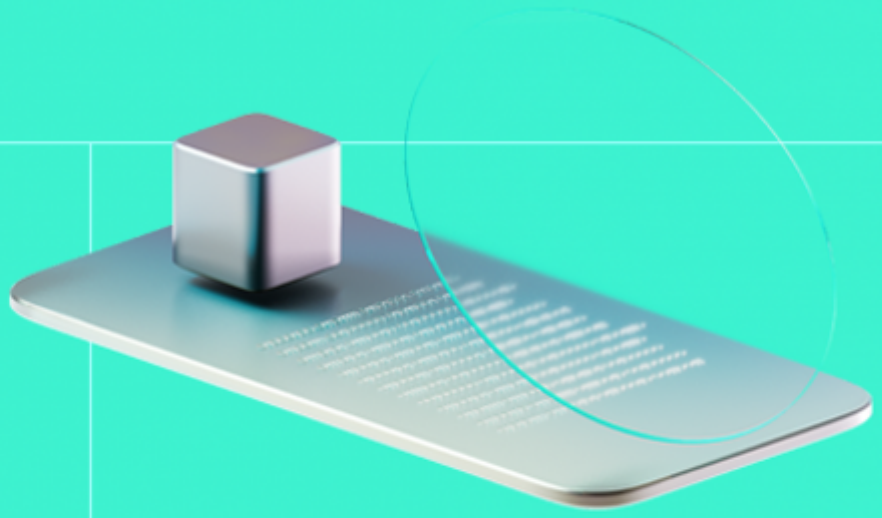
HACKEN

Smart Contract Code Review And Security Analysis Report

Customer: ENKI

Date: 12/03/2024



We express our gratitude to the Enki team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

ENKI is an blockchain ecosystem that can distribute rewards to its users on both chains. It combines ERC4626 standart with the Vesting opportunities. Users holding Staked ENKI Metis receive their rewards in eMetis, maintaining consistency and simplicity in the reward cycle. The vesting mechanism, tied to the ENKI token, encourages sustained engagement and investment in the ecosystem, aligning user incentives with the long-term success of ENKI.

Platform: Ethereum (L1), Metis Andromeda (L2)

Language: Solidity

Tags: Multi-chain, Vesting, Incentive

Timeline: 20/02/2024 - 12/03/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts
Commit	934933f
Remediation Commit	addd1fd

Audit Summary

10/10	9/10	100.0%	6/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.4/10

The system users should acknowledge all the risks summed up in the risks section of the report

7	6	1	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	1
Medium	1
Low	5

Vulnerability	Status
F-2024-1068 - Risk of centralization due to authority of Timelock and Operator roles in Vesting settings	Accepted
F-2024-1063 - Price manipulation on the first liquidation	Fixed
F-2024-1064 - Unlimited approval granted to unrelated contract	Fixed
F-2024-1066 - Timelock and Operator roles can renounce their permissions	Fixed
F-2024-1071 - Owner can renounce its ownership	Fixed
F-2024-1079 - `Dealer.lockFor()` and `Dealer.setActive()` functions do not check previous agent entries	Fixed
F-2024-1101 - The contract managers are able to withdraw tokens from the contracts	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for ENKI
Audited By	Ataberk Yavuzer, Vladyslav Khomenko
Approved By	Grzegorz Trawinski
Website	https://www.enkixyz.com/
Changelog	28/02/2024 - Preliminary Report
	11/03/2024 - Final Report

Table of Contents

System Overview

ENKI Protocol is a blockchain ecosystem with the following contracts:

ENKI Metis (eMetis) — an ERC-20 token, pegged 1:1 to Metis tokens circulating in ENKI system. It supports gasless transfers.
ENKI Metis Minter — contract that allows the exchange of Metis for eMetis. Exchanging the other way is currently disabled due to outside factors (restrictions in the Metis Sequencer).
Staked ENKI Metis (seMetis) — a token of an ERC4626 vault. Staking eMetis in it lets users earn rewards in form of eMetis for participating in ENKI system. Unstaking eMetis is done in 2 steps: part of the tokens are released immediately, other part is temporarily locked in vesting contract.
Vesting — this contract holds eMetis and postpones its withdrawal to ensure participants of the ENKI system are committed to it. To withdraw eMetis users have to deposit ENKI coins and wait for vesting period to pass. eMetis is then unlocked gradually.
ENKI — utility token of the ENKI protocol, used to provide access to the eMetis in the Vesting contract.
Rewards Dispatcher — this contract distributes rewards for participating in ENKI system in form of eMetis. The protocol takes it's cut of the yield and sends the rest to seMetis vault for users to claim.
Config — common contract that stores configuration for the whole system.
Dealer and **SequenserAgent** — contracts that manage the layer 1 part of the system.

Privileged roles

Owner & Timelock:

- Grant timelock and eMetis minter roles.

Timelock & Operator:

- Grant and revoke various roles and capabilities.
- Pause and unpause the protocol.
- Make protocol public and vice versa.
- Set protocol fee.
- Set a cut of eMetis that goes to Vesting contract when eMetis is unstaked.

Minter:

- Mint and burn eMetis tokens.

User:

- Exchange Metis for eMetis.
- Stake eMetis to generate rewards.
- Unstake it and get part of the stake and rewards. Other part is kept in Vesting contract.
- Stake ENKI to unlock the eMetis in Vesting contract.
- Unstake ENKI and claim eMetis.

Beta User:

- Can participate while the system is not public.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **6** out of **10**.

- Whitepaper is not provided.
- Litepaper is provided.
- Functional requirements are partially missed.
- Technical description is not provided.

Code quality

The total Code Quality score is **9** out of **10**.

- Several template code patterns were found.
- The development environment is configured.

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- ERC4626 test coverage is missed.
- Interactions by several users are not tested thoroughly.

Security score

Upon auditing, the code was found to contain **0** critical, **1** high, **1** medium, and **5** low severity issues, leading to a security score of **4** out of **10**. All significant findings were resolved during the remediation phase. The security score increased to **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **9.4**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The `minter_burn_from()` function allows accounts, whitelisted by the contract managers to burn eMetis tokens of any account.
- The `unstake()` function has modifier `forUser` which prevent users from calling it when the protocol is paused. This way ENKI tokens can be temporarily withheld from users.
- There is no safeguard against price manipulation on `seMetis` contract.
- The `RewardDispatcher` contract grants unlimited approval to `eMetisMinter` contract. That contract can consume all eMetis tokens from `RewardDispatcher` contract.
- There is a couple of centralization risks on `Config` contracts. Timelock and Operator roles can change vesting configuration while its already active.

Findings

Vulnerability Details

F-2024-1063 - Price manipulation on the first liquidation - High

Description:

The `seMetis` contract is designed as ERC-4626 Tokenized Vault standart and it uses **eMetis** token as its main asset.

There is a loss of funds risk according to the OpenZeppelin's ERC4626.sol documentation:

> In empty (or nearly empty) ERC-4626 vaults, deposits are at high risk of being stolen through frontrunning with a "donation" to the vault that inflates the price of a share. This is variously known as a donation or inflation attack and is essentially a problem of slippage. Vault deployers can protect against this attack by making an initial deposit of a non-trivial amount of the asset, such that price manipulation becomes infeasible. Withdrawals may similarly be affected by slippage.

It was identified that the `seMetis` vault does not have any security precautions to prevent such scenario. Therefore, the first liquidation is open to manipulation. A malicious user can front-run the first liquidity in order to profit.

Also, share prices will be determined by the first liquidator. Inflating share prices to very high amount could be another risky scenario. In such scenario, other users will not be able to purchase shares even if they have paid large amounts.

Steps to reproduce:

1. seMetis contract is deployed.
2. The initial liquidity provider tries to call two different deposits. (**0.5e18** eMetis + **0.75e18** eMetis ⇒ **1.25e18** eMetis in total)
3. Malicious user front-runs that call.
4. Malicious user calls **`eMetisMinter.mintAndDeposit(user1, 1)`** function. (**1** asset = **1** share)
5. Malicious user also transfers **1e18** eMetis to the **seMetis** contract . (**1e18** asset = **1** share)
6. Initial liquidity providers' calls takes place secondly due to gas fee priority.
7. **0.5e18** eMetis ⇒ **0** share
8. **0.75e18** eMetis ⇒ **0** share
9. In total, malicious user sent **1e18 + 1** eMetis to the contract and got **1** share. Liquidity provider sent **1.25e18** eMetis in total and got 0 share.

Path:

- contracts/SeMetis.sol#L17

Assets:

- SeMetis.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Fixed

Classification

Severity: High

Impact:
Likelihood [1-5]: 4
Impact [1-5]: 5
Exploitability [0-2]: 0
Complexity [0-2]: 1
Final Score: 4.3 (High)
Hacken Calculator Version: 0.6

Recommendations

Recommendation: Consider requiring a minimal amount of share tokens to be minted for the first minter, and send a portion of the initial mints as a reserve to the `seMetis` so that the price per share can be more resistant to manipulation.

Remediation (revised commit: **addd1fd)**: The finding was eliminated after the ENKI team added an initial minting to the `eMetisMinter.initialize()` function in order to prevent potential ERC4626 price manipulation issues.

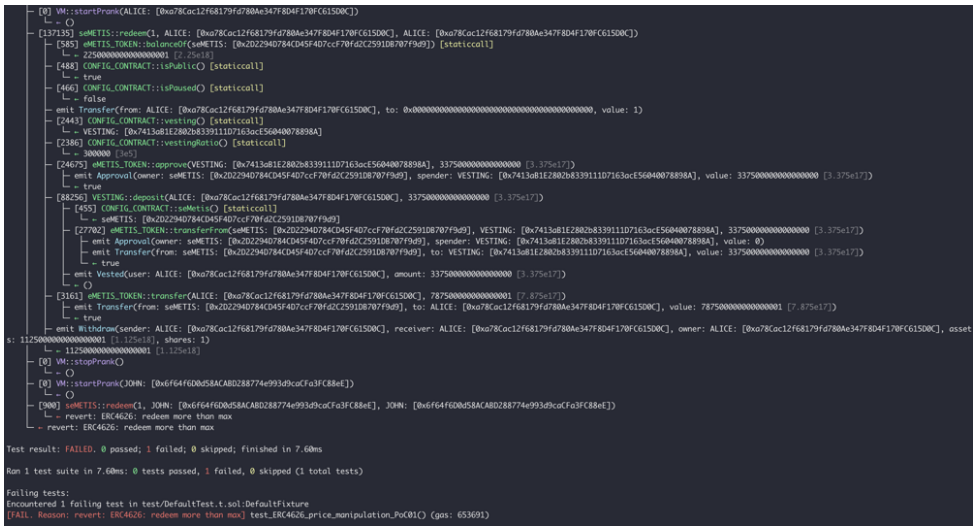
Evidences

PoC

Reproduce:

```
function test_ERC4626_price_manipulation_PoC01() public {
vm.startPrank(user1);
l1MetisToken.approve(address(eMetisMinter), type(uint256).max);
eMetisMinter.mint(user1, 1 ether + 1);

eMetisMinter.mintAndDeposit(user1, 1);
eMetis.transfer(config.seMetis(), 1 ether); // total deposit 1e18 + 1 --> 1s
hare:1e18

changePrank(user2);
l1MetisToken.approve(address(eMetisMinter), type(uint256).max);
eMetisMinter.mintAndDeposit(user2, 0.5 ether);
eMetisMinter.mintAndDeposit(user2, 0.75 ether); // total deposit 1.25e18 ->
share amount: 0 even user2 holds higher amount of total deposits

changePrank(user1);
seMetis.redeem(1, user1, user1);

changePrank(user2);
seMetis.redeem(1, user2, user2); // fails due to rounding error.
vm.stopPrank();
}
```

Results:


[0] VM::startPrank(ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C])
└─ ← ()
[137135] seMETIS::redeem(1, ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C], ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C])
[585] eMETIS_TOKEN::balanceOf(seMETIS: [0x2D2294D784CD45F4D7ccF70fd2C2591DB707f9d9]) [staticcall]
← 2250000000000000001 [2.25e18]
[488] CONFIG_CONTRACT::isPublic() [staticcall]
← true
[466] CONFIG_CONTRACT::isPaused() [staticcall]
← false
emit Transfer(from: ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C], to: 0x0000000000000000000000000000000000000000, value: 1)
[2443] CONFIG_CONTRACT::vesting() [staticcall]
← VESTING: [0x7413aB1E2802b8339111D7163acE56040078898A]
[2386] CONFIG_CONTRACT::vestingRatio() [staticcall]
← 300000 [3e5]
[24675] eMETIS_TOKEN::approve(VESTING: [0x7413aB1E2802b8339111D7163acE56040078898A], 337500000000000000 [3.375e17])
emit Approval(owner: seMETIS: [0x2D2294D784CD45F4D7ccF70fd2C2591DB707f9d9], spender: VESTING: [0x7413aB1E2802b8339111D7163acE56040078898A], value: 337500000000000000 [3.375e17])
← true
[88256] VESTING::deposit(ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C], 337500000000000000 [3.375e17])
[455] CONFIG_CONTRACT::seMetis() [staticcall]
← seMETIS: [0x2D2294D784CD45F4D7ccF70fd2C2591DB707f9d9]
[27702] eMETIS_TOKEN::transferFrom(seMETIS: [0x2D2294D784CD45F4D7ccF70fd2C2591DB707f9d9], VESTING: [0x7413aB1E2802b8339111D7163acE56040078898A], 337500000000000000 [3.375e17])
emit Approval(owner: seMETIS: [0x2D2294D784CD45F4D7ccF70fd2C2591DB707f9d9], spender: VESTING: [0x7413aB1E2802b8339111D7163acE56040078898A], value: 0)
emit Transfer(from: seMETIS: [0x2D2294D784CD45F4D7ccF70fd2C2591DB707f9d9], to: VESTING: [0x7413aB1E2802b8339111D7163acE56040078898A], value: 337500000000000000 [3.375e17])
← true
emit Vested(user: ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C], amount: 337500000000000000 [3.375e17])
← ()
[3161] eMETIS_TOKEN::transfer(ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C], 7875000000000001 [7.875e17])
emit Transfer(from: seMETIS: [0x2D2294D784CD45F4D7ccF70fd2C2591DB707f9d9], to: ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C], value: 7875000000000001 [7.875e17])
← true
emit Withdraw(sender: ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C], receiver: ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C], owner: ALICE: [0xa78Cac12f68179fd780Ae347F8D4F170FC615D0C], asset
s: 1125000000000000001 [1.125e18], shares: 1)
← 1125000000000000001 [1.125e18]
[0] VM::stopPrank()
← ()
[0] VM::startPrank(JOHN: [0x6f64f6D0d58ACABD288774e993d9caCFa3FC88eE])
← ()
[900] seMETIS::redeem(1, JOHN: [0x6f64f6D0d58ACABD288774e993d9caCFa3FC88eE], JOHN: [0x6f64f6D0d58ACABD288774e993d9caCFa3FC88eE])
← revert: ERC4626: redeem more than max
← revert: ERC4626: redeem more than max
Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 7.60ms
Ran 1 test suite in 7.60ms: 0 tests passed, 1 failed, 0 skipped (1 total tests)
Failing tests:
Encountered 1 failing test in test/DefaultTest.t.sol:DefaultFixture
[FAIL. Reason: revert: ERC4626: redeem more than max] test_ERC4626_price_manipulation_PoC01() (gas: 653691)

F-2024-1101 - The contract managers are able to withdraw tokens from the contracts - Medium

Description:

The `Base` contract has a function that allows the `OPERATOR_ROLE` or `TIMELOCK_ROLE` (contract manager addresses) to withdraw tokens that were sent to the contract by mistake. This also allows these actors to withdraw tokens that are supposed to stay in the contract.
Contracts that inherit the Base.sol and tokens they are expected to hold:

- EMetisMinter - Metis
- RewardDispatcher - eMetis
- SeMetis - eMetis
- Vesting - eMetis, ENKI

Assets:

- Base.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Fixed

Classification

Severity: Medium

Impact:

Likelihood [1-5]: 4
Impact [1-5]: 5
Exploitability [0-2]: 2
Complexity [0-2]: 0
Final Score: 2.7 (Medium)
Hacken Calculator Version: 0.6

Recommendations

Recommendation:

Prevent withdrawal of the tokens that the contract is expected to work with. Optionally - keep track of the relevant tokens on the contract and only allow to withdraw tokens from direct transfers.

Remediation (revised commit: **addd1fd)**: The `holdTokens` variable was implemented to track which tokens will be used in the protocol. Therefore, it will be forbidden to recover/transfer these tokens by the contract owner.

F-2024-1064 - Unlimited approval granted to unrelated contract - Low

Description:

The `RewardDispatcher` contract is designed for transferring **eMetis** token to the Protocol treasury and `seMetis` vault. Even, there is a communication between `eMetisMinter` and `RewardDispatcher` contracts, the `eMetisMinter` contract does not make any **eMetis** function call for `RewardDispatcher` contract directly.

The `initialize()` function of `RewardDispatcher` contract grants unlimited approval for `eMetisMinter` contract which is irrelevant. If any security vulnerability occurs in the `eMetisMinter` contract, all eMetis tokens on RewardDispatcher can be transferred by the `eMetisMinter` contract.

It was also confirmed with the ENKI team that the `IERC20Upgradeable(metis).approve()` call is irrelevant.

```
function initialize(address _config) public initializer {
__Base_init(_config);
metis = config.metis();
eMetisMinter = config.eMetisMinter();
eMetis = config.eMetis();
seMetis = config.seMetis();

IERC20Upgradeable(metis).approve(eMetisMinter, type(uint256).max);
}
```

Path:

- contracts/RewardDispatcher.sol#L37

Assets:

- RewardDispatcher.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 2
Impact [1-5]: 3
Exploitability [0-2]: 0
Complexity [0-2]: 0
Final Score: 2.5 (Low)
Hacken Calculator Version: 0.6

Recommendations

Recommendation: Consider removing unneeded `IERC20Upgradeable(metis).approve()` call from the `RewardDispatcher.initialize()` function in order to prevent any unwanted situations occur.

Remediation (revised commit: **addd1fd)**: Unlimited approval was removed from the contract with the latest code update.

F-2024-1066 - Timelock and Operator roles can renounce their permissions - Low

Description:

The **Timelock** and **Operator** roles of the `Config` contract are designed for changing protocol settings. These roles can perform some privileged functions such as `setPublic()`, `setProtocolTreasury()`, `setProtocolTreasuryRatio()`, `setVestingRatio()` and `setVestingDuration()`. In the `Config` contract, the `renounceRole()` function is used to renounce **Timelock** and **Operator** roles. Renouncing roles before transferring would result in the contract having no **Timelock** or **Operator** users, eliminating the ability to call these privileged functions.

```
function renounceRole(bytes32 role, address account) external override {
require(account == _msgSender(), "AccessControl: can only renounce roles for
self");

_revokeRole(role, account);
}
```

Path:

- contracts/Config.sol#L172-L176

Assets:

- Config.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 3
Impact [1-5]: 4
Exploitability [0-2]: 2
Complexity [0-2]: 0
Final Score: 2.3 (Low)
Hacken Calculator Version: 0.6

Recommendations

Recommendation:

It is recommended to override the `renounceRole()` function in a way that prevents revoking roles without transferring roles.

Remediation (revised commit: addd1fd): This finding was eliminated after the `renounceRole()` function was removed from the code.

F-2024-1068 - Risk of centralization due to authority of Timelock and Operator roles in Vesting settings - Low

Description:

The `Config` contract is designed to store all protocol related configurations in a place. This contract has two different roles (**Timelock** and **Operator**) to manage configuration changes. Basically, when this contract is initialized, the **msg.sender** of initialization call becomes the **Timelock** and will be responsible from managing this contract.

The **Timelock** role can grant also grant **Operator** role to other users which has really similar permissions. These roles should be granted to multi-sig wallet addresses.

In other case, the **Timelock** and **Operator** roles can change:

- Vesting Ratio
- Vesting Duration
- Protocol Treasury Ratio

Changing this values to very small and very high amounts may prevent `Vesting` contract working properly.

Assets:

- Config.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Accepted

Classification

Severity: Low

Impact:

Likelihood [1-5]: 3
Impact [1-5]: 4
Exploitability [0-2]: 2
Complexity [0-2]: 0
Final Score: 2.3 (Low)
Hacken Calculator Version: 0.6

Recommendations

Recommendation:

Consider using Multi-sig wallet for `Timelock` and `Operator` addresses.

Remediation (Accepted): This finding was acknowledged by the ENKI team.

F-2024-1071 - Owner can renounce its ownership - Low

Description:

The **Owner** of the contract is usually the account that deploys/initializes the contract. Additionally, the **Owner** can perform some privileged functions. The `ERC20PermitPermissionedMint` contracts uses `Ownable` library of OpenZeppelin which has `renounceOwnership()` function.

Renouncing ownership before transferring would result in the contract having no Owner, eliminating the ability to call privileged functions.

Assets:

- ERC20PermitPermissionedMint.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 3
Impact [1-5]: 4
Exploitability [0-2]: 2
Complexity [0-2]: 0
Final Score: 2.3 (Low)
Hacken Calculator Version: 0.6

Recommendations

Recommendation:

Consider replacing `Ownable` library with `Ownable2Step` that includes a two-step mechanism to transfer ownership.

Remediation (revised commit: **addd1fd)**: The `Ownable` library was replaced with `Ownable2Step` to fix this finding.

F-2024-1079 - `Dealer.lockFor()` and `Dealer.setActive()` functions do not check previous agent entries - Low

Description:

It was identified that the `Dealer.lockFor()` and `Dealer.setActive()` functions on the L1Dealer contract, which help to add new SequencerAgent, do not control the **activeSequencerIds** variable before adding new agents. As a result, the admin of that contract can add duplicate entries to the protocol by mistake. This situation may pose unexpected situations since the **activeSequencerIds** variable takes so many places in the `Dealer` contract.

```
function lockFor(uint32 agentId, address sequencerSigner, uint256 amount, by
tes memory signerPubKey) external onlyRole(DEFAULT_ADMIN_ROLE) {
address agent = sequencerAgents[agentId];
metis.safeTransferFrom(msg.sender, agent, amount);
SequencerAgent(agent).lockFor(sequencerSigner, amount, signerPubKey);
activeSequencerIds.push(agentId);
}
```

```
function setActive(uint32 agentId, bool active) external onlyRole(DEFAULT_AD
MIN_ROLE) {
if (active) {
activeSequencerIds.push(agentId);
} else {
_removeFromActiveList(agentId);
}
}
```

Assets:

- L1/Dealer.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 3
Impact [1-5]: 2
Exploitability [0-2]: 2
Complexity [0-2]: 0
Final Score: 1.8 (Low)
Hacken Calculator Version: 0.6

Recommendations

Recommendation:

It is recommended to verify that specified agents are not previously included to the **activeSequencerIds** array.

Remediation (revised commit: **addd1fd)**: The `_setActive()` function was added to the `Dealer` contract to prevent adding existing entries.

Observation Details

F-2024-1065 - `__ReentrancyGuard_init()` function is not called during the initialization - Info

Description:

The `__ReentrancyGuard_init()` function updates the value of the `_status` variable to **1** when the contracts are initialized. Although this does not cause any known security vulnerabilities, calling this function is considered best-practice.

There are multiple instances that use the **ReentrancyGuardUpgradeable** library without initializing it.

Path:

- contracts/Config.sol#L118-L119
- contracts/EMetisMinter.sol#L44-L45
- contracts/RewardDispatcher.sol#L30-L31
- contracts/SeMetis.sol#L14-L17
- contracts/Vesting.sol#L56-L57

Status: Fixed

Recommendations

Recommendation:

It is recommended to call the `__ReentrancyGuard_init()` function during the initialization of contracts.

Remediation (revised commit: **addd1fd)**: The `__ReentrancyGuard_init()` was implemented for the `Base` contract in order to initialize the **ReentrancyGuard**.

F-2024-1067 - Long revert messages consumes extra gas - Info

Description:

Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime Gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional `mstore` introduction, along with additional overhead to calculate memory offset, etc.

To optimize gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short. Doing so can reduce the amount of gas used during deployment and runtime when the revert condition is met.

There are multiple instances of **Long revert messages** in the protocol.

Status: Accepted

Recommendations

Recommendation:

Consider shorting all revert strings longer than 32 characters in order to optimise gas usage. In addition, Custom Errors can be used to decrease gas usage.

Remediation (Accepted): This finding was acknowledged by the ENKI team.

External References:

- Custom Errors

F-2024-1069 - Missing lower and upper bounds on `setVestingDuration()` function - Info

Description:

The `Config.setVestingDuration()` function is designed to change the `configMap[UINT64_VESTING_DURATION]` variable. That variable is one of the key elements of vesting calculations. Currently, there is no lower and upper bounds for `_vestDuration` variable `Config.setVestingDuration()` function. Setting very small or very high amounts as this variable can drastically affect the vesting logic. Therefore, it is necessary to have lower and upper bounds for this function.

```
function setVestingDuration(uint64 _vestDuration) public override onlyOperat
orOrTimeLock {
require(_vestDuration > 0, "Config: vestingDuration must be greater than 0")
;
uint64 oldValue = uint64(configMap[UINT64_VESTING_DURATION]);
configMap[UINT64_VESTING_DURATION] = _vestDuration;
emit VestingDurationSet(oldValue, _vestDuration);
}
```

Assets:

- Config.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Accepted

Recommendations

Recommendation:

Consider implementing lower and upper bounds for `_vestDuration` variable of the `Config.setVestingDuration()` function.

Remediation (Accepted): This finding was acknowledged by the ENKI team.

F-2024-1070 - Missing amount control - Info

Description:

There are multiple instances of functions that do not check if the `amount` variable is higher than zero. These instances can be executed with zero amount which can produce unexpected behaviors. Various token transfers are also callable with these `amount` variables. Invoking these token transfers with zero amounts will only waste users' gas. It is important to prevent functions from being called with a zero amount.

Path:

- contracts/EMetisMinter.sol#L57
- contracts/EMetisMinter.sol#L66
- contracts/EMetisMinter.sol#L72
- contracts/EMetisMinter.sol#L82
- contracts/Vesting.sol#L65

Assets:

- EMetisMinter.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]
- Vesting.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Fixed

Recommendations

Recommendation:

It is a best-practice to having `amount > 0` checks in contracts to prevent any unexpected situations occur.

Remediation (revised commit: **addd1fd)**: The finding was eliminated after the `amount > 0` checks were implemented to the code.

F-2024-1073 - TokenMinterBurned event creates confusion due to invalid naming of event parameter - Info

Description:

The `TokenMinterBurned` event is implemented for providing details about `ERC20PermitPermissionedMint.minter_burn_from()` function. These event uses the following structure:

```
event TokenMinterBurned(address indexed from, address indexed to, uint256 am
ount);
```

The variable names used here do not seem correct considering the logic of the `minter_burn_from()` function. The naming here has created confusion as if the burned tokens were sent to the `to` address. However, the `to` variable exhibits who calls the `minter_burn_from()` function.

Assets:

- ERC20PermitPermissionedMint.sol [https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts]

Status: Accepted

Recommendations

Recommendation:

Consider renaming `to` field of `TokenMinterBurned` event to clear that confusion.

Remediation (Accepted): This finding was acknowledged by the ENKI team.

F-2024-1074 - Floating pragma - Info

Description:

A **floating pragma** in Solidity refers to the practice of using a pragma statement that does not specify a fixed compiler version but instead allows the contract to be compiled with any compatible compiler version. This issue arises when pragma statements like `pragma solidity ^0.8.0;` are used without a specific version number, allowing the contract to be compiled with the latest available compiler version. This can lead to various compatibility and stability issues.

It was identified that there are multiple examples of floating pragma exists.

In addition, the project folder contains multiple pragma versions which. Each pragma statement should be identical within a contract or file, as it sets the compiler version and potentially other compiler-specific configurations.

Status: Fixed

Recommendations

Recommendation:

Consider using the same pragma version and locking all of the pragma version in the protocol whenever possible and avoid using a floating pragma in the final deployment.

Remediation (revised commit: **addd1fd)**: This finding is eliminated after the ENKI team locked the pragma version to `0.8.24`.

External References:

- Solidity Releases

F-2024-1076 - Unused imports - Info

Description: The following identifiers are imported but never used within contracts:

```
import "@openzeppelin/contracts-upgradeable/proxy/ClonesUpgradeable.sol";
```

```
import "../interface/ICrossDomainEnabled.sol";
```

It is suggested to clear unused imports from the code in order to prevent code clutters.

Path:

- contracts/L1/SequencerAgent.sol#L6
- contracts/L1/SequencerAgent.sol#L8

Status: Fixed

Recommendations

Recommendation: Regularly review your Solidity code to remove unused imports. This practice declutters the codebase, making it easier to understand and maintain. In addition, consider using tools or IDE features that can automatically detect and highlight unused imports for cleanup. Keeping imports limited to only what is necessary helps maintain a clear understanding of the contract's dependencies and reduces potential confusion for developers working on or reviewing the code.

Remediation (revised commit: **addd1fd)**: Unused imports were removed from the code.

F-2024-1077 - For loop optimizations - Info

Description: It has been observed for loops in the protocol were not optimized properly. Having not optimized for loops can cost too much gas usage. These for loops can be optimized with suggestions below:

- In Solidity (pragma 0.8.0 and later), adding `unchecked` keyword for arithmetical operations can reduce gas usage on contracts where underflow/underflow is unrealistic. It is possible to save gas by using this keyword on multiple code locations.
- In all for loops, the index variable is incremented using `i++`. It is known that, in loops, using `++i` costs less gas per iteration than `i++`. This also affects incremented variables within the loop code block.
- Do not initialize index variables with **0**, Solidity already initializes these uint variables as zero.

Path:

- contracts/ERC20PermitPermissionedMint.sol#L82
- contracts/L1/Dealer.sol#L144
- contracts/L1/Dealer.sol#L205
- contracts/L1/Dealer.sol#L243

Status: Accepted

Recommendations

Recommendation: It is recommended to apply the following pattern for Solidity pragma version between 0.8.0 and 0.8.20.

```
for ( uint256 i; i < arrayLength ; ) {
. . .
unchecked {
++ i;
}
}
```

Remediation (Accepted): This finding was acknowledged by the ENKI team.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details

Repository	https://github.com/ENKIXYZ/lsd-contracts/tree/v2/contracts
Commit	934933fc5963075ab6cbfa3a13bcf17b36480268
Remediation Commit	addd1fd48d70a68ce6366088ab8fe80abd983a3b
Whitepaper	N/A
Requirements	N/A
Technical Requirements	N/A

Contracts in Scope

Base.sol

Config.sol

EMetis.sol

EMetisMinter.sol

ERC20PermitPermissionedMint.sol

RewardDispatcher.sol

SeMetis.sol

Vesting.sol

L1/Dealer.sol

L1/SequencerAgent.sol

interfaces/*.sol