QuillAudits

AUDIT REPORT

January 2026

For

SWARM

Table of Content

Executive Summary

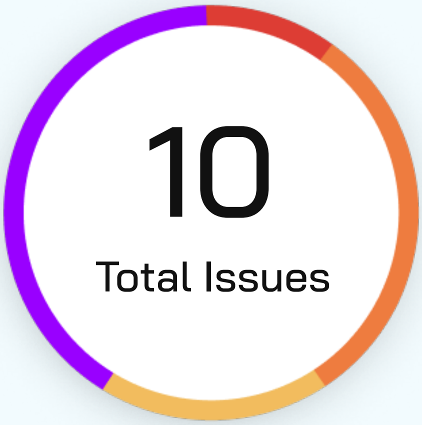
Project Name	VAULT by Swarm Network
Protocol Type	Encrypted Vault & Secret Management Protocol
Project URL	https://swarmnetwork.ai/
Overview	VAULT is a decentralized secrets management platform built on the Sui blockchain, offering zero-knowledge encryption for secure storage of sensitive credentials and confidential data. The platform employs military-grade encryption protocols ensuring that only authorized users can decrypt their stored secrets, while maintaining complete data sovereignty. The solution enables organizations to securely share vaults among team members through blockchain-based access controls, eliminating traditional centralized key management vulnerabilities. As a Web3-native application, VAULT leverages Sui's high-performance blockchain infrastructure to provide cryptographically secure, distributed storage with no single point of failure. The platform targets enterprises and development teams requiring robust secrets management without compromising on decentralization principles or privacy guarantees.
Branch	main
Contracts in Scope	vault.move user_registry.move
Commit Hash	b72725075cea32736efe067b97da1a42140181c9
Language	Move
Blockchain	Sui
Method	Manual Analysis, Functional Testing, Automated Testing
Audit Scope	The scope of this Audit was to analyze the Swarm Sui Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Swarm-Network/vault-contract-sui/tree/main/move/sources

Review 1	15th December 2025 - 24th December 2025
Updated Code Received	15th January 2025
Review 2	15th January 2025 - 20th January 2025
Fixed In	https://github.com/Swarm-Network/vault-contract-sui/commit/684973c3d081b835926c388933ccbcc08ec99b75

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	0 (0.0%)
🟥	High	0 (0.0%)
🟧	Medium	1 (10.0%)
🟨	Low	4 (40.0%)
🟪	Informational	5 (50.0%)

Severity

Issues	🟥 Critical	🟥 High	🟧 Medium	🟨 Low	🟪 Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	1	1
Partially Resolved	0	0	0	0	0
Resolved	0	1	3	1	3

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Package Credits Can Be Unfairly Consumed on First Free Vault	High	Resolved
2	Package Names Are Not Enforced to Be Unique	Medium	Resolved
3	Secrets Can Be Added for Non-Members	Medium	Resolved
4	Vault Lifetime Overflow Check Is Ineffective	Medium	Resolved
5	Unlimited Secrets per Vault Remove Incentive for New Vaults	Low	Acknowledged
6	No Validation That Registered Public Key Belongs to Sender	Low	Resolved
7	Owner Removal Protection Is Misleading	Informational	Resolved
8	Package Credit Selection Is Order-Dependent and Non-Deterministic	Informational	Resolved
9	Testnet Vault Lifetime Constant Left in Code	Informational	Acknowledged
10	Redundant Fee Assignment When Using Package Credits	Informational	Resolved

Checked Vulnerabilities

- ☑ Transaction-ordering dependence
- ☑ Timestamp dependence
- ☑ Denial of service / logical oversights
- ☑ Timestamp dependence
- ☑ Access control
- ☑ Code clones, functionality duplication
- ☑ Witness Type
- ☑ Access Management
- ☑ Integer overflow/underflow by bit operations
- ☑ Number of rounding errors
- ☑ Business logic contradicting the specification
- ☑ Number of rounding errors
- ☑ Gas usage
- ☑ Unchecked CALL Return Values
- ☑ Centralization of power
- ☑ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

🟥 Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

🟥 High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

🟧 Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

🟨 Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

🟪 Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

		🟥 High	🟧 Medium	🟨 Low
Likelihood	🟥 High	Critical	High	Medium
	🟧 Medium	High	Medium	Low
	🟨 Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

High Severity Issues

Package Credits Can Be Unfairly Consumed on First Free Vault

Resolved

Path

vault.move

Function Name

`create_vault()`

Description

When creating a vault, the contract checks for available package credits before applying tier-based pricing. This causes a package credit to be consumed even when the user is creating their first vault, which is intended to be free for all users.

As a result, users who already own package credits lose one credit on a vault that should not require payment or credits, creating unfair treatment compared to users without credits.

Impact

Users may unintentionally lose paid package credits for a vault that should have been free, leading to economic unfairness and reduced trust in the pricing model.

Likelihood

High

Recommendation

Explicitly bypass package credit usage when vaults_created == 0, ensuring the first vault is always free and does not consume package credits.

Medium Severity Issues

Package Names Are Not Enforced to Be Unique

Resolved

Path

vault.move

Function Name

`add_package(), update_package()`

Description

The add_package function does not check whether a package name already exists. Since user credits are tracked by package name, packages with duplicate names can cause ambiguous or incorrect credit usage. Also changing package name in update-package can cause users to loss that package.

Impact - High

Duplicate package names may lead to incorrect credit usage or loss/theft when packages differ in price or vault count.

Likelihood

Low - Only admin can make this mistake.

Recommendation

Enforce uniqueness of package names at creation time. Also avoid changing package name in update-package.

Secrets Can Be Added for Non-Members

Resolved

Path

vault.move

Function Name

`add_secret()`

Description

When adding a secret, encrypted data can be provided for any address without verifying that the address is a member of the vault.

```
 public fun add_secret(
        …
        while (I < len) {
            let member = *member_addresses.borrow(i);
            let enc_data = EncryptedData {
                data: *encrypted_data.borrow(i),
                ephemeral_public_key: *ephemeral_keys.borrow(i),
            };
            table::add(&mut secret.encrypted_versions, member, enc_data); //
@audit no check if member is part of vault.members
            I = I + 1;
            …
        };
```

Impact - Medium

Encrypted secrets may be unnecessarily stored for addresses that have no access rights, bypassing payment per member invite, increasing storage costs or creating confusion around access guarantees.

Likelihood

Medium

Recommendation

Validate that each member_address exists in vault.members before accepting encrypted data.

Vault Lifetime Overflow Check Is Ineffective

Resolved

Path

vault.move

Function Name

`check_vault_validity();`

Description

The overflow check in check_vault_validity occurs after computing expiration_epoch. If the addition overflows, the transaction aborts before the check is reached, making the check ineffective.

```
    public fun check_vault_validity(_config: &GlobalConfig, vault: &Vault, ctx:
&TxContext) {
        let current_epoch = ctx.epoch();
        let expiration_epoch = vault.creation_epoch + MAX_VAULT_LIFETIME;

        // If max_vault_lifetime is huge (e.g. u64::MAX), we might overflow if
we add,
        // but since creation_epoch is at most current epoch, and we want to
support “infinite”,
        // we can just check if max_vault_lifetime is a special “forever” value
or just handle basic logic.
        // Actually, simple addition is fine. If it overflows, it’s definitely
future.
        // But to be safe from u64 overflow aborts:
        if (MAX_VAULT_LIFETIME > 18446744073709551615 - vault.creation_epoch) {
// @audit-issue If that addition overflows, the transaction aborts immediately,
so this check is completely pointless because execution would never reach it.
             return // Treated as infinite
        };

        if (current_epoch > expiration_epoch) {
            abort EVaultExpired
        };
    }
```

Impact - High

No immediate vulnerability, but the logic is misleading and unnecessary.

Likelihood

Low

Recommendation

Perform overflow-safe checks before arithmetic or remove the redundant logic entirely.

Low Severity Issues

Unlimited Secrets per Vault Remove Incentive for New Vaults

Acknowledged

Path

vault.move

Function Name

`add_secret()`

Description

There is no limit on the number of secrets that can be stored in a single vault. Users can indefinitely reuse one vault without needing to create or purchase additional vaults.

Impact

This weakens the economic model around vault creation and may reduce demand for additional vault purchases.

Likelihood

High

Recommendation

Clarify whether vaults are intended as so or enforce limits/pricing per secret.

No Validation That Registered Public Key Belongs to Sender

Resolved

Path

user_registry.move

Function Name

`register_user()`

Description

When a user registers their public encryption key, the contract accepts any public_key value provided by the caller without verifying that the key actually belongs to the transaction sender. Since registration is allowed only once per address, an incorrect or malicious key cannot be corrected later.

Impact

This can lead to permanent loss of access to encrypted secrets intended for that user, as future vault sharing relies on the registered key.

Recommendation

Consider adding an off-chain verification flow or on-chain proof mechanism to ensure the registered public key is controlled by the sender, or allow a controlled key rotation mechanism to recover from mistakes.

Informational Issues

Owner Removal Protection Is Misleading

Resolved

Path

vault.move

Function Name

`remove_member()`

Description

The code comments state that the vault owner cannot be removed, but the owner was never added to vault.members on vault creation. As a result, removal is prevented implicitly rather than explicitly.

```
    /// Remove a member from the vault
    public fun remove_member(
        config: &GlobalConfig,
        vault: &mut Vault,
        member: address,
        ctx: &mut TxContext
    ) {
        check_vault_validity(config, vault, ctx);
        assert!(is_admin(vault, ctx.sender()), ENotAuthorized);
        assert!(vault.members.contains(&member), EMemberNotFound);
        assert!(member != vault.owner, ENotAuthorized); // Can't remove owner
<= here
```

Impact

No functional risk, but this adds unnecessary logic and reduces code clarity.

Recommendation

Have the specs align with the implementation contracts as described.

Package Credit Selection Is Order-Dependent and Non-Deterministic

Resolved

Path

vault.move

Function Name

`create_vault()`

Description

When multiple package credits exist, the contract automatically consumes the first matching package based on the order of config.packages. Users cannot choose which package credit is applied to a vault.

Impact

Users may unintentionally consume higher-value or unintended package credits on the wrong vault choice, leading to poor UX and unpredictable behavior especially if package ordering changes.

Recommendation

Allow users to explicitly specify which package credit to use, or document this behavior clearly.

Testnet Vault Lifetime Constant Left in Code

Acknowledged

Path

vault.move

Function Name

`MAX_VAULT_LIFETIME`

Description

MAX_VAULT_LIFETIME is set to a testnet value with a comment reminding to switch to mainnet.

```
// MAINNET: Approx 2,739,726,027.39726 years
// const MAX_VAULT_LIFETIME: u64 = 1_000_000_000_000;

 // TESTNET: Approx 2 days
const MAX_VAULT_LIFETIME: u64 = 2; //
```

Impact

If deployed as-is, vaults may expire much sooner than intended.

Recommendation

Ensure the correct lifetime constant is set before mainnet deployment.

Redundant Fee Assignment When Using Package Credits

Resolved

Path

vault.move

Function Name

`create_vault()`

Description

When a package credit is used, fee is explicitly set to zero, even though the default value is already zero.

```
       public fun create_vault(
                 …
        // Determine fee and free invites
        let mut fee = 0; // Default to 0, will be set by tier
        let mut free_invites = 0;
        let mut used_credit = false;

        // Check for package credits
        let mut i = 0;
        let len = config.packages.length();
        while (I < len) {
            let pkg = config.packages.borrow(i);
            if (table::contains(&user_info.package_credits, pkg.name)) {
                let credit = *table::borrow(&user_info.package_credits,
pkg.name);
                if (credit > 0) {
                    fee = 0; // @audit this is a pointless operation
                    free_invites = pkg.free_invites_per_vault;
                    table::remove(&mut user_info.package_credits, pkg.name);
                    table::add(&mut user_info.package_credits, pkg.name, credit
- 1);
                    used_credit = true;
                    break
                };
            };
            I = I + 1;
        };
            …
```

Impact

Low - This does not break any existing functionality.

Recommendation

Remove the redundant assignment to improve readability.

Centralization Risk

The above audit assumes that all privileged roles act honestly and remain uncompromised under normal operating conditions.

The Swarm Vault protocol relies on centralized administrative authority through the use of privileged roles and shared configuration objects. These roles have the ability to control critical protocol behavior, including pricing, packages, treasury withdrawals, and system-wide configuration.

Privileged Capabilities Include:

- Managing vault pricing tiers and package definitions
- Updating invite fees and economic parameters
- Withdrawing protocol funds from the treasury
- Modifying global configuration affecting all users

Risks of Centralization

A compromised or malicious admin authority could:

- Modify pricing or packages in a way that unfairly disadvantages users
- Drain protocol funds via treasury withdrawals
- Change economic assumptions without user consent
- Disrupt protocol usage by misconfiguring global parameters

Roles should be assigned to multisig/timelock and add observability.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
GlobalConfig	Admin configuration updates	Malicious or compromised admin can arbitrarily change pricing, packages, or invite fees, impacting user economics and fairness.
GlobalConfig	Treasury withdrawal	Full withdrawal of protocol funds if admin role is compromised, resulting in loss of treasury assets.
Vault	create_vault	Economic bypass if vault pricing logic is misconfigured or admin-controlled parameters are abused.
Vault	add_secret	Unbounded secret storage can be abused to bloat on-chain state or bypass intended economic incentives.
Vault	remove_member	Incorrect assumptions about owner membership may lead to future logic errors or unintended access control changes.
UserRegistry	Package management	Package reordering or misconfiguration can cause unexpected credit usage or unfair advantages.
GlobalConfig	check_vault_validity	Misconfigured vault lifetime parameters could prematurely expire or indefinitely extend vault access.
AdminCap	Any admin-gated function	Full protocol control if admin capability is compromised; represents the highest-risk trust assumption.

Closing Summary

In this report, we have considered the security of VAULT by Swarm Network Smart Contract. We performed our audit according to the procedure described above.

Issues of Medium, low and informational were found. The Swarm Network team acknowledged a few issues and fixed the others.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

January 2026

For

SWARM

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com