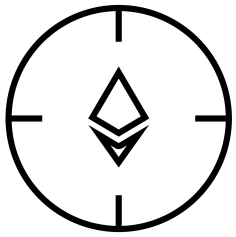

DedEth

Velocore Security Review

Reviewers

DedOhwale & ArzDev

Table Of Contents

1. High Vulnerability
2. Medium Vulnerability
3. Analysis
4. Disclaimer

High Vulnerability

H-01 Delegatecall to extort() fails when ether is sent which can be abused to receive more rewards

Summary

When `execute()` is called with opType 3, a delegatecall is made to `extort()`. The problem is that when using delegatecall the `msg.value` is retained but `extort()` is not payable so the delegatecall will fail if ether is sent.

Vulnerability Details

When `extort` is invoked via delegatecall from execute, it operates in execute's storage and execution context. This means `msg.value` is inherited. However, `extort()` is not a payable function, so if ether is sent, the delegatecall will fail. The whole transaction will not revert, but `r.snapshots[user]` is not updated while `g.userVotes[user]` is, and both of these values are used in `extort()` to calculate `userClaimed`:

```
uint256 userClaimed = (r.current - r.snapshots[user]) * uint256(g.userVotes[user]) / 1e18;
```

An attacker could execute opType 3 and send 1 wei causing the extort() function to fail.

```
r.snapshots[user] = r.current;
```

Because the delegatecall failed, `r.snapshots[user]` won't be set and will remain 0. This is the attacker's first time voting for a gauge. After extort(), the attacker can still cast votes, thus increasing his `g.userVotes`.

```
gauge.userVotes[user] = (int256(uint256(gauge.userVotes[user])) + deltaVote).toUint256().toU
```

When the attacker calls execute with opType 3 again, but without sending ether, `extort()` will execute. However, `userClaimed` won't be 0 because the attacker increased his `g.userVotes`, resulting in `userClaimed` being `(r.current - 0) * g.userVotes[user]`.

This will allow the attacker to get much more rewards than he is supposed to get.

Impact

The attacker will improperly obtain tokens from the vault because he gets many more rewards than he is supposed to get.

Also, as stated in the docs, users can vote directly with ether so if they do, `extort()` will fail and `r.snapshots[user]` is not going to be updated while the `gauge.userVotes[user]` will be updated leaving the users with more rewards than they are supposed to get

Proof of Concept

1. Attacker calls `execute()` with opType 3 and sends 1 wei, he has never voted before for that gauge
2. The delegatecall to `extort()` fails because it's not payable and the `r.snapshots[user]` is not updated
3. After `extort()`, the attacker uses veLVC to vote which increases his `g.userVotes`
4. The attacker calls execute() opType 3 again but without sending ether
5. `extort()` is not going to fail this time and r.snapshots[user] will be 0, the g.userVotes is not going to be 0 so the `userClaimed` will be a large number and the attacker will receive a lot of rewards
6. He can then unvote to receive his veLVC back

Recommended Mitigation Steps

The fix is simple: make the extort function payable.

Medium Vulnerability

M-02 Missing a deadline check for `execute()`

Summary

The execute() function is missing a deadline check. Without a deadline, the transaction might be left hanging in the mempool and it can be executed way later than the user wanted.

Vulnerability Details

DEXs should provide their users with an option to limit the execution of their pending actions, such as swaps or adding and removing liquidity. A deadline check can be a useful tool to ensure that your transaction cannot be "saved for later".

The execute() function in `SwapFacet.sol` is missing a deadline check. The result of this is that it becomes more profitable for a miner to deny the transaction from being mined until the transaction incurs the maximum amount of slippage.

The most common solution is to include a deadline timestamp as a parameter(see Uniswap). This is used to prevent users from swapping after significant price movements and this limits miners holding signed transactions for extended durations and executing them based on market movements. The user might submit a transaction fee that is too low for miners to be interested in including the transaction in the block.

Impact

The users may execute swaps following significant market movements or a miner could deny the transaction from being mined until the transaction incurs the maximum amount of slippage

Recommended Mitigation Steps

It is advisable to implement a deadline check like Uniswap and other DEXs have, and to require users to pass the deadline as a parameter in the function call.

Acknowledged by the team.

Analysis

Diamond

- On creation (`code` block of "Diamond"), it uses `delegatecall` to call back into the contract that initiated the creation. The expectation is that this caller initializes the storage.
- If the `delegatecall` fails, the initialization process will revert with no message.
- If it succeeds, it copies the "Runtime" code into memory and returns that, essentially setting the runtime code of the Diamond.

Runtime Behavior

The `Runtime` code provides the operational functionality of the Diamond:

1. **'read' ingrained function (0x72656164)**:
   - A cheap mechanism to directly read storage slots.
   - Expects a series of storage slots in the calldata (starting from the 5th byte).
   - For each slot, it fetches the corresponding storage value and stores it in memory.
   - Returns these storage values.
2. **'view' ingrained function (0x76696577)**:
   - Uses `delegatecall` to any contract and expects that call to revert.
   - The purpose is to simulate the result of a state-modifying function without actually changing state.
   - If the delegate call succeeds, it reverts with the returned data.
   - If the delegate call reverts, it simply returns that data.
3. **Function Calls via Diamond Storage**:
   - Fetches the function's implementation address from storage using the function's selector as a key.
   - If there's an implementation address:
     - If it's registered as a normal function, the code uses `delegatecall` to execute that function.
     - If it's registered as a view function, the Diamond's ingrained 'view' function is used to execute it.

- If no matching function is found, the call reverts.

VaultStorage

1. Overview

The `VaultStorage` contract acts as a base storage contract for the "Vault" system, maintaining its critical state and struct definitions. It appears to be

built using the diamond pattern, as seen with the use of facets and function routing.

2. Intentional Functionality

1. **Storage Structs**:
    - `EmissionInformation`: Contains information about token emissions, including a mapping for individual `GaugeInformation`.
    - `GaugeInformation`: Contains details for individual gauges, including votes, bribes, and rewards.
    - `Rewards`: Tracks token distribution for rewards.
    - `RoutingTable`: Maintains the routing table for the diamond pattern, with function selectors tied to implementations.
    - `FacetCut`: Defines an action to be executed in the diamond pattern (add, replace, remove facet).
2. **Events**: Various events like `Swap`, `Gauge`, `Convert`, `Vote`, etc., provide important insights into the operations performed on the contract.
3. **Facet Management**: Using `_setFunction` and `_setViewer`, facets (small modular contracts) can be added, replaced, or removed. This is fundamental to the diamond pattern.
4. **Storage Accessors**: The contract uses functions like `_routingTable`, `_poolBalances`, `_e`, and `_userBalances` to get specific storage slots. This is essential to ensure that storage doesn't collide and is efficiently accessed.
5. **Modifiers**:
    - `nonReentrant`: Protects against reentrancy attacks.
    - `whenNotPaused`: Ensures certain functions can't be run while the contract is paused.
    - `authenticate`: Validates if the caller is authenticated to perform an action.
6. **Authentication**: The `authenticateCaller` function checks permissions via an external `IAuthorizer` contract.

3. Notable Observations

1. **Diamond Pattern**: This design pattern is apparent in the contract, especially with function routing and facet management. The diamond pattern provides a way to maximize contract storage and utility by composing behaviors from multiple small contracts (facets).
2. **Bribe Mechanism**: The system allows for the concept of "bribes", contracts which can be extorted for bribes on demand. This is a unique feature not typically found in standard vault or DeFi designs.

3. **Emission System**: The contract has a detailed emission system where rewards can be tracked on a per-vote and per-bribe basis. This offers a granular insight into how rewards are distributed.
4. **Authorization and Authentication**: Authorization is delegated to an external contract (`IAuthorizer`). This can be a powerful design pattern for separating concerns, but its security and correctness are contingent on the external contract.
5. **Assembly Usage**: The contract uses inline assembly for specific storage operations and optimizations, particularly for the diamond pattern's function routing.

States:

1. **Uninitialized**: The contract is yet to be properly initialized.
2. **Initialized**: The contract has been initialized and is ready for operation.
3. **Paused**: The contract operation is halted temporarily.
4. **Operational**: The contract is actively accepting and processing operations.
5. **Deprecated**: The contract is marked obsolete and will not support further operations.

Transitions:

1. InitializeContract

- **From**: `Uninitialized`
- **To**: `Initialized`
- **Function**: `initialize()`
- **Attack Vectors**:
  - Ensure that the function can be called only once.
  - Ensure initial parameters, if any, are passed securely.
  - Utilize OpenZeppelin's initializer modifier to prevent double initialization.

2. PauseContract

- **From**: `Operational`
- **To**: `Paused`
- **Function**: `pause()`
- **Attack Vectors**:
  - Restrict function to authorized addresses.
  - Prevent potential denial of service by verifying current state.

3. **ResumeContract**

- **From**: `Paused`
- **To**: `Operational`
- **Function**: `resume()`
- **Attack Vectors**:
  - Restrict function to authorized addresses.
  - Verify current state to ensure correct transition.

4. **DeprecateContract**

- **From**: `Operational` or `Paused`
- **To**: `Deprecated`
- **Function**: `deprecate()`
- **Attack Vectors**:
  - Restrict function to authorized addresses.
  - Ensure irreversible transition to `Deprecated` state.

SwapFacet

The `SwapFacet` contract provides functionalities related to token swaps, staking, and voting.

General Observations:

- The contract uses libraries extensively for a modular design.
- Uses UncheckedMemory for operations that might involve overflows and underflows.
- Many of the functions utilize assembly for optimized operations, but this makes the code harder to read and understand.

States

1. **Uninitialized**: When the `SwapFacet` contract has just been deployed but hasn't been initialized yet.
2. **Initialized**: Once the `initializeFacet` function has been called, and the contract is ready for token swap operations.
3. **Executed**: After the `execute` function is called for token operations.
4. **Queried**: After the `query` function is called.

Transitions

1. **Uninitialized -> Initialized**

- **Function**: `initializeFacet`
- **Attack vectors**:
  - If not properly secured, unauthorized users can call this function and manipulate the initialization process.
  - If the contract logic is not idempotent, multiple initializations can cause unwanted behaviors.

2. **Initialized -> Executed**
   - **Function**: `execute`
   - **Attack vectors**:
     - Reentrancy attacks since external calls to token contracts are made. However, the `nonReentrant` modifier is used to prevent this attack.
     - Possible overflows or underflows in calculations, although the contract uses SafeCast which should handle most of these issues.
     - The unchecked block can be a cause for concern if not used correctly.
     - If token operations aren't correctly sorted or managed, there's a risk of tokens being lost or mishandled.
     - Dependence on external contracts such as the token contracts, pool, or the converter contracts which can introduce vulnerabilities if those contracts are malicious or buggy.
3. **Initialized -> Queried**
   - **Function**: `query`
   - **Attack vectors**:
     - Similar to the `execute` function, reentrancy attacks and mishandling of tokens are concerns here.
     - As this function reads and potentially manipulates states, front-running attacks might be feasible where attackers observe a transaction and try to act before it.

AdminFacet

Overview

`AdminFacet` is a facet of a diamond architecture (based on the EIP-2535 Diamond Standard). It provides administrative functionality to deploy and configure the diamond.

Details & Vulnerabilities

Constructor:

- The constructor initializes the `deployer`, `initialAuth`, and `thisImplementation` state variables. It's crucial that the deployer address is trustworthy, as it has permission to deploy using this facet.

Function: `deploy`

- **Attack Vector:** The function allows the `deployer` to deploy any bytecode provided. If an attacker gains control of the `deployer` address, they can deploy malicious contracts.

Function: `deploy_zksync`

- This function allows the deployer to create a new Diamond contract. Same attack vector as the above function if the `deployer` address gets compromised.

Fallback Function:

- The fallback function is expected to be called by the `Diamond.yul` constructor.
- **Attack Vector:** It performs a delegate call to `thisImplementation` with `initializeFacet.selector`. If there's an issue with `thisImplementation` or the target function, the delegate call will execute unintended logic.

Function: `initializeFacet`

- Initializes function selectors for the admin functions and sets the authorizer if it hasn't been set before.
- **Note:** Ensure that `StorageSlot` utilities are well-audited and correctly implemented to avoid storage collision vulnerabilities.

Function: `admin_setFunctions`

- Allows an authenticated admin to set multiple function selectors to point to a given implementation.
- **Attack Vector:** If an attacker gains the capability to authenticate (e.g., compromised admin key, issues in the `authenticate` logic), they can redirect function calls to malicious implementations.

Function: `admin_pause`

- Pauses/unpauses functionalities based on a boolean parameter.

Function: `admin_addFacet`

- Authenticates the caller and then performs a delegate call to `initializeFacet` of another facet implementation.
- **Attack Vector:** Similar to `admin_setFunctions`, if an attacker can authenticate, they can initialize a malicious facet.

Function: `admin_setAuthorizer`

- Updates the authorizer for the contract.
- **Attack Vector:** If the authentication mechanism gets compromised, attackers can change the authorizer to a malicious or incorrect one, affecting access controls and leading to a potential loss of funds or data.

Function: `admin_setTreasury`

- Allows authenticated admin to set the treasury address.
- **Attack Vector:** If the authentication mechanism gets compromised, attackers can redirect funds or assets to a malicious address.

States:

1. **Uninitialized** - The diamond is deployed but has not had its functions or facets initialized.
2. **Initialized** - The diamond has its initial facets and functions set up.
3. **Paused** - Some functionalities of the diamond are paused.
4. **Resumed** - The functionalities that were paused have been resumed.
5. **AuthorizerUpdated** - The authorizer has been changed.
6. **TreasuryUpdated** - The treasury address has been changed.
7. **FunctionsUpdated** - Function routing has been updated.
8. **NewFacetAdded** - A new facet has been added.

Transitions:

1. Uninitialized -> Initialized
- Function: `initializeFacet()` - Attack Vectors: - Unauthorized access if the `authenticate` mechanism is compromised. - Possible storage collision if `StorageSlot` utility is not well-audited.

2. Any State (except Uninitialized) -> Paused
- Function: `admin_pause(true)` - Attack Vectors: - Unauthorized pausing if the `authenticate` mechanism is compromised.

3. Paused -> Resumed
- Function: `admin_pause(false)` - Attack Vectors: - Unauthorized resumption if the `authenticate` mechanism is compromised.

4. Any State (except Uninitialized) -> AuthorizerUpdated
- Function: `admin_setAuthorizer(new_authorizer)` - Attack Vectors: - Unauthorized change of authorizer if the `authenticate` mechanism is compromised. - Redirecting to a malicious or incorrect authorizer, leading to other potential vulnerabilities.

5. Any State (except Uninitialized) -> TreasuryUpdated
- Function: `admin_setTreasury(new_treasury)` - Attack Vectors: - Unauthorized change of treasury if the `authenticate` mechanism is compromised. - Redirecting funds or assets to a malicious address.

6. Any State (except Uninitialized) -> FunctionsUpdated
- Function: `admin_setFunctions(new_implementation, function_signatures)` - Attack Vectors: - Unauthorized redirection of function calls if the `authenticate` mechanism is compromised. - Pointing to malicious implementations.

7. Any State (except Uninitialized) -> NewFacetAdded
- Function: `admin_addFacet(new_facet)` - Attack Vectors: - Unauthorized addition of a facet if the `authenticate` mechanism is compromised. - Initializing a malicious facet.

Disclaimer

This audit report aims to identify vulnerabilities and potential issues within the reviewed smart contracts. Although the review process follows a meticulous methodology, it should not be viewed as an exhaustive audit and should not be construed as financial or investment advice.

Limitation of Scope

The audit is limited to the scope defined in the contract, focusing on but not limited to potential security vulnerabilities, logic issues, and best practices. The auditors do not undertake any responsibility to update the audit to account for events or changes occurring after the date of the audit.

No Liability

No party involved in this audit accepts any liability whatsoever for any direct or consequential loss arising from the use of this report or its contents.

Multiple Reviews Recommended

Given the evolving and complex nature of blockchain technology, and the potential for unforeseen vulnerabilities, it is strongly advised to seek multiple independent audits and security reviews.

No Warranty

This report is provided “as is” and all representations, warranties, whether expressed or implied, are disclaimed.