
New budget balance (campaign1): 1000000
Total eligible rewards for user per month: 38.46153846153846
Current BlockTimestamp: 1672327634. Forwarding...New BlockTimestamp: 1672932434
rewardDue user1 (1 WEEK): 38356164383561660160

Balance user1 before claim: 8600998.978759961

Balance user1 after claim: 8601037.334806815

     Should correctly send rewards upon claiming for different freezing periods (MONTH) (115ms)
Campaigns address: 0x77CCee4bD24111BE39DAF3AfD757F0fc9bD60b3D
Campaign 0 address: 0xCd43568029C852A6D17fa18A4FB7228c63a2D5b3
BudgetContract address of campaign 0: 0x2EF8cfDf0f23106A4503FE34B0a3Ee8c5526b341
Budget balance: 1000000
Sending 100000 to freeze to campaign1...
User1 balance after freezing: 8501037334591303701080641
Frozen funds (campaign1): 100000
Budget funds (campaign1): 1000000
Current BlockTimestamp: 1672327642. Forwarding...New BlockTimestamp: 1672414042
rewardDue user1 (1 DAY): 5479452054794522880

User1 balance after claiming: 8501042813925828494663281
Frozen funds after claim (campaign1): 100000
Budget funds after claim (campaign1): 999994.5205479452
rewardDue user1 (after claim): 0

Current BlockTimestamp: 1672414042. Forwarding...New BlockTimestamp: 1672500442
rewardDue user1 (after first claim, before next claim): 5479452054794522880

     Should allow claiming the rewards (137ms)
Campaigns address: 0x77518F1EaE1429f3D6714bF75e8574F1c95976c7
Campaign 0 address: 0x09F1b1a6134f4e95dBC12aCdC225C755B949a22b
BudgetContract address of campaign 0: 0x521c82fdD2917022808f29291580996116D60E3D
Budget balance: 1000000
Sending 100000 to freeze to campaign1...
User1 balance after freezing: 8401042813710317492939193
Frozen funds (campaign1): 100000
Budget funds (campaign1): 1000000
Current BlockTimestamp: 1672327651. Forwarding...New BlockTimestamp: 1672414051
rewardDue user1 (1 DAY): 5479452054794522880

User1 balance after claim 1: 8401048293044842286521833
Frozen funds after claim 1 (campaign1): 100000
Budget funds after claim 1 (campaign1): 999994.5205479452
Current BlockTimestamp: 1672414051. Forwarding...New BlockTimestamp: 1672500451
rewardDue user1 (before claim 2): 5479452054794522880
```

```
User1 balance after claim 2: 8401053772413567080378073
Frozen funds after claim 2 (campaign1): 100000
Budget funds after claim 2 (campaign1): 999989.0410958905
     Should calculate multiple claims correctly (157ms)
Campaigns address: 0x2E0C5bc81785D5566971400061ed9930E699D5ED
Campaign 0 address: 0xD98B9442501a225F6d42ff797Bf9FB5F55582E47
BudgetContract address of campaign 0: 0xbB541Faec517302e74e56A3bc2E7B9E4E0F7C999
Budget balance: 1000000
rewardDue user1 (1 DAY): 0

Frozen funds after claim (campaign1): 0
Budget funds after claim (campaign1): 1000000
     Should allow 0 claim without any freezings
Campaigns address: 0xE31192eBb636d730F9906d1eEF7A439BC399EB93
Campaign 0 address: 0x4C3766118d65205F0a4D5573a3edDdfAbeFcA609
BudgetContract address of campaign 0: 0x1DbFfeE8b283ac3766873306d16687B43B36D6F1
     Should not allow non-owners to call reserve() on budget contract
Campaigns address: 0x17C71F0c51897F39Ea6621ea8Bd22D71a51f5fe3
Campaign 0 address: 0xCA07F92Be8c94C36D60B6d618c5cB4Bf7e2cE499
BudgetContract address of campaign 0: 0x6E5bAD88fC96b62Bd7B177F2aa56Dfe18d111ced
     Should not allow non-owners to call claim() on campaign contract
Campaigns address: 0x8a69880332C758502700dFF47e146C224475cf83
Campaign 0 address: 0xa05191A59D39FeE04CDDA1acD22E325B405aEBa5
BudgetContract address of campaign 0: 0x436460172Df3D2ebe6e90e3d0A7857Ac2bcA8111
     Should not allow non-owners to call claimForSender() on campaign contract
Campaigns address: 0x871AA5904A5d6c1DD2a75a7E623E1fc2b3D78Ed6
Campaign 0 address: 0xFC2abB2cF9BaD95524cD918ACF858803BC613882
BudgetContract address of campaign 0: 0x9890609219833E5f7E44fAe095fBEA87d2afdD92
     Should not allow non-owners to call claimRewards() on budget contract
Campaigns address: 0x2aB63470ee423A47F5EC0398bB6035cd8BfD4005
Campaign 0 address: 0x9D1A2271ecFe303102B1e6d06A3F6E604107ebE4
BudgetContract address of campaign 0: 0xf68E2C8e9E638fbE94e4c7ccd9870D0f99fe91de
Budget balance: 1000000
Sending 100000 to freeze to campaign...
Freezing block: 231
Current BlockTimestamp: 1672327687. Forwarding...New BlockTimestamp: 1703863687
rewardDue user1: 102000
     Should return freezings after 1 year automatically (168ms)
Campaigns address: 0xE0A7D91cF936797a5408bFab9E6359c7736bA378
Campaign 0 address: 0xC78B34b813C3dB1e7B39e1389CFa430D020135f2
BudgetContract address of campaign 0: 0xAC1d7c9C00a8cf047c769CB25484a6762Ccbb3B5
Budget balance: 1000000
Sending 100000 to freeze to campaign...
Freezing block: 239
Campaign funds after freeze (campaign1): 100000
Current BlockTimestamp: 1672327695. Forwarding...New BlockTimestamp: 1703863695
rewardDue user1: 102000
Campaign funds after claim (campaign1): 0
Budget balance (campaign1): 998000
Current BlockTimestamp: 1703863695. Forwarding...New BlockTimestamp: 1704468495
Balance user1 before claim: 8405053.771565337

Balance user1 after claim: 8405053.77150783

     Should give rewards and unlocked amount only once if claimed twice after 1 year (191ms)
Campaigns address: 0x3d95769B4e17938c0b6248E33E94da947CB3c976
Campaign 0 address: 0x559084f355DD326efa9E8CEc7906F1Ef43B1233D
BudgetContract address of campaign 0: 0xe06bB3cdd5bEFd31ECB7A66d14a88aE009711a35
Budget balance: 1000000
Sending 100000 to freeze to campaign...
Freezing block1: 249
Current BlockTimestamp: 1672327705. Forwarding...New BlockTimestamp: 1672932505
rewardDue user1 (block1): 38.35616438356166
Sending 100000 to freeze to campaign...
Freezing block2: 252
Current BlockTimestamp: 1672932505. Forwarding...New BlockTimestamp: 1673537305
Sending 100000 to freeze to campaign...
Freezing block3: 254
     Should return correct values for getFreezeInfo() (242ms)
Campaigns address: 0x485727F6C4b14791F009587e57e2bb88E4Bc09C1
Campaign 0 address: 0x75f7294b612821AAe7b1db94b5c8005Cc7394205
BudgetContract address of campaign 0: 0x6364153B93d9bEb110e799350ebB8Ea334fB3F65
Budget balance: 1000000
     Should prevent multiple campaigns running in parallel as long as there is budget on previous campaign (41ms)
```

```
Campaigns address: 0x04869fe8E3E9e8c192fa8d388F5dFCF4D8454d2c
Campaign 0 address: 0x9223ae8cbB7805D6EE4bB19705d860Bb362F3f92
BudgetContract address of campaign 0: 0xd3a4f9fAA01271c58267483cE670085c08F21e85
Budget balance: 21000
     Should allow a new campaign if old one ran out of budget (98ms)
Campaigns address: 0x5530996c6059792317Df2ec0ee446575A3f1a390
Campaign 0 address: 0x96C88F66F4bC15eF7a412b18096961C35b566eFe
BudgetContract address of campaign 0: 0xbA1394afC5b3a0e0c5FBe28ed24129c4136E78f8
Budget balance: 20002
Freezable campaign1: 99000000000000000000
Freezing...
Freezable campaign1 new: 0
Current BlockTimestamp: 1672327730. Forwarding...New BlockTimestamp: 1672932530
Claimable 1 user1: 37972602739726043
Current BlockTimestamp: 1672932530. Forwarding...New BlockTimestamp: 1673537330
Claimable 2 user1: 37972602739726044
availableFreezingBudget: 19999999999999158
Campaign2 address: 0x16E2481982a519467EB6676b803Fe97E0fA5C5fe
Campaign2 id: 1
Budget balance 0x16E2481982a519467EB6676b803Fe97E0fA5C5fe: 30000
Freezable campaign2: 499999000000000000000000
Current BlockTimestamp: 1673537330. Forwarding...New BlockTimestamp: 1674142130
Claimable 3 user1: 76262302765093893 (0.0762623027650939 ECS)
User balance before claim: 7554994164163920364373612
User balance after claim: 7554994240565986413690639
User balance diff: +0.07640206604931703 ECS
Total frozen: 500098 ECS
     Should calculate multiple freezings over multiple campaigns correctly (357ms)
Campaigns address: 0x1Ed23C8C0696A84B2b4D43f714eA44ec3c202A89
Campaign 0 address: 0x88712eBBD93FC5ce91044D665926A4434bfeE9c4
BudgetContract address of campaign 0: 0xDF09d3ba82B6933b762b6EF8494c523371d4A3D9
Budget balance: 1000000
     Should prevent freezings of same sender in same block (108ms)
Campaigns address: 0x8Ea6578E8f1d08Da0e53931C258e8f19986D4185
Campaign 0 address: 0xA65C5DD6280cEE97E44eB49369525b4Dd4C28914
BudgetContract address of campaign 0: 0xdd076bC82076032AfD23c5A61b5Adf6986796DA9
Budget balance: 1000000
     Should allow freezings of different senders in same block (95ms)
Campaigns address: 0xaEe4529Fb8a27DDD7a2f6d80F282A39353Faa180
Campaign 0 address: 0xf8a971268d045b2065dd6474B117524baED08902
BudgetContract address of campaign 0: 0x5c9A259a4b61dB2Bb73B7c38581F4103Ab3d52F9
Campaigns address: 0xf7c07ebB9C70852a1f432AACFe9F3f1F850FB66D
Freezing period: 15 sec
Campaign 0 address: 0x03F19E964fe573d8789deb6eEEa0bc7e86511788
BudgetContract address of campaign: 0x11603e44F724B8c299EBE3C9e54c6329A31F9AcA
Budget balance: 500000
Current block height: 304
Current block height: 308
Claimable user1: 634195839675290 (0.00063419583967529 ECS)
User balance before claim: 6754994238859235400036631
User balance after claim: 6754994239537965157369055
User balance diff: +0.000678729757332424 ECS
     test claim with reduced locking (384ms)

  SubscriptionNFT contract
    1) Should set the owner correctly
    mint
      2) Non-owner of the contract cannot mint
       User receives Gold token
       User cannot mint Gold token twice (56ms)
       User receives Platinum token
       User cannot mint Platinum token twice (80ms)
       User with Gold token receives Platinum token (86ms)
       User with Platinum token cannot mint Gold token (51ms)
       Tokens cannot be transfered (68ms)


  39 passing (16s)
  2 failing
```

```
1) SubscriptionNFT contract
     Should set the owner correctly:
   TypeError: tokenContract.owner is not a function
    at Context.<anonymous> (test/SubscriptionNFT.test.js:19:32)
    at runMicrotasks (<anonymous>)
    at processTicksAndRejections (internal/process/task_queues.js:95:5)
    at runNextTicks (internal/process/task_queues.js:64:3)
    at listOnTimeout (internal/timers.js:526:9)
    at processTimers (internal/timers.js:500:7)

2) SubscriptionNFT contract
     mint
       Non-owner of the contract cannot mint:
   AssertionError: Expected transaction to be reverted with Ownable: caller is not the owner, but other exception was
   ↪  thrown: Error: VM Exception while processing transaction: reverted with reason string 'AccessControl: account
   ↪  0xb04a130920446963efaaf51c4a9ba7d471a2c428 is missing role
   ↪  0x9f2df0fed2c77648de5860a4cc508cd0818c85b8b8a1ab4ceeef8d981c8956a6'
```

8.3 Code Coverage

```
npx hardhat coverage
```

The output is presented below.

```
--------------------------------------------|----------|----------|----------|----------|----------------|
File                                        |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
--------------------------------------------|----------|----------|----------|----------|----------------|
 contracts/                                 |    89.31 |    80.23 |    80.82 |    88.72 |                |
  EVaultCampaign.sol                        |    97.37 |      100 |     91.3 |    97.87 |         71,278 |
  EVaultCampaignBudget.sol                  |       75 |      100 |       80 |    83.33 |             23 |
  EVaultCampaigns.sol                       |    90.91 |    83.33 |    84.62 |     92.5 |       51,52,57 |
  SubscriptionNFT.sol                       |    70.37 |    56.67 |    57.14 |    64.52 |... 89,91,92,96 |
  TestEVaultCampaign.sol                    |     87.5 |       75 |    85.71 |       80 |          20,28 |
  TestEVaultCampaignWithReducedLocking.sol  |    66.67 |       50 |    66.67 |     62.5 |       26,34,49 |
  TestEVaultCampaigns.sol                   |      100 |      100 |      100 |      100 |                |
  TestEVaultCampaignsWithReducedLocking.sol |      100 |      100 |      100 |      100 |                |
--------------------------------------------|----------|----------|----------|----------|----------------|
All files                                   |    89.31 |    80.23 |    80.82 |    88.72 |                |
--------------------------------------------|----------|----------|----------|----------|----------------|
```

8.4 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

9 About Nethermind

Founded in 2017 by a small team of world-class technologists, Nethermind builds Ethereum solutions for developers and enterprises. Boosted by a grant from the Ethereum Foundation in August 2018, our team has worked tirelessly to deliver the fastest Ethereum client in the market. Our flagship Ethereum client is all about performance and flexibility. Built on .NET core, a widespread, enterprise-friendly platform, Nethermind makes integration with existing infrastructures simple, without losing sight of stability, reliability, data integrity, and security. **Nethermind is made up of several engineering teams across various disciplines, all collaborating to realize the Ethereum roadmap, by conducting research and building high-quality tools**. Teams focus on specific areas of the Ethereum problem space. Each consists of specialists and experienced developers working alongside interns, learning the ropes in the Nethermind Internship Program. **Our mission is to gather passionate talent from around the world, and to tackle some of the blockchain's most complex problems**. Nethermind provides software solutions and services for developers and enterprises building the Ethereum ecosystem. We offer security reviews to projects built on EVM compatible chains and StarkNet. We have expertise in multiple areas of the Ethereum ecosystem, including protocol design, smart contracts (written in Solidity and Cairo), MEV, etc. We develop some of the most used tools on Starknet and one of the most used Ethereum clients. Learn more about us at `https://nethermind.io`.

Disclaimer: This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.