SALUS SECURITY

DEC 2023

CODE SECURITY ASSESSMENT

DVOL FINANCE

Overview

Project Summary

- Name: dVOL Finance - devault contracts
- Platform: BNB Smart Chain
- Address Set:
  - Vault:
    - Proxy: 0x6652f1B0531C4C75B523e74BCf5D0CD009b7BBB8
    - Implementation: 0x2e389c2F28BfAEcA7a358F1415bd00116c77b40d
  - LPTokenFactory: 0x5c28aaA0486499bc722dd44113844cd80Ed36E19
- Repository: https://github.com/dvol-finance/devault-contracts
- Language: Solidity
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	dVOL Finance - devault contracts
Version	v3
Type	Solidity
Date	Dec 04 2023
Logs	Nov 02 2023; Nov 20 2023; Dec 04 2023

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	3
Total Low-Severity issues	6
Total informational issues	6
Total	15

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction **4**

1.1 About SALUS 4

1.2 Audit Breakdown 4

1.3 Disclaimer 4

Findings **5**

2.1 Summary of Findings 5

2.2 Notable Findings 6

1. Lack of return value check on transfer and transferFrom 6
2. Insufficient data validation 8
3. Centralization risk 9
4. Should not leave the implementation contract uninitialized 11
5. Unsafe external call in reinvest() 12
6. Insufficient info in events 13
7. Should use call instead of transfer when transferring native token 14
8. Potential DoS in reinvest() function 16
9. Incompatibility with deflationary/fee-on-transfer tokens 17

2.3 Informational Findings 18

10. Use of magic value 18
11. Inconsistency between code implementation and error message 19
12. Missing zero address check 20
13. Floating compiler version 21
14. Redundant code 22
15. Gas optimization 23

Appendix **24**

Appendix 1 - Files in Scope 24

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Lack of return value check on transfer and transferFrom	Medium	Data Validation	Resolved
2	Insufficient data validation	Medium	Data Validation	Resolved
3	Centralization risk	Medium	Centralization	Mitigated
4	Should not leave the implementation contract uninitialized	Low	Access Control	Acknowledged
5	Unsafe external call in reinvest()	Low	Data Validation	Acknowledged
6	Insufficient info in events	Low	Logging	Acknowledged
7	Should use call instead of transfer when transferring native token	Low	Code Quality	Acknowledged
8	Potential DoS in reinvest() function	Low	Denial of Service	Acknowledged
9	Incompatibility with deflationary	Low	Business Logic	Acknowledged
10	Use of magic value	Informational	Data Validation	Acknowledged
11	Inconsistency between code implementation and error message	Informational	Inconsistency	Acknowledged
12	Missing zero address check	Informational	Data Validation	Acknowledged
13	Floating compiler version	Informational	Configuration	Acknowledged
14	Redundant code	Informational	Redundancy	Acknowledged
15	Gas optimization	Informational	Gas Optimization	Acknowledged

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Lack of return value check on transfer and transferFrom

Severity: Medium	Category: Data Validation

Target:
- contracts/Vault.sol
- contracts/LPToken.sol

Description

Some tokens, like BAT, HT, and cUSDC, do not revert when transfers fail. Instead, they return false.
However, in the following places, the return value of `transfer()` and `transferFrom()` is not checked.

contracts/Vault.sol:L192

```
function depositFor(uint256 vaultId, uint256 amount, address account)
requireExists(vaultId) requireOnSale(vaultId) public payable {
        ...
        } else {
            IERC20Metadata(vaultInfo.depositToken).transferFrom(msg.sender,
address(this), amount);
        }
        ...
}
```

contracts/Vault.sol:L232

```
function withdraw(uint256 vaultId) requireExists(vaultId) requireFailed(vaultId)
external {
        ...
        } else {
            IERC20Metadata(depositToken).transfer(msg.sender, lpTokenBalance);
        }

        ...
}
```

contracts/Vault.sol:L305-306

```
function transfer(uint256 vaultId, address payable to, uint256 fee)
requireExists(vaultId) external {
        ...
        IERC20Metadata(vaultInfo.depositToken).transfer(
payable(vaultInfo.organization), fee);
        IERC20Metadata(vaultInfo.depositToken).transfer(to, investAmount);
            }
        }
        ...
```

```
}
```

contracts/LPToken.sol:L34

```
function transferTo(address payable account, uint256 amount, address token) onlyOwner
external  {
        ...
        } else {
            ...
            IERC20(token).transfer(account, amount);
        }
}
```

Recommendation

Consider using the safeTransfer() and safeTransferFrom() functions from OpenZeppelin's SafeERC20 library to replace transfer() and transferFrom() for token transfer.

This ensures that the above function can only continue execution after the transfer is successful. If the transfer fails, the transaction should be reverted.

Status

The team has resolved this issue with commit 3223b08 and commit cb59171.

2. Insufficient data validation

Severity: Medium	Category: Data Validation
Target: - contracts/Vault.sol	

Description

After the creation of a new vault, the claimTokens array is used in subsequent claim and reinvest operations. However, the `createVault()` function does not verify if the claimTokens's length is non-zero, nor does it ensure that `claimTokens[0]` and `depositToken` are identical. The absence of these checks can potentially disrupt the subsequent execution of the claim and reinvest processes.

Additionally, the function does not verify the uniqueness of the elements in the claimTokens array. Consequently, if the claimTokens array has duplicate tokens, it could lead to inconsistencies during the `settle()` function's execution. Specifically, a single transfer of claimToken to lpTokenContract will double-record the amount. As a result, during the user claim process, it is possible to receive twice the intended amount of claimToken funds.

Recommendation

Consider adding checks to ensure the following:

1. The length of the claimTokens array is not zero
2. The first item from the claimTokens array matches depositToken
3. There are no duplicate elements in claimTokens array

Status

The team has resolved this issue with commit cb59171.

3. Centralization risk

Severity: Medium	Category: Centralization
Target: - contracts/Vault.sol	

Description

1. Centralization risk regarding the transferSigner role

When the 'soldAmount' reaches the 'maxVaultCapacity' or when the time exceeds the 'saleEndTime' and the 'soldAmount' is not less than the 'minVaultLimit', the transferSigner, which is set by the manager in createVault(), can use the transfer() function to transfer 'fee' amount to the 'organization' address and transfer users' remaining deposits to the 'to' address.

The 'to' address and 'fee' amount are determined by the transferSigner. Therefore, if the transferSigner's private key is compromised by an attacker, the attacker can use the transfer() function to transfer users' deposits to his/her own account.

contracts/Vault.sol:L280-311

```
function transfer(uint256 vaultId, address payable to, uint256 fee)
requireExists(vaultId) external {
        ...
        require(vaultInfo.transferSigner == msg.sender, "not signer");

        ...

        if (totalSupply > 0) {
            uint256 investAmount = totalSupply - fee;

            if (vaultInfo.depositToken == address(0)) {
                payable(vaultInfo.organization).transfer(fee);
                to.transfer(investAmount);
            } else {
                IERC20Metadata(vaultInfo.depositToken).transfer(
payable(vaultInfo.organization), fee);
                IERC20Metadata(vaultInfo.depositToken).transfer(to, investAmount);
            }
        }

        emit Transfer(vaultId, to);
}
```

2. Centralization risk regarding the manager role

After the deposited funds and yields are returned to the lpTokenContract, the manager is responsible for calling the settle() function to settle the investment. Only then can users claim their investment and yields.

However, it should be noted that if the settle() function is called before the funds are transferred to the lpTokenContract, the 'claimTokenAmounts' will all be set to zero and can not be changed. This means that users will not be able to claim their funds in this situation.

If the manager's private key is compromised by attackers, they can prematurely call settle() to prevent users from claiming their funds.

contracts/Vault.sol:L313-345

```
function settle(uint256 vaultId) requireManager requireExists(vaultId) external {
        ...

        require(vaultState.hasSettled == false, "has settled");
        require(block.timestamp >= vaultInfo.termStartTime && vaultState.soldAmount >=
vaultInfo.minVaultLimit, "can not settle");

        vaultState.hasSettled = true;

        bool allZero = true;
        for (uint16 i = 0; i < vaultInfo.claimTokens.length; i++) {
            ...
        }

        ...
}
```

Recommendation

We recommend transferring privileged accounts to multi-sig accounts with timelock governors for enhanced security. This ensures that no single person has full control over the accounts and that any changes must be authorized by multiple parties.

Status

The team has mitigated the issue by configuring the transferSigners as multi-sig accounts.

4. Should not leave the implementation contract uninitialized

Severity: Low	Category: Access Control
Target: - contracts/Vault.sol	

Description

According to OpenZeppelin, the implementation contract should not be left uninitialized. It's recommended that you invoke the _disableInitializers function in the constructor to automatically lock it, so that the implementation contract can not be initialized by malicious users.

Recommendation

Consider invoke the _disableInitializers function in the constructor:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Status

This issue has been acknowledged by the team.

5. Unsafe external call in reinvest()

Severity: Low	Category: Data Validation
Target: - contracts/Vault.sol	

Description

The ‘to’ address in reinvest() is a user-provided parameter, and there is no guarantee that it is a valid vault address. This opens an attack surface to the attackers, as they can input a malicious contract as the 'to' address and make reinvest() do external calls to the malicious 'to' address, potentially leading to unexpected outcomes for the system.

contracts/Vault.sol:L347-427

```
function reinvest(uint256 vaultId, uint256 investVaultId, address to)
requireExists(vaultId) external {
        ...
        } else {
            if (status == STATUS_FAILED) {
                if (vaultInfo.depositToken == address(0)) {
                    IVault(to).depositFor{value: lpTokenBalance}(investVaultId,
lpTokenBalance, msg.sender);
                } else {
                    IERC20Metadata(idVaultInfoMap[vaultId].depositToken).approve(to,
lpTokenBalance);
                    IVault(to).depositFor(investVaultId, lpTokenBalance, msg.sender);
                }
            } else if (status == STATUS_ENDED) {
        ...
}
```

Recommendation

It is recommended not to allow users to reinvest in another vault address. This will eliminate the need for the 'to' parameter in the reinvest() function.

Status

This issue has been acknowledged by the team.

6. Insufficient info in events

Severity: Low	Category: Logging
Target: - contracts/Vault.sol	

Description

Triggering events can facilitate off-chain tracking of important variables or configuration changes, but the following events, in our opinion, do not provide enough information.

contracts/Vault.sol:L138

```
function updateVault(UpdateVaultReq calldata req) requireManager requireExists(req.id)
external {
        ...
        emit Update(req.id);
}
```

The Update() event should include parameters indicating which part of the vault has been updated and the corresponding updated value.

contracts/Vault.sol:L310

```
function transfer(uint256 vaultId, address payable to, uint256 fee)
requireExists(vaultId) external {
        ...

        emit Transfer(vaultId, to);
}
```

The Transfer() event should include parameters indicating the transferred token and the amount transferred to the ‘to’ address, as well as the fee transferred to ‘organization’ address.

Recommendation

Consider including additional parameters for events. This allows off-chain event tracking to gather more information about on-chain state changes.

Status

This issue has been acknowledged by the team.

7. Should use call instead of transfer when transferring native token

Severity: Low	Category: Code Quality

Target:
- contracts/Vault.sol
- contracts/LPToken.sol

Description

The transfer function is not recommended for sending ETH due to its 2300 gas unit limit. Instead, `address.call{value:...}("")` can be used to circumvent the gas limit.

contracts/Vault.sol:L230

```
function withdraw(uint256 vaultId) requireExists(vaultId) requireFailed(vaultId)
external {
        ...
        if (depositToken == address(0)) {
            payable(msg.sender).transfer(lpTokenBalance);
        }
        ...
}
```

contracts/Vault.sol:L302-303

```
function transfer(uint256 vaultId, address payable to, uint256 fee)
requireExists(vaultId) external {
        ...
        if (totalSupply > 0) {
            uint256 investAmount = totalSupply - fee;

            if (vaultInfo.depositToken == address(0)) {
                payable(vaultInfo.organization).transfer(fee);
                to.transfer(investAmount);
            }
            ...
}
```

contracts/LPToken.sol:L29

```
function transferTo(address payable account, uint256 amount, address token) onlyOwner
external  {
        if (token == address(0)) {
            require(address(this).balance >= amount, "balance not enough");
            account.transfer(amount);
        }
        ...
}
```

Recommendation

Consider using the ‘call’ method instead of the ‘transfer’ method.

Status

This issue has been acknowledged by the team.

8. Potential DoS in reinvest() function

Severity: Low	Category: Denial of Service
Target: - contracts/Vault.sol	

Description

In the `reinvest()` function, if the original vault successfully executes to ENDED, the user can transfer his deposit from one vault to another vault. This function will determine that `vaultInfo.claimTokenAmounts[i]>0`, but if a malicious user transfers claimToken to lpTokenContract in advance before `settle()` executing, as a result, this variable is not 0, and the function will revert, which will also prevent other users from using `reinvest()` function to reinvest.

contracts/Vault.sol:L374, L404

```
function reinvest(uint256 vaultId, uint256 investVaultId, address to)
requireExists(vaultId) external {
        ...
        if (address(this) == to) {
            ...
                for (uint16 i = 1; i < vaultInfo.claimTokens.length; i++) {
                    if (vaultInfo.claimTokenAmounts[i] > 0) {
                        revert("invalid deposit token");
                    }
                }
            ...
        } else {
            ...
                for (uint16 i = 1; i < vaultInfo.claimTokens.length; i++) {
                    if (vaultInfo.claimTokenAmounts[i] > 0) {
                        revert("invalid deposit token");
                    }
                }
            ...
}
```

Recommendation

Consider adding permission control to the change of `claimTokenAmounts`, or executing it directly in the vault contract to make the operation public.

Status

This issue has been acknowledged by the team.

9. Incompatibility with deflationary/fee-on-transfer tokens

Severity: Low	Category: Business Logic
Target: - contracts/Vault.sol	

Description

The Vault contract does not support deflationary/fee-on-transfer tokens. When transferring these tokens, the received amount may be lower than the specified transfer amount.

If the depositToken is a deflationary/fee-on-transfer token, the contract will mint more lpToken than it should to the user during deposit.

If the claimToken is a deflationary/fee-on-transfer token, the user may receive fewer than expected when withdrawing or claiming.

Recommendation

Consider transferring the tokens first and compare pre-/after token balances to compute the actual transferred amount.

Status

This issue has been acknowledged by the team.

2.3 Informational Findings

10. Use of magic value

Severity: Informational	Category: Data Validation
Target: - contracts/Vault.sol	

Description

In the given bytes32 variables, the developers manually re-enter the string for encryption rather than utilizing the previously declared string variable. If typo errors occur during the coding process, it will cause an inconsistency between the calculated status and the predetermined bytes32 status.

contracts/Vault.sol:L59-71

```
    string constant private NOT_STARTED = "NOT_STARTED";
    string constant private ON_SALE = "ON_SALE";
    string constant private FAILED = "FAILED";
    string constant private SALE_CLOSED = "SALE_CLOSED";
    string constant private RUNNING = "RUNNING";
    string constant private ENDED = "ENDED";

    bytes32 constant private STATUS_NOT_STARTED =
keccak256(abi.encodePacked("NOT_STARTED"));
    bytes32 constant private STATUS_ON_SALE = keccak256(abi.encodePacked("ON_SALE"));
    bytes32 constant private STATUS_FAILED = keccak256(abi.encodePacked("FAILED"));
    bytes32 constant private STATUS_SALE_CLOSED =
keccak256(abi.encodePacked("SALE_CLOSED"));
    bytes32 constant private STATUS_RUNNING = keccak256(abi.encodePacked("RUNNING"));
    bytes32 constant private STATUS_ENDED = keccak256(abi.encodePacked("ENDED"));
```

Recommendation

Consider using the declared string variables to calculate the vault statuses of bytes32 type.

Take “NOT_STARTED” for example:

Can change

```
bytes32 constant private STATUS_NOT_STARTED = keccak256(abi.encodePacked("NOT_STARTED"));
```

to

```
bytes32 constant private STATUS_NOT_STARTED = keccak256(abi.encodePacked(NOT_STARTED));
```

Status

This issue has been acknowledged by the team.

11. Inconsistency between code implementation and error message

Severity: Informational	Category: Inconsistency

Target:
- contracts/Vault.sol

Description

1. The following revert situations in reinvest() are unrelated to the deposit token, so the error message “invalid deposit token” is not appropriate.

contracts/Vault.sol:L373-377, L403-407

```
function reinvest(uint256 vaultId, uint256 investVaultId, address to)
requireExists(vaultId) external {
        ...
                for (uint16 i = 1; i < vaultInfo.claimTokens.length; i++) {
                    if (vaultInfo.claimTokenAmounts[i] > 0) {
                        revert("invalid deposit token");
                    }
                }
        ...
                for (uint16 i = 1; i < vaultInfo.claimTokens.length; i++) {
                    if (vaultInfo.claimTokenAmounts[i] > 0) {
                        revert("invalid deposit token");
                    }
                }
        ...
}
```

2. The error message “has settled” in reinvest() should be “not settled” or “require settled”.

contracts/Vault.sol:L378-380

```
function reinvest(uint256 vaultId, uint256 investVaultId, address to)
requireExists(vaultId) external {
        ...
          if (vaultState.hasSettled != true) {
              revert("has settled");
          }
       ...
}
```

Recommendation

Consider modifying the error message to make it consistent with the code implementation.

Status

This issue has been acknowledged by the team.

12. Missing zero address check

Severity: Informational	Category: Data Validation

Target:
- contracts/Vault.sol

Description

It is considered a security best practice to verify addresses against the zero address in the constructor or setting. However, this precautionary step is absent for the variables highlighted below.

contracts/Vault.sol:L79-82

```
function initialize(address _lpTokenFactory, address _manager) public initializer {
        lpTokenFactory = _lpTokenFactory;
        manager = _manager;
}
```

contracts/Vault.sol:L102

```
function createVault(CreateVaultReq memory req) requireManager requireNotExists(req.id)
external {
        ...
        vaultInfo.organization = req.organization;
        ...
}
```

contracts/Vault.sol:L198-212

```
function depositInternal(uint256 vaultId, uint256 amount, address account)
requireExists(vaultId) requireOnSale(vaultId) internal {
        ...
}
```

contracts/Vault.sol:L280-311

```
function transfer(uint256 vaultId, address payable to, uint256 fee)
requireExists(vaultId) external {
        ...
}
```

Recommendation

Consider adding zero-address checks.

Status

This issue has been acknowledged by the team.

13. Floating compiler version

Severity: Informational	Category: Configuration
Target: - all	

Description

```
pragma solidity ^0.8.9;
```

The codebase uses a floating Solidity compiler version, ^0.8.9.

However, we discourage this practice. It's best to deploy contracts with the same compiler version and flags that they have been thoroughly tested with. Locking the compiler version helps to prevent contracts from being accidentally deployed using an outdated compiler version, which could introduce bugs that have a negative impact on the system.

Recommendation

It is recommended to use a locked Solidity compiler version.

For example:

```
pragma solidity 0.8.9;
```

Status

This issue has been acknowledged by the team.

14. Redundant code

Severity: Informational	Category: Redundancy
Target: - contracts/Vault.sol	

Description

It is unnecessary to compare boolean variables with true or false.

contracts/Vault.sol:L240, L262, L317, L378, L408

```
function reinvest(uint256 vaultId, uint256 investVaultId, address to)
requireExists(vaultId) external {
        ...
          if (vaultState.hasSettled != true) {
              revert("has settled");
          }

       ...
}
```

Recommendation

It is recommended to use boolean variables themselves in condition checks.

For example:

`if (vaultState.hasSettled != true)` can be changed to `if (!vaultState.hasSettled)` , and `if (vaultState.hasSettled == true)` can be changed to `if (vaultState.hasSettled)`

Status

This issue has been acknowledged by the team.

15. Gas optimization

Severity: Informational	Category: Gas Optimization
Target: - contracts/Vault.sol	

Description

When the storage variable is only read and not modified in a function, the variable can be temporarily stored in memory instead of storage to save gas.

contracts/Vault.sol:L183-184, L199-200, L219, L239, L261, L348-349

```
VaultInfo storage vaultInfo = idVaultInfoMap[vaultId];
VaultState storage vaultState = idVaultStateMap[vaultId];
```

Recommendation

Consider loading vaultInfo and vaultState to memory to save gas.

Status

This issue has been acknowledged by the team.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit ff89428:

File	SHA-1 hash
contracts/ILPToken.sol	eab958774f194000cb851058d3628bdc03bb478b
contracts/ILPTokenFactory.sol	b8d3ab20efb1c8a5faccd87468446f387f4c06ec
contracts/IVault.sol	c30b8b288cff94fc7032a560b9e0d4a7a19649f8
contracts/LPToken.sol	b5fc73a104be385dbc00aebe24479245fdcf8b46
contracts/LPTokenFactory.sol	3434ccd6adead1653a77af37b126861c727abd26
contracts/Vault.sol	166a6ee003d8788174ca2d5ce1762e1e2b83479c