Security Audit Report for Stable Token Contracts

Date: April 30, 2025 **Version:** 1.0
Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**

1.1 About Target Contracts 1
1.2 Disclaimer 1
1.3 Procedure of Auditing 2
1.3.1 Software Security 2
1.3.2 DeFi Security 2
1.3.3 NFT Security 2
1.3.4 Additional Recommendation 3
1.4 Security Model 3

Chapter 2 Findings **4**

2.1 DeFi Security 4
2.1.1 Potential DoS when the variable `required` equals to the number of owners 4
2.1.2 Potential DoS due to the unbounded array `transactions` 5
2.1.3 Improper handling of the variable `transaction.executed` 6
2.2 Recommendation 7
2.2.1 Lack of zero address checks 7
2.2.2 Add blacklist restrictions to approval operations 8
2.2.3 Add state change checks in the `blacklist()`, `unBlacklist()`, `removeMinter()`, and `configureMinter()` functions 9
2.2.4 Unify signature handling across the protocol for consistency 10
2.3 Note 11
2.3.1 Potential centralization risks 11
2.3.2 Potential front-running risks 11
2.3.3 Correct value assignments for the variable `required` 12

Report Manifest

Item	Description
Client	Solidus
Target	Stable Token Contracts

Version History

Version	Date	Description
1.0	April 30, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-04-30 05:46:55+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of Stable Token Contracts of Solidus. The Stable Token Contracts of the Solidus implements an ERC20-compliant stablecoin based on the EIP-3009 and EIP-2612 standards. Additionally, the stablecoin incorporates security features such as role-based access control. Note this audit only focuses on the smart contracts in the following directories/files:

- src/*

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Stable Token Contracts	Version 1	b7ce5dcb052058b8b760ba263b383c993c1bc691
	Version 2	79ca0e9115adce46ffa253ce2f13bbb310bdaf7a
	Version 3	0c53348b47e442acf4cb3c1c86657a723af6f852

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying

[1] https://github.com/solidusfin/stablecoin-evm

compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver

* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **four** recommendations and **three** notes.

- Medium Risk: 1
- Low Risk: 2
- Recommendation: 4
- Note: 3

ID	Severity	Description	Category	Status
1	Medium	Potential DoS when the variable `required` equals to the number of owners	DeFi Security	Fixed
2	Low	Potential DoS due to the unbounded array `transactions`	DeFi Security	Fixed
3	Low	Improper handling of the variable `transaction.executed`	DeFi Security	Fixed
4	-	Lack of zero address checks	Recommendation	Fixed
5	-	Add blacklist restrictions to approval operations	Recommendation	Fixed
6	-	Add state change checks in the `blacklist()`, `unBlacklist()`, `removeMinter()`, and `configureMinter()` functions	Recommendation	Confirmed
7	-	Unify signature handling across the protocol for consistency	Recommendation	Fixed
8	-	Potential centralization risks	Note	-
9	-	Potential front-running risks	Note	-
10	-	Correct value assignments for the variable `required`	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential DoS when the variable `required` equals to the number of owners

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the contract `MultiSigWallet`, the owners can submit transactions and the pending transaction cannot be executed until the number of confirmed owners equals to the variable `required`. According to the design, the variable `required` can be equal to `owners.length`, which means all the owners need to confirm the transaction before execution. In this case, a malicious or compromised owner can deny to confirm any pending transactions, leading to a DoS issue.

```
function removeOwner(address owner) public onlyWallet {
    if (!isOwner[owner]) {
        revert NoExistOwner(owner);
    }

    isOwner[owner] = false;
    for (uint256 i = 0; i < owners.length - 1; i++) {
        if (owners[i] == owner) {
            owners[i] = owners[owners.length - 1];
            break;
        }
    }
    owners.pop();
    if (required > owners.length) changeRequirement(owners.length);
    emit OwnerRemoval(owner);
}
```

Listing 2.1: src/MultiSigWallet.sol

Impact The contract `MultiSigWallet` can suffer a DoS issue.

Suggestion Add a proper check on the variables `required`.

Note The project mitigates the issue by ensuring that the variable `required` is less than `owners.length`. However, the risk still exists when multiple owners are malicious or compromised. It is crucial for the project to implement a rigorous selection process for all owners to mitigate these risks.

2.1.2 Potential DoS due to the unbounded array `transactions`

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the contract `MultiSigWallet`, the array `transactions` records submitted transactions via the function `_addTransaction()` without any cleanup mechanism. This design introduces a potential DoS risk to the functions `getTransactionCount()` and `getTransactionIds()`, as they iterate over the entire `transactions` array.

```
function _addTransaction(
    address destination,
    uint256 value,
    bytes calldata data
) internal returns (uint256 transactionId) {
    transactionId = transactions.length;
    transactions.push(Transaction(destination, value, data, false));
    emit Submission(transactionId, destination, value, data);
}
```

Listing 2.2: src/MultiSigWallet.sol

```
function getTransactionCount(
    bool pending,
    bool executed
) public view returns (uint256 count) {
    for (uint256 i = 0; i < transactions.length; i++)
        if (
            (pending && !transactions[i].executed) ||
            (executed && transactions[i].executed)
        ) count += 1;
}
```

Listing 2.3: src/MultiSigWallet.sol

```
function getTransactionIds(
    uint256 from,
    uint256 to,
    bool pending,
    bool executed
) public view returns (uint256[] memory _transactionIds) {
    uint256[] memory transactionIdsTemp = new uint256[](
        transactions.length
    );
    uint256 count = 0;
    for (uint256 i = 0; i < transactions.length; i++)
        if (
            (pending && !transactions[i].executed) ||
            (executed && transactions[i].executed)
        ) {
            transactionIdsTemp[count] = i;
            count += 1;
        }
    _transactionIds = new uint256[](to - from);
    for (uint256 i = from; i < to; i++)
        _transactionIds[i - from] = transactionIdsTemp[i];
}
```

Listing 2.4: src/MultiSigWallet.sol

Impact Potential DoS due to the unbounded array `transactions`.

Suggestion Revise the code accordingly.

2.1.3 Improper handling of the variable `transaction.executed`

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the contract `MultiSigWallet`, the function `executeTransaction()` executes transactions and records their execution status (i.e., `transaction.executed`) in the array `transactions`. However, if a transaction fails, the execution status remains `false`, which means the transaction can be executed multiple times.

```
function executeTransaction(uint256 transactionId) public {
    if (transactions[transactionId].executed) {
        revert ExecutedTransaction(transactionId);
    }

    if (isConfirmed(transactionId)) {
        Transaction storage transaction = transactions[transactionId];
        transaction.executed = true;
        (bool success, ) = transaction.destination.call{
            value: transaction.value
        }(transaction.data);
        if (success) {
            emit Execution(transactionId);
        } else {
            emit ExecutionFailure(transactionId);
            transaction.executed = false;
        }
    }
}
```

Listing 2.5: src/MultiSigWallet.sol

Impact The improper handling of the variable `transaction.executed` allows the transaction to be executed multiple times, which is not an expected behavior.

Suggestion Revise the code accordingly.

2.2 Recommendation

2.2.1 Lack of zero address checks

Status Fixed in `Version 3`

Introduced by `Version 1 & 2`

Description In the contract `StableTokenV1` and `MultiSigWallet`, inputs of several functions (i.e., `initialize()`, `configureMinter()`, and `submitTransaction()`) are not checked to ensure they are not zero. It is recommended to add such checks to prevent potential mis-operations.

```
function initialize(
    string calldata name,
    string calldata symbol,
    address defaultAdmin,
    address upgrader,
    address pauser,
    address rescuer,
    address blacklister,
    address mainMinter
) public initializer {
    __UUPSUpgradeable_init();
    __Pausable_init();
    __ERC20_init(name, symbol);
    __ERC20Permit_init(name);
```

```
        __AccessControlDefaultAdminRules_init(3 days, defaultAdmin);

        _grantRole(UPGRADER_ROLE, upgrader);
        _grantRole(PAUSER_ROLE, pauser);
        _grantRole(RESCUER_ROLE, rescuer);
        _grantRole(BLACKLISTER_ROLE, blacklister);
        _grantRole(MAIN_MINTER_ROLE, mainMinter);

        _setRoleAdmin(MINTER_ROLE, MAIN_MINTER_ROLE);
    }
```

Listing 2.6: src/StableTokenV1.sol

```
    function configureMinter(
        address minter,
        uint256 minterAllowedAmount
    ) public override whenNotPaused returns (bool) {
        return super.configureMinter(minter, minterAllowedAmount);
    }
```

Listing 2.7: src/StableTokenV1.sol

```
    function submitTransaction(
        address destination,
        uint256 value,
        bytes calldata data
    ) public returns (uint256 transactionId) {
        transactionId = _addTransaction(destination, value, data);
        confirmTransaction(transactionId);
    }
```

Listing 2.8: src/MultiSigWallet.sol

Suggestion Add non-zero address checks accordingly.

Note The project removed certain role assignments (i.e., `upgrader`, `pauser`, `rescuer`, and `blacklister`) from the function `initialize()` and will assign these roles as needed.

2.2.2 Add blacklist restrictions to approval operations

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `StableTokenV1`, blacklisted users are restricted from performing transfer operations but are still allowed to approve their assets. It is recommended to add restrictions on blacklisted users for performing approval operations.

```
    function _approve(
        address owner,
        address spender,
        uint256 value,
        bool emitEvent
    ) internal override whenNotPaused {
```

```
        super._approve(owner, spender, value, emitEvent);
    }
```

Listing 2.9: src/StableTokenV1.sol

Suggestion Restrict blacklisted users from performing approval operations.

2.2.3 Add state change checks in the `blacklist()`, `unBlacklist()`, `removeMinter()`, and `configureMinter()` functions

Status Confirmed

Introduced by `Version 1`

Description In the protocol, the `MAIN_MINTER_ROLE` and `BLACKLISTER_ROLE` roles can manage users' permissions through the functions `blacklist()`, `unBlacklist()`, `configureMinter()`, and `removeMinter()`. It is recommended to implement state change checks on accounts' current statuses to ensure that they are different from the newly updated ones.

```
function blacklist(address account) public onlyRole(BLACKLISTER_ROLE) {
    _isBlacklisted[account] = true;
    emit Blacklisted(account);
}

/**
 * @notice Removes account from blacklist.
 * @param account The address to remove from the blacklist.
 * @dev Only callable by accounts with BLACKLISTER_ROLE.
 */
function unBlacklist(address account) public onlyRole(BLACKLISTER_ROLE) {
    _isBlacklisted[account] = false;
    emit UnBlacklisted(account);
}
```

Listing 2.10: src/libraries/Blacklistable.sol

```
function configureMinter(
    address minter,
    uint256 minterAllowedAmount
) public virtual onlyRole(MAIN_MINTER_ROLE) returns (bool) {
    _grantRole(MINTER_ROLE, minter);
    _minterAllowed[minter] = minterAllowedAmount;
    emit MinterConfigured(minter, minterAllowedAmount);
    return true;
}

/**
 * @notice Removes a minter from the system
 * @dev Can only be called by an account with MAIN_MINTER_ROLE
 * @param minter Address of the minter to remove
 * @return bool True if the operation was successful
 */
function removeMinter(
    address minter
```

```
) public virtual onlyRole(MAIN_MINTER_ROLE) returns (bool) {
    _revokeRole(MINTER_ROLE, minter);
    _minterAllowed[minter] = 0;
    emit MinterRemoved(minter);
    return true;
}
```

Listing 2.11: src/libraries/MintManager.sol

Suggestion Add state change checks on accounts' current status in the functions `blacklist()`, `unBlacklist()`, `configureMinter()`, and `removeMinter()`.

Feedback from the project The project will not add the restrict in the functions `blacklist()` and `unBlacklist()` for a clear code like USDT and USDC. For the function `configureMinter()`, it is used to reconfigure the allowance.

2.2.4 Unify signature handling across the protocol for consistency

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the protocol, the function `permit()` supports both 65-byte and 64-byte (i.e., the compact signature) signatures. However, in the functions `transferWithAuthorization()`, `receiveWithAuthorization()` and `cancelAuthorization()` of the contract `EIP3009`, only 65-byte signatures are supported. It is recommended to unify signature handling across the protocol for consistency.

```
function permit(
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    bytes calldata signature
) public {
    (bytes32 r, bytes32 s, uint8 v) = signature.decodeRSV();

    permit(owner, spender, value, deadline, v, r, s);
}
```

Listing 2.12: src/StableTokenV1.sol

```
function decodeRSV(
    bytes calldata signature
) internal pure returns (bytes32 r, bytes32 s, uint8 v) {
    if (signature.length == 65) {
        // Standard signature format
        (r, s) = abi.decode(signature, (bytes32, bytes32));
        v = uint8(signature[64]);
    } else if (signature.length == 64) {
        // EIP-2098 compact signature format
        bytes32 vs;
        (r, vs) = abi.decode(signature, (bytes32, bytes32));
```

```
            s = vs & UPPER_BIT_MASK;
            v = uint8(uint256(vs >> 255)) + 27;
        } else {
            revert InvalidSignatureLength();
        }
    }
```

Listing 2.13: src/libraries/Utils.sol

```
    function _requireValidSignature(
        address signer,
        bytes32 dataHash,
        bytes memory signature
    ) private view {
        if (
            !SignatureChecker.isValidSignatureNow(
                signer,
                MessageHashUtils.toTypedDataHash(
                    _domainSeparatorV4(),
                    dataHash
                ),
                signature
            )
        ) revert InvalidSignature();
    }
```

Listing 2.14: src/libraries/EIP3009.sol

Suggestion Unify signature handling across the protocol for consistency.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description Several protocol roles (e.g., the roles `DEFAULT_ADMIN_ROLE`, `MAIN_MINTER_ROLE`, and `UPGRADER_ROLE`) could conduct privileged operations, which introduces potential centralization risks. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

Feedback from the project The project mitigated centralization risks by introducing the contract `MultiSigWallet` to hold critical roles such as `DEFAULT_ADMIN_ROLE`, `MAIN_MINTER_ROLE` and `UPGRADER_ROLE`.

2.3.2 Potential front-running risks

Introduced by `Version 1`

Description The protocol implements a stable token contract by integrating the `EIP-3009` and `EIP-2612` standards, which enables approving and transferring assets via signatures. However, these standards expose a front-running risk. Specifically, attackers can front-run the invocations of some functions (i.e., `transferWithAuthorization()`, `receiveWithAuthorization()`, and `permit()`). As a result, if the other protocols do not properly catch and handle errors, the transactions will revert. This may influence the other normal users in the scenarios like batch transfer. The protocol should notify other protocols, who integrate the stable token contract, about the potential front-running risks.

2.3.3 Correct value assignments for the variable `required`

Introduced by `Version 2`

Description In the contract `MultiSigWallet`, the variable `required` is used as a requirement for transaction executions. Specifically, the variable `required` can be set and modified via the functions `constructor()` and `changeRequirement()`, respectively. The project should assign a proper value to the variable `required` to ensure security.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS