Symbioticfi core

Competition

December 9, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Symbiotic is a shared security protocol that serves as a thin coordination layer, empowering network builders to control and adapt their own (re)staking implementation in a permissionless manner.

From Sep 11th to Oct 2nd Cantina hosted a competition based on symbioticfi-core. The participants identified a total of **39** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 4
- Low Risk: 26
- Gas Optimizations: 0
- Informational: 9

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 Slashable amounts are calculated wrong and its possible for depositors to abuse it

Submitted by deadrosesxyz, also found by cryptomoon, Audinarey, armormadeofwoe, typicalHuman, pwnforce, Jiri123, 0xNirix, ktl, rilwan99, Flint, 0xleadwizard, heeze, ni8mare and trachev

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The way a vault's slashable amount is calculated is by getting the active stake at a certain timestamp and from it to subtract any slashed amounts that have happened since. However, this does not work as expected in case withdraws have been made in the time between `captureTimestamp` and timestamp of slashing.

To showcase the issue:

1. Consider an active stake of 100 at timestamp `T`.
2. At timestamp `T + 1 days` `VetoSlasher` requests a slash with `captureTimestamp == T` for 50.
3. Then, users withdraw 50 from the active stake (this could both be intentional if they're aware of the issue or unintentional if they're not).
4. At timestamp `T + 5 days` the slashing gets executed. The slashing cumulative is increased at timestamp `T + 5 days`.
5. Now, if any slashing is requested (or has been already requested) for the 4-day time period of `T + 1 days -> T + 5 days`, the slashable amount returned will be calculated as follows: `activeStakeAt - cumulativeSlashableDelta = 50 - 50 = 0`.
6. Despite the fact that there's been non-slashed 50 active stake during that 4-day time period, the post-slashing 25 active stake and 25 sitting in withdraw request are impossible to be slashed.

The issue always occurs when there's been withdraw requests made during the time period between `captureTimestamp` and slash execution.

In the cases where `VetoSlasher` is used, as soon as users see a slash requested, they. can create withdraw requests to protect the rest of their funds. If there's a slash for 50%+, they can fully protect the rest of their funds.

Recommendation: Fix is non-trivial.

Symbiotic: Symbiotic core contracts are a very basic and thin layer that allows networks to bootstrap their development and deployment process without almost any limitations. However, its thinness also leads to the need for the networks to fully understand all the aspects of its workflow (and further process them accordingly) to create a truly secure system.

Therefore, we are expanding our documentation (e.g., see networks' flow) and are developing an on-chain SDK to handle as much of the integration hassle as possible for the networks.

3.1.2 Vetoslashing might fail

Submitted by mxuse, also found by cryptomoon, Silvermist, 0xNirix, Anirruth, yttriumzz, vinicaboy, 0xaman, 0xG0P1, flacko, DemoreXTess, DemoreXTess, 4gontuk, ruhum, 1337web3, Shubham, Greed, 0xlookman and Chinmay Farkya

Severity: Medium Risk

Context: VetoSlasher.sol#L80

Description: As per the sponsor:

> `captureTimestamp` may be performed several times per Vault epoch (depends on the Network epoch - let it be a duration during which a captured validator set is used).

Let's assume `requestSlash()` is called to slash Bob with a specific `captureTimestamp`, say `captureTimestamp = 1000`. Within the function, the timestamp will be stored in the request as follows:

```
function requestSlash(
    bytes32 subnetwork,
    address operator,
    uint256 amount,
    uint48 captureTimestamp,
    bytes calldata hints
) external initialized nonReentrant onlyNetworkMiddleware(subnetwork) returns (uint256 slashIndex) {

    slashIndex = slashRequests.length;
    slashRequests.push(
        SlashRequest({
            subnetwork: subnetwork,
            operator: operator,
            amount: amount,
            captureTimestamp: captureTimestamp, // <<<
            vetoDeadline: vetoDeadline,
            completed: false
        })
    );

    emit RequestSlash(slashIndex, subnetwork, operator, amount, captureTimestamp, vetoDeadline);
}
```

At this point, Bob's request is pending. Next, a `requestSlash()` call is made for Alice in the same epoch, but this time with a new `captureTimestamp` of `1200`. When `executeSlash()` is called for Alice, it updates the `latestSlashedCaptureTimestamp` for the subnetwork to `1200`. Later, `executeSlash` is called for Bob, but this happens after Alice's slash has already been executed. This could be because Bob's request took longer, as it was not vetoed and is now being executed close to the end of the allowed duration.

Upon entering `executeSlash()` for Bob, various checks are performed, including the following:

```
function _checkLatestSlashedCaptureTimestamp(bytes32 subnetwork, uint48 captureTimestamp) internal view {
    if (captureTimestamp < latestSlashedCaptureTimestamp[subnetwork]) {
        revert OutdatedCaptureTimestamp();
    }
}
```

In this case, `captureTimestamp` will be `1000` since `request.captureTimestamp` is used, but the `latestSlashedCaptureTimestamp` has already been updated to `1200` after Alice's slash. As a result, this check will fail, and Bob's slash will not be processed leaving Bob unslashed.

Recommendation: Handle the logic to prevent this from happening.

Symbiotic: Generally, the given sequencing restriction shouldn't cause any problems. However,

1. We've weakened the restriction from the network level to the lower operator level so that it gives more flexibility for the networks without any additional risks for stakeholders or operators (see commit c3853546).
2. We are aware that even after the code changes, such a situation still may exist in case of weak slashing requests' validation on the networks' middleware side. Therefore, we've added such information to our risks mitigation manual for the networks. We will continue enhancing the networks' security directly via work-in-progress on-chain SDK and indirectly via the documentation.

3.1.3 `upperLookupRecentCheckpoint` may revert due to access of index out of bounds

Submitted by cryptomoon, also found by Nexarion, 0xiceman, ni8mare and 0xpiken

Severity: Medium Risk

Context: FullRestakeDelegator.sol#L29-L32, FullRestakeDelegator.sol#L56, FullRestakeDelegator.sol#L77, NetworkRestakeDelegator.sol#L58, NetworkRestakeDelegator.sol#L78, NetworkRestakeDelegator.sol#L99, Checkpoints.sol#L286, Checkpoints.sol#L330-L333, BaseSlasher.sol#L44, BaseSlasher.sol#L73, VaultStorage.sol#L117-L121, VaultStorage.sol#L176, VaultStorage.sol#L190, VaultStorage.sol#L204

Description: The function `upperLookupRecentCheckpoint(Trace256 storage self, uint48 key, bytes memory hint_)` may revert when some valid `hint_` is provided because it will try to access the element in `Trace256._values` array which will most likely more than length of array.

The function `upperLookupRecentCheckpoint(Trace256 storage self, uint48 key, bytes memory hint_)` is a variant of {upperLookupRecentCheckpoint} that can be optimized by getting the hint. It contains following snippet in it's implementation:

```
uint32 hint = abi.decode(hint_, (uint32));
Checkpoint256 memory checkpoint = at(self, hint);
if (checkpoint._key == key) {
    // @audit will revert when accessing self._values[checkpoint._value]
    return (true, checkpoint._key, self._values[checkpoint._value], hint);
}
if (checkpoint._key < key && (hint == length(self) - 1 || at(self, hint + 1)._key > key)) {
    // @audit will revert when accessing self._values[checkpoint._value]
    return (true, checkpoint._key, self._values[checkpoint._value], hint);
}
```

The above implementation will most likely revert in the cases where it goes in any of the above mentioned `if` blocks. To understand reason for revert, we need to look at `function at`:

```
struct Checkpoint256 {
    uint48 _key;
    uint256 _value;
}

function at(Trace256 storage self, uint32 pos) internal view returns (Checkpoint256 memory) {
    OZCheckpoints.Checkpoint208 memory checkpoint = self._trace.at(pos);
    return Checkpoint256({_key: checkpoint._key, _value: self._values[checkpoint._value]});
}
```

As we can see in the implementation of function `at`, it returns `key` as `checkpoint._key` and `value` as `self._values[checkpoint._value]`. As we can see, the `value` returned is already accessed from `Trace256._values` array. In `upperLookupRecentCheckpoint` function, it access following:

```
Checkpoint256 memory checkpoint = at(self, hint);
self._values[checkpoint._value]
```

`checkpoint._value` is the `value` corresponding to the `checkpoint` but it again tries to access it from `self._values` array. This will revert when `checkpoint._value` is more than or equal to `self._values.length`. It will most likely always be the case.

Impact: All the functions using `upperLookupRecentCheckpoint(Trace256 storage self, uint48 key, bytes memory hint_)` will revert when valid `hint_` is passed. This breaks the core functionality of the protocol.

Likelihood: This library function will break when valid `hint_` is passed. `hint_` is expected to be sent most of the times for optimizing search.

Recommendation: Update the vulnerable implementation to following fix:

```
uint32 hint = abi.decode(hint_, (uint32));
Checkpoint256 memory checkpoint = at(self, hint);
if (checkpoint._key == key) {
    // @fix
    return (true, checkpoint._key, checkpoint._value, hint);
}
if (checkpoint._key < key && (hint == length(self) - 1 || at(self, hint + 1)._key > key)) {
    // @fix
    return (true, checkpoint._key, checkpoint._value, hint);
}
```

Symbiotic: We agree that the issue existed; however, it didn't cause any security breaches in the core contracts, as the vulnerable function isn't used there in any way (it is a helper function for off-chain needs). Now, it is fixed (see commit e6a46bfb).

3.1.4 Potential Over-Slashing in Multi-Operator and Multi-Subnetwork Scenarios

Submitted by 0xNirix, also found by 0xCiphky and 0xleadwizard

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The current implementation of slashable stake calculation in the `BaseSlasher` contract does not adequately account for cumulative slashes across multiple operators within a subnetwork, nor does it consider the implications across multiple subnetworks. This oversight could lead to over-slashing. It violates a key protocol logic that the slashable amount at a slashing capture time does not exceed the active stake at that capture time.

In the `BaseSlasher` contract, the `slashableStake` function calculates the amount of stake that can be slashed for a specific operator. However, it only considers the cumulative slash amount for that individual operator, without taking into account the total amount slashed across all operators in the subnetwork and across all the subnetworks. The current implementation:

```
function slashableStake(
    bytes32 subnetwork,
    address operator,
    uint48 captureTimestamp,
    bytes memory hints
) public view returns (uint256) {
    // ... [input validation] ...

    uint256 stakeAmount = IBaseDelegator(IVault(vault).delegator()).stakeAt(
        subnetwork, operator, captureTimestamp, slashableStakeHints.stakeHints
    );
    return stakeAmount
        - Math.min(
            cumulativeSlash(subnetwork, operator)
                - cumulativeSlashAt(subnetwork, operator, captureTimestamp,
↪   slashableStakeHints.cumulativeSlashFromHint),
            stakeAmount
        );
}
```

This approach fails to consider the cumulative slashed amount across all operators and subnetworks. As a result, the calculated slashable amount may be higher than it should be if the total slashed amount for all operators and subnetworks were properly accounted for, potentially leading to excessive slashing. On the vault, this over slashing can actually be accomplished, as during slashing the entire active stake including new deposits are considered.

Impact: The impact of this issue is significant, as it would lead to over-slashing.

Likelihood: The likelihood of this issue occurring is moderate to high, in any scenario where multiple operators and subnetworks are present, this issue could manifest

Proof of Concept: Consider the following scenario:

1. At T_0, a subnetwork has 1000 tokens total stake.
2. At T_1, a slashing event occurs for Operator A:
    - 500 tokens are slashed.
    - The contract updates Operator A's cumulative slash to 500 tokens.
    - The total stake in the subnetwork is now 500 tokens.
3. At T_2, new deposits increase the total stake to 1000 tokens again.
4. At T_3, another slashing event occurs, this time for Operator B:
    - The contract only considers Operator B's individual cumulative slash (which is 0).
    - It doesn't account for the 500 tokens already slashed from the subnetwork due to Operator A's slashing.
    - The contract calculates Operator B's slashable amount based on the current total stake of 1000 tokens.

5. The issue:
    - The contract allows slashing of up to 1000 tokens for Operator B.
    - This could bring the total slashed amount to 1500 tokens (500 from Operator A + up to 1000 from Operator B).
    - However, only 500 tokens of the original stake should have been available for slashing.
    - The extra slashable amount is effectively including the new deposits, which should not have been subject to slashing based on the earlier event.

This scenario demonstrates how the current implementation fails to consider the subnetwork's total slashed amount across all operators, potentially leading to over-slashing and incorrectly including newer deposits in the slashable amount deliberately. For simplicity, above example shows multi-operators within a subnetwork, however a similar issue is present due to not considering multiple subnetworks.

Recommendation: Implement a network-wide cumulative slash tracker.

Symbiotic: Symbiotic is a highly flexible protocol that allows the creation of an extensive set of products/strategies on top of it. We aim to provide as much freedom to integrators as possible without breaking the only rule -- a network must be able to slash a captured stake during one vault epoch after the capturing timestamp. Therefore, leading the contracts' state into a highly risky configuration without proper management/integration is also possible. We are aware of it and are working on educational resources like documentation (e.g., ready-to-go vault deployment guide or risk mitigation manuals and blog posts).