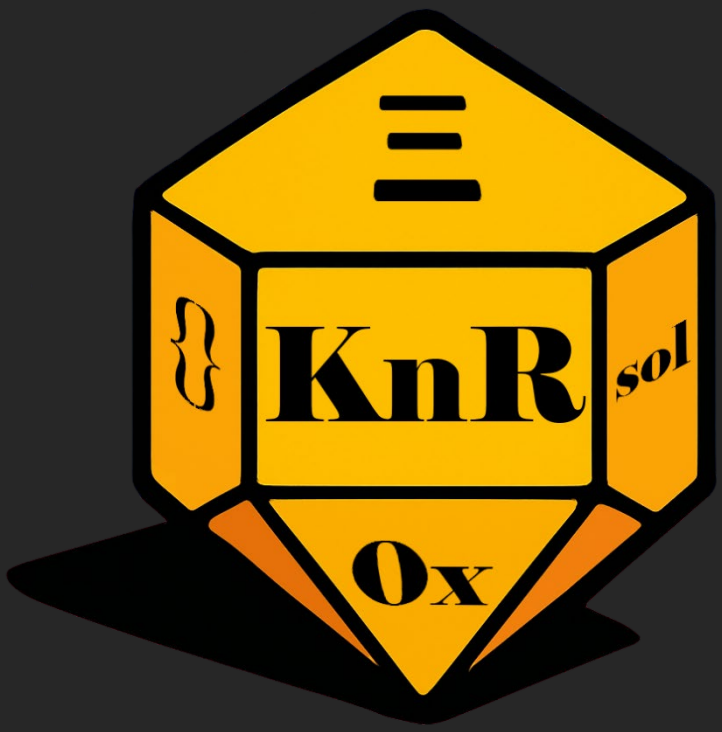

Kode-n-Rolla

Secret Vault on Aptos Security Review

Aug 14th 2025 - Aug 21st 2025

Table of Contents

- **Whoami**.....3
- **Disclaimer**.....3
- **Risk Classification**.....3
- **Protocol Summary**.....4
- **Audit Details**.....5
  - **Scope**.....5
  - **Roles**.....5
- **Executive Summary**.....5
  - **Issue found**.....6
- **Findings**.....7
- High.....7
- [H-01] Unauthorized Secret Disclosure via Broken Authentication.....7

Whoami

I am Pavel (aka *kode-n-rolla*), an independent security researcher.

Driven by a passion for decentralization and security, I focus on strengthening blockchain ecosystems.

I believe that security is an ongoing process, and my work contributes to making systems more resilient and reliable.

Disclaimer

This report reflects a time-boxed security review performed as part of a ***CodeHawks*** audit contest.

The analysis focuses exclusively on the security aspects of the reviewed ***Move*** codebase and does not guarantee the discovery of all potential vulnerabilities.

Findings represent the issues identified during the allotted review period and should not be considered an endorsement of the underlying protocol, business model, or operational security practices.

Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	H	H/M	M
Likelihood: Medium	H/M	M	M/L
Likelihood: Low	M	M/L	L

I use *CodeHawks* severity matrix to determine severity.

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- *High Impact*:
    - Funds are directly or nearly directly at risk

    - There's a severe disruption of protocol functionality or availability.
- *Medium Impact*:
    - Funds are indirectly at risk.
    - There's some level of disruption to the protocol's functionality or availability.
- *Low Impact*:
    - Funds are not at risk.
    - However, a function might be incorrect, the state might not be handled appropriately, etc.

Protocol Summary

`SecretVault` is a *Move-based* smart contract application deployed on the Aptos blockchain.

The protocol is designed to store a secret value in on-chain global storage, ensuring that only the designated owner is authorized to set and retrieve the secret.

The contract leverages Move's resource-oriented programming model and ***Aptos*** account-based authentication to enforce ownership and access control.

Unauthorized accounts should not be able to read or modify the stored secret under any circumstances.

From a security perspective, the protocol touches on several fundamental concepts specific to the ***Move*** language and the ***Aptos*** execution model, including ownership semantics, resource safety, global storage access patterns, and event emission behavior.

This audit review focuses on assessing whether the implemented access controls and ownership assumptions are correctly enforced and whether the contract logic can be bypassed or misused to gain unauthorized access to the protected data.

Audit Details

This audit was conducted as part of a time-boxed competitive security review hosted on the ***CodeHawks*** platform. The assessment focused exclusively on the *Move* smart contract logic and its adherence to intended game mechanics, security assumptions, and fund safety.

The review did not include off-chain components, frontend integrations, or economic modeling beyond what is explicitly enforced by the contract code.

The review was performed during the "Secret Vault on Aptos" ***CodeHawks*** audit event.

Scope

The audit scope consisted of the following smart contract:

```
./sources/
└── secret_vault.move
./Move.toml
```

Roles

Owner

The *Owner* is the account authorized to initialize the vault and is the only entity permitted to set and retrieve the stored secret. All sensitive operations are expected to be restricted exclusively to this role.

Executive Summary

This security review evaluated the *SecretVault* smart contract deployed on the Aptos blockchain, with a focus on

access control, ownership enforcement, and data confidentiality within Move's global storage model.

The review identified one high-severity issue. The core design assumption of the protocol — that a secret can be securely stored on-chain and accessed exclusively by its owner — does not hold in practice. Due to the inherent transparency of blockchain state, the stored secret is publicly accessible, regardless of the implemented access control logic.

While the contract correctly restricts write and read operations at the function level, this does not prevent external observers from accessing the secret directly from global on-chain storage. As a result, the protocol fails to provide the confidentiality guarantees it claims to offer.

This issue does not stem from an implementation bug, but from a fundamental mismatch between the protocol's intended security properties and the underlying execution and data availability model of the blockchain. Consequently, the protocol cannot be considered secure for use cases requiring secrecy or confidentiality of stored data.

Issue found

Severity	Issue
High	Unauthorized Secret Disclosure via Broken Authentication

Findings

High

[H-01] Unauthorized Secret Disclosure via Broken Authentication

Description

The protocol intends to restrict access to the stored secret exclusively to the owner account. Only the owner should be able to store a secret and retrieve it later.

However, the *get_secret* function performs authorization by comparing a **user-supplied address parameter** to the named address *@owner*, instead of authenticating the actual transaction sender. The function does not require a &signer, and therefore does not verify who is calling it.

As a result, any account can call *get_secret* and supply the owner's address as an argument, successfully bypassing access control and retrieving the secret.

Authentication is implemented using a caller-controlled input (*address*) rather than the actual caller identity (*&signer*). This breaks the trust boundary between the caller and the protected resource.

```
    //// view functions

    #[view]
@>  public fun  get_secret (caller: address):String acquires Vault{
    assert! (caller == @owner,NOT_OWNER);
    let vault = borrow_global<Vault >(@owner);

        vault.secret
    }
```

Impact

- Any account can read the owner's secret without authorization.
- The core security guarantee of the protocol ("only the owner can access the secret") is violated.

- The data breach is irreversible: once the secret is read from on-chain state, confidentiality is permanently lost.

Likelihood

- The function is marked as public and `#[view]`, making it callable by any account.
- The owner's address is publicly discoverable after deployment (e.g., via on-chain metadata or explorers).
- No special privileges or setup are required to exploit the issue.

Proof of Concept

A unit test demonstrates the issue by storing a secret as the owner and then calling `get_secret(owner_address)` from an arbitrary account. The call succeeds and returns the secret, confirming that access control is enforced solely via a user-supplied parameter rather than caller authentication.

```
#[test(owner = @0xcc, attacker = @0x123)]
fun test_leak_owners_secret(owner: &signer, attacker: &signer) acquires Vault {
    use aptos_framework::account;

    // Prepare account(s)
    account::create_account_for_test(signer::address_of(owner));
    account::create_account_for_test(signer::address_of(attacker)); // not
necessary

    // Owner stores a secret
    let s = b"i'm a secret";
    set_secret(owner, s);

    // Save owner`s address
    let owner_addr = signer::address_of(owner);

    // Exploit: anyone can pass `owner_addr` to the view and read the secret
    let leaked = get_secret(owner_addr);

    // Verify leakage
    assert!(leaked == string::utf8(s), 100);
}
```

Recommended Mitigation

Authenticate the actual caller via &signer (and keep storage/reads bound to *@owner*).

```
-    public fun get_secret(caller: address): String acquires Vault {
-        assert!(caller == @owner, NOT_OWNER);
-        let vault = borrow_global<Vault>(@owner);
-        vault.secret
-    }
+    public fun get_secret(caller: &signer): String acquires Vault {
+        // Authorize by signer, not by user-supplied address
+        assert!(signer::address_of(caller) == @owner, NOT_OWNER);
+        let v = borrow_global<Vault>(@owner);
+        // Return a copy of the secret; use your project's preferred string copy
helper
+        string::clone(&v.secret)
+    }
```