SALUS SECURITY

DEC 2023

CODE SECURITY ASSESSMENT

BITSTABLEPOOL

Overview

Project Summary

- Name: BitStablePool
- Platform: EVM-compatible chains
- Language: Solidity
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	BitStablePool
Version	v3
Type	Solidity
Dates	Dec 08 2023
Logs	Dec 01 2023; Dec 04 2023; Dec 08 2023

Vulnerability Summary

Total High-Severity issues	3
Total Medium-Severity issues	3
Total Low-Severity issues	1
Total informational issues	4
Total	11

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

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	Arbitrary calls execution in the build() function	High	Business Logic	Resolved
2	Incorrect conversion from amountOut to amountIn	High	Numerics	Resolved
3	Loss of precision due to tokens with different decimals	High	Numerics	Resolved
4	The build() function may lock ethers in the contract	Medium	Business Logic	Resolved
5	Incorrect DAII balance validation	Medium	Business Logic	Resolved
6	Centralization risk	Medium	Centralization	Acknowledged
7	Implementation contract could be initialized by everyone	Low	Business Logic	Resolved
8	Use of floating pragma	Informational	Configuration	Acknowledged
9	Lack of duplication check	Informational	Data Validation	Resolved
10	Use of the ERC20 transfer()	Informational	Business Logic	Resolved
11	Gas optimization suggestions	Informational	Gas Optimization	Resolved

2.2 Notable Findings

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

1. Arbitrary calls execution in the build() function

Severity: High	Category: Business Logic
Target: - BitStablePool.sol	

Description

In the build() function, parameters recipient and calldata are completely controlled by the caller. An attacker can set the recipient to a token address and transfer tokens from the contract.

BitStablePool.sol:L72-L96

```
function build(
    address tokenIn,
    uint256 amountIn,
    address recipient,
    bytes memory data
) external payable lock whenNotPaused {
    ...
    if (data.length > 0) {
        DAII.safeIncreaseAllowance(recipient, amountOut);
        (bool success, ) = recipient.call{value: msg.value}(data);
        require(success, "Unable to send funds");
    }
    ...
}
```

Recommendation

It is recommended to remove the callback logic and use safeTransfer() to directly transfer DAII tokens. If callbacks are required, consider calling a specific function rather than defined by the caller, e.g. IBitStablePoolCallee(to).bitStablePoolCall().

Status

This issue has been resolved by the team.

2. Incorrect conversion from amountOut to amountIn

Severity: High	Category: Numerics
Target: - BitStablePool.sol	

Descriptions

In the redeem() function, users can exchange DAII with specific tokens. The amountOut is the amount of tokenOut that the user wants to redeem from the contract. The amount of DAII tokens that the user requires to transfer into the contract is calculated from amountOut based on the decimal of two tokens. To convert to a DAII denominated value, the amountOut needs to be divided by tokenOut's decimals and multiplied by DAII's decimals.

BitStablePool.sol:L97-L112

```
function redeem(
        address tokenOut,
        uint256 amountOut
) external lock whenNotPaused {
        ...
        uint256 amountIn = amountOut * (10 ** IERC20Metadata(tokenOut).decimals()) / (10
** IERC20Metadata(address(DAII)).decimals());
        ...
        DAII.safeTransferFrom(msg.sender, address(this), amountIn);
        ...
}
```

Recommendation

Consider changing `(10 ** IERC20Metadata(tokenOut).decimals()) / (10 ** IERC20Metadata(address(DAII)).decimals())` to `(10 ** IERC20Metadata(address(DAII)).decimals()) / (10 ** IERC20Metadata(tokenOut).decimals())`.

Status

This issue has been resolved by the team.

3. Loss of precision due to tokens with different decimals

Severity: High	Category: Numerics
Target: - BitStablePool.sol	

Descriptions

In both build() and redeem() function, the token amount will be converted to a DAII denominated value. However, the decimals of some tokens may be different from DAII tokens. For example, USDC has 6 decimals and YAM V2 has 24 decimals. In the build() function, if the tokenIn's decimals are larger than DAII's, users may have spent tokens without receiving DAII in return. Similarly, if the tokenOut's decimals are larger than DAII's in the redeem() function, users may redeem tokens for free.

BitStablePool.sol:L72-L96

```
function build(
        address tokenIn,
        uint256 amountIn,
        ...
) external payable lock whenNotPaused {
        ...
        IERC20(tokenIn).safeTransferFrom(msg.sender, address(this), amountIn);
        uint256 amountOut = amountIn * (10 ** IERC20Metadata(address(DAII)).decimals()) /
(10 ** IERC20Metadata(tokenIn).decimals());
...
}
```

Recommendation

Consider reverting the transaction if:

- The amountOut is 0 in the build() function;
- The amountIn is 0 in the redeem() function.

Status

This issue has been resolved by the team.

4. The build() function may lock ethers in the contract

Severity: Medium	Category: Business Logic
Target: - BitStablePool.sol	

Description

The build() function uses the payable modifier to receive ethers, and ethers will be directly transferred to the recipient when the data is not empty. However, if the data is empty, ethers will be locked in the contract and can only be withdrawn by the owner.

BitStablePool.sol:L72-L96

```
function build(
        ...
) external payable lock whenNotPaused {
        ...
        if (data.length > 0) {
            DAII.safeIncreaseAllowance(recipient, amountOut);
            (bool success, ) = recipient.call{value: msg.value}(data);
            require(success, "Unable to send funds");
        } else {
            DAII.safeTransfer(msg.sender, amountOut);
        }
        ...
}
```

Recommendation

Consider removing the payable modifier of the build() function.

Status

This issue has been resolved by the team.

5. Incorrect DAII balance validation

Severity: Medium	Category: Business Logic
Target: - BitStablePool.sol	

Descriptions

In the build() function, users can exchange DAII tokens with other tokens. The comparison logic should check if the contract's DAII balance is greater than or equal to amountOut, but it mistakenly compares it to amounIn. This could lead to unexpected behavior in the contract.

BitStablePool.sol:L81

```
require(DAII.balanceOf(address(this)) >= amountIn, "Insufficient DAII balance");
```

Recommendation

Consider changing amountIn to amountOut.

Status

This issue has been resolved by the team.

6. Centralization risk

Severity: Medium	Category: Centralization
Target: - BitStablePool.sol	

Description

There is a privileged owner role in the BitStablePool contract. The owner of the BitStablePool contract can manage all the configurations and withdraw tokens from the contract.

BitStablePool.sol:L181-L196

```
function withdrawToken(
    address[] calldata tokens,
    address to
) external onlyOwner {
    uint256 balance = address(this).balance;
    if (balance > 0) {
        (bool success, ) = to.call{value: balance}(new bytes(0));
        require(success, "Unable to send funds");
    }
    for (uint256 index = 0; index < tokens.length; ++index) {
        balance = IERC20(tokens[index]).balanceOf(address(this));
        if (balance > 0) {
                IERC20(tokens[index]).transfer(to, balance);
        }
    }
}
```

Should the owner's private key be compromised, an attacker could withdraw all the funds in the contract. If the privileged account is a plain EOA account, this can be worrisome and pose a risk to other users.

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

This issue has been acknowledged by the team.

7. Implementation contract could be initialized by everyone

Severity: Low	Category: Business Logic
Target: - BitStablePool.sol	

Description

According to OpenZeppelin, the implementation contract should not be left uninitialized.

An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy. There is nothing preventing the attacker from calling the initialize() function in BitStablePool's implementation contract.

Recommendation

To prevent the implementation contract from being used, consider invoking the _disableInitializers() function in the constructor of the BitStablePool contract to automatically lock it when it is deployed.

Status

This issue has been resolved by the team.

2.3 Informational Findings

8. Use of floating pragma

Severity: Informational	Category: Configuration
Target: - BitStablePool.sol	

Description

```
pragma solidity ^0.8.20;
```

The BitStablePool uses a floating compiler version ^0.8.20.

Using a floating pragma ^0.8.20 statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

Status

This issue has been acknowledged by the team.

9. Lack of duplication check

Severity: Informational	Category: Data Validation
Target: - BitStablePool.sol	

Description

During the initialization, the function does not perform a duplication check on the input tokens. However, when setting the token and its validity in the setExchange() function, it loops through the entire exchanges array to determine if any elements are duplicated with the input token. This inconsistency exists between the two implementations.

Recommendation

Consider adding a duplication check when adding tokens to the exchanges array during initialization.

Status

This issue has been resolved by the team.

10. Use of the ERC20 transfer()

Severity: Informational	Category: Business Logic
Target: - BitStablePool.sol	

Description

Tokens not compliant with the ERC20 specification could return false from the transfer function call to indicate the transfer fails, while the calling contract would not notice the failure if the return value is not checked.

Since SafeERC20 has been imported and the methods safeTransferFrom() and safeTransfer() have been utilized, the token transfer in the withdrawToken() function should also use safeTransfer().

BitStablePool.sol:L193

```
IERC20(tokens[index]).transfer(to, balance);
```

Recommendation

Consider replacing transfer() with safeTransfer().

Status

This issue has been resolved by the team.

11. Gas optimization suggestions

Severity: Informational	Category: Gas Optimization
Target: - BitStablePool.sol	

Description

BitStablePool.sol:L54

```
function initialize(
        address initialOwner,
        address daii,
        address[] memory tokens
) public virtual initializer
```

BitStablePool.sol:L76

```
function build(
        address tokenIn,
        uint256 amountIn,
        address recipient,
        bytes memory data
) external payable lock whenNotPaused
```

The data location of parameters highlighted above could be calldata.

BitStablePool.sol:L79-80

```
function build(
        address tokenIn,
        uint256 amountIn,
        address recipient,
        bytes memory data
    ) external payable lock whenNotPaused {
        ...
        require(IERC20(tokenIn).balanceOf(msg.sender) >= amountIn && amountIn > 0,
"Insufficient balance");
        require(IERC20(tokenIn).allowance(msg.sender, address(this)) >= amountIn,
"Insufficient allowance");
        ...
}
```

BitStablePool.sol:L106-107

```
function redeem(
        address tokenOut,
        uint256 amountOut
    ) external lock whenNotPaused {
        ...
        require(IERC20(tokenOut).balanceOf(address(this)) >= amountOut && amountOut > 0,
"Insufficient balance");
        ...
        require(DAII.balanceOf(msg.sender) >= amountIn, "Insufficient DAII balance");
        require(DAII.allowance(msg.sender, address(this)) >= amountIn, "Insufficient
DAII allowance");
        ...
}
```

During a token transfer, the balance of the sender and allowance will be checked in the transferFrom() or transfer() function inside the token contract, taking the ERC20 token as an example. Thus, these highlighted checks can be safely removed.

BitStablePool.sol:L117

```
function setExchange(address token, bool value) external onlyOwner {
        ...
        bool exist = false;
        ...
}
```

BitStablePool.sol:L158

```
function getExchanges() public view returns (address[] memory) {
        ...
        uint256 index = 0;
        ...
}
```

The uint256 and bool variables are initialized to by default 0 and false respectively. The assignments can be removed to save gas.

BitStablePool.sol:L118-L123

```
function setExchange(address token, bool value) external onlyOwner {
       ...
       for (uint256 i = 0; i < exchanges.length; i++) {
              if (exchanges[i] == token) {
                            exist = true;
                            break;
              }
       }
       ...
}
```

BitStablePool.sol:L159-L164

```
function getExchanges() public view returns (address[] memory) {
    ...
    for (uint256 i = 0; i < exchanges.length; i++) {
        if (validity[exchanges[i]]) {
            tokens[index] = exchanges[i];
            index++;
        }
    }
    ...
}
```

In the setExchange() and getExchanges() functions, the length of the exchanges array will be read during each iteration. Since the exchanges array is a storage array, this is an extra sload operation. It is recommended to use a memory variable to cache the length to save gas.

Recommendation

Consider applying the gas optimization suggestions where needed.

Status

This issue has been resolved by the team.

Appendix

Appendix 1 - Files in Scope

This audit covered the following file provided by the client:

File	SHA-1 hash
BitStablePool.sol	e02c7efbf67d03595259d6cbbd823e2408fb4bc6

The file and hash after revision are as follows:

File	SHA-1 hash
BitStablePool.sol	65e270013c307c7a48435f571082f7b251fd395a