Aptos Secret Vault Security Review

Reviewer: Muhammad Faran **Date:** July 27, 2025 **Repo:** CodeHawks-Contests/2025-07-secret-vault

---

Table of Contents

Disclaimer

I made all effort to find as many vulnerabilities in the code in the given time period, but hold no responsibilities for the findings provided in this document. A security audit is not an endorsement

of the underlying business or product. The audit was time-boxed and view of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk-Classification

The severity of each finding combines **Impact** (the damage if exploited) and **Likelihood** (the chance of exploitation). The matrix below illustrates how these two axes map to a single severity label:

Likelihood ↓ / Impact →	**High**	**Medium**	**Low**
High	**High**	**High/Medium**	**Medium**
Medium	**High/Medium**	**Medium**	**Medium-Low**
Low	**Medium**	**Medium-Low**	**Low**

Executive Summary

Severity	**Number Of Issues Found**
High	**2**
Medium	**1**
Low	**1**

Audit Details

Scope

- In Scope:

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

Protocol Summary

SecretVault is a Move smart contract application for storing a secret on the Aptos blockchain. Only the owner should be able to store a secret and then retrieve it later. Others should not be able to access the secret.

Actors

Owner - Only the owner may set and retrieve their secret

[H-1] On-Chain Data Privacy Illusion and Unauthorized Secret Access

Description

- The `Vault` struct stores sensitive information in the `secret` field, with the expectation that access control through `get_secret()` will keep this data private
- The fundamental issue is that all on-chain state is publicly readable by anyone with access to the blockchain, regardless of access control functions. The `secret` field is stored in plain text on-chain and can be accessed directly through state reads, completely bypassing the `get_secret()` authorization check

```
struct Vault has key {
    secret: String  // @> This is stored on-chain in plain text, accessible
to everyone
}

#[view]
public fun get_secret(caller: address): String acquires Vault {
    assert!(caller == @owner, NOT_OWNER);  // @> Access control is
meaningless for on-chain data
    let vault = borrow_global<Vault>(@owner);
    vault.secret
}
```

Risk

Likelihood: * Any user with blockchain access can read the global state directly using `borrow_global<Vault>(@owner)` * Block explorers and indexing services automatically expose all on-chain state data * Full nodes maintain complete state history, making all historical secrets permanently accessible * State dumps and debugging tools provide direct access to struct fields

Impact: * Complete exposure of confidential information to unauthorized parties * Violation of user privacy expectations and potential regulatory compliance issues * Permanent data exposure (blockchain immutability means secrets cannot be retroactively hidden) * Loss of trust in the application's security model

Proof of Concept

- Add this test to `secret_vault.move` file and see the results

```
#[test(owner = @0xcc)]
fun test_secret_leak(owner: &signer) acquires Vault {
```

```
    use aptos_framework::account;

    account::create_account_for_test(signer::address_of(owner));

    // Owner sets a secret
    let secret = b"I Love ...";
    set_secret(owner, secret);

    // Attacker reads the state directly (bypasses get_secret)
    let vault = borrow_global<Vault>(@0xcc);
    let stolen_secret = &vault.secret;
    debug::print(stolen_secret);  // Successfully prints the "secret"
}
```

Recommended Mitigation

```
- struct Vault has key {
-     secret: String
- }

+ // Option 1: Remove on-chain storage entirely
+ // Store sensitive data off-chain with on-chain references only
+
+ struct Vault has key {
+     secret_hash: vector<u8>,  // Store hash for verification
+     // other non-sensitive metadata
+ }
+
+ // Option 2: Use commitment scheme
+ public entry fun commit_secret(caller: &signer, commitment: vector<u8>) {
+     let secret_vault = Vault { secret_hash: commitment };
+     move_to(caller, secret_vault);
+ }
+
+ public fun verify_secret(caller: address, secret: vector<u8>): bool
acquires Vault {
+     let vault = borrow_global<Vault>(caller);
+     // Verify against commitment/hash
+     aptos_hash::sha3_256(secret) == vault.secret_hash
+ }
```

[H-2] Authentication Bypass in Access Control leads to Complete Authorization Failure

Description

- The `get_secret()` function is designed to restrict access to only the owner by checking if the `caller` parameter equals `@owner`
- The critical flaw is that `get_secret()` is a view function that accepts an arbitrary `caller` parameter, allowing any user to simply pass `@owner` as the argument, making the authentication check meaningless since attackers control the input being validated

```
#[view]
public fun get_secret(caller: address): String acquires Vault {
    assert!(caller == @owner, NOT_OWNER);  // @> Attacker can pass @owner
directly
    let vault = borrow_global<Vault>(@owner);
    vault.secret  // @> Access granted to anyone who passes @owner
}
```

Risk

Likelihood: * Any external caller can invoke `get_secret(@owner)` directly, bypassing authentication * No actual verification of the caller's identity occurs - only validation of a user-controlled parameter * The vulnerability is trivially exploitable with a single function call

Impact: * Complete bypass of intended access controls for secret retrieval * Unauthorized users gain full access to sensitive information through legitimate function calls * Authentication mechanism provides false security, masking the actual vulnerability * Could lead to privilege escalation if similar patterns exist elsewhere in the codebase

Proof of Concept

- Add this test to `secret_vault.move` file and see the results

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

    account::create_account_for_test(signer::address_of(owner));

    // Owner sets the secret
    let secret = b"I Love ...";
    set_secret(owner, secret);

    // Attacker calls get_secret with caller=@owner
    let leaked_secret = get_secret(@0xcc);

    // Debug-print what attacker got
    debug::print(&string::utf8(b"Attacker bypassed auth and got:"));
    debug::print(&leaked_secret);
}
```

Recommended Mitigation

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

+ // Option 1: Use &signer instead for authentication
+ public entry fun get_secret(caller: &signer): String acquires Vault {
+     assert!(signer::address_of(caller) == @owner, NOT_OWNER);
```

```
+     let vault = borrow_global<Vault>(@owner);
+     vault.secret
+ }
+
+ // Option 2: Remove the function entirely if not needed
+ // Since blockchain state is public anyway, this function provides no real
value
+
```

[M-1] Resource Existence Collision DoS on Secret Updates

Description

- The `set_secret()` function is intended to allow owners to set and update their vault secrets
- The function uses `move_to()` unconditionally, which will abort if a `Vault` resource already exists at the caller's address. After the first successful call to `set_secret()`, all subsequent calls will fail with a runtime error, permanently preventing secret updates or rotation

```
public entry fun set_secret(caller: &signer, secret: vector<u8>) {
    let secret_vault = Vault { secret: string::utf8(secret) };
    move_to(caller, secret_vault);  // @> This aborts if Vault already exists
    event::emit(SetNewSecret {});
}
```

Risk

Likelihood: * Any second call to `set_secret()` by the same user will trigger an abort * The issue occurs immediately after the first successful secret creation * No workaround exists within the current contract design * The problem is deterministic and affects all users who attempt to update their secrets

Impact: * Complete denial of service for secret rotation functionality after first use * Users cannot update compromised or outdated secrets * Poor user experience with cryptic runtime errors on legitimate operations * Security degradation as users cannot respond to potential secret compromises * Business logic failure preventing normal application workflows

Proof of Concept

- Add this test to `secret_vault.move` file and see the results

```
#[test(owner = @0xcc)]
fun test_set_secret_dos(owner: &signer) {
    use aptos_framework::account;

    account::create_account_for_test(signer::address_of(owner));

    let secret1 = b"first-secret";
    let secret2 = b"rotated-secret";
```

```
    // First set_secret call should succeed
    set_secret(owner, secret1);

    // This proves the DoS bug - second call will abort
    set_secret(owner, secret2);  // ABORT: Resource already exists
}
```

Recommended Mitigation

- Here is the recommended mitigation

```
public entry fun set_secret(caller: &signer, secret: vector<u8>) {
-   let secret_vault = Vault { secret: string::utf8(secret) };
-   move_to(caller, secret_vault);
-   event::emit(SetNewSecret {});

+   let caller_address = signer::address_of(caller);
+   let secret_vault = Vault { secret: string::utf8(secret) };
+
+   // Check if vault already exists and handle accordingly
+   if (exists<Vault>(caller_address)) {
+       // Update existing vault
+       let vault_ref = borrow_global_mut<Vault>(caller_address);
+       vault_ref.secret = string::utf8(secret);
+   } else {
+       // Create new vault
+       move_to(caller, secret_vault);
+   };
+
+   event::emit(SetNewSecret {});
}
```

[L-1] Empty Event Payload leads to Poor Observability

Description

- The `SetNewSecret` event is designed to notify external systems and users when a secret is set or updated in the vault
- The event structure contains no fields or payload data, making it impossible for observers to distinguish between different secret operations, identify which user performed the action, or gather any meaningful context about the state change

```
#[event]
struct SetNewSecret has drop, store {}  // @> No fields - provides no useful
information

public entry fun set_secret(caller: &signer, secret: vector<u8>) {
    let secret_vault = Vault { secret: string::utf8(secret) };
    move_to(caller, secret_vault);
    event::emit(SetNewSecret {});  // @> Emits empty event with no context
}
```

Risk

Likelihood: * Every call to `set_secret()` emits an event with zero informational value * External monitoring systems receive events but cannot determine the actor or context * Off-chain applications relying on event data will lack critical operational information * The issue affects all users and all secret operations consistently

Impact: * Severely degraded observability for security monitoring and auditing * External systems cannot track which users are setting secrets or when * Impossible to correlate events with specific addresses or operations

Proof of Concept

- No proof of concept needed - the issue is evident from the empty event structure definition.

Recommended Mitigation

- Here is the fix for the event

```
#[event]
- struct SetNewSecret has drop, store {}
+ struct SetNewSecret has drop, store {
+     owner: address,
+     timestamp: u64,
+     // Note: Don't include the actual secret for privacy
+ }

public entry fun set_secret(caller: &signer, secret: vector<u8>) {
    let secret_vault = Vault { secret: string::utf8(secret) };
    move_to(caller, secret_vault);
-   event::emit(SetNewSecret {});
+   event::emit(SetNewSecret {
+       owner: signer::address_of(caller),
+       timestamp: aptos_framework::timestamp::now_microseconds(),
+   });
}
```