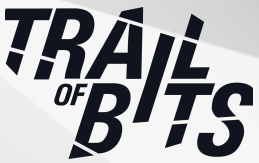

Onchain Pass App Contracts

Security Assessment (Summary Report)

September 10, 2024

Prepared for:
Manu Siddalingegowda
Pass App Ltd.

Prepared by: **Guillermo Larregay**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits 1
Notices and Remarks 2
Table of Contents 3
Project Summary 4
Project Targets 5
Executive Summary 6
Codebase Maturity Evaluation 8
Summary of Findings 11
A. Vulnerability Categories 12
B. Code Maturity Categories 14
C. Token Integration Checklist 16
D. Incident Response Recommendations 19
E. Security Best Practices for Using Multisignature Wallets 21
F. Upgradability Checks with Slither 23
G. Fix Review Results 24
Detailed Fix Review Results 25
H. Fix Review Status Categories 27

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
August 5, 2024	Pre-project kickoff call
August 13, 2024	Delivery of report draft
August 13, 2024	Report readout meeting
September 10, 2024	Delivery of summary report

Project Targets

The engagement involved a review and testing of the following target.

Pass App Contracts

Repository	https://github.com/PassHQ/pass-contracts
Version	commit deb8e33
Type	Solidity
Platform	EVM compatible

Executive Summary

Engagement Overview

Pass App Ltd. engaged Trail of Bits to review the security of Pass App Contracts commit deb8e33. The contract implements a multisignature-owned Vault that accepts ERC20 tokens and native assets deposits and withdrawals. Additionally, there is a swap functionality that performs swapping and bridging of assets using a third-party aggregator exchange, and a function to fund an external paymaster.

A team of one consultant conducted the review from August 5 to August 9, 2024, for a total of one engineer-week of effort. With full access to source code and documentation, we performed static and dynamic testing of the target using automated and manual processes.

Observations and Impact

The engagement was scoped to provide a security assessment of the Pass App Vault contract. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can the funds be withdrawn from the vault by an attacker, bypassing the owner's role?
- Can the Swapper perform any other privileged actions other than the expected swaps?
- Can an external user lock funds or otherwise affect the owner's or swapper's ability to access the funds in the vault?

The audit found seven issues: one of medium severity, one of low severity, and three informational issues. The severity of the remaining two issues could not be determined from the information available at the time of the engagement. In particular, the medium-severity issue (TOB-PASS-1) could lead to an unexpected loss of funds from the Vault contract. One of the undetermined-severity issues (TOB-PASS-5) is related to the Swapper role, which can be considered a single point of failure in the Vault if it were misassigned to a non-trusted contract or address.

Please note that this review's coverage was limited; we did not have access to the back-end code, including the event handlers, the Rango swap calldata generation, the multisignature wallet contracts, or any other code besides the content of the repository mentioned above.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Pass App Ltd. take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactoring that may occur when addressing other recommendations.
- **Improve the test suite.** Add tests for the upgrade procedure, and review the conditions for the failing fuzzing tests.
- **Improve the user and developer documentation and source code comments.** Include information about how the system is expected to behave, such as how the back end works, how the swapper obtains the calldata for the swap, how the integration between the paymaster and the swapper works, and any other information that can be useful for users or developers integrating with the Vault contract.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Because the code uses Solidity compiler version 0.8.23, SafeMath is included by default. The code does not depend on arithmetic algorithms. However, we found two usages of unchecked arithmetic in for-loop variable incrementing, with no further documentation or explanation.	**Satisfactory**
Auditing	The vault contract interacts with the back end through events. However, it is documented that deposit and withdrawal functions deliberately miss event emissions, as they rely on the ERC20 standard events. This leads to issue TOB-PASS-3. At the time of the audit, we were not aware of an incident monitoring system or response plan. For recommendations on incident response plans, see appendix D.	**Moderate**
Authentication / Access Controls	There are two actors in the Vault contract, and their privileges are clearly specified in the documentation. Anyone can deposit assets into the vaults, but withdrawing, transferring, and swapping assets are privileged functions. The owner role is assigned to a 2-of-4 multisignature wallet controlled by the team, and the swapper role is assumed to be fully trusted. The swapper role is assigned to a single external contract address and can be changed. The owner can also be changed, but it is not a two-step process. If the swapper is set to a malicious contract or EOA, it can drain the vaults or call any external contract	**Satisfactory**

	function on behalf of the vault.	
Complexity Management	The contract's functions are short, easy to read, and clear in purpose. There is no redundant code, and the naming conventions are easy to understand. Some functions present unnecessary low-level code.	**Satisfactory**
Decentralization	The Vault contract is centralized by design. All withdrawal functions are callable only by the owner role, and the swap function is callable by the swapper and owner roles. The swapper role is a single point of failure in the contract, as assigning it to a malicious actor can allow them to drain the contract or to interact with any third-party contract (TOB-PASS-5).	**Weak**
Documentation	The documentation provided for the audit consisted of a document explaining the purpose of the functions and roles in the contract. Also, a diagram of the high-level architecture of the Pass App contracts was provided. The code is well commented in general, using NatSpec for most functions. There was no specific documentation for the back end, the interaction between the Vault and external contracts (Rango and Biconomy), or the privileged roles.	**Moderate**
Low-Level Manipulation	The code uses unjustified low-level (assembly) calls for native asset transfers instead of using the Solady Safe Transfer library, which is also used for token transfers (TOB-PASS-4). Although the low-level parts of the code do not have specific comments, most are self-explanatory and consist of a single line.	**Moderate**
Testing and Verification	Test coverage is not 100% for the Vault contract. No tests implement a vault upgrade procedure. Some test cases for the expected functionality are missing. In particular, the test suite should have detected issue TOB-PASS-1. Unit and basic Fuzz tests have been implemented.	**Weak**

	However, one of the fuzzing tests occasionally fails (`testFuzz_RevertNotOwner_FillPaymasterGasTank`). Testing is part of the CI/CD pipeline.	
Transaction Ordering	We detected a transaction ordering risk when the owner updates the Swapper account address (TOB-PASS-5).	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Value is set for all calls to Rango, even when swapping tokens	Data Validation	**Medium**
2	The back end can be event-spammed	Auditing and Logging	**Undetermined**
3	Lack of event emission for ERC20 deposits and withdrawals	Data Validation	**Low**
4	Ether transfers use low-level calls	Data Validation	**Informational**
5	Swapper can front-run the `updateSwapperAddress` call	Timing	**Undetermined**
6	Unused libraries in the project	Configuration	**Informational**
7	Lack of a two-step process for ownership transfer	Data Validation	**Informational**

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.

Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified and its associated risks understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the following output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath` or Solidity 0.8.0+.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath`/Solidity 0.8.0+ usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can misuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot denylist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a denylist. Identify denylisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for misuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC-20 Tokens

ERC-20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a Boolean.** Several tokens do not return a Boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC-20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is less than 255.
- ❏ **The token mitigates the known ERC-20 race condition.** The ERC-20 standard has a known ERC-20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC-20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC-777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is accounted for.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not accounted for.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

D. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Pass App Ltd. will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

E. Security Best Practices for Using Multisignature Wallets

Consensus requirements for sensitive actions, such as spending funds from a wallet, are meant to mitigate the risks of the following:

- Any one person overruling the judgment of others
- Failures caused by any one person's mistake
- Failures caused by the compromise of any one person's credentials

For example, in a 2-of-3 multisignature wallet, the authority to execute a "spend" transaction would require a consensus of two individuals in possession of two of the wallet's three private keys. For this model to be useful, the following conditions are required:

1. The private keys must be stored or held separately, and access to each one must be limited to a unique individual.
2. If the keys are physically held by third-party custodians (e.g., a bank), multiple keys should not be stored with the same custodian. (Doing so would violate requirement #1.)
3. The person asked to provide the second and final signature on a transaction (i.e., the cosigner) should refer to a pre-established policy specifying the conditions for approving the transaction by signing it with his or her key.
4. The cosigner should also verify that the half-signed transaction was generated willingly by the intended holder of the first signature's key.

Requirement #3 prevents the cosigner from becoming merely a "deputy" acting on behalf of the first signer (forfeiting the decision-making responsibility to the first signer and defeating the security model). If the cosigner can refuse to approve the transaction for any reason, the due-diligence conditions for approval may be unclear. That is why a policy for validating transactions is needed. A verification policy could include the following:

- A protocol for handling a request to cosign a transaction (e.g., a half-signed transaction will be accepted only via an approved channel)
- An allowlist of specific addresses allowed to be the payee of a transaction
- A limit on the amount of funds spent in a single transaction or in a single day

Requirement #4 mitigates the risks associated with a single stolen key. For example, say that an attacker somehow acquired the unlocked Ledger Nano S of one of the signatories. A voice call from the cosigner to the initiating signatory to confirm the transaction would reveal that the key had been stolen and that the transaction should not be cosigned. If the signatory were under an active threat of violence, he or she could use a duress code (a code word, a phrase, or another signal agreed upon in advance) to covertly alert the others that the transaction had not been initiated willingly, without alerting the attacker.

F. Upgradability Checks with Slither

Trail of Bits developed the `slither-check-upgradability` tool to aid in the development of secure proxies; it performs safety checks relevant to both upgradeable and immutable `delegatecall` proxies. Consider using this tool during the development of the Pass App contracts' codebase.

- Use `slither-check-upgradeability` to check for issues such as a corrupted storage layout between the previous and new implementations.

```
slither-check-upgradeability . ContractV1 --new-contract-name ContractV2
```

Figure F.1: An example of how to use `slither-check-upgradeability`

For example, if a variable `a` is incorrectly added in the new implementation before other storage variables, `slither-check-upgradeability` will issue a warning like the one shown in figure F.2.

```
...
INFO:Slither:
Different variables between ContractV1 (contracts/versions/ContractV1.sol#9-54)
and ContractV2 (contracts/versions/ContractV2.sol#11-133)
        ContractV1.__gap (contracts/versions/ContractV1.sol#15)
        ContractV2.a (contracts/versions/ContractV2.sol#20)
Reference:
https://github.com/crytic/slither/wiki/Upgradeability-Checks#incorrect-variables-w
ith-the-v2
...
```

Figure F.2: Example `slither-check-upgradeability` output

- Use `slither-read-storage` with the new implementation to manually check that the expected storage layout matches the previous implementation. Running the command shown in figure F.3 for the previous and new implementations will list the storage locations for variables in both versions. Make sure that variables in the previous implementation have the same slot number in the upgraded version.

```
slither-read-storage . --contract ContractV1 --table
```

Figure F.3: An example of how to use `slither-read-storage`

- If gaps are used in parent contracts, check the storage layout with `slither-read-storage` after adding new functionality and decrease the gap size accordingly. Make sure that storage is not overwritten or shifted in child contracts.

- Simulate the upgrade with a local mainnet fork, verify that all storage variables have the expected values, and run the tests on the new implementation.

G. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On August 21, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Pass App Ltd. team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In addition to fixing the issues mentioned in this report, the team also updated and shared the documentation regarding the backend and its interaction with the vault, clarifying the actions executed by the backend.

In summary, of the seven issues described in this report, Pass App Ltd. has resolved six issues, and has not resolved the remaining issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Value is set for all calls to Rango, even when swapping tokens	**Resolved**
2	The back end can be event-spammed	**Resolved**
3	Lack of event emission for ERC20 deposits/withdrawals	**Resolved**
4	Ether transfers use low-level calls	**Resolved**
5	Swapper can front-run the updateSwapperAddress call	**Unresolved**
6	Unused libraries in the project	**Resolved**
7	Lack of a two-step process for ownership transfer	**Resolved**

Detailed Fix Review Results

TOB-PASS-1: Value is set for all calls to Rango, even when swapping tokens
Resolved in PR #20. The call to the Rango contract is now handled differently for the native asset and ERC20 token. The value field is set only for the native asset.

New unit and fuzzing tests have been added to ensure Rango receives the correct parameters and assets in each case.

TOB-PASS-2: The back end can be event-spammed
Resolved. From the updated documentation, the back end checks for allowlisted token balances only once a day and does not listen for events, as previously stated. It is worth noting that no changes were made to the contracts to address this issue, and the back end is out of scope for this audit.

The client provided the following context for this finding's fix status:

> *We have a system in the backend to check token whitelisting criteria, including tokens supported by Rango. This check is performed before contract interaction. We do not listen to events to check if the token is supported by Rango. Hence, even if there are a lot of event-emitting transactions from a cheaper network, the backend is safe from being overloaded with requests.*

TOB-PASS-3: Lack of event emission for ERC20 deposits/withdrawals
Resolved in PR #21. Two new events were added to the Vault contract: `Erc20TokenWithdraw` and `Erc20TokenReceived`. Existing events have been renamed to match the naming conventions, and new unit tests have been implemented to check the emission of events.

TOB-PASS-4: Ether transfers use low-level calls
Resolved in PR #22. Low-level native asset transfers and the associated failure event have been replaced by `SafeTransferLib` functions.

TOB-PASS-5: Swapper can front-run the updateSwapperAddress call
Unresolved. The swapper is assumed to be fully trusted, and no changes were made to the contracts.

The client provided the following context for this finding's fix status:

> *We trust the swapper smart account. In case swapper becomes malicious, we have the provision to change swapper address from the multisig owner.*

TOB-PASS-6: Unused libraries in the project
Resolved in PR #24. Unused library modules have been removed.

TOB-PASS-7: Lack of a two-step process for ownership transfer
Resolved in PR #23. Inheritance from OpenZeppelin's `Ownable2StepUpgradeable` is now used for the vault instead of `OwnableUpgradeable`.

Additionally, new unit and fuzz tests for the two-step ownership change have been added.

H. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.