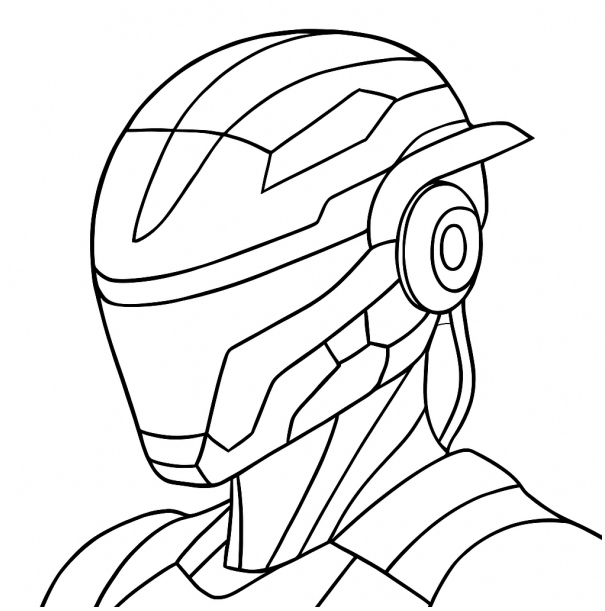

Secret Vault on Aptos Audit Report

Version 1.0

X: @AlexScherbatyuk

August 21, 2025

Secret Vault on Aptos Audit Report

Alexander Scherbatyuk

August 21, 2025

Prepared by: X: @AlexScherbatyuk Lead Auditors: - Alexander Scherbatyuk

Table of Contents

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.

Disclaimer

The @AlexScherbatyuk makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

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

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

```
sources/
--- secret_vault.move
```

Roles

Owner - Only the owner may set and retrieve their secret

Executive Summary

Issues found

Severity	Number of issues found
High	0
Medium	1
Low	1
Info	0
Gas Optimizations	0
Total	2

Findings

Medium

[H-1] Unauthorized retrieve, anyone can read an owner secret

Description: Normal behavior is that only the owner may set and retrieve their secret. Current behavior is that anyone can read the owner's secret by simply passing @owner into the view function. The function authorizes based on a user-provided address parameter, not the transaction signer.

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

Impact: Breaks core confidentiality and main functionality. The protocols that trusted their users' secrets to be stored using this contract are at risk of a security breach. Users may potentially lose data or funds.

Proof of Concept: The test shows that a non-owner can read the owner's secret by calling the view function with @owner as the argument. There's no signer-based auth; the function authorizes purely on the caller-supplied parameter.

```
    #[test(owner = @0xcc, attacker = @0x456)]
    public fun test_anyone_can_read_owner_secret(owner: &signer,
        attacker: &signer) {
        account::create_account_for_test(signer::address_of(owner));
        account::create_account_for_test(signer::address_of(attacker));

        let secret = b"top secret";
        vault::set_secret(owner, secret);

        let leaked: String = vault::get_secret(@owner);
        assert!(leaked == string::utf8(secret), 100);
    }
```

Recommended Mitigation: Enforce signer-based authentication for secrets to be read only by actual owner.

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

Low

[L-1] Secret cannot be updated, no update path (owner cannot change secret due to move_to overwrite abort)

Description: Normal behaviour is that owner can use update vault Current behaviour owner cannot update secret, due to move_to overwrite abort.

```

  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 {});
  }
```

Impact: The current architecture allows one secret per vault, so the update is the intended behavior. Breaks functionality and disrupts the intended path.

Proof of Concept: This test proves that the owner cannot update the secret because move_to cannot overwrite an existing resource.

```
    #[test(owner = @0xcc)]
    #[expected_failure]
    public fun test_owner_cannot_update_secret(owner: &signer) {
        account::create_account_for_test(signer::address_of(owner));
        let secret1 = b"first";
        let secret2 = b"second";
        vault::set_secret(owner, secret1);
        // This should abort because move_to cannot overwrite an
           existing resource
        vault::set_secret(owner, secret2);
    }
```

Recommended Mitigation: This mitigation allows the owner to update the secret or create a vault and set an initial secret. Additionally, this code enforces access restrictions, allowing only the owner to set or update a secret.

```
- public entry fun set_secret(caller:&signer,secret:vector<u8>){
+ public entry fun  set_secret(caller: &signer, secret: vector<u8>)
   acquires Vault {
+    assert!(signer::address_of(caller) == @owner, NOT_OWNER);
+    if (exists<Vault>(@owner)) {
+        let vault_ref = borrow_global_mut<Vault>(@owner);
+        vault_ref.secret = string::utf8(secret);
+    } else {
+        move_to(caller, Vault { secret: string::utf8(secret) });
+    }
      let secret_vault = Vault{secret: string::utf8(secret)};
      move_to(caller,secret_vault);
     }
    event::emit(SetNewSecret {});
}
```