



SMART CONTRACT AUDIT REPORT

for

Velix Protocol



Prepared By: Xiaomi Huang

PeckShield
June 19, 2024

Document Properties

Client	Velix
Title	Smart Contract Audit Report
Target	Velix
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 19, 2024	Xuxian Jiang	Final Release
1.0-rc	June 17, 2023	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About Velix	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Improved Constructor/Initialization Logic in Base	11
3.2	Suggested Adherence Of Checks-Effects-Interactions Pattern	12
3.3	Trust Issue of Admin Keys	13
4	Conclusion	15
	References	16

1 | Introduction

Given the opportunity to review the design document and related source code of the `Velix` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Velix

`Velix` is a liquid staking derivative protocol that allows users on the `Metis` L2 network to stake their `Metis` in return for a liquid staked token allowing them to maintain liquidity while earning yield on their staked `Metis`. `Velix` uses the dual token model unlike the mono token model in most protocols. `Velix` users mint `veMETIS` with `Metis` and then submit the `veMETIS` for `sveMETIS` which is a proof of their staked `veMETIS` earning yield. Rewards earned from sequencers are converted to `veMETIS` via a `veMETISMinter` and gets distributed to `sveMETIS` holders with a fee going to the protocol treasury which will be used for future protocol initiatives. Users of `Velix` protocol also get rewarded with `VELIX` tokens from their accumulated staking points as additional incentive for participating in the protocol. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Velix

Item	Description
Name	Velix
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 19, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in

this audit. Note this audit covers only the following contracts: `Base.sol`, `Config.sol`, `RewardDispatcher.sol`, `SveMetis.sol`, `VeMetis.sol`, and `VeMetisMinter.sol`.

- <https://github.com/Velix-protocol/velix-smart-contract.git> (999b8e9)

And here is the commit ID after fixes for the issues found in the audit have been checked in:

- <https://github.com/Velix-protocol/velix-smart-contract.git> (3c21124)

1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [6], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices



Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `velix` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	2	
Informational	0	
Total	3	

We have so far identified a list of potential issues. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in [Section 3](#).

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Velix Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor/Initialization Logic in Base	Coding Practices	Resolved
PVE-002	Low	Suggested Adherence of Checks-Effects-Interactions	Times And State	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Constructor/Initialization Logic in Base

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [2]

Description

To facilitate possible future upgrade, the `Base` contract is instantiated as a proxy with actual logic contract in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., `_disableInitializers()`; . Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initialize()` function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
49     /// @notice Initializes the contract with the config contract address
50     /// @param _config The address of the config contract
51     function __Base_init(address _config) internal onlyInitializing {
52         config = IConfig(_config);
53         __ReentrancyGuard_init();
54     }
```

Listing 3.1: `Base::__Base_init()`

Recommendation Improve the above-mentioned constructor routine in the upgradeable contract, i.e., `Base`. Moreover, the current initialization routine may be improved by calling `__Context_init()` as well.

Status This issue has been fixed by the following commit: [e4cd489d](#).

3.2 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: SveMetis
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [10] exploit, and the Uniswap/Lendf.Me hack [9].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the SveMetis as an example, the `_withdraw()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. Apparently, the interaction with the external contract (line 69) starts before effecting the update on internal states (line 70), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be abused of launching `re-entrancy` via the same entry function.

```
55     function _withdraw(  
56         address caller,  
57         address receiver,  
58         address owner,  
59         uint256 assets,  
60         uint256 shares  
61     ) internal override {  
62         if (caller != owner) {  
63             _spendAllowance(owner, caller, shares);  
64         }  
65  
66         _burn(owner, shares);  
67  
68         IERC20 asset = IERC20(asset());
```

```

69     SafeERC20.safeTransfer(asset, receiver, assets);
70     _totalAssets -= assets;
71
72     emit Withdraw(caller, receiver, owner, assets, shares);
73 }

```

Listing 3.2: SveMetis::_withdraw()

Recommendation Revisit the above routine to follow the best practice of the checks-effects-interactions pattern.

Status This issue has been fixed by the following commit: [e4cd489d](#).

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [3]

Description

In the Velix protocol, there is a special admin account (with the `ROLE_DEFAULT_ADMIN_ROLE` role). This admin account plays a critical role in governing and regulating the protocol-wide operations (e.g., assign roles, configure parameters, and upgrade the proxy). Our analysis shows that the admin account and other privileged roles need to be scrutinized. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

```

91     function setL1Dealer(
92         address _l1Dealer
93     ) public onlyOperatorOrAdmin {
94         configMap[ADDRESS_L1_DEALER] = uint256(uint160(_l1Dealer));
95     }
96
97     /**
98      * @dev Sets the veMetis address.
99      * @param _veMetis Address of the veMetis contract.
100     */
101     function setVeMetis(address _veMetis) public onlyOperatorOrAdmin {
102         configMap[ADDRESS_VEMETIS] = uint256(uint160(_veMetis));
103     }
104
105     /**
106      * @dev Sets the veMetis minter address.
107      * @param _veMetisMinter Address of the veMetis minter contract.

```

```

108     */
109     function setVeMetisMinterAddress(
110         address _veMetisMinter
111     ) public onlyOperatorOrAdmin {
112         configMap[ADDRESS_VEMETIS_MINTER] = uint256(uint160(_veMetisMinter));
113     }

115     /**
116     * @dev Sets the sVeMetis address.
117     * @param _sveMetis Address of the sVeMetis contract.
118     */
119     function setSveMetis(
120         address _sveMetis
121     ) public onlyOperatorOrAdmin {
122         configMap[ADDRESS_SVEMETIS] = uint256(uint160(_sveMetis));
123     }

125     /**
126     * @dev Sets the reward dispatcher address.
127     * @param _rewardDispatcher Address of the reward dispatcher contract.
128     */
129     function setRewardDispatcher(
130         address _rewardDispatcher
131     ) public onlyOperatorOrAdmin {
132         configMap[ADDRESS_REWARD_DISPATCHER] = uint256(
133             uint160(_rewardDispatcher)
134         );
135     }

```

Listing 3.3: Example Privileged Operations in Config

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

In the meantime, the above contract makes use of the proxy contract to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

Recommendation Promptly transfer the privileged accounts to the intended DAO-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been resolved as the team confirms the use of a timelocked contract as the admin.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `velix` protocol, which is a liquid staking derivative protocol. It allows users on the `Metis` L2 network to stake their `Metis` in return for a liquid staked token allowing them to maintain liquidity while earning yield on their staked `Metis`. Users of `velix` protocol also get rewarded with `VELIX` tokens from their accumulated staking points as additional incentive for participating in the protocol. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



References

- [1] MITRE. CWE-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [3] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [8] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [9] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.

- [10] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

