// Security Assessment

09.01.2025 - 09.25.2025

DSToken

Securitize

HALBORN

DSToken - Securitize

Prepared by: HALBORN

Last Updated 10/08/2025

Date of Engagement: September 1st, 2025 - September 25th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
13	0	0	2	4	7

TABLE OF CONTENTS

1. Introduction

`Securitize Protocol` engaged Halborn to conduct a security assessment on their smart contracts beginning on September 1st, 2025 and ending on September 25th, 2025. The scope of this assessment was limited to the smart contracts provided to the Halborn team. Commit hashes and additional details are documented in the Scope section of this report.

`Securitize Protocol` is a digital securities framework where tokens are fully backed and governed by compliance rules. The TokenIssuer handles issuance and minting, while the ComplianceService enforces transfer restrictions, lockups, and jurisdictional investor limits using data from the RegistryService. Core actions such as transfer, swap, buy, and redemption automatically route through these compliance checks, ensuring every movement of tokens respects regulations. Governance is managed by the TrustService, assigning roles like Master, Issuer, and Transfer Agent to control permissions and upgrades. Together, these components create a regulated on-chain ecosystem for compliant tokenized securities.

2. Assessment Summary

`Halborn` assigned a full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, `Halborn` identified several areas for improvement to reduce both the likelihood and impact of potential risks, **which were successfully addressed by the `Securitize team`. The main recommendations were:**

- `Correct the totalIssued handling in burn logic to prevent a permanent cap lock state where the protocol becomes stuck and no new participants can mint tokens.`
- `Enforce wallet validation in issueTokensCustom to prevent issuance of tokens to unregistered or non-compliant addresses.`

3. Test Approach And Methodology

`Halborn` performed a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is essential to uncover flaws in logic, process, and implementation, automated testing techniques enhance coverage of smart contracts and can quickly identify issues that do not follow security best practices.

The following phases and associated tools were used throughout the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual code review and walkthrough of the smart contracts to identify potential logic issues.
- Manual testing of all core functions, including deposit, withdraw, repay, and borrow, to validate expected behavior and identify edge-case vulnerabilities.
- Local testing to simulate contract interactions and validate functional and security assumptions.
- Local deployment and testing with Foundry.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: dstoken

(b) Assessed Commit ID: f288c35

(c) Items in scope:

- contracts/rebasing/RebasingLibrary.sol
- contracts/rebasing/SecuritizeRebasingProvider.sol
- contracts/registry/RegistryService.sol
- contracts/registry/WalletRegistrar.sol
- contracts/swap/BaseSecuritizeSwap.sol
- contracts/swap/SecuritizeSwap.sol
- contracts/token/DSToken.sol
- contracts/token/StandardToken.sol
- contracts/token/TokenLibrary.sol
- contracts/trust/TrustService.sol
- contracts/utils/BaseDSContract.sol
- contracts/utils/BulkBalanceChecker.sol
- contracts/utils/CommonUtils.sol
- contracts/utils/MultiSigWallet.sol
- contracts/utils/TransactionRelayer.sol
- contracts/compliance/ComplianceConfigurationService.sol
- contracts/compliance/ComplianceService.sol
- contracts/compliance/ComplianceServiceNotRegulated.sol
- contracts/compliance/ComplianceServiceRegulated.sol
- contracts/compliance/ComplianceServiceWhitelisted.sol
- contracts/compliance/InvestorLockManager.sol
- contracts/compliance/InvestorLockManagerBase.sol
- contracts/compliance/LockManager.sol
- contracts/compliance/WalletManager.sol
- contracts/data-stores/BaseLockManagerDataStore.sol
- contracts/data-stores/ComplianceConfigurationDataStore.sol
- contracts/data-stores/ComplianceServiceDataStore.sol
- contracts/data-stores/InvestorLockManagerDataStore.sol
- contracts/data-stores/LockManagerDataStore.sol
- contracts/data-stores/RegistryServiceDataStore.sol
- contracts/data-stores/ServiceConsumerDataStore.sol
- contracts/data-stores/TokenDataStore.sol
- contracts/data-stores/TrustServiceDataStore.sol
- contracts/data-stores/WalletManagerDataStore.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- c2e62c9
- f14ea54
- 2d539a8
- 37e0a7b
- fb92b4d

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	2	4

INFORMATIONAL
7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PERMANENT CAP LOCK DUE TO TOTALISSUED NOT REDUCED ON BURN LEADS TO PROTOCOL STUCK AND NO ONE CAN PARTICIPATE	MEDIUM	SOLVED - 10/03/2025
UNTRACKED WALLETS FROM NO-COMPLIANCE ISSUANCE CAN BREAK INVESTOR RECORDS BECAUSE CHECKWALLETSFORLIST ISN'T CALLED	MEDIUM	SOLVED - 10/29/2025
BULK ISSUANCE ENFORCES A SINGLE ISSUANCETIME FOR ALL RECIPIENTS, PREVENTING FLEXIBLE SCHEDULING	LOW	RISK ACCEPTED - 10/06/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT REVERT REASON PARSING IN MULTICALLPROXY OBSCURES CUSTOM ERRORS	LOW	SOLVED - 10/29/2025
INEFFECTIVE LOCK TIME CONDITION IN GETCOMPLIANCETRANSFERABLETOKENS LEADS TO MISLEADING LOGIC	LOW	RISK ACCEPTED - 10/06/2025
UNSAFE ERC20 TRANSFERFROM AND APPROVE USAGE IN EXECUTESTABLECOINTRANSFER	LOW	SOLVED - 09/29/2025
MISSING TWO-STEP OWNERSHIP TRANSFER PROTECTION IN SERVICECONSUMER	INFORMATIONAL	ACKNOWLEDGED - 09/19/2025
UNNECESSARY DYNAMIC ARRAY ALLOCATION IN ISSUETOKENSCUSTOM WASTES GAS	INFORMATIONAL	ACKNOWLEDGED - 10/06/2025
POTENTIAL OUT-OF-GAS RISK IN GETTOKENBALANCES FOR LARGE INPUT ARRAYS	INFORMATIONAL	ACKNOWLEDGED - 10/06/2025
INCORRECT MAXIMUM HOLDINGS CHECK PREVENTS VALID TRANSFERS	INFORMATIONAL	SOLVED - 10/06/2025
INEFFICIENT USE OF DYNAMIC ARRAY FOR FIXED-LENGTH PARAMETERS	INFORMATIONAL	SOLVED - 09/29/2025
MISSING INPUT VALIDATION IN ISSUETOKENSWITHNOCOMPLIANCE FUNCTION	INFORMATIONAL	SOLVED - 09/29/2025
INCORRECT LOOP VARIABLE TYPE IN _REGISTERNEWINVESTOR	INFORMATIONAL	SOLVED - 09/29/2025

7. FINDINGS & TECH DETAILS

7.1 PERMANENT CAP LOCK DUE TO TOTALISSUED NOT REDUCED ON BURN LEADS TO PROTOCOL STUCK AND NO ONE CAN PARTICIPATE

// MEDIUM

Description

In the `TokenLibrary` contract's `issueTokensCustom` function, the protocol enforces the token cap using `totalIssued`. However, the `burn` function only reduces `totalSupply` and does not update `totalIssued`. This mismatch means that once the cap is reached, even if all tokens are burned, `totalIssued` still reflects the maximum issued amount. As a result, the protocol enters a permanent cap lock state where it is stuck and no one can participate in future token issuance.

Code Location

In `TokenLibrary.sol`, the cap check uses `totalIssued` instead of `totalSupply`:

```
69  function issueTokensCustom(
70          TokenData storage _tokenData,
71          address[] memory _services,
72          IDSLockManager _lockManager,
73          IssueParams memory _params
74      ) public returns (uint256) {
75          //Check input values
76          require(_params._to != address(0), "Invalid address");
77          require(_params._value > 0, "Value is zero");
78          require(_params._valuesLocked.length == _params._releaseTimes.length, "Wrong length c
79
80          uint256 totalIssuedTokens = _params._rebasingProvider.convertSharesToTokens(_tokenDat
81
82          //Make sure we are not hitting the cap
83          require(_params._cap == 0 || totalIssuedTokens + _params._value <= _params._cap, "Tok
84
85          //Check issuance is allowed (and inform the compliance manager, possibly adding locks
86          IDSComplianceService(_services[COMPLIANCE_SERVICE]).validateIssuance(_params._to, _pa
```

Proof of Concept

The following proof of concept demonstrates how the mismatch between `totalIssued` and `totalSupply` leads to a permanent cap lock. The step-by-step flow is as follows:

- The cap is set to 100,000 tokens.
- User1 is registered as an investor and issues the full cap amount (100,000 tokens).
- At this point, `totalSupply = 100,000 and totalIssued = 100,000`
- User1 immediately burns all 100,000 tokens.
- After burn: `totalSupply = 0 and totalIssued = 100,000` (unchanged, because burn does not reduce it)

- This creates a mismatch where the protocol sees the cap as “fully consumed” even though no tokens are circulating.
- Now when User2 (or any user) tries to issue even 1 token, the transaction reverts with `"Token Cap Hit"`.
- Here, `totalIssuedTokens = 100,000` and `_params._value = 1`, so `100,001 > 100,000`.

```
import { expect } from 'chai';
import { loadFixture, time } from '@nomicfoundation/hardhat-toolbox/network-helpers';
import hre from 'hardhat';
import { deployDSTokenRegulatedWithRebasingAndEighteenDecimal, INVESTORS } from './utils/fix
import { registerInvestor } from './utils/test-helper';

describe('Permanent Cap Lock State', function() {

  it.only('POC: Protocol gets permanently stuck after cap reached and tokens burned', async
    const [user1, user2] = await hre.ethers.getSigners();
    const { dsToken, registryService } = await loadFixture(deployDSTokenRegulatedWithRebasing

    // Register both users
    await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, user1, registryService);
    await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, user2, registryService);

    // Set cap to 100,000 tokens
    const CAP_AMOUNT = 100000;
    await dsToken.setCap(CAP_AMOUNT);

    // User1 issues tokens up to the cap (100,000 tokens)
    await dsToken.issueTokensCustom(user1.address, CAP_AMOUNT, await time.latest(), 0, "", 0

    const user1BalanceAfterIssue = await dsToken.balanceOf(user1.address);
    const totalSupplyAfterIssue = await dsToken.totalSupply();
    const totalIssuedAfterIssue = await dsToken.totalIssued();

    // Verify cap is reached
    expect(totalIssuedAfterIssue).to.equal(CAP_AMOUNT);
    expect(user1BalanceAfterIssue).to.equal(CAP_AMOUNT);

    // User1 immediately burns all their tokens
    await dsToken.burn(user1.address, CAP_AMOUNT, 'test burn');

    const user1BalanceAfterBurn = await dsToken.balanceOf(user1.address);
    const totalSupplyAfterBurn = await dsToken.totalSupply();
    const totalIssuedAfterBurn = await dsToken.totalIssued();

    // Verify user gets their money back (balance is 0)
    expect(user1BalanceAfterBurn).to.equal(0);
    expect(totalSupplyAfterBurn).to.equal(0);

    // This is the root cause of the permanent cap lock
    expect(totalIssuedAfterBurn).to.equal(CAP_AMOUNT); // Should be 0, but remains 100,000

    // Now when user2 tries to issue even 1 token, it will fail because:
    // totalIssuedTokens (100,000) + _params._value (1) > _params._cap (100,000)
    // This triggers: require(_params._cap == 0 || totalIssuedTokens + _params._value <= _par

    await expect(
      dsToken.issueTokensCustom(user2.address, 1, await time.latest(), 0, "", 0)
    ).to.be.revertedWith("Token Cap Hit");


    // Try different amounts - all should fail
    await expect(
      dsToken.issueTokensCustom(user2.address, 1, await time.latest(), 0, "", 0)
    ).to.be.revertedWith("Token Cap Hit");

    await expect(
      dsToken.issueTokensCustom(user2.address, 100, await time.latest(), 0, "", 0)
    ).to.be.revertedWith("Token Cap Hit");
```

```
    await expect(
      dsToken.issueTokensCustom(user1.address, 1, await time.latest(), 0, "", 0)
    ).to.be.revertedWith("Token Cap Hit");

  });
});
```

```
npx hardhat test
```

OutPut

```
Permanent Cap Lock State
    ✔ POC: Protocol gets permanently stuck after cap reached and tokens burned (599ms)


  1 passing (605ms)
```

BVSS

AO:A/AC:M/AX:L/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (6.7)

Recommendation

Update the cap check to use `totalSupply` instead of `totalIssued`, ensuring that burned tokens are correctly accounted for. This prevents the protocol from getting permanently stuck after reaching the cap.

Remediation Comment

SOLVED: The **Securitize team** resolved the issue in the specified commit ID by removing the `cap` attribute, as the `authorizedSecurities` compliance rule already covers it.

Remediation Hash

https://github.com/securitize-io/dstoken/commit/c2e62c9c1137bb7c6f548b72f960d864c42445fc

7.2 UNTRACKED WALLETS FROM NO-COMPLIANCE ISSUANCE CAN BREAK INVESTOR RECORDS BECAUSE CHECKWALLETSFORLIST ISN'T CALLED

// MEDIUM

Description

In the `DSToken` contract, the `issueTokensWithNoCompliance` function is used to mint tokens directly to a wallet but does not call `checkWalletsForList`. As a result, wallets that receive tokens through this function are not added to the protocol's official `walletsList`. This leads to inconsistencies where investors may show a positive balance with `balanceOf`, but their addresses are missing from the tracked `walletsList`.

Code Location

In `DSToken.sol`, the function omits a call to `checkWalletsForList`:

```
148 function issueTokensWithNoCompliance(address _to, uint256 _value) public virtual override onl
149         require(getRegistryService().isWallet(_to), "Unknown wallet");
150         ISecuritizeRebasingProvider rebasingProvider = getRebasingProvider();
151         uint256 shares =  TokenLibrary.issueTokensWithNoCompliance(
152             tokenData,
153             getCommonServices(),
154             _to,
155             _value,
156             block.timestamp,
157             cap,
158             rebasingProvider
159         );
160         emit Transfer(address(0), _to, _value);
161         emit TxShares(address(0), _to, shares, rebasingProvider.multiplier());
162 }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (5.0)

Recommendation

Add a call to `checkWalletsForList(address(0), _to)` inside `issueTokensWithNoCompliance`, just like in `issueTokensWithMultipleLocks`.

Remediation Comment

SOLVED: The **Securitize team** resolved the issue in the specified commit ID by removing the `issueTokensWithNoCompliance` function.

Remediation Hash

https://github.com/securitize-io/dstoken/commit/f14ea54928678fb56f3dbaa11b6e27236b306a41

7.3 BULK ISSUANCE ENFORCES A SINGLE ISSUANCETIME FOR ALL RECIPIENTS, PREVENTING FLEXIBLE SCHEDULING

// LOW

Description

In the `BulkOperator` contract, the `bulkIssuance` function sets the same `issuanceTime` for all recipients in a single call. Therefore, when multiple addresses receive tokens through bulk issuance, they are all forced to share an identical issuance timestamp. This prevents issuers from assigning individualized issuance times in one operation.

Code Location

In `BulkOperator.sol`, `bulkIssuance` enforces the same `issuanceTime` for all recipients:

```
51  function bulkIssuance(address[] memory addresses, uint256[] memory values, uint256 issuanceT
52          require(addresses.length == values.length, "Addresses and values length mismatch");
53
54          for (uint256 i = 0; i < addresses.length; i++) {
55              dsToken.issueTokensCustom(addresses[i], values[i], issuanceTime, 0, "", 0);
56          }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

Modify the function to accept an array of `issuanceTime` values aligned with `addresses` and `values`.

```
51  function bulkIssuance(
52      address[] memory addresses,
53      uint256[] memory values,
54      uint256[] memory issuanceTimes
55  ) whenNotPaused onlyIssuerOrAbove external {
56      require(
57          addresses.length == values.length && values.length == issuanceTimes.length,
58          "Input arrays length mismatch"
59      );
60
61      for (uint256 i = 0; i < addresses.length; i++) {
62          dsToken.issueTokensCustom(addresses[i], values[i], issuanceTimes[i], 0, "", 0);
63      }
64  }
```

Remediation Comment

RISK ACCEPTED: The **Securitize team** has accepted the risk of this finding.

7.4 INCORRECT REVERT REASON PARSING IN MULTICALLPROXY OBSCURES CUSTOM ERRORS

// LOW

Description

The `MulticallProxy` contract's helper function `_getRevertReason` assumes all revert payloads follow the `Error(string)` format used by `require` and `revert("...")`. This works for standard error messages but fails for custom errors, which encode a 4-byte selector and ABI-encoded arguments instead of a string. When `_getRevertReason` attempts to decode such payloads as a string, it either reverts or produces meaningless output. As a result, the original revert cause is lost, and `MulticallFailed` reports misleading or generic error messages.

For example, if the error is `require(false, "Not enough balance");`, the encoded revert data is:

```
0x08c379a0                                                              // selector for Error(string)
0000000000000000000000000000000000000000000000000000000000000020       // offset to string data (32 by
0000000000000000000000000000000000000000000000000000000000000014       // string length = 20 bytes
4e6f7420656e6f7567682062616c616e6365000000000000000000000000000000     // "Not enough balance"
```

If error is custom such as `NotAllowed(msg.sender)`, assuming `msg.sender = 0x1111111111111111111111111111111111111111`:

```
0xd09de08a                                                              // selector for NotAllowed(addr
0000000000000000000000001111111111111111111111111111111111111111       // msg.sender argument
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

To handle both standard and custom errors safely, verify the value of the first 4 bytes of the revert payload and:

- If the selector equals `0x08c379a0` (`Error(string)`), decode the payload as `(string)`.
- Otherwise, treat it as a custom or unknown error and return a generic fallback string such as `"Custom error (raw data: 0x...)"`.

Remediation Comment

SOLVED: The **Securitize team** resolved the issue in the specified commit ID by removing the `MulticallProxy` contract.

Remediation Hash

https://github.com/securitize-io/dstoken/commit/2d539a81da45377bb11415f59c15bdbe4d296da7

7.5 INEFFECTIVE LOCK TIME CONDITION IN GETCOMPLIANCETRANSFERABLETOKENS LEADS TO MISLEADING LOGIC

// LOW

Description

In the `ComplianceServiceRegulated` contract, the `getComplianceTransferableTokens` function contains a condition that always evaluates to false in practice. The first condition `uint256(_lockTime) > _time` will always evaluate to false in realistic scenarios, since `_lockTime` represents a duration (e.g., 1 day, 30 days) and `_time` is a current timestamp. A lock duration (in seconds) cannot exceed an absolute timestamp in the future, so this check never contributes to the condition.

Code Location

In `ComplianceServiceRegulated.sol`, the first clause of the lock time check is ineffective:

```
701 function getComplianceTransferableTokens(
702         address _who,
703         uint256 _time,
704         uint64 _lockTime
705     ) public view override returns (uint256) {
706     ....
707     ....
708      if (uint256(_lockTime) > _time || issuanceTimestamp > (_time - uint256(_lockTime))) {
709                 uint256 tokens = getRebasingProvider().convertSharesToTokens(issuancesValues
710                 totalLockedTokens = totalLockedTokens + tokens;
711      }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

Remove the ineffective check to make the logic clear and correct.

Remediation Comment

RISK ACCEPTED: The **Securitize team** has accepted the risk of this finding.

7.6 UNSAFE ERC20 TRANSFERFROM AND APPROVE USAGE IN EXECUTESTABLECOINTRANSFER

// LOW

Description

In the `SecuritizeSwap` contract's `executeStableCoinTransfer` function, the contract directly calls `transferFrom` and `approve` on the ERC20 token without verifying return values or handling non-standard implementations. Many tokens, such as **USDT**, do not fully comply with the ERC20 specification and may return no boolean value or require resetting allowances before reapproving.

Code Location

In `SecuritizeSwap.sol`, `executeStableCoinTransfer` interacts with ERC20 tokens using unsafe calls:

```
243 function executeStableCoinTransfer(address from, uint256 value) private {
244         if (bridgeChainId != 0 && address(USDCBridge) != address(0)) {
245             stableCoinToken.transferFrom(from, address(this), value);
246             stableCoinToken.approve(address(USDCBridge), value);
247             USDCBridge.sendUSDCCrossChainDeposit(bridgeChainId, issuerWallet, value);
248         } else {
249             stableCoinToken.transferFrom(from, issuerWallet, value);
250 }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (2.1)

Recommendation

Use OpenZeppelin's SafeERC20 library (safeTransferFrom, safeApprove) to ensure compatibility with both standard and non-standard ERC20 tokens.

Remediation Comment

SOLVED: The **Securitize team** resolved the issue in the specified commit ID by removing the `SecuritizeSwap` smart contract.

Remediation Hash

https://github.com/securitize-io/dstoken/commit/37e0a7b878ca6fc1a677720a0b8e68f54a596484

7.7 MISSING TWO-STEP OWNERSHIP TRANSFER PROTECTION IN SERVICECONSUMER

// INFORMATIONAL

Description

The `ServiceConsumer` contract inherits from `OwnableUpgradeable`, which uses a single-step ownership transfer model. In this model, ownership is directly transferred to the new address in a single transaction. This creates the risk of accidental loss of ownership if an invalid or incorrect address is provided, or if the new owner does not have access to the account. Without a confirmation step, control of the contract could be irreversibly lost.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

Replace `OwnableUpgradeable` with `Ownable2StepUpgradeable` from OpenZeppelin. This enforces a secure two-step process where ownership transfer requires both an initiation by the current owner and explicit acceptance by the new owner, significantly reducing the risk of accidental loss of contract control.

Remediation Comment

ACKNOWLEDGED: The **Securitize team** acknowledged the issue.

7.8 UNNECESSARY DYNAMIC ARRAY ALLOCATION IN ISSUETOKENSCUSTOM WASTES GAS

// INFORMATIONAL

Description

In `TokenLibrary` contract's `issueTokensCustom`, the function creates dynamic arrays `uint256[] valuesLocked` and `uint64[] releaseTimes` even though at most **one element** is ever used.

```
94  function issueTokensCustom(address _to, uint256 _value, uint256 _issuanceTime, uint256 _valu
95      public
96      virtual
97      override
98      returns (
99      /*onlyIssuerOrAbove*/
100         bool
101     )
102     {
103         uint256[] memory valuesLocked;
104         uint64[] memory releaseTimes;
105         if (_valueLocked > 0) {
106             valuesLocked = new uint256[](1);
107             releaseTimes = new uint64[](1);
108             valuesLocked[0] = _valueLocked;
109             releaseTimes[0] = _releaseTime;
110         }
111
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

Replace dynamic arrays with simple fixed-size variables or pass values directly to `issueTokensWithMultipleLocks`.

Remediation Comment

ACKNOWLEDGED: The **Securitize team** acknowledged this finding.

7.9 POTENTIAL OUT-OF-GAS RISK IN GETTOKENBALANCES FOR LARGE INPUT ARRAYS

// INFORMATIONAL

Description

The `BulkBalanceChecker` contract's `getTokenBalances` accepts an unbounded `address[] calldata wallets` array from external callers. It allocates a new `uint256[]` of the same size and performs an ERC20 `balanceOf` call for each wallet. While Solidity itself prevents true “out-of-bounds” errors, providing a large input array can cause the transaction to run out of gas due to the unbounded loop.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

Impose an upper bound on `wallets.length` to prevent denial-of-service.

Remediation Comment

ACKNOWLEDGED: The **Securitize team** acknowledged this finding.

7.10 INCORRECT MAXIMUM HOLDINGS CHECK PREVENTS VALID TRANSFERS

// INFORMATIONAL

Description

In the `ComplianceServiceLibrary` contract's `isMaximumHoldingsPerInvestorOk` function, the maximum holdings check uses the strict `>` operator instead of `>=`. This creates an off-by-one error where a user cannot hold exactly the maximum allowed number of tokens.

For example, if `_maximumHoldingsPerInvestor = 100`, `_balanceOfInvestorTo = 90`, and `_value = 10`, then the condition evaluates as: `90 + 10 > 100 → false`.

The transfer is allowed, but the resulting balance is **exactly 100**, which should be permitted since it does not exceed the cap.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

Update the condition to include equality, ensuring that the maximum value is respected.

```
1 | return _maximumHoldingsPerInvestor != 0 && _balanceOfInvestorTo + _value >= _maximumHoldingsP
```

Remediation Comment

SOLVED: The **Securitize team** resolved the issue in the specified commit ID.

Remediation Hash

https://github.com/securitize-io/dstoken/commit/fb92b4d389f8a24c670cd5d157894fd29f508c02

7.11 INEFFICIENT USE OF DYNAMIC ARRAY FOR FIXED-LENGTH PARAMETERS

// INFORMATIONAL

Description

The `SecuritizeSwap` contract's `executePreApprovedTransaction` function accepts a `uint256[] memory params` argument but immediately enforces that `params.length == 2`. Since the array size is fixed at 2 elements, using a dynamic array introduces unnecessary memory allocation and validation overhead. Dynamic arrays are more expensive in gas cost compared to fixed-size arrays, and they signal flexibility that does not exist in this case. This increases runtime gas consumption.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.8)

Recommendation

Use a fixed-length array type (`uint256[2] memory params`) instead of a dynamic array.

Remediation Comment

SOLVED: The **Securitize team** resolved the issue in the specified commit ID by removing the `SecuritizeSwap` smart contract.

Remediation Hash

https://github.com/securitize-io/dstoken/commit/37e0a7b878ca6fc1a677720a0b8e68f54a596484

7.12 MISSING INPUT VALIDATION IN ISSUETOKENSWITHNOCOMPLIANCE FUNCTION

// INFORMATIONAL

Description

In the `TokenLibrary` contract's `issueTokensWithNoCompliance` mints tokens directly to a recipient but does not perform basic sanity checks on input values. In contrast, the similar function `issueTokensCustom` enforces that the recipient address is not the zero address, the value is greater than zero, and the lock arrays have consistent lengths. Since `issueTokensWithNoCompliance` does not perform these checks, it allows tokens to be minted to the zero address or with a value of zero, which can lead to inconsistent behavior and potential accounting issues.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.8)

Recommendation

Add the same input validation present in `issueTokensCustom` to `issueTokensWithNoCompliance`.

Remediation Comment

SOLVED: The **Securitize team** resolved the issue in the specified commit ID by removing the `issueTokensWithNoCompliance` function.

Remediation Hash

https://github.com/securitize-io/dstoken/commit/f14ea54928678fb56f3dbaa11b6e27236b306a41

7.13 INCORRECT LOOP VARIABLE TYPE IN _REGISTERNEWINVESTOR

// INFORMATIONAL

Description

In the `SecuritizeSwap` contract's `_registerNewInvestor` function, the contract iterates through `_investorAttributeIds` using a `uint256` index, even though `_investorAttributeIds` is defined as a `uint8[]`. This mismatch is not a functional bug but introduces unnecessary gas costs because the index type does not match the element type.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Use `uint8` indexing instead `uint256`.

Remediation Comment

SOLVED: The **Securitize team** resolved the issue in the specified commit ID by removing the `SecuritizeSwap` smart contract.

Remediation Hash

https://github.com/securitize-io/dstoken/commit/37e0a7b878ca6fc1a677720a0b8e68f54a596484
