Securitize DSToken Rebasing Audit Report

Prepared by Cyfrin
Version 2.1

Lead Auditors

Dacian

Stalin

Jorge

October 10, 2025

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The DSToken protocol implements a comprehensive regulated security token system designed to bridge traditional securities compliance with blockchain technology. At its core, the protocol consists of a rebasing ERC20 token `DSToken` that maintains regulatory compliance through an interconnected suite of smart contracts managing investor registration, transfer restrictions, and multi-jurisdictional regulatory requirements.

The architecture employs a modular service-oriented design where each component handles specific regulatory or operational functions. The `RegistryService` maintains KYC/AML investor records, tracking attributes like accreditation status and country of residence, while supporting multiple wallets per investor identity. The `ComplianceService` enforces transfer restrictions based on configurable rules including investor limits by jurisdiction, lockup periods for newly issued tokens, minimum holding requirements, and forced transfer conditions. The system distinguishes between US accredited/non-accredited investors and EU qualified/retail investors, implementing different regulatory frameworks for each jurisdiction.

Token economics are managed through a rebasing mechanism where a multiplier adjusts token balances to reflect dividends or corporate actions without requiring explicit distributions. The protocol implements sophisticated locking mechanisms operating at both wallet and investor levels, including manual locks for administrative purposes and automatic lockup periods for regulatory compliance. These locks use different accounting methods - manual locks track fixed token amounts while compliance lockups track share amounts that scale with rebasing events.

The system supports multiple operational roles including Master, Issuer, Exchange, and Transfer Agent, each with specific permissions enforced through the `TrustService`. Token issuance flows through the `TokenIssuer` contract which coordinates compliance validation, registry updates, and lock creation. Additional infrastructure includes a `MultiSigWallet` for governance operations, a `TransactionRelayer` for pre-signed transactions, and a `SecuritizeSwap` contract enabling compliant token purchases using stablecoins with integrated KYC onboarding.

The protocol's design prioritizes regulatory compliance and operational flexibility, supporting features like investor liquidation modes, cross-chain USDC bridging, and comprehensive event logging for audit trails. The upgradeable proxy pattern allows for protocol evolution while maintaining consistent contract addresses and state continuity.

5 Audit Scope

The scope of this audit was limited to:

```
contracts/bulk/BulkOperator.sol
contracts/compliance/ComplianceConfigurationService.sol
contracts/compliance/ComplianceService.sol
contracts/compliance/ComplianceServiceNotRegulated.sol
contracts/compliance/ComplianceServiceRegulated.sol
contracts/compliance/ComplianceServiceWhitelisted.sol
contracts/compliance/IDSComplianceConfigurationService.sol
contracts/compliance/IDSComplianceService.sol
contracts/compliance/IDSLockManager.sol
contracts/compliance/IDSWalletManager.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/TokenDataStore.sol
contracts/data-stores/ServiceConsumerDataStore.sol
contracts/data-stores/TrustServiceDataStore.sol
contracts/data-stores/WalletManagerDataStore.sol
contracts/issuance/IDSTokenIssuer.sol
contracts/issuance/TokenIssuer.sol
contracts/multicall/IssuerMulticall.sol
contracts/multicall/MulticallProxy.sol
contracts/rebasing/RebasingLibrary.sol
contracts/rebasing/SecuritizeRebasingProvider.sol
contracts/registry/IDSRegistryService.sol
contracts/registry/IDSWalletRegistrar.sol
contracts/registry/RegistryService.sol
contracts/registry/WalletRegistrar.sol
contracts/service/IDSServiceConsumer.sol
contracts/service/ServiceConsumer.sol
contracts/swap/BaseSecuritizeSwap.sol
contracts/swap/SecuritizeSwap.sol
contracts/token/DSToken.sol
contracts/token/IDSToken.sol
contracts/token/StandardToken.sol
contracts/token/TokenLibrary.sol
contracts/trust/IDSTrustService.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
```

6 Executive Summary

Over the course of 15 days, the Cyfrin team conducted an audit on the Securitize DSToken Rebasing smart contracts provided by Securitize. In this period, a total of 78 issues were found.

The findings consist of 1 Critical, 9 Medium and 23 Low severity issues with the remainder being informational and gas optimizations.

The single Critical finding allowed investors to steal tokens from other investors as spending approvals were not being checked for `transferFrom` operations.

The 9 Medium and 23 Lows included a wide variety of issues such as:

- various ways total investor counts could be corrupted, triggering DoS on transfers or not correctly enforcing compliance rules related to maximum number of investors
- inconsistencies in rules being enforced for similar operations
- various ways investors could abuse the ability to add their own wallets
- issues related to new rebasing functionality
- incorrect EIP-712 type hashes

Centralization Risks:

Due to the regulated nature of the underlying assets being tokenized and applicable regulatory requirements, the protocol is highly centralized by design including the ability for protocol admins to burn and seize user assets; users must place a high degree of trust in the protocol team.

The protocol if correctly configured should not allow anonymous attackers to call state-changing functions; consequently it has a reduced attack surface compared to permissionless/anonymous DeFi protocols.

Summary

Project Name	Securitize DSToken Rebasing
Repository	dstoken
Commit	f288c359ee85...
Fix Commit	f20fd8d0fc3f...
Audit Timeline	Sep 1st - Sep 19th, 2025
Methods	Manual Review

Issues Found

Critical Risk	1
High Risk	0
Medium Risk	9
Low Risk	23
Informational	25
Gas Optimizations	20
Total Issues	78

Summary of Findings

[C-1] Investors can steal tokens from other investors since `StandardToken::transferFrom` never checks spending approvals	Resolved
[M-1] Country updates is not reducing from the storage the prev country	Resolved
[M-2] Incorrect EIP-712 `TXTYPE_HASH` in `SecuritizeSwap`	Resolved
[M-3] Changing investor country to the same country inflates investor count erroneously triggering max investor errors	Resolved
[M-4] Historical token issuances become subject to new lock-up requirements when users change residency	Acknowledged
[M-5] No way to revert `setInvestorLiquidateOnly`	Resolved
[M-6] `euRetailInvestorsCount` is susceptible to underflow when transferring tokens between retail and qualified investors of the same country on the EU region	Acknowledged
[M-7] `ComplianceServiceRegulated` applies double locking to investor transferable tokens leading to DoS of transfers	Acknowledged
[M-8] Token value locks done as part of issuance restrict subsequently transferred unlocked tokens after negative rebasing	Acknowledged
[M-9] Investors transferring all their balances among their wallets or self-transferring on the same wallet causes to incorrectly decrement the investor counters causing DoS for other investors transfers	Resolved
[L-01] Protocol will leak value to users due to rounding in `RebasingLibrary`	Acknowledged
[L-02] Multiplication could overflow in `RebasingLibrary` for tokens with greater than 18 decimals	Resolved
[L-03] Asymmetry enforcement between `TokenIssuer::registerInvestor`, `WalletRegistrar::registerWallet` and `SecuritizeSwap::_registerNewInvestor`	Resolved
[L-04] Protocol will leak value to users due to rounding in `SecuritizeSwap::buy`	Resolved
[L-05] Missing `checkWalletsForList` in `issueTokensWithNoCompliance`	Resolved
[L-06] Transferring all the investor balance from a non-us investor to a new us investor allows to bypass the `usInvestorLimit`	Resolved
[L-07] `TrustService::changeEntityOwner` can overwrite existing `_newOwner` record, breaking 1-1 relationship between owners and addresses	Resolved
[L-08] `TrustService::removeRole` doesn't delete already owned entities so address which lost role can still manage existing entities	Resolved
[L-09] No way for users to invalidate nonce used to sign	Acknowledged
[L-10] No deadline for user signatures	Resolved
[L-11] Stale Issuance Records After `Burn` and `Seize` Operations	Acknowledged
[L-12] Malicious investor can register wallets that belong to other investors	Resolved
[L-13] `SecuritizeSwap::executeStableCoinTransfer` should use `SafeERC20` approval and transfer functions instead of standard `IERC20` functions	Resolved
[L-14] `MulticallProxy::_callTarget` doesn't check if `target` has code	Resolved

[L-15] Seized tokens will be stuck on the receiver of the seized funds because it is not an investor and never receives investorsBalance	Acknowledged
[L-16] `ComplianceServiceRegulated::preIssuanceCheck` allows issuance to non-accredited investors when `forceAccredited` or `forceAccreditedUS` is set, and allows issuance below regional minimum thresholds, violating compliance requirements	Resolved
[L-17] Burn and seize functions can be DoS when investor has several wallets that they control	Resolved
[L-18] Not maximum amount of issuances enforce	Acknowledged
[L-19] Platform Wallet not exception for Maximum Holdings Per Investor Limits	Resolved
[L-20] Attribute changes via `setAttribute` or `updateInvestor` do NOT trigger compliance count updates.	Acknowledged
[L-21] Investor can prevent themselves from being removed by making `removeInvestor` revert	Resolved
[L-22] Resolve inconsistency between `DSToken::checkWalletsForList` and `RegistryService::removeWallet`	Resolved
[L-23] `RegistryService::addWallet` should revert if the wallet being added has positive balance of `DSToken`	Resolved
[I-01] Use named imports	Resolved
[I-02] Upgradeable contracts should call `_disableInitializers` in constructor	Resolved
[I-03] Prefer explicit unsigned integer sizes	Resolved
[I-04] Use named mapping parameters to explicitly note the purpose of keys and values	Resolved
[I-05] Emit missing events	Resolved
[I-06] Consider reverting in `RebasingLibrary` functions if rounding down to zero occurs	Resolved
[I-07] Use named constants instead of hard-coded literals for important values	Acknowledged
[I-08] Remove obsolete `return` statements when already using named return variables	Resolved
[I-09] Prefer `ECDSA::tryRecover` to using `ecrecover` directly	Resolved
[I-10] `TransactionRelayer` and `SecuritizeSwap` should use `CommonUtils::encodeString`	Resolved
[I-11] Not mechanism to automatically cleanup locks that are already unlockable on LockManagers	Acknowledged
[I-12] `SecuritizeSwap::buy` should revert if `stableCoinAmount` is zero	Resolved
[I-13] Possible to escape burning of DSTokens by removing the wallet from the current investor and adding it as the wallet of another investor	Acknowledged
[I-14] Refactor duplicated checks into modifiers	Acknowledged
[I-15] `InvestorLockManager::createLockForInvestor`, `removeLockRecordForInvestor` should revert for invalid investor id	Acknowledged

[I-16] Refactor away duplicated code between `ComplianceService::newPreTransferCheck` and `preTransferCheck`	Resolved
[I-17] Cache compliance service and compliance configuration service for much cleaner code in `ComplianceServiceRegulated::completeTransferCheck`	Acknowledged
[I-18] `ComplianceServiceRegulated::getServices` should use constants from `ComplianceServiceLibrary` when setting array indexes	Acknowledged
[I-19] Using `block.number` instead of `block.timestamp` as an expiration check to execute signature	Acknowledged
[I-20] Remove useless function `ComplianceServiceRegulated::adjustTransferCounts`	Resolved
[I-21] Protocol classifies retail investors based solely on whether they are qualified investors, without checking if they are accredited investors	Resolved
[I-22] Not possible to rehash `DOMAIN_SEPARATOR` in `MultiSigWallet` to update chainId if is ever required	Resolved
[I-23] `ComplianceConfigurationService::getWorldWideForceFullTransfer` is not applied to US investors	Acknowledged
[I-24] `ComplianceServiceRegulated` and its parent `ComplianceServiceWhitelisted` uses a chain of `initializer` modifiers when calling the `initialize`	Resolved
[I-25] Feature flags are not being used in the `TokenLibrary`	Acknowledged
[G-01] Don't perform storage reads unless necessary	Resolved
[G-02] Cache result of identical external calls when the result can't change	Acknowledged
[G-03] Use `calldata` for read-only external inputs	Resolved
[G-04] In Solidity don't initialize to default values	Acknowledged
[G-05] Cache storage to prevent identical storage reads	Acknowledged
[G-06] Use named return variables where this optimizes away a local variable definition	Acknowledged
[G-07] Since attribute expiration is deprecated, remove as input parameters, don't write it to storage and put comment explaining this	Acknowledged
[G-08] Remove setting deprecated `lastUpdatedBy` in RegistryService	Resolved
[G-09] Cheaper not to cache `calldata` array length	Resolved
[G-10] More efficient way of checking for empty string in `CommonUtils::isEmptyString`	Resolved
[G-11] More efficient way of comparing two strings for equality in `CommonUtils::isEqualString`	Acknowledged
[G-12] Cache computation results instead of repeatedly performing the same computation	Acknowledged
[G-13] Remove deprecated collision hash and proof hash from function calls	Acknowledged
[G-14] Return fast in `ComplianceServiceRegulated::checkHoldUp` if platform wallet	Resolved

[G-15] Perform local variable checks first prior to external calls in composite `if` statement conditions	Acknowledged
[G-16] Fast fail without performing unnecessary storage reads or external calls	Resolved
[G-17] Not possible to send native via the `TransactionRelayer` to the target contract when executing `executeByInvestorWithBlockLimit`	Resolved
[G-18] Don't write to the same storage slot multiple times	Resolved
[G-19] `ComplianceServiceRegulated::getComplianceTransferableTokens` should call `IDSLockManager::getTransferableTokensForInvestor`	Resolved
[G-20] Remove return value from `DSToken::updateInvestorBalance` as it is never checked	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Investors can steal tokens from other investors since `StandardToken::transferFrom` never checks spending approvals

Description: Investors can steal tokens from other investors since `StandardToken::transferFrom` never checks spending approvals.

Proof of Concept: Add PoC to `test/dstoken-regulated.test.ts`:

```
describe('TransferFrom', function () {
  it('Investors can steal tokens from other investors', async function () {
    // setup 2 investors
    const [investor, investor2] = await hre.ethers.getSigners();
    const { dsToken, registryService, rebasingProvider } = await loadFixture(deployDSTokenRegulated);
    await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, investor, registryService);
    await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, investor2, registryService);

    // give first investor some tokens
    await dsToken.issueTokens(investor, 500);

    const valueToTransfer = 100;
    const shares = await rebasingProvider.convertTokensToShares(valueToTransfer);
    const multiplier = await rebasingProvider.multiplier();

    // connect as second investor
    const dsTokenFromInvestor = await dsToken.connect(investor2);

    // use `transferFrom` to steal tokens from first investor, even though
    // first investor never approved second investor as a spender
    await expect(dsTokenFromInvestor.transferFrom(investor, investor2, valueToTransfer))
      .to.emit(dsToken, 'TxShares')
      .withArgs(investor.address, investor2.address, shares, multiplier);
  });
});
```

Run with: `npx hardhat test --grep "Investors can steal tokens from other investors"`.

Recommended Mitigation: `StandardToken::transferFrom` must enforce spending approvals.

Securitize: Fixed in commit aefb895.

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 Country updates is not reducing from the storage the prev country

Description: The `adjustInvestorCountsAfterCountryChange` function in `ComplianceServiceRegulated.sol` contains an accounting flaw that leads to double counting of investors across countries. When an investor changes their country through `RegistryService.setCountry()`, the function correctly increases the investor counts for the new country but fails to decrease the counts for the previous country:

```
function setCountry(string calldata _id, string memory _country) public override onlyExchangeOrAbove
↪  investorExists(_id) returns (bool) {
        string memory prevCountry = getCountry(_id); <----------

        getComplianceService().adjustInvestorCountsAfterCountryChange(_id, _country, prevCountry);
        ...
    }
```

In the `adjustInvestorCountsAfterCountryChange` the `prevCountry` is ignored:

```
function adjustInvestorCountsAfterCountryChange(
    string memory _id,
    string memory _country,
    string memory /*_prevCountry*/  // Previous country parameter is ignored
) public override onlyRegistry returns (bool) {
    if (getToken().balanceOfInvestor(_id) == 0) {
        return false;
    }

    // Only increases counts for new country
    adjustInvestorsCountsByCountry(_country, _id, CommonUtils.IncDec.Increase);
    // Missing: adjustInvestorsCountsByCountry(_prevCountry, _id, CommonUtils.IncDec.Decrease);

    return true;
}
```

Then in `adjustInvestorsCountsByCountry` several investor counting variables are increased without decreasing the prevCountry.

Each country change inflates the following counters without decrementing the previous country:

- `accreditedInvestorsCount`
- `usInvestorsCount`
- euRetailInvestorsCount[country]
- jpInvestorsCount

These variables are important since those are use to impose maximum investor limits.

Impact: * Incorrect calculation of investor limits per jurisdiction

- Potential regulatory violations due to inaccurate reporting

Proof of Concept: Run the next proof of concept in `file:compliance-service-regulated.ts`

```
 it('PoC: Country change double counting bug', async function() {
      const [wallet] = await hre.ethers.getSigners();
      const { dsToken, registryService, complianceConfigurationService, complianceService } = await
      ↪  loadFixture(deployDSTokenRegulated);

      // Setup
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.USA,
      ↪  INVESTORS.Compliance.US);
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.JAPAN,
      ↪  INVESTORS.Compliance.JP);
```

```
    await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, wallet, registryService);
    await registryService.setAttribute(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, 2, 1, 0, ""); // Make
    ↪  accredited

    // Set initial country and issue tokens
    await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.USA);
    await dsToken.setCap(1000);
    await dsToken.issueTokens(wallet, 100);

    // Verify initial state: 1 US investor
    expect(await complianceService.getUSInvestorsCount()).to.equal(1);
    expect(await complianceService.getJPInvestorsCount()).to.equal(0);

    // Change country from USA to JAPAN
    await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.JAPAN);

    // VULNERABILITY: Both countries now count the same investor
    expect(await complianceService.getUSInvestorsCount()).to.equal(1); // Should be 0
    expect(await complianceService.getJPInvestorsCount()).to.equal(1);  // Should be 1
  });
```

Recommended Mitigation: Modify the `adjustInvestorCountsAfterCountryChange` function to properly decrease the previous country's counts before increasing the new country's counts:

```
function adjustInvestorCountsAfterCountryChange(
    string memory _id,
    string memory _country,
    string memory _prevCountry  // Use this parameter
) public override onlyRegistry returns (bool) {
    if (getToken().balanceOfInvestor(_id) == 0) {
        return false;
    }

+    if (bytes(_prevCountry).length > 0) {
+        adjustInvestorsCountsByCountry(_prevCountry, _id, CommonUtils.IncDec.Decrease);
+    }

    adjustInvestorsCountsByCountry(_country, _id, CommonUtils.IncDec.Increase);

    return true;
}
```

Securitize: Fixed in commit 2c76e2f.

Cyfrin: Verified.

7.2.2 Incorrect EIP-712 `TXTYPE_HASH` in `SecuritizeSwap`

Description: `SecuritizeSwap::TXTYPE_HASH` does not match the actual message structure being encoded for EIP-712 signature verification. The constant is defined as the hash of:

```
"ExecutePreApprovedTransaction(string memory _senderInvestor, address _destination,address
↪  _executor,bytes _data, uint256[] memory _params)"
```

But the actual `abi.encode` call in `SecuritizeSwap::doExecuteByInvestor` encodes a different structure:

```
function doExecuteByInvestor(
        uint8 _securitizeHsmSigV,
        bytes32 _securitizeHsmSigR,
        bytes32 _securitizeHsmSigS,
        string memory _senderInvestorId,
        address _destination,
        bytes memory _data,
        address _executor,
        uint256[] memory _params
    ) internal {
        bytes32 txInputHash = keccak256(
            abi.encode(
                TXTYPE_HASH,
                _destination,
                _params[0], //value
                keccak256(_data),
                noncePerInvestor[_senderInvestorId],
                _executor,
                _params[1], //gasLimit
                keccak256(abi.encodePacked(_senderInvestorId))
            )
        );
        ...
    }
```

The type string declares 5 parameters in this order:

1) string _senderInvestor
2) address _destination
3) address _executor
4) bytes _data
5) uint256[] _params

But the actual encoding has 7 parameters:

1) address destination
2) uint256 value (from params[0])
3) bytes32 dataHash (keccak of _data)
4) uint256 nonce
5) address executor
6) uint256 gasLimit (from params[1])
7) bytes32 investorIdHash (keccak of _senderInvestor)

This is a significant mismatch between the expected message type and the actual encoded parameters that violates EIP-712 structured data standards.

Another problem is that the EIP-712 spec does not have storage qualifiers like `storage` and `memory` in the type hash string. So even if the above mismatch didn't occur, the current hash is still incorrect as it has been created using an incorrect type hash string.

Impact: The implementation violates EIP-712 standards which could cause compatibility issues with wallets and signing tools that expect proper EIP-712 compliance.

Recommended Mitigation: Correct `TXTYPE_HASH`:

- remove `memory` keyword from type hash string
- change type hash string to match the actual encoded parameters
- generate a new hash using the corrected type hash string

A potential fix looks like:

```
// keccak256("ExecutePreApprovedTransaction(address destination,uint256 value,bytes32 data,uint256
↪  nonce,address executor,uint256 gasLimit,bytes32 investorIdHash)")
bytes32 constant TXTYPE_HASH = 0xf13da213cea16aa5bb997703966334f85e5aa4d2b25964a7191e3a7bc08b7690;
```

Securitize: `SecuritizeSwap` was removed as it was deprecated.

Cyfrin: Verified.

7.2.3 Changing investor country to the same country inflates investor count erroneously triggering max investor errors

Description: Changing investor country to the same country inflates investor count.

Impact: Inflating the investor count will erroneously trigger `MAX_INVESTORS_IN_CATEGORY` error in `ComplianceServiceRegulated::completeTransferCheck`. This does not require any error on the admin's part since the admin can call `RegistryService::updateInvestor` where the country remains the same but other investor properties are being changed, and this ends up calling `ComplianceServiceRegulated::adjustInvestorCountsAfterCountryChange` and inflating the investor count.

Proof of Concept: Add PoC to `test/compliance-service-regulated.test.ts`:

```
    it('Changing to the same country inflates investor count', async function() {
      const [wallet] = await hre.ethers.getSigners();
      const { dsToken, registryService, complianceConfigurationService, complianceService } = await
      ↪  loadFixture(deployDSTokenRegulated);

      // Setup
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.USA,
      ↪  INVESTORS.Compliance.US);

      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, wallet, registryService);
      await registryService.setAttribute(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, 2, 1, 0, ""); // Make
      ↪  accredited

      // Set initial country and issue tokens
      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.USA);
      await dsToken.setCap(1000);
      await dsToken.issueTokens(wallet, 100);

      // Verify initial state: 1 US investor
      expect(await complianceService.getUSInvestorsCount()).to.equal(1);

      // Change country from USA to USA
      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.USA);

      // bug: us investor count increased even though it is the same investor
      //      and their country hasn't actually changed
      expect(await complianceService.getUSInvestorsCount()).to.equal(2);
    });
```

Run with: `npx hardhat test --grep "Changing to the same country inflates investor count"`

Recommended Mitigation: `ComplianceServiceRegulated::adjustInvestorCountsAfterCountryChange` should revert or simply not adjust anything if `_country` and `_prevCountry` are the same.

The second option (not adjust anything) may be preferred since this can end up being called from an original call to `RegistryService::updateInvestor` where the country remains the same but other investor properties are being changed.

Securitize: Fixed in commit 86d4135 by changing `RegistryService::setCountry` (which calls `ComplianceServiceRegulated::adjustInvestorCountsAfterCountryChange` to not process if the countries are the same.

As part of another issue `ComplianceServiceRegulated::adjustInvestorCountsAfterCountryChange` was also changed to correctly decrement previous country so calling this function directly with identical countries would decrement then increment resulting in no net change to investor counts which is correct.

Cyfrin: Verified.

7.2.4 Historical token issuances become subject to new lock-up requirements when users change residency

Description: The `checkHoldUp` function and `getComplianceTransferableTokens` function contain a flaw in how lock periods are applied to token issuances. The system incorrectly applies a single, current lock period to all historical issuances instead of using the lock period that was applicable when each token was originally issued. The vulnerability stems from two key issues:

1. Lock periods are calculated at transfer time based on the investor's current country, not their country at issuance time (this is in `checkHoldUp`):

```
function checkHoldUp(
    address[] memory _services,
    address _from,
    uint256 _value,
    bool _isUSLockPeriod,      // ← Determined at TRANSFER time
    bool _isPlatformWalletFrom
) internal view returns (bool) {
    uint256 lockPeriod;
    if (_isUSLockPeriod) {
        lockPeriod = getUSLockPeriod();     // ← Current US lock period
    } else {
        lockPeriod = getNonUSLockPeriod();  // ← Current Non-US lock period
    }

    // ← Applies same lock period to ALL issuances!
    return complianceService.getComplianceTransferableTokens(_from, block.timestamp,
    ↪  uint64(lockPeriod)) < _value;
}
```

2. The same lock period is applied to all historical issuances for an investor, regardless of when those tokens were issued or what the applicable regulations were at that time (This is in `getComplianceTransferableTokens()`):

```
 function getComplianceTransferableTokens(
        address _who,
        uint256 _time,
        uint64 _lockTime
    ) public view override returns (uint256) {
        ...

        for (uint256 i = 0; i < investorIssuancesCount; i++) {
            uint256 issuanceTimestamp = issuancesTimestamps[investor][i];
             // Uses same _lockTime for ALL issuances <-------------
            if (uint256(_lockTime) > _time || issuanceTimestamp > (_time - uint256(_lockTime))) {
                uint256 tokens =
                ↪  getRebasingProvider().convertSharesToTokens(issuancesValues[investor][i]);
                totalLockedTokens = totalLockedTokens + tokens;
            }
        }

        //there may be more locked tokens than actual tokens, so the minimum between the two
        uint256 transferable = balanceOfInvestor - Math.min(totalLockedTokens, balanceOfInvestor);

        return transferable;
    }
```

Impact: * Investors may be unable to transfer tokens that should legally be unlocked(Temporally freeze of funds), causing liquidity issues.

- Incorrect application of lock periods can lead to violations of securities regulations that require specific lock-up periods based on investor status at issuance time.

Proof of Concept: Consider the following real-world scenario:

1. January 1, 2024: Alice is a German investor (Non-US, 6-month lock period)
2. June 1, 2024: Alice relocates to USA and updates her investor profile ((US, 12-month lock period))
3. December 1, 2024: Alice attempts to transfer tokens

issuance history of Alice:

Issuance 1: January 1, 2024 - 1,000 tokens (issued when Alice was German) Issuance 2: March 1, 2024 - 500 tokens (issued when Alice was German) Issuance 3: August 1, 2024 - 200 tokens (issued when Alice was US resident)

The tokens issued when Alice was in Germany should be unlocked after 6 months, and the tokens that Alice issued in the US should be locked for 12 months. However, since the contract uses lock periods based on the investor's current country, not their country at issuance time, Alice will have all her tokens locked incorrectly.

How it should be: December 1, 2024 transfer check:

- Issuance 1: Jan 1 + 6 months = July 1, 2024 UNLOCKED (issued under German rules)
- Issuance 2: Mar 1 + 6 months = Sep 1, 2024 UNLOCKED (issued under German rules)
- Issuance 3: Aug 1 + 12 months = Aug 1, 2025 LOCKED (issued under US rules)

Total transferable: 1,500 tokens Total locked: 200 tokens

How it is: December 1, 2024 transfer check:

- Issuance 1: Jan 1 + 12 months = Jan 1, 2025 LOCKED
- Issuance 2: Mar 1 + 12 months = Mar 1, 2025 LOCKED
- Issuance 3: Aug 1 + 12 months = Aug 1, 2025 LOCKED

Recommended Mitigation: Consider storing Lock Periods at Issuance Time:

```
// Add new mapping to ComplianceServiceDataStore.sol
mapping(string => mapping(uint256 => uint256)) issuanceLockPeriods;

function createIssuanceInformation(
    string memory _investor,
    uint256 _shares,
    uint256 _issuanceTime
) internal returns (bool) {
    ...

    issuancesValues[_investor][issuancesCount] = _shares;
    issuancesTimestamps[_investor][issuancesCount] = _issuanceTime;
+  issuanceLockPeriods[_investor][issuancesCount] = lockPeriod; // <----------Store lock period
    issuancesCounters[_investor] = issuancesCount + 1;

    return true;
}
```

Then in `getComplianceTransferableTokens`:

```
function getComplianceTransferableTokens(
    address _who,
    uint256 _time,
    uint64 _lockTime // This parameter can be removed
) public view override returns (uint256) {
    ...

    for (uint256 i = 0; i < investorIssuancesCount; i++) {
        uint256 issuanceTimestamp = issuancesTimestamps[investor][i];
+       uint256 applicableLockPeriod = issuanceLockPeriods[investor][i]; // Use stored lock period

-         if (uint256(_lockTime) > _time || issuanceTimestamp > (_time - uint256(_lockTime))) {
+        if (_time < issuanceTimestamp + applicableLockPeriod) {
            uint256 tokens = getRebasingProvider().convertSharesToTokens(issuancesValues[investor][i]);
            totalLockedTokens = totalLockedTokens + tokens;
        }
    }

    ...
}
```

Securitize: Acknowledged.

7.2.5 No way to revert `setInvestorLiquidateOnly`

Description: The `setInvestorLiquidateOnly` function in `InvestorLockManagerBase.sol` contains a logic error that prevents the disabling of liquidate-only mode once it has been enabled. The function includes a require statement that checks if the investor is already in liquidate-only mode and reverts if they are, making it impossible to toggle the state back to false.

```
function setInvestorLiquidateOnly(string memory _investorId, bool _enabled) public
↪   onlyTransferAgentOrAbove returns (bool) {
    require(!investorsLiquidateOnly[_investorId], "Investor is already in liquidate only mode");
    investorsLiquidateOnly[_investorId] = _enabled;
    emit InvestorLiquidateOnlySet(_investorId, _enabled);
    return true;
}
```

Impact: Once an investor is set to liquidate-only mode, there is no way to disable this state

Recommended Mitigation: Remove the require statement to allow toggling of the liquidate-only state:

```
function setInvestorLiquidateOnly(string memory _investorId, bool _enabled) public
↪   onlyTransferAgentOrAbove returns (bool) {
-   require(!investorsLiquidateOnly[_investorId], "Investor is already in liquidate only mode");
    investorsLiquidateOnly[_investorId] = _enabled;
    emit InvestorLiquidateOnlySet(_investorId, _enabled);
    return true;
}
```

Securitize: Fixed in commit 74a6675 by reverting if the current state is the same as the input state; this allows state to be toggled on/off.

Cyfrin: Verified.

7.2.6 `euRetailInvestorsCount` is susceptible to underflow when transferring tokens between retail and qualified investors of the same country on the EU region

Description: The root cause of this problem is that a qualified EU investor with tokens on its balance would not increment the `euRetailInvestorsCount` when the investor is downgraded to a retail investor. This puts the system in an inconsistent state. The qualified investor could've received tokens while he was qualified, and after he is downgraded to retail, he transfers all of his balance to another investor. This will cause the `euRetailInvestorsCount` to be decremented because a retail investor is no longer a holder.

- This problem occurs because the qualified investor was downgraded to retail while he had tokens on his balance, but the `euRetailInvestorsCount` was not incremented.

```
    function adjustInvestorsCountsByCountry(
        ...
    ) internal {
        ...
//@audit-info => euRetailInvestorsCount are not modified for Qualified EU Investors
        } else if (countryCompliance == EU && !getRegistryService().isQualifiedInvestor(_id)) {
            if(_increase == CommonUtils.IncDec.Increase) {
                euRetailInvestorsCount[_country]++;
            }
            else {
                euRetailInvestorsCount[_country]--;
            }
        }
        ...
    }
```

As is demonstrated on the PoC, the `euRetailInvestorsCount` for France is messed up in a scenario as shown below:

1. A French retail EU investor transfers some of its balance (partial transfer) to a qualified (non-retail) French investor
2. The qualified French investor becomes retail and transfers all of its balance back to the French retail investor1
    - Here, the `euRetailInvestorsCount` will be messed up
3. The retail French investor1 attempts to transfer all of its balance out and gets an underflow because the retail counter for France is already 0.

Impact: In addition to the risk of reaching the status of underflow, the `euRetailInvestorsCount` will be off from the real number of investors that should be tracked.

Proof of Concept: Add the next PoC to `dstoken-regulated.test.ts` test file

```
    it.only('underflow on euRetailInvestorsCount PoC', async function () {
      const [investor, investor2, usInvestor] = await hre.ethers.getSigners();
      const { dsToken, registryService, complianceService, complianceConfigurationService } = await
      ↪  loadFixture(deployDSTokenRegulated);
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.FRANCE,
      ↪  INVESTORS.Compliance.EU);
      await complianceConfigurationService.setEURetailInvestorsLimit(10);
      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, investor, registryService);
      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, investor2, registryService);
      await registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID, usInvestor, registryService);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.FRANCE);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, INVESTORS.Country.FRANCE);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID, INVESTORS.Country.USA);

      //@audit-info => Set investor2 as a qualified investor!
      await registryService.setAttribute(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, 4, 1, 0, 'abcde');

      await dsToken.issueTokens(investor, 500);
      const valueToTransfer = 100;

      //@audit-info => a retail EU investor transfers to a qualified (non-retail) EU investor
      const dsTokenFromInvestor = await dsToken.connect(investor);
      await dsTokenFromInvestor.transfer(investor2, valueToTransfer);
      expect(await complianceService.getEURetailInvestorsCount(INVESTORS.Country.FRANCE)).equal(1);
```

```
        //@audit-info => Set investor2 as retail investor - non-qualified!
        await registryService.setAttribute(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, 4, 0, 0, 'abcde');
        const dsTokenFromInvestor2 = await dsToken.connect(investor2);
        //@audit-info => The old qualified investor, currently retail, transfers all its balance to
        ↪   another retail EU investor
        await dsTokenFromInvestor2.transfer(investor, valueToTransfer);
        expect(await complianceService.getEURetailInvestorsCount(INVESTORS.Country.FRANCE)).equal(0);

        //@audit-info => investor1 tries to transfer all of its balance to an investor on another country
        ↪   and reverts bcs of underflow on euRetailInvestorsCount
        await expect(dsTokenFromInvestor.transfer(usInvestor, 500)).to.be.reverted;
    });
```

Recommended Mitigation: Consider incrementing the `euRetailInvestorsCount` when converting a qualified EU investor to a retail EU investor, and such an investor has an investorBalance.

Securitize: Acknowledged; it is not realistic to account for changes between qualified and not qualified statuses. Implementing this logic would add complexity to investor and attribute updates and in any case can be solved operationally.

7.2.7 `ComplianceServiceRegulated` applies double locking to investor transferable tokens leading to DoS of transfers

Description: `ComplianceServiceRegulated::getComplianceTransferableTokens()` enforces a double locking on the investors' transferable tokens, causing more tokens to be considered locked up than what they really are, leading to investors getting DoS when transferring tokens when they should be allowed to.

The root problem is that the investorBalance used on the `ComplianceServiceRegulated::getComplianceTransferableTokens` is actually the transferable tokens rather than the actual investorBalance. This causes the lockups enforced on the `InvestorLockManager` to be accumulated on top of the lockups enforced on the Compliance contract.

For example, Investor1 owns WalletA and WalletB.

- WalletA has 500 transferable tokens.
- WalletB has 1000 locked tokens. In aggregation, investor1 has 1500 tokens, of which 1000 are locked and 500 are transferable.

Here is what happens on the `getComplianceTransferableTokens()`

1. balanceOfInvestor would be set as 500 because `Investor.getTransferableTokens()` determines investor1 has 500 unlocked tokens (1500 total minus 1000 locked).
2. `getComplianceTransferableTokens()` iterates over the issuancesCounters and calculates 1000 `totalLockedTokens`.
3. `getComplianceTransferableTokens()` calculates `transferable` as zero because `balanceOfInvestor (500)` - `min(totalLocked, balanceOfInvestor` => `500 - min(1000,500)` => `500 - 500 =>` 0.

This results in the investor getting DoSed from transferring the 500 transferable tokens because of the double counting applied by the Compliance contract, which causes the investor's locked balance to outweigh the investor's transferable tokens.

```
    function getComplianceTransferableTokens(
        address _who,
        uint256 _time,
        uint64 _lockTime
    ) public view override returns (uint256) {
        ...

//@audit-info => amount of transferable tokens that are not locked anymore (500)
        uint256 balanceOfInvestor = getLockManager().getTransferableTokens(_who, _time);
        ...

        for (uint256 i = 0; i < investorIssuancesCount; i++) {
            uint256 issuanceTimestamp = issuancesTimestamps[investor][i];

            if (uint256(_lockTime) > _time || issuanceTimestamp > (_time - uint256(_lockTime))) {
                uint256 tokens =
                ↪   getRebasingProvider().convertSharesToTokens(issuancesValues[investor][i]);
//@audit-info => amount of locked tokens (1000)
                totalLockedTokens = totalLockedTokens + tokens;
            }
        }

//@audit-issue => transferable is calculated as 0 even though the investor has 500 transferable tokens
        uint256 transferable = balanceOfInvestor - Math.min(totalLockedTokens, balanceOfInvestor);

        return transferable;
    }
```

Note: This issue also occurs if the LockManager is an instance of the `LockManager` contract. In this contract, the locks are applied at the wallet level, but even then, the double locking counting also messes up the calculation of the real transferable tokens. The developers have stated that `LockManager` contract will be deprecated; therefore, the recommended mitigation to fix this issue is to focus solely on the `InvestorLockManager` contract

Impact: Investors get fewer transferable tokens than they have, potentially leading to complete DoS of transfers.

Proof of Concept: Add the next PoC to `dstoken-regulated.test.ts`

```
    it.only('Double Locking Count leads to DoS even when using InvestorLockManager', async function () {
      const [owner, wallet1, wallet2, walletOtherInvestor] = await hre.ethers.getSigners();
      const { dsToken, registryService, complianceConfigurationService, lockManager, complianceService
      ↪  } = await loadFixture(deployDSTokenRegulated);

      const usLockPeriod = 1000;
      await complianceConfigurationService.setUSLockPeriod(usLockPeriod);
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.USA,
      ↪  INVESTORS.Compliance.US);

      await registryService.registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, '');
      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.USA);
      await registryService.addWallet(wallet1, INVESTORS.INVESTOR_ID.INVESTOR_ID_1);//wallet1 =>
      ↪  INVESTOR_ID_1
      await registryService.addWallet(wallet2, INVESTORS.INVESTOR_ID.INVESTOR_ID_1);//wallet2 ->
      ↪  INVESTOR_ID_1

      await registryService.registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, '');
      await registryService.addWallet(walletOtherInvestor,
      ↪  INVESTORS.INVESTOR_ID.INVESTOR_ID_2);//walletOtherInvestor -> INVESTOR_ID_2

      const currentTime = await time.latest();

      await dsToken.issueTokensCustom(wallet1, 500, currentTime, 500, 'TEST', currentTime + 1_000);
      ↪  //lock for INVESTOR_ID_1

      //@audit-info => forward time so 500 issued tokens to Wallet1 are transferable
      await time.increaseTo(currentTime + 1000);

      //@audit-info => Any of the investor's wallet has 500 transferable tokens bcs the Investor as a
      ↪  whole has 500 transferable tokens!
      expect(await lockManager.getTransferableTokens(wallet1, await time.latest())).equal(500);
      expect(await lockManager.getTransferableTokens(wallet2, await time.latest())).equal(500);
      //@audit-info => complianceTransferableTokens for any of the wallets is 500 before the new
      ↪  issuance
      expect(await complianceService.getComplianceTransferableTokens(wallet1, time.latest(),
      ↪  usLockPeriod)).equals(500);
      expect(await complianceService.getComplianceTransferableTokens(wallet2, time.latest(),
      ↪  usLockPeriod)).equals(500);

      //@audit-info => issue 1000 locked tokens to Wallet2
      await dsToken.issueTokensCustom(wallet2, 1000, await time.latest(), 1000, 'TEST', await
      ↪  time.latest() + 1_000); //lock for INVESTOR_ID_1

      //@audit-info => Any of the investor's wallet has 500 transferable tokens bcs the Investor as a
      ↪  whole has 500 transferable tokens!
      expect(await lockManager.getTransferableTokens(wallet1, await time.latest())).equal(500);
      expect(await lockManager.getTransferableTokens(wallet2, await time.latest())).equal(500);

//@audit-issue => Both wallets have 0 transferable tokens because of the double locking count
      expect(await complianceService.getComplianceTransferableTokens(wallet1, time.latest(),
      ↪  usLockPeriod)).equals(0);
      expect(await complianceService.getComplianceTransferableTokens(wallet2, time.latest(),
      ↪  usLockPeriod)).equals(0);

//@audit-issue => Investor as a whole (can't transfer from any of his wallets) is DoS from transferring
↪  even though it has transferable tokens because of the double locking count
      const dsTokenWallet1 = await dsToken.connect(wallet1);
      await expect(dsTokenWallet1.transfer(walletOtherInvestor, 500)).revertedWith('Under lock-up');

      const dsTokenWallet2 = await dsToken.connect(wallet2);
      await expect(dsTokenWallet2.transfer(walletOtherInvestor, 500)).revertedWith('Under lock-up');
    });
```

Recommended Mitigation: Instead of querying the `InvestorLockManager::getTransferableTokens()` to set the base `balanceOfInvestor`, query the `token::balanceOfInvestor()`. This will remove the double locking counting and use the investor's actual balance as the base to determine the transferable tokens in case of compliance locks.

- If there are no compliance locks but a portion of the balance is locked on the LockManager, that locking count is handled on the `InvestorLockManager` itself and applied at the investor level.

```
function getComplianceTransferableTokens(
        address _who,
        uint256 _time,
        uint64 _lockTime
    ) public view override returns (uint256) {
        require(_time != 0, "Time must be greater than zero");
        string memory investor = getRegistryService().getInvestor(_who);

-       uint256 balanceOfInvestor = getLockManager().getTransferableTokens(_who, _time);
+      uint256 balanceOfInvestor = getToken().balanceOfInvestor(investor);

        ...
}
```

Securitize: Acknowledged; this is working as intended Investor: Level Locks apply over the current amount of transferrable tokens for the investor, which is the amount of the balance minus the total amount of tokens issued that are still under a lockup period.

In commit 7bbe0b1 we've updated the comments and `LockManager` and `ComplianceServiceNotRegulated` have been removed as they were deprecated.

7.2.8 Token value locks done as part of issuance restrict subsequently transferred unlocked tokens after negative rebasing

Description: When a token issuance occurs, as part of that token issuance two types of locks can be enacted:

- share value based lock via:

```
TokenIssuer::issueTokens
-- DSToken::issueTokensWithMultipleLocks
---- TokenLibrary::issueTokensCustom
------ ComplianceService::validateIssuance
-------- ComplianceServiceRegulated::recordIssuance -> createIssuanceInformation
```

- token value based lock via:

```
TokenIssuer::issueTokens
-- DSToken::issueTokensWithMultipleLocks
---- TokenLibrary::issueTokensCustom
------ InvestorLockManager::addManualLockRecord -> createLock -> createLockForInvestor
```

Impact: In a scenario where:

- Investor1 has a token issuance for 1000 tokens with both shared-base and token-value based locks for that issuance
- Negative rebasing occurs
- Investor2 transfers 1000 unlocked tokens to Investor1

The token value lock from Investor1's original token issuance will restrict the transfer of the unlocked tokens which Investor2 subsequently transferred to Investor1.

Proof of Concept: Add PoC to `test/token-issuer.test.ts`:

```
it('Token value locks done as part of issuance restrict subsequently transferred tokens after negative
↪  rebasing', async function() {
  const [ owner, investor1, investor2 ] = await hre.ethers.getSigners();
  const { dsToken, tokenIssuer, lockManager, rebasingProvider, registryService } = await
  ↪  loadFixture(deployDSTokenRegulated);

  // Register investors
  await registryService.registerInvestor('INVESTOR1', '');
  await registryService.setCountry('INVESTOR1', 'US');
  await registryService.addWallet(investor1.address, 'INVESTOR1');

  await registryService.registerInvestor('INVESTOR2', '');
  await registryService.setCountry('INVESTOR2', 'US');
  await registryService.addWallet(investor2.address, 'INVESTOR2');

  const currentTime = await time.latest();
  const lockRelease = currentTime + 100000;

  // Issue 1000 tokens to Investor1 with 1000 manually locked
  await tokenIssuer.issueTokens(
    'INVESTOR1',
    investor1.address,
    [ 1000, 1 ],
    '',
    [ 1000 ], // Lock ALL tokens
    [ lockRelease ],
    'INVESTOR1',
    'US',
    [0, 0, 0],
    [0, 0, 0]
  );
```

```
    // Issue 2000 tokens to Investor2 (no locks)
    await tokenIssuer.issueTokens(
      'INVESTOR2',
      investor2.address,
      [ 2000, 1 ],
      '',
      [],
      [],
      'INVESTOR2',
      'US',
      [0, 0, 0],
      [0, 0, 0]
    );

    // Verify initial state
    expect(await dsToken.balanceOf(investor1.address)).to.equal(1000);
    expect(await lockManager.getTransferableTokens(investor1.address, currentTime)).to.equal(0);

    // Get current multiplier and halve it
    const currentMultiplier = await rebasingProvider.multiplier();
    const halfMultiplier = currentMultiplier / BigInt(2);
    await rebasingProvider.setMultiplier(halfMultiplier);

    // After rebasing, Investor1 has 500 tokens but 1000 still locked
    expect(await dsToken.balanceOf(investor1.address)).to.equal(500);
    const transferableAfterRebasing = await lockManager.getTransferableTokens(investor1.address,
    ↪  currentTime);
    expect(transferableAfterRebasing).to.equal(0); // Over-locked

    // Investor2 transfers 1000 tokens to Investor1
    await dsToken.connect(investor2).transfer(investor1.address, 1000);

    // Final state
    const finalBalance = await dsToken.balanceOf(investor1.address);
    const finalTransferable = await lockManager.getTransferableTokens(investor1.address, currentTime);

    expect(finalBalance).to.equal(1500); // 500 from rebasing + 1000 transfer

    // Investor1 has 500 tokens from their own issuance and 1000 tokens
    // they received as a transfer with no associated lock
    // But the token value lock placed as part of their original issuance
    // affects subsequent transfers which had no locks attached
    //
    // Unlocked tokens received via transfer from Investor2 are partially
    // locked by Investor1's token-value based lock from Investor1's original
    // token issuance
    expect(finalTransferable).to.equal(500);
  });
```

Recommended Mitigation: Convert the lock system in `InvestorLockManager` and `LockManager` to store all lock amounts in shares rather than tokens. Alternatively the issue can simply be acknowledged to accept the current behavior.

Securitize: Acknowledged; locks are always accounted for in tokens, and yes, for a negative rebasing scenario that could happen. But could also be managed by operational procedures.

7.2.9 Investors transferring all their balances among their wallets or self-transferring on the same wallet causes to incorrectly decrement the investor counters causing DoS for other investors transfers

Description: `ComplianceServiceRegulated::recordTransfer()` is in charge of adjusting the investor counters when the receiver's investor is new, or the sender's investor is transferring all of its balance, but there are edge cases that cause the investor counters to be incorrectly decremented.

1. The first edge case is when an existing investor transfers all their balance from one wallet to another.
2. An investor executing self-transfers from the same wallet.

The root cause of the problem is found in the `ComplianceServiceRegulated::recordTransfer()`, the function does not check if the sender and receiver investor are the same, and it straight checks if the sender's investor is transferring all of his balance, regardless of whether the receiver is the same investor.

```
    function compareInvestorBalance(
        address _who,
        uint256 _value,
        uint256 _compareTo
    ) internal view returns (bool) {
       //@audit => true when the sender is transferring all of its balance
        return (_value != 0 && getToken().balanceOfInvestor(getRegistryService().getInvestor(_who)) ==
        ↪  _compareTo);
    }

    function recordTransfer(
        address _from,
        address _to,
        uint256 _value
    ) internal override returns (bool) {
//@audit-issue => Not checking if sender is the same as the receiver

        if (compareInvestorBalance(_to, _value, 0)) {
            adjustTransferCounts(_to, CommonUtils.IncDec.Increase);
        }

        if (compareInvestorBalance(_from, _value, _value)) {
//@audit=> When from's investor transfers all of its balance, investor counters are decremented.
            adjustTotalInvestorsCounts(_from, CommonUtils.IncDec.Decrease);
        }

        ...
    }
```

Impact:

- Investor counters will be decremented when they should not be, leading to DoS transfers for other investors.
- An additional impact is that investor limits can be bypassed because the investor counters won't correctly track the real number of investors in the system.

Proof of Concept: Add the PoCs to `dstoken-regulated.test.ts`. First PoC demonstrates the problem by transferring among the wallets of the same investor:

```
    it.only('Mess up counters via transfers among wallets owned by the same investor', async function
    ↪  () {
      const [owner, wallet1Investor1, wallet2Investor1, walletInvestor2, walletInvestor3] = await
      ↪  hre.ethers.getSigners();
      const { dsToken, registryService, complianceConfigurationService, lockManager, complianceService
      ↪  } = await loadFixture(deployDSTokenRegulated);

      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.USA,
      ↪  INVESTORS.Compliance.US);
      await complianceConfigurationService.setUSInvestorsLimit(10);

      await registryService.registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID, '');
      await registryService.registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2, '');
      await registryService.registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_3, '');
      await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID, INVESTORS.Country.USA);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2, INVESTORS.Country.USA);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_3, INVESTORS.Country.USA);

      await registryService.addWallet(wallet1Investor1,
      ↪  INVESTORS.INVESTOR_ID.US_INVESTOR_ID);//wallet1Investor1 => US_INVESTOR_ID
      await registryService.addWallet(wallet2Investor1,
      ↪  INVESTORS.INVESTOR_ID.US_INVESTOR_ID);//wallet2Investor1 -> US_INVESTOR_ID

      await registryService.addWallet(walletInvestor2,
      ↪  INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2);//walletInvestor2 -> US_INVESTOR_ID_2
      await registryService.addWallet(walletInvestor3,
      ↪  INVESTORS.INVESTOR_ID.US_INVESTOR_ID_3);//walletInvestor3 -> US_INVESTOR_ID_3

      const currentTime = await time.latest();

      //@audit-info => Issue unlocked tokens to investors

      await dsToken.issueTokensCustom(wallet1Investor1, 10_000, currentTime, 0, 'TEST', 0);
      await dsToken.issueTokensCustom(walletInvestor2, 10_000, currentTime, 0, 'TEST', 0);
      await dsToken.issueTokensCustom(walletInvestor3, 10_000, currentTime, 0, 'TEST', 0);

      expect(await complianceService.getUSInvestorsCount()).equal(3);
      expect(await complianceService.getTotalInvestorsCount()).equal(3);

      //@audit-info => investor1 transfers all the balance among his wallets
      const dsTokenWallet1Investor1 = await dsToken.connect(wallet1Investor1);
      const dsTokenWallet2Investor1 = await dsToken.connect(wallet2Investor1);

      await dsTokenWallet1Investor1.transfer(wallet2Investor1, 10_000);
      await dsTokenWallet2Investor1.transfer(wallet1Investor1, 10_000);
      await dsTokenWallet1Investor1.transfer(wallet2Investor1, 10_000);

      //@audit-issue => Investor counters have been brought down to 0 while the 3 investors still have
      ↪  balances!
      expect(await complianceService.getUSInvestorsCount()).equal(0);
      expect(await complianceService.getTotalInvestorsCount()).equal(0);

      expect(await dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID)).equal(10_000);
      expect(await dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2)).equal(10_000);
      expect(await dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_3)).equal(10_000);

      //@audit-info => Will revert because investor counter has been brought down to 0
      const dsTokenWalletInvestor2 = await dsToken.connect(walletInvestor2);
      await expect(dsTokenWalletInvestor2.transfer(walletInvestor3, 10_000)).to.be.revertedWith('Not
      ↪  enough investors');
    });
```

The second PoC demonstrates the problem by self-transferring on the same wallet:

```
  it.only('self transfer from the same wallet messes up investor counters', async function () {
    const [owner, wallet1Investor1, wallet2Investor1, walletInvestor2, walletInvestor3] = await
    ↪  hre.ethers.getSigners();
    const { dsToken, registryService, complianceConfigurationService, lockManager, complianceService
    ↪  } = await loadFixture(deployDSTokenRegulated);

    await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.USA,
    ↪  INVESTORS.Compliance.US);
    await complianceConfigurationService.setUSInvestorsLimit(10);

    await registryService.registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID, '');
    await registryService.registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2, '');
    await registryService.registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_3, '');

    await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID, INVESTORS.Country.USA);
    await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2, INVESTORS.Country.USA);
    await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_3, INVESTORS.Country.USA);

    await registryService.addWallet(wallet1Investor1,
    ↪   INVESTORS.INVESTOR_ID.US_INVESTOR_ID);//wallet1Investor1 => US_INVESTOR_ID

    await registryService.addWallet(walletInvestor2,
    ↪   INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2);//walletInvestor2 -> US_INVESTOR_ID_2
    await registryService.addWallet(walletInvestor3,
    ↪   INVESTORS.INVESTOR_ID.US_INVESTOR_ID_3);//walletInvestor3 -> US_INVESTOR_ID_3

    const currentTime = await time.latest();

    //@audit-info => Issue unlocked tokens to investors

    await dsToken.issueTokensCustom(wallet1Investor1, 10_000, currentTime, 0, 'TEST', 0);
    await dsToken.issueTokensCustom(walletInvestor2, 10_000, currentTime, 0, 'TEST', 0);
    await dsToken.issueTokensCustom(walletInvestor3, 10_000, currentTime, 0, 'TEST', 0);

    expect(await complianceService.getUSInvestorsCount()).equal(3);
    expect(await complianceService.getTotalInvestorsCount()).equal(3);

    //@audit-info => investor1 transfers all the balance among his wallets
    const dsTokenWallet1Investor1 = await dsToken.connect(wallet1Investor1);

    await dsTokenWallet1Investor1.transfer(wallet1Investor1, 10_000);
    await dsTokenWallet1Investor1.transfer(wallet1Investor1, 10_000);
    await dsTokenWallet1Investor1.transfer(wallet1Investor1, 10_000);

    //@audit-issue => Investor counters have been brought down to 0 while the 3 investors still have
    ↪   balances!
    expect(await complianceService.getUSInvestorsCount()).equal(0);
    expect(await complianceService.getTotalInvestorsCount()).equal(0);

    expect(await dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID)).equal(10_000);
    expect(await dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2)).equal(10_000);
    expect(await dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_3)).equal(10_000);

    //@audit-info => Will revert because investor counter has been brought down to 0
    const dsTokenWalletInvestor2 = await dsToken.connect(walletInvestor2);
    await expect(dsTokenWalletInvestor2.transfer(walletInvestor3, 10_000)).to.be.revertedWith('Not
    ↪   enough investors');
  });
```

Recommended Mitigation: Consider skipping the check and decrement of investor counters when the investor of the `from` and `receiver` wallets is the same:

```
    function recordTransfer(
        address _from,
        address _to,
        uint256 _value
    ) internal override returns (bool) {
        if (compareInvestorBalance(_to, _value, 0)) {
            adjustTransferCounts(_to, CommonUtils.IncDec.Increase);
        }

-       if (compareInvestorBalance(_from, _value, _value)) {
-           adjustTotalInvestorsCounts(_from, CommonUtils.IncDec.Decrease);
-       }

+       string memory investorFrom = getRegistryService().getInvestor(_from);
+       string memory investorTo = getRegistryService().getInvestor(_to);

+       if(!CommonUtils.isEqualString(investorFrom, investorTo)) {
+           if (compareInvestorBalance(_from, _value, _value)) {
+               adjustTotalInvestorsCounts(_from, CommonUtils.IncDec.Decrease);
+           }
+       }

        cleanupInvestorIssuances(_from);
        cleanupInvestorIssuances(_to);
        return true;
    }
```

A more gas optimized version of this fix would be to refactor the functions `compareInvestorBalance` and `cleanupInvestorIssuances` to take the investor id as input as opposed to receiving the wallet address and internally getting the investor id on each function.

Securitize: Fixed in commit 6b94242.

Cyfrin: Verified.

7.3 Low Risk

7.3.1 Protocol will leak value to users due to rounding in `RebasingLibrary`

Description: Protocols should always round against users in favor of the protocol. `RebasingLibrary` uses "rounding to nearest" method which leaks value to users:

```
// In convertTokensToShares
return (_tokens * DECIMALS_FACTOR + _rebasingMultiplier / 2) / _rebasingMultiplier;
                                  // ^^^^^^^^^^^^^^^^^^^^
                                  // This rounds to nearest

// In convertSharesToTokens
return (_shares * _rebasingMultiplier + DECIMALS_FACTOR / 2) / DECIMALS_FACTOR;
                                      // ^^^^^^^^^^^^^^^^^^^^
                                      // This also rounds to nearest
```

Impact: The rounding helps users in both directions:

- When depositing: Users might get 1 extra share
- When withdrawing: Users might get 1 extra token

Over thousands of transactions, these wei-level losses accumulate.

Proof of Concept: First add Foundry integration. Then add new PoC contract to `test/RebasingRoundingTest.t.sol`:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.22;

import "forge-std/Test.sol";
import "../contracts/rebasing/RebasingLibrary.sol";

contract RebasingRoundingTest is Test {
    using RebasingLibrary for *;

    function testRoundingInconsistency() public {
        // Setup: multiplier = 1.5e18 (1.5x rebasing)
        uint256 multiplier = 1.5e18;
        uint8 decimals = 18;

        // Test 1: Convert 1 token to shares and back
        uint256 initialTokens = 1e18;

        // Convert tokens to shares
        uint256 shares = RebasingLibrary.convertTokensToShares(
            initialTokens,
            multiplier,
            decimals
        );

        // Convert shares back to tokens
        uint256 finalTokens = RebasingLibrary.convertSharesToTokens(
            shares,
            multiplier,
            decimals
        );

        console.log("Initial tokens:", initialTokens);
        console.log("Shares received:", shares);
        console.log("Final tokens:", finalTokens);

        // This should fail if there's inconsistency
        assertEq(initialTokens, finalTokens, "Rounding inconsistency detected");
    }
    function testAccumulatedRoundingErrors() public {
        uint256 multiplier = 1.7e18; // Non-round multiplier
        uint8 decimals = 18;

        uint256 totalSharesIssued;
        uint256 totalTokensIn;

        // Simulate 1000 small deposits
        for(uint i = 0; i < 1000; i++) {
            // Each user deposits a small odd amount
            uint256 tokens = 1e15 + i; // 0.001 tokens + i wei

            uint256 shares = RebasingLibrary.convertTokensToShares(
                tokens,
                multiplier,
                decimals
            );

            totalTokensIn += tokens;
            totalSharesIssued += shares;
        }

        // Now convert total shares back to tokens
        uint256 totalTokensOut = RebasingLibrary.convertSharesToTokens(
            totalSharesIssued,
            multiplier,
            decimals
        );

        console.log("Total tokens in:", totalTokensIn);
        console.log("Total tokens out:", totalTokensOut);
        console.log("Difference:", totalTokensOut > totalTokensIn ?
            totalTokensOut - totalTokensIn : totalTokensIn - totalTokensOut);

        // Check if protocol lost tokens
        if(totalTokensOut > totalTokensIn) {
            console.log("Protocol LOST tokens due to rounding!");
        }
    }
}
```

Run with: `forge test --match-contract RebasingRoundingTest -vv`

Recommended Mitigation: Rounding should always favor the protocol; normally rounding in favor is rounding down when for example issuing tokens to users but rounding up when users are charged fees etc. But consider:

- `convertTokensToShares` is used during issuance (`TokenLibrary::issueTokensCustom`) where rounding down is in favor of the protocol to give the user slightly less shares
- `convertTokensToShares` is also used during burning (`TokenLibrary::burn`) where rounding down is actually in favor of the user to burn slightly less of their shares

So this is tricky; what newer protocols are doing now:

- in functions such as `convertTokensToShares` that are used throughout the code, they add an input rounding parameter
- all callers to `convertTokensToShares` must specify an explicit rounding direction via the additional input parameter
- all callers to `convertTokensToShares` should have a comment above the call explaining the logic for why the specified rounding direction is correct

This forces developers to carefully consider the rounding direction in the context of every call which is a good practice to follow and can help eliminate rounding bugs.

I'd do the same for `convertSharesToTokens`; make every caller specify the rounding direction and put a comment before every call explaining why the rounding direction is correct in that context.

Securitize: Acknowledged; the current implementation doesn't appear exploitable. In practice due to the decimals it very often won't even occur, and trying to fix it creates other potential problems. We'll leave it as is for now but have added some explanatory comments and additional tests in commit 8b74550.

7.3.2 Multiplication could overflow in `RebasingLibrary` for tokens with greater than 18 decimals

Description: `RebasingLibrary` contains special handling for tokens with greater than 18 decimals:

```
// convertTokensToShares
        } else {
            uint256 scale = 10**(_tokenDecimals - 18);
            return (_tokens * DECIMALS_FACTOR + (_rebasingMultiplier * scale) / 2) /
            ↪  (_rebasingMultiplier * scale);
        }

// convertSharesToTokens
        } else {
            uint256 scale = 10**(_tokenDecimals - 18);
            return (_shares * _rebasingMultiplier * scale + DECIMALS_FACTOR / 2) / DECIMALS_FACTOR;
        }
```

Impact: When using tokens with high decimal values, the multiplication here could overflow causing denial of service.

Recommended Mitigation: Use OpenZeppelin's Math::mulDiv, fixed code also incorporates suggested fix for L-1:

```
pragma solidity 0.8.22;

import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

library RebasingLibrary {
    uint256 private constant DECIMALS_FACTOR = 1e18;

    function convertTokensToShares(
        uint256 _tokens,
        uint256 _rebasingMultiplier,
        uint8 _tokenDecimals
    ) internal pure returns (uint256 shares) {
        require(_rebasingMultiplier > 0, "Invalid rebasing multiplier");

        if (_tokenDecimals == 18) {
            return Math.mulDiv(_tokens, DECIMALS_FACTOR, _rebasingMultiplier);
        } else if (_tokenDecimals < 18) {
            uint256 scale = 10**(18 - _tokenDecimals);
            // tokens * scale * DECIMALS_FACTOR / multiplier
            return Math.mulDiv(_tokens * scale, DECIMALS_FACTOR, _rebasingMultiplier);
        } else {
            uint256 scale = 10**(_tokenDecimals - 18);
            // tokens * DECIMALS_FACTOR / (multiplier * scale)
            return Math.mulDiv(_tokens, DECIMALS_FACTOR, _rebasingMultiplier * scale);
        }
    }

    function convertSharesToTokens(
        uint256 _shares,
        uint256 _rebasingMultiplier,
        uint8 _tokenDecimals
    ) internal pure returns (uint256 tokens) {
        require(_rebasingMultiplier > 0, "Invalid rebasing multiplier");

        if (_tokenDecimals == 18) {
            return Math.mulDiv(_shares, _rebasingMultiplier, DECIMALS_FACTOR);
        } else if (_tokenDecimals < 18) {
            uint256 scale = 10**(18 - _tokenDecimals);
            // (shares * multiplier / DECIMALS_FACTOR) / scale
            return Math.mulDiv(_shares, _rebasingMultiplier, DECIMALS_FACTOR * scale);
        } else {
            uint256 scale = 10**(_tokenDecimals - 18);
            // shares * multiplier * scale / DECIMALS_FACTOR
            return Math.mulDiv(_shares * scale, _rebasingMultiplier, DECIMALS_FACTOR);
        }
    }
}
```

Securitize: Fixed in commit 9b81e76 by reverting for tokens with greater than 18 decimals.

Cyfrin: Verified.

7.3.3 Asymmetry enforcement between `TokenIssuer::registerInvestor`, `WalletRegistrar::registerWallet` and `SecuritizeSwap::_registerNewInvestor`

Description: In `TokenIssuer::registerInvestor`, if the user isn't already an investor they get registered and must have 3 specific attributes set:

```
if (!getRegistryService().isInvestor(_id)) {
    getRegistryService().registerInvestor(_id, _collisionHash);
    getRegistryService().setCountry(_id, _country);

    if (_attributeValues.length > 0) {
        require(_attributeValues.length == 3, "Wrong length of parameters");
        getRegistryService().setAttribute(_id, KYC_APPROVED, _attributeValues[0],
        ↪  _attributeExpirations[0], "");
        getRegistryService().setAttribute(_id, ACCREDITED, _attributeValues[1],
        ↪  _attributeExpirations[1], "");
        getRegistryService().setAttribute(_id, QUALIFIED, _attributeValues[2],
        ↪  _attributeExpirations[2], "");
    }
```

But in `WalletRegistrar::registerWallet` and `SecuritizeSwap::_registerNewInvestor` if the user isn't already an investor, they get registered but the same attribute logic is not there. Instead it is more generic appearing to over-write anything that exists and not enforcing existence of KYC_APPROVED, ACCREDITED or QUALIFIED attributes:

```
if (!registryService.isInvestor(_id)) {
    registryService.registerInvestor(_id, _collisionHash);
    registryService.setCountry(_id, _country);
}

for (uint256 i = 0; i < _wallets.length; i++) {
    if (registryService.isWallet(_wallets[i])) {
        require(CommonUtils.isEqualString(registryService.getInvestor(_wallets[i]), _id), "Wallet
        ↪  belongs to a different investor");
    } else {
        registryService.addWallet(_wallets[i], _id);
    }
}

for (uint256 i = 0; i < _attributeIds.length; i++) {
    registryService.setAttribute(_id, _attributeIds[i], _attributeValues[i], _attributeExpirations[i],
    ↪  "");
}
```

Impact: Going through `WalletRegistrar::registerWallet` or `SecuritizeSwap::_registerNewInvestor` an investor can be registered without the required attributes.

Recommended Mitigation: Harmonize the investor registration process to remove duplicated code and enforce the same requirements.

Securitize: Fixed in commit 72e54d2.

Cyfrin: Verified.

7.3.4 Protocol will leak value to users due to rounding in `SecuritizeSwap::buy`

Description: Protocols should always round against users in favor of the protocol. The `buy` function in the `SecuritizeSwap.sol` is rounding the stable coin need it in favor of the user:

```
 function buy(uint256 _dsTokenAmount, uint256 _maxStableCoinAmount) public override whenNotPaused {
        ...
        uint256 stableCoinAmount = calculateStableCoinAmount(_dsTokenAmount); <----
        dsToken.issueTokensCustom(msg.sender, _dsTokenAmount, block.timestamp, 0, "", 0);

        emit Buy(msg.sender, _dsTokenAmount, stableCoinAmount, navProvider.rate());
    }

  function calculateStableCoinAmount(uint256 _dsTokenAmount) public view returns (uint256) {
        return _dsTokenAmount * navProvider.rate() / (10 ** ERC20(address(dsToken)).decimals()); //
        ↪  rounding down
    }
```

With this rounding down, in theory a user could be minting free DSTokens. However, since the DSToken has only 2 decimals, this is not possible.

Impact: The rounding helps users in both directions:

- When depositing: Users might get 1 extra share
- When withdrawing: Users might get 1 extra token

Over thousands of transactions, these wei-level losses accumulate.

Recommended Mitigation: Rounding should always favor the protocol; use `Math::mulDiv` which supports specifying rounding direction.

Securitize: `SecuritizeSwap` was deleted as it is obsolete.

Cyfrin: Verified.

7.3.5 Missing `checkWalletsForList` in `issueTokensWithNoCompliance`

Description: The `issueTokensWithNoCompliance` function in `DSToken.sol` fails to call `checkWalletsForList(address(0), _to)` after token issuance, unlike all other issuance functions. This prevents wallets from being added to the contract's internal enumeration list used for administrative operations.

Impact: * `walletCount()` returns incorrect values and `getWalletAt()` cannot retrieve wallets that received tokens via no-compliance issuance

- Failed Bulk Operations: Any administrative function relying on wallet enumeration will miss these token holders

Recommended Mitigation: Add the `checkWalletsForList` in `issueTokensWithNoCompliance`

Securitize: `DSToken:: issueTokensWithNoCompliance` was removed.

Cyfrin: Verified.

7.3.6 Transferring all the investor balance from a non-us investor to a new us investor allows to bypass the `usInvestorLimit`

Description: `ComplianceServiceLibrary.completeTransferCheck()` runs a list of validations to prevent investor limits (regional and global) from being bypassed in case the recipient of the transfer is a new investor and the number of investors has reached the limit.

When validating the `usInvestorsLimit`, it is not validated whether the sender's region is also the US.

- This means that transferring all the investor balance of a non-US investor will bypass the `usInvestorsLimit`

Thanks to bypassing the check on `ComplianceServiceRegulated.completeTransferCheck()` (as explained in the snippet below), the `usInvestorLimit` will be increased on the `ComplianceServiceRegulated.adjustInvestorsCountsByCountry()`, which is called from the `ComplianceServiceRegulated.recordTransfer()` to increase the investor's counters because the `to` recipient is a new investor (execution flow explained in the second snippet).

```
//completeTransferCheck()//
        } else if (toRegion == US) {
            ...

            uint256 usInvestorsLimit = getUSInvestorsLimit(_services);
            if (
                usInvestorsLimit != 0 &&
//@audit-info => Transferring the full balance turns out the entire conditional to evaluate to false
//@audit => A single false on any of the individual conditions causes all the conditions to evaluate to
↪   false because all of them are &&
                _args.fromInvestorBalance > _args.value &&
                ComplianceServiceRegulated(_services[COMPLIANCE_SERVICE]).getUSInvestorsCount() >=
                ↪   usInvestorsLimit &&
                isNewInvestor(toInvestorBalance)
            ) {
                return (40, MAX_INVESTORS_IN_CATEGORY);
            }

            ...
        }
```

```
//ComplianceServiceRegulated.sol//

    function recordTransfer(
        address _from,
        address _to,
        uint256 _value
    ) internal override returns (bool) {
//@audit => The `to` recipient is a new investor, therefore, calls adjustTransferCounts to increase the
↪   counters
//@audit => `to` investor is a us investor, therefore, the usInvestorLimit will be incremented
        if (compareInvestorBalance(_to, _value, 0)) {
            adjustTransferCounts(_to, CommonUtils.IncDec.Increase);
        }

//@audit => `from` investor is not a us investor
//@audit-info => The usInvestorLimit won't be decremented here, it will be decremented the counter of
↪   the from investor's region
        if (compareInvestorBalance(_from, _value, _value)) {
            adjustTotalInvestorsCounts(_from, CommonUtils.IncDec.Decrease);
        }

        cleanupInvestorIssuances(_from);
        cleanupInvestorIssuances(_to);
        return true;
    }
```

Impact: * US securities regulations requiring strict investor limits can be bypassed.

- Note that this could lead to violations of country-specific transfer restrictions.

Proof of Concept: Run the next proof of concept in `dstoken-regulated.test.ts`:

```
describe('US Investor Limit Bypass Vulnerability POC', function() {
    it('Should demonstrate that US investor limit can be bypassed with full transfer from any country',
    ↪  async function() {
      const [usInvestor1, usInvestor2, nonUsInvestor] = await hre.ethers.getSigners();
      const { dsToken, registryService, complianceConfigurationService, complianceService } = await
      ↪  loadFixture(deployDSTokenRegulatedWithRebasingAndEighteenDecimal);

      // Setup: US limit = 1, disable other limits
      await complianceConfigurationService.setAll(
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 150, 1, 1, 0], // No other limits, short lock period
        [false, false, false, false, false] // Disable all compliance checks initially
      );
      await complianceConfigurationService.setUSInvestorsLimit(1);// set one investor limit for us
      ↪  investor
      await complianceConfigurationService.setBlockFlowbackEndTime(1); // Disable flowback restriction
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.USA,
      ↪  INVESTORS.Compliance.US);
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.GERMANY,
      ↪  INVESTORS.Compliance.EU);

      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, usInvestor1.address, registryService);
      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, usInvestor2.address, registryService);
      await registerInvestor(INVESTORS.INVESTOR_ID.GERMANY_INVESTOR_ID, nonUsInvestor.address,
      ↪  registryService);

      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.USA);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, INVESTORS.Country.USA);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.GERMANY_INVESTOR_ID,
      ↪  INVESTORS.Country.GERMANY);

      // Issue tokens to non-US investor first
      await dsToken.issueTokens(nonUsInvestor.address, 1000);
      console.log('Non-US investor count:', await complianceService.getUSInvestorsCount());

      // Issue tokens to first US investor (reaches limit)
      await dsToken.issueTokens(usInvestor1.address, 1000);
      console.log('After US investor count:', await complianceService.getUSInvestorsCount());
      expect(await complianceService.getUSInvestorsCount()).to.equal(1);

      // Direct issuance to second US investor should fail
      await expect(dsToken.issueTokens(usInvestor2.address, 100))
        .to.be.revertedWith('Max investors in category');

      // VULNERABILITY: Non-US investor can bypass US limit with full transfer
      // This should be blocked but isn't due to missing country check in US logic
      await dsToken.connect(nonUsInvestor).transfer(usInvestor2.address, 1000);

      // Verify bypass succeeded - usInvestor2 now has tokens despite limit being reached
      expect(await dsToken.balanceOf(usInvestor2.address)).to.equal(1000);
      expect(await dsToken.balanceOf(nonUsInvestor.address)).to.equal(0);
      expect(await complianceService.getUSInvestorsCount()).to.equal(2);
    });
  });
```

Recommended Mitigation: Consider updating the conditionals to evaluate to true when the sender (`from`) investor is not a US investor, or when it is a US investor and is not transferring all of its balance; If it's transferring all of its balance, the `from` investor would be decremented from the `usInvestorLimit`.

```diff
        } else if (toRegion == US) {
            ...

            uint256 usInvestorsLimit = getUSInvestorsLimit(_services);
            if (
                usInvestorsLimit != 0 &&
-               _args.fromInvestorBalance > _args.value &&
+               (_args.fromRegion != US || _args.fromInvestorBalance > _args.value) &&
                ComplianceServiceRegulated(_services[COMPLIANCE_SERVICE]).getUSInvestorsCount() >=
                ↪  usInvestorsLimit &&
                isNewInvestor(toInvestorBalance)
            ) {
                return (40, MAX_INVESTORS_IN_CATEGORY);
            }

            ...
        }
```

Securitize: Fixed in commit 5d52ceb.

Cyfrin: Verified.

7.3.7 `TrustService::changeEntityOwner` can overwrite existing `_newOwner` record, breaking 1-1 relationship between owners and addresses

Description: `TrustService::changeEntityOwner` can overwrite existing `_newOwner` record, breaking 1-1 relationship between owners and addresses.

Proof of Concept: Add PoC to `test/trust-service.test.ts`:

```
    it('overwrite existing owner breaks 1-1 relationship', async function() {
      const [owner, firstOwner, secondOwner] = await hre.ethers.getSigners();
      const { trustService } = await loadFixture(deployDSTokenRegulated);

      // Setup: Create two entities with different owners
      await trustService.setRole(firstOwner, DSConstants.roles.ISSUER);
      await trustService.setRole(secondOwner, DSConstants.roles.ISSUER);

      const entity1 = "Entity1";
      const entity2 = "Entity2";

      const trustServiceFromFirst = await trustService.connect(firstOwner);
      await trustServiceFromFirst.addEntity(entity1, firstOwner);

      const trustServiceFromSecond = await trustService.connect(secondOwner);
      await trustServiceFromSecond.addEntity(entity2, secondOwner);

      // Verify initial state
      expect(await trustService.getEntityByOwner(firstOwner)).equal(entity1);
      expect(await trustService.getEntityByOwner(secondOwner)).equal(entity2);

      // Change entity1 owner from firstOwner to secondOwner
      await trustService.changeEntityOwner(entity1, firstOwner, secondOwner);
      // Bug: secondOwner now owns both entities in the forward mapping
      // but reverse mapping shows only entity1
      expect(await trustService.getEntityByOwner(secondOwner)).equal(entity1);
      // entity2 is now orphaned - no way to find its owner through getEntityByOwner

      // firstOwner has no entity in reverse mapping
      expect(await trustService.getEntityByOwner(firstOwner)).equal("");
    });
```

Run with: `npx hardhat test --grep "overwrite existing owner"`.

Recommended Mitigation: Add modifier `onlyNewEntityOwner(_newOwner)` to function `changeEntityOwner`.

Securitize: Fixed in commit 6cd6cca; the relevant storage slots were deprecated and the associated functions were removed.

Cyfrin: Verified.

7.3.8 `TrustService::removeRole` doesn't delete already owned entities so address which lost role can still manage existing entities

Description: `TrustService::removeRole` doesn't delete entities so address which lost role can still manage existing entities.

Proof of Concept: Add PoC to `test/trust-service.test.ts`:

```
  it('Should demonstrate orphaned entity relationships after role removal', async function() {
    const [owner, entityOwner, operator, resource] = await hre.ethers.getSigners();
    const { trustService } = await loadFixture(deployDSTokenRegulated);

    // Step 1: Give entityOwner ISSUER role
    await trustService.setRole(entityOwner, DSConstants.roles.ISSUER);
    expect(await trustService.getRole(entityOwner)).equal(DSConstants.roles.ISSUER);

    // Step 2: Create an entity owned by entityOwner
    const entityName = "TestEntity";
    const trustServiceFromEntityOwner = await trustService.connect(entityOwner);
    await trustServiceFromEntityOwner.addEntity(entityName, entityOwner);

    // Verify entity ownership
    expect(await trustService.getEntityByOwner(entityOwner)).equal(entityName);

    // Step 3: Add operator and resource to the entity
    await trustServiceFromEntityOwner.addOperator(entityName, operator);
    await trustServiceFromEntityOwner.addResource(entityName, resource);

    // Verify operator and resource are linked to entity
    expect(await trustService.getEntityByOperator(operator)).equal(entityName);
    expect(await trustService.getEntityByResource(resource)).equal(entityName);

    // Step 4: Remove entityOwner's ISSUER role
    await trustService.removeRole(entityOwner);
    expect(await trustService.getRole(entityOwner)).equal(DSConstants.roles.NONE);

    // Step 5: Demonstrate the bug - entity relationships still exist
    // These should ideally be cleaned up but they're not:
    expect(await trustService.getEntityByOwner(entityOwner)).equal(entityName);  // Still owns entity!
    expect(await trustService.getEntityByOperator(operator)).equal(entityName);   // Still linked!
    expect(await trustService.getEntityByResource(resource)).equal(entityName);  // Still linked!

    // Step 6: Show the security issue - entityOwner can still manage the entity
    // even without any role
    await expect(
      trustServiceFromEntityOwner.addOperator(entityName, hre.ethers.Wallet.createRandom())
    ).to.not.be.reverted;  // This should fail but doesn't!

    // The onlyEntityOwnerOrAbove modifier still passes because:
    // - entityOwner has NONE role (not MASTER/ISSUER)
    // - But ownersEntities[entityOwner] still equals entityName
    // - So the check passes even though they shouldn't have access
  });
```

Recommended Mitigation: When an address loses its role, delete entities it previously owned by clearing the entity ownership mappings.

Securitize: Fixed in commit 6cd6cca; the relevant storage slots were deprecated and the associated functions were removed.

Cyfrin: Verified.

7.3.9 No way for users to invalidate nonce used to sign

Description: After signing a transaction, users may change their mind and wish to invalidate their signature. In this case users should have a function to call that invalidates their nonce.

In places such as `SecuritizeSwap::executePreApprovedTransaction`, there is no function which allows users to increase (and thereby invalidate) their current nonce.

Impact: Users are unable to revoke a signature before it has been used as they can't invalidate their current nonce.

Recommended Mitigation: Create a function that users can call which just increments their current nonce. Also check `MultiSigWallet`, `TransactionRelayer` and other places using signatures.

Securitize: Acknowledged.

7.3.10 No deadline for user signatures

Description: Signatures signed by users should always have an expiration or timestamp deadline, such that after that time the signature is no longer valid.

When users sign a transaction they should sign using a future expiry deadline and the function processing that signature should revert if the deadline has elapsed.

Recommended Mitigation: Functions such as `SecuritizeSwap::executePreApprovedTransaction` should take a user-supplied deadline as input, and `doExecuteByInvestor` should revert if it has expired otherwise include it when calculating the hash along with the other parameters. Also check `MultiSigWallet`, `TransactionRelayer` and other places using signatures.

Could also consider adding a deadline to functions such as `SecuritizeSwap::buy` even if they don't use a signature, though since that function has a slippage parameter the benefit is not as great.

Securitize: Most of the affected contracts were deleted as they were obsolete, the only one that remains is `TransactionRelayer` which we'll leave as is.

Cyfrin: Verified.

7.3.11 Stale Issuance Records After `Burn` and `Seize` Operations

Description: The `recordBurn()` and `recordSeize()` functions in `ComplianceServiceRegulated.sol` do not clean up issuance records when tokens are burned or seized. This causes stale issuance records to persist in the system, which can lead to incorrect lock calculations for newly issued tokens. The `cleanupInvestorIssuances()` function only removes expired issuance records based on lock periods, but does not remove records for tokens that have been completely burned or seized.

```
  function recordBurn(address _who, uint256 _value) internal override returns (bool) {
        if (compareInvestorBalance(_who, _value, _value)) {
            adjustTotalInvestorsCounts(_who, CommonUtils.IncDec.Decrease);
        }
        return true;
    }
```

Impact: Issuance records remain in the system for tokens that no longer exist, When an investor receives new tokens after a burn/seize operation, the lock calculation in `getComplianceTransferableTokens()` will include stale issuance records from the previously burned/seized tokens

Proof of Concept:

1. Investor receives 100 tokens at timestamp T1 - Creates issuance record: {value: 100, timestamp: T1}
2. All 100 tokens are burned via burn() function
    - recordBurn() is called but does NOT clean up issuance records
    - Stale issuance record persists: {value: 100, timestamp: T1}
3. Investor receives 50 new tokens at timestamp T2 (after lock period)
    - Creates new issuance record: {value: 50, timestamp: T2}
4. When calculating transferable tokens, `getComplianceTransferableTokens()` uses BOTH records:

- Stale record: 100 tokens from T1 (should not exist)
- New record: 50 tokens from T2

5. If lock period hasn't expired since T1, ALL 150 tokens are considered locked
6. Result: Investor cannot transfer any of their 50 new tokens due to stale lock calculation

Recommended Mitigation: Delete the old issuance when a investor is complete burn or seize.

Securitize: Acknowledged; generally it is impossible to determine what issuance record should be deleted when a burn or seize occurs, unless there is only 1 issuance record and the burn or seize is for all its tokens. We also have some compliance rules which effectively prevent the described scenario from happening in the first place.

7.3.12 Malicious investor can register wallets that belong to other investors

Description: The `addWalletByInvestor` function in `RegistryService.sol` allows any registered investor to claim ownership of any wallet address that is not currently in the `investorsWallets` mapping. This creates a critical vulnerability where attackers can front-run legitimate wallet registration transactions to hijack wallet ownership:

```
function addWalletByInvestor(address _address) public override newWallet(_address) returns (bool) {
        require(!getWalletManager().isSpecialWallet(_address), "Wallet has special role");

        string memory owner = getInvestor(msg.sender);
        require(isInvestor(owner), "Unknown investor");

        investorsWallets[_address] = Wallet(owner, msg.sender, msg.sender);
        investors[owner].walletCount++;

        emit DSRegistryServiceWalletAdded(_address, owner, msg.sender);

        return true;
    }
```

The function only checks that the caller is a registered investor and that the wallet is not a special wallet, but does not verify that the caller actually controls the wallet address being registered.

With that being say a malicious register investor can front run (or if he already know what wallet will be registered) any `addWallet`, `updateInvestor`, and `addWalletByInvestor` call setting the wallet to himself DoSing those function and possible Redirect token issuances meant for other investors to themselves.

Impact: * `addWallet`, `updateInvestor`, `addWalletByInvestor` in the registry and `TokenIssuer:issueTokens` and `SecuritySwap:swap` functions can be DoS.

- Since the token issuant process is using those wallets to mint tokens(see `issueTokensCustom`)

```
function issueTokensCustom(address _to, uint256 _value, uint256 _issuanceTime, uint256 _valueLocked,
↪  string memory _reason, uint64 _releaseTime) // _to could be the wallet that malicious investor just
↪  take
   public
   virtual
   override
   returns (
   /*onlyIssuerOrAbove*/
       bool
   )
   {...}
```

Proof of Concept: Run the next proof of concept in `registry-service.test.ts` in describe `Wallet By Investor`

```
it('Steal wallet', async function() {
       // victim wallet will be another wallet that the investor2 want to register
       const [owner, investor1, investor2, victimWallet] = await hre.ethers.getSigners();
       const { registryService, dsToken } = await loadFixture(deployDSTokenRegulated);

       // Setup: Register two investors
       await registryService.registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1,
       ↪  INVESTORS.INVESTOR_ID.INVESTOR_COLLISION_HASH_1);
       await registryService.registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2,
       ↪  INVESTORS.INVESTOR_ID.INVESTOR_COLLISION_HASH_2);

       // Setup: Add wallets to investors
       await registryService.addWallet(investor1, INVESTORS.INVESTOR_ID.INVESTOR_ID_1);
       await registryService.addWallet(investor2, INVESTORS.INVESTOR_ID.INVESTOR_ID_2);

       // Verify initial state
       expect(await registryService.getInvestor(investor1.address)).to.equal(INVESTORS.INVESTOR_ID.INV ⌋
       ↪  ESTOR_ID_1);
       expect(await registryService.getInvestor(investor2.address)).to.equal(INVESTORS.INVESTOR_ID.INV ⌋
       ↪  ESTOR_ID_2);
       expect(await registryService.isWallet(victimWallet.address)).to.be.false;

       // ATTACK: Investor1 front-runs and steals victimWallet before Investor2 can register it
       const registryServiceFromInvestor1 = await registryService.connect(investor1);
       await registryServiceFromInvestor1.addWalletByInvestor(victimWallet.address);

       // Verify attack succeeded - victimWallet now belongs to Investor1
       expect(await registryService.getInvestor(victimWallet.address)).to.equal(INVESTORS.INVESTOR_ID. ⌋
       ↪  INVESTOR_ID_1);
       expect(await registryService.isWallet(victimWallet.address)).to.be.true;

       // Now if Investor2 tries to register the same wallet, it will fail
       const registryServiceFromInvestor2 = await registryService.connect(investor2);
       await expect(
         registryServiceFromInvestor2.addWalletByInvestor(victimWallet.address)
       ).to.be.revertedWith("Wallet already exists");

       // Demonstrate token hijacking - issue tokens to victimWallet
       // The tokens will go to Investor1 instead of Investor2
       await dsToken.setCap(1000);
```

```
        await dsToken.issueTokens(victimWallet.address, 100);

        // Verify Investor1 received the tokens (through their wallet)
        expect(await dsToken.balanceOf(victimWallet.address)).to.equal(100);
        expect(await registryService.getInvestor(victimWallet.address)).to.equal(INVESTORS.INVESTOR_ID.
        ↪   INVESTOR_ID_1);

      });
```

Recommended Mitigation: Remove `addWalletByInvestor`.

Securitize: Fixed in commit 05c5bad.

Cyfrin: Verified.

7.3.13 `SecuritizeSwap::executeStableCoinTransfer` should use `SafeERC20` approval and transfer functions instead of standard `IERC20` functions

Description: In `SecuritizeSwap::executeStableCoinTransfer` use `SafeERC20::forceApprove` and `SafeERC20::safeTransferFrom`:

```diff
    function executeStableCoinTransfer(address from, uint256 value) private {
        if (bridgeChainId != 0 && address(USDCBridge) != address(0)) {
-           stableCoinToken.transferFrom(from, address(this), value);
-           stableCoinToken.approve(address(USDCBridge), value);
+           SafeERC20.safeTransferFrom(stableCoinToken, from, address(this), value);
+           SafeERC20.forceApprove(stableCoinToken, address(USDCBridge), value);
            USDCBridge.sendUSDCCrossChainDeposit(bridgeChainId, issuerWallet, value);
        } else {
-           stableCoinToken.transferFrom(from, issuerWallet, value);
+           SafeERC20.safeTransferFrom(stableCoinToken, from, issuerWallet, value);
        }
    }
```

Securitize: `SecuritizeSwap` was removed as it was deprecated.

Cyfrin: Verified.

7.3.14 `MulticallProxy::_callTarget` doesn't check if `target` has code

Description: `IssuerMulticall::multicall` allows permissioned users to perform multiple calls against arbitrary targets, calling `MulticallProxy::_callTarget` for each target.

`MulticallProxy::_callTarget` doesn't check whether each `target` has code:

```solidity
function _callTarget(address target, bytes memory data, uint256 i) internal returns (bytes memory) {
    // @audit returns true if `target` has no code
    (bool success, bytes memory returndata) = target.call(data);
    if (!success) {
        if (returndata.length > 0) {
            // Assumes revert reason is encoded as string
            revert MulticallFailed(i, _getRevertReason(returndata));
        } else {
            revert MulticallFailed(i, "Call failed without revert reason");
        }
    }
    return returndata;
}
```

Impact: When using `call` on an address with no code, `call` returns `true`. This can mislead the caller to think that the multi-call succeeded when in fact it didn't if one of the targets within the multi-call has no code.

Recommended Mitigation: Verify `target` has code:

```
    function _callTarget(address target, bytes memory data, uint256 i) internal returns (bytes memory) {
+        require(target.code.length > 0, "Target is not a contract");
```

Securitize: The affected contracts were deleted as they were obsolete.

Cyfrin: Verified.

7.3.15 Seized tokens will be stuck on the receiver of the seized funds because it is not an investor and never receives investorsBalance

Description: When seizing DSTokens, the receiver of the confiscated assets must be an issuer wallet. A wallet that is an issuer wallet must not be an investor wallet. This means the receiver wallet won't point to an investor, so reading `getRegistryService().getInvestor(_wallet);` would return an empty string.

This has the side effect that during the seizure of funds, when updating the `investorsBalance` of the wallet receiving the confiscated funds won't register any investorsBalance because there is not an investor associated to that wallet.

```
    function seize(
        TokenData storage _tokenData,
        address[] memory _services,
        address _from,
        address _to,
        uint256 _value,
        uint256 _shares
)
    public
    validSeizeParameters(_tokenData, _from, _to, _shares)
    {
        ...
@>      updateInvestorBalance(_tokenData, registryService, _to, _shares, CommonUtils.IncDec.Increase);
    }

    function updateInvestorBalance(TokenData storage _tokenData, IDSRegistryService _registryService,
    ↪   address _wallet, uint256 _shares, CommonUtils.IncDec _increase) internal returns (bool) {
//@audit => investor for the wallet receiving the seized funds would return an empty string
        string memory investor = _registryService.getInvestor(_wallet);
//@audit => An empty string would skip the code to register the investorsBalance for the seized funds
        if (!CommonUtils.isEmptyString(investor)) {
            uint256 balance = _tokenData.investorsBalances[investor];
            if (_increase == CommonUtils.IncDec.Increase) {
                balance += _shares;
            } else {
                balance -= _shares;
            }
            _tokenData.investorsBalances[investor] = balance;
        }
        return true;
    }
```

Not having an `investorsBalance` for the seized funds means that `InvestorLockManager.getTransferableTokens()` will return 0 available tokens to be transferred when attempting to move the seized funds.

- 0 available tokens will cause the `ComplianceServiceRegulated.completeTransferCheck()` to return code 16 with error message `TOKENS_LOCKED` because the `sender` is not a platform wallet, it's an issuer wallet, and it has 0 investor balance to process the transfer.

```
    function completeTransferCheck(
        address[] memory _services,
        CompletePreTransferCheckArgs memory _args
    ) internal view returns (uint256 code, string memory reason) {
        ...
        if (
            !isPlatformWalletFrom &&
@>      IDSLockManager(_services[LOCK_MANAGER]).getTransferableTokens(_args.from, block.timestamp) <
↪   _args.value
        ) {
@>          return (16, TOKENS_LOCKED);
        }

}
```

Impact: Seized tokens won't be transferable from the address receiving the confiscated funds.

Proof of Concept: Add the next PoC to the `dstoken-regulated.test.ts`

```
    it.only('seized tokens are stuck on the receiver', async function () {
      const [owner, investor1, investor2, seizedReceiver, investor3] = await hre.ethers.getSigners();
      const { dsToken, registryService, complianceService, complianceConfigurationService,
      ↪  walletManager } = await loadFixture(deployDSTokenRegulated);
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.FRANCE,
      ↪  INVESTORS.Compliance.EU);
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.USA,
      ↪  INVESTORS.Compliance.US);
      await complianceConfigurationService.setEURetailInvestorsLimit(10);
      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, investor1, registryService);
      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, investor2, registryService);
      // await registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID, seizedReceiver, registryService);
      await registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2, investor3, registryService);

      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.FRANCE);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, INVESTORS.Country.USA);
      // await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID, INVESTORS.Country.USA);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2, INVESTORS.Country.USA);

      const issuedTokens = 500;
      const seizedTokens = issuedTokens;
      await dsToken.issueTokens(investor1, issuedTokens);
      await dsToken.issueTokens(investor2, issuedTokens);

      expect(await complianceService.getEURetailInvestorsCount(INVESTORS.Country.FRANCE)).equal(1);
      expect(await complianceService.getUSInvestorsCount()).equal(1);

      await walletManager.addIssuerWallet(seizedReceiver)

      await dsToken.seize(investor1, seizedReceiver, await
      ↪  dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1), "");
      expect(await dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1)).equal(0);
      expect(await dsToken.balanceOf(investor1)).equal(0);
      expect(await complianceService.getEURetailInvestorsCount(INVESTORS.Country.FRANCE)).equal(0);
      expect(await complianceService.getUSInvestorsCount()).equal(1);

      expect(await dsToken.balanceOf(seizedReceiver)).equal(500);

      time.increase(10_000);

      const dsTokenFromSeizedReceiver = await dsToken.connect(seizedReceiver);
      //@audit-issue => Seized tokens will be stuck on the receiver of the seized funds because it is
      ↪  not an investor and never receives investorsBalance
      await expect(dsTokenFromSeizedReceiver.transfer(investor3, seizedTokens)).to.be.reverted;
    });
```

Securitize: Acknowledged; according to the operations team, this is the workflow: seized funds stay in wallet until reallocation, wallet is locked in advance. Then we burn and reissue the tokens to the correct wallet/holder.

7.3.16 `ComplianceServiceRegulated::preIssuanceCheck` allows issuance to non-accredited investors when `forceAccredited` or `forceAccreditedUS` is set, and allows issuance below regional minimum thresholds, violating compliance requirements

Description: `ComplianceServiceRegulated::completeTransferCheck` has several checks that `preIssuanceCheck` is missing:

1) Force Accredited

`completeTransferCheck` verifies whether force accredited is enabled and if so only allow transfers to accredited investors:

```
bool isAccreditedTo = isAccredited(_services, _args.to);
if (
    IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]).getForceAccredited()
    ↪  && !isAccreditedTo
) {
    return (61, ONLY_ACCREDITED);
}

} else if (toRegion == US) {
    if (
        IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]).getForceAccredit ⌋
        ↪  edUS() &&
        !isAccreditedTo
    ) {
        return (62, ONLY_US_ACCREDITED);
    }
```

But `preIssuanceCheck` doesn't enforce this, so it could allow issuance to unaccredited investors even when `ForceAccredited` or `ForceAccreditedUS` is enabled.

2) Regional Minimal Token Holdings

`completeTransferCheck` verifies regional minimum token holdings eg:

```
if (toInvestorBalance + _value <
↪   IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]).getMinUSTokens()) {
    return (51, AMOUNT_OF_TOKENS_UNDER_MIN);
}
```

But preIssuanceCheck only verifies the generic minimum holdings:

```
        if (
            !walletManager.isPlatformWallet(_to) &&
        balanceOfInvestorTo + _value < complianceConfigurationService.getMinimumHoldingsPerInvestor()
        ) {
            return (51, AMOUNT_OF_TOKENS_UNDER_MIN);
        }
```

Hence `preIssuanceCheck` could result in users being issued an amount of tokens that violates their regional minimum token holdings.

Impact: `ComplianceServiceRegulated::preIssuanceCheck` allows issuance to non-accredited investors when `forceAccredited` or `forceAccreditedUS` is set, and allows issuance below regional minimum thresholds, violating compliance requirements

Recommended Mitigation: Enforce the above checks in `ComplianceServiceRegulated::preIssuanceCheck`.

Securitize: Fixed in commits 4991826, 0656ccd.

Cyfrin: Verified.

7.3.17 Burn and seize functions can be DoS when investor has several wallets that they control

Description: The `burn` function in `TokenLibrary.sol` checks `walletsBalances[_who]` (individual wallet balance, see the arrow above) instead of `investorsBalances[investorId]` (total investor balance across all wallets). This allows investors with multiple registered wallets to make their tokens unburnable by transferring tokens between their own wallets.

```
function burn(
        TokenData storage _tokenData,
        address[] memory _services,
        address _who,
        uint256 _value,
        ISecuritizeRebasingProvider _rebasingProvider
    ) public returns (uint256) {
        uint256 sharesToBurn = _rebasingProvider.convertTokensToShares(_value);

        require(sharesToBurn <= _tokenData.walletsBalances[_who], "Not enough balance"); <---------

        IDSComplianceService(_services[COMPLIANCE_SERVICE]).validateBurn(_who, _value);

        _tokenData.walletsBalances[_who] -= sharesToBurn;
        updateInvestorBalance(
            _tokenData,
            IDSRegistryService(_services[REGISTRY_SERVICE]),
            _who,
            sharesToBurn,
            CommonUtils.IncDec.Decrease
        );

        _tokenData.totalSupply -= sharesToBurn;
        return sharesToBurn;
    }
```

When an investor transfers tokens from their original wallet to another wallet they control, the burn function will fail with "Not enough balance" even though the investor still owns the tokens in their total balance.

Impact: If the admin doesn't use private mempools when performing sensitive operations such as burn or seize, investors can front-run to temporarily prevent token burning by transferring tokens between their own wallets.

Proof of Concept: Run the next proof of concept in `dstoken-regulated.test.ts`:

```
describe('Burn DoS Vulnerability POC', function() {
        it('Should demonstrate that burn can be DoS by transferring between investor wallets', async
        ↪  function() {
          const [investor, wallet2, wallet3] = await hre.ethers.getSigners();
          const { dsToken, registryService } = await
          ↪  loadFixture(deployDSTokenRegulatedWithRebasingAndEighteenDecimal);

          // Register investor with multiple wallets
          await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, investor.address,
          ↪  registryService);
          await registryService.addWallet(wallet2.address, INVESTORS.INVESTOR_ID.INVESTOR_ID_1);
          await registryService.addWallet(wallet3.address, INVESTORS.INVESTOR_ID.INVESTOR_ID_1);

          // Issue tokens to investor
          await dsToken.issueTokens(investor.address, 1000);

          // Transfer tokens between investor's own wallets
          await dsToken.connect(investor).transfer(wallet2.address, 1000);
          // Now investor has 0 balance in original wallet but 1000 total
          expect(await dsToken.balanceOf(investor.address)).to.equal(0);
          expect(await dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1)).to.equal(1000);

          // VULNERABILITY: Burn fails because wallet balance is 0, even though investor has tokens
          await expect(dsToken.burn(investor.address, 100, 'DoS test'))
            .to.be.revertedWith('Not enough balance');
        });
      });
```

Recommended Mitigation: Possible mitigation options include:

- perform sensitive admin transactions such as burn and seize through private mempool services like flashbots so they can't be front-run
- remove the `addWalletByInvestor` function to prevent investors from continually adding more wallets and distributing their tokens to them
- add `burnAll` and `seizeAll` functions to `DSToken` which iterate over every wallet belonging to an investor and burn/seize all their tokens

Securitize: Fixed in commit 05c5bad by removing `addWalletByInvestor`. Operations team to consider advice regarding running sensitive admin transactions via private mempools.

Cyfrin: Verified.

7.3.18 Not maximum amount of issuances enforce

Description: The `createIssuanceInformation` function in `ComplianceServiceRegulated.sol` allows unlimited issuance records to be created per investor without any maximum limit. Unlike the lock system which has a `MAX_LOCKS_PER_INVESTOR = 30` constant to prevent run out of gas, the issuance system has no such protection. Each time tokens are issued to an investor, a new issuance record is created and stored in unbounded mappings (`issuancesValues`, `issuancesTimestamps`, `issuancesCounters`). These records are processed in loops during compliance checks (`getComplianceTransferableTokens` and `cleanupExpiredIssuances`), which could potentially cause gas limit issues if an investor accumulates too many issuance records.

```
function createIssuanceInformation(
        string memory _investor,
        uint256 _shares,
        uint256 _issuanceTime
    ) internal returns (bool) {
        uint256 issuancesCount = issuancesCounters[_investor];//@audit-ok (low) there is not max limit
        ↪  for the issuancesCounters?

        issuancesValues[_investor][issuancesCount] = _shares;
        issuancesTimestamps[_investor][issuancesCount] = _issuanceTime;
        issuancesCounters[_investor] = issuancesCount + 1;

        return true;
    }
```

Impact: If an investor accumulates many issuance records, compliance checks could hit gas limits; While the attack vector is limited due to the issuance is an access control operation the lock manager is also an access control operation and have a maximum lock

Recommended Mitigation: Implement a maximum limit for issuance records per investor, similar to the existing lock system.

Securitize: Acknowledged; the cleanup method was created for this very reason, and we assume that we will clean enough records to avoid hitting those scenarios because lockup periods are not that long that we need to keep very old issuance records.

7.3.19 Platform Wallet not exception for Maximum Holdings Per Investor Limits

Description: The `preIssuanceCheck` function in `ComplianceServiceRegulated.sol` exhibits inconsistent behavior regarding platform wallet exemptions for investor holdings limits. While platform wallets are properly exempted from the minimum holdings per investor check, they are not exempted from the maximum holdings per investor check.

```
if (
    !_args.isPlatformWalletTo &&
    toInvestorBalance + _args.value < IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURAT ⌋
    ↪  ION_SERVICE]).getMinimumHoldingsPerInvestor()
) {
    return (51, AMOUNT_OF_TOKENS_UNDER_MIN);
}
```

The minimum holdings check correctly includes `!_args.isPlatformWalletTo &&` to exempt platform wallets, but the maximum holdings check lacks this exemption, subjecting platform wallets to the same maximum holdings limits as regular investors.

```
if (
            isMaximumHoldingsPerInvestorOk(
                IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]).getMaxim ⌋
                ↪  umHoldingsPerInvestor(),
                toInvestorBalance, _args.value)
        ) {
            return (52, AMOUNT_OF_TOKENS_ABOVE_MAX);
        }
```

Impact: Platform wallets may be unable to hold sufficient tokens for operational purposes due to maximum holdings limits.

Recommended Mitigation: Add platform wallet exemption to the maximum holdings check:

```
if (
+    !_args.isPlatformWalletTo &&
    isMaximumHoldingsPerInvestorOk(
        complianceConfigurationService.getMaximumHoldingsPerInvestor(),
        balanceOfInvestorTo,
        _value)
) {
    return (52, AMOUNT_OF_TOKENS_ABOVE_MAX);
}
```

Securitize: Fixed in commits 5b96460, 2cab0c2.

Cyfrin: Verified.

7.3.20 Attribute changes via `setAttribute` or `updateInvestor` do NOT trigger compliance count updates.

Description: The compliance system maintains several count variables (`accreditedInvestorsCount`, `usAccreditedInvestorsCount`, `euRetailInvestorsCount`, between others) to enforce investor limits and regulatory compliance. However, when investor attributes are changed through `setAttribute()` or `updateInvestor()` functions, these count variables are not updated to reflect the changes.

The system only updates counts when investors are added or removed via `adjustInvestorCountsAfterCountryChange()`, but attribute changes that affect investor classification (`ACCREDITED`, `QUALIFIED`) bypass this mechanism entirely. This creates a fundamental disconnect between the actual investor status and the compliance tracking system.

This affect specifically `accreditedInvestorsCount`, `usAccreditedInvestorsCount`, `euRetailInvestorsCount` for example:

```
} else if (countryCompliance == EU && !getRegistryService().isQualifiedInvestor(_id)) {//@audit  if
↪  there are an investor that become qualified after he enter the system counting  it will not be
↪  discounting the investor when he is leaving
            if(_increase == CommonUtils.IncDec.Increase) {
                euRetailInvestorsCount[_country]++;
            }
            else {
                euRetailInvestorsCount[_country]--;
            }
```

In this case if EU retail investor enter the system he will count in `euRetailInvestorsCount` but if he become `QUALIFIED` and then leave the system the `euRetailInvestorsCount` will remain inflated.

Impact: investor limit enforcement becomes unreliable as counts don't reflect actual count

Proof of Concept: Run the next proof of concept in `compliance-service-regulated.test.ts`:

```
describe('Proof of Concept: Attribute Change Bug', function () {
    it('should demonstrate euRetailInvestorsCount inflation when investor becomes qualified', async
    ↪  function () {
      const [wallet, wallet2, transferAgent] = await hre.ethers.getSigners();
      const { dsToken, registryService, complianceService, complianceConfigurationService, trustService
      ↪  } = await loadFixture(deployDSTokenRegulated);

      // Set up EU compliance
      await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.FRANCE,
      ↪  INVESTORS.Compliance.EU);
      await complianceConfigurationService.setEURetailInvestorsLimit(1);

      // Register two investors in France
      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, wallet, registryService);
      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, wallet2, registryService);

      // Set both investors to France (EU)
      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.FRANCE);
      await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, INVESTORS.Country.FRANCE);

      // Issue tokens to first investor (non-qualified) - should be counted as EU retail
      await dsToken.issueTokens(wallet, 100);

      // Check initial EU retail count
      const initialEuRetailCount = await
      ↪  complianceService.getEURetailInvestorsCount(INVESTORS.Country.FRANCE);
      expect(initialEuRetailCount).to.equal(1, "Initial EU retail count should be 1");

      // Try to issue tokens to second investor - should fail due to EU retail limit
      await expect(dsToken.issueTokens(wallet2, 100)).revertedWith('Max investors in category');

      // Now make the first investor qualified via setAttribute
      // QUALIFIED = 4, APPROVED = 1
      await registryService.setAttribute(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, 4, 1, 0, "");


      // Check EU retail count - it should still be 1 (BUG: should be 0)
      const euRetailCountAfterQualification = await
      ↪  complianceService.getEURetailInvestorsCount(INVESTORS.Country.FRANCE);
      expect(euRetailCountAfterQualification).to.equal(1, "BUG: EU retail count should be 0 after
      ↪  qualification, but it's still 1");

      // Now try to issue tokens to second investor - this should still fail due to inflated count
      await expect(dsToken.issueTokens(wallet2, 100)).revertedWith('Max investors in category');
    });
  });
```

Recommended Mitigation: Ensure that any attribute change that affects investor classification immediately updates the corresponding compliance count variables to maintain system integrity.

Securitize: Acknowledged; these functions are never currently invoked.

7.3.21 Investor can prevent themselves from being removed by making `removeInvestor` revert

Description: The `removeInvestor` function in `RegistryService.sol` contains a flaw that allows any investor to permanently prevent their removal from the system. The function requires that `investors[_id].walletCount == 0` before allowing investor removal, but investors can add unlimited wallets via `addWalletByInvestor` without any restrictions, while only `EXCHANGE` roles can remove wallets via `removeWallet`.

```
function removeInvestor(string calldata _id) public override onlyExchangeOrAbove investorExists(_id)
↪   returns (bool) {
        require(getTrustService().getRole(msg.sender) != EXCHANGE || investors[_id].creator ==
        ↪   msg.sender, "Insufficient permissions");
        require(investors[_id].walletCount == 0, "Investor has wallets"); <----------

        for (uint8 index = 0; index < 16; index++) {
            delete attributes[_id][index];
        }

        delete investors[_id];

        emit DSRegistryServiceInvestorRemoved(_id, msg.sender);

        return true;
    }
```

This creates a permanent DoS condition where malicious investors can add wallets to prevent their own removal

Impact: `removeInvestor` can be DoS, making an investor unremovable.

Recommended Mitigation: Consider removing `addWalletByInvestor`.

Securitize: Fixed in commit 05c5bad by removing `addWalletByInvestor`.

Cyfrin: Verified.

7.3.22 Resolve inconsistency between `DSToken::checkWalletsForList` and `RegistryService::removeWallet`

Description: `DSToken::checkWalletsForList` only removes wallets if their balance is zero:

```
function checkWalletsForList(address _from, address _to) private {
    if (super.balanceOf(_from) == 0) {
        removeWalletFromList(_from);
    }
```

But `RegistryService::removeWallet` allows removing wallets with positive balances:

```
function removeWallet(address _address, string memory _id) public override onlyExchangeOrAbove
↪   walletExists(_address) walletBelongsToInvestor(_address, _id) returns (bool) {
    require(getTrustService().getRole(msg.sender) != EXCHANGE || investorsWallets[_address].creator ==
    ↪   msg.sender, "Insufficient permissions");

    delete investorsWallets[_address];
    investors[_id].walletCount--;

    emit DSRegistryServiceWalletRemoved(_address, _id, msg.sender);
    return true;
}
```

Impact: Wallets with positive balances can be removed via `RegistryService::removeWallet` which prevents token transfers and other related activity that depends on functions from `RegistryService` returning investor ids for given wallet addresses.

Recommended Mitigation: `RegistryService::removeWallet` shouldn't allow removing wallets with positive balances.

Securitize: Fixed in commit 1eaec18.

Cyfrin: Verified.

7.3.23 `RegistryService::addWallet` should revert if the wallet being added has positive balance of `DSToken`

Description: `RegistryService::addWallet` doesn't update investor wallet or total balances if it is used to add a wallet that already has a positive `DSToken` balance.

Using it to add a wallet with a positive balance results in a number of incorrect states:

Compliance Validation Problems

- Investor might not be counted in total investors
- Won't trigger investor limit checks
- Could bypass US/EU investor limits

Transfer Validation Problems

- Compliance checks expect investor balance to match wallet balance
- Some checks compare investor balance to transfer amount
- Could trigger incorrect "new investor" logic
- Transfers could revert due to underflow when subtracting from existing internal wallet / investor balances resulting in the tokens being stuck

Recommended Mitigation: `RegistryService::addWallet` should revert if the wallet being added has positive balance of `DSToken`. The same applies to `addWalletByInvestor` but that function is being removed.

Alternatively another option is to register the existing tokens for the investor by calling `DSToken::updateInvestorBalance` and `addWalletToList` though these functions are currently private.

Securitize: Fixed in commit 3e9c754 by preventing adding wallets with positive balance.

Cyfrin: Verified.

7.4 Informational

7.4.1 Use named imports

Description: The codebase in some places used named imports but not in others; recommend using named imports everywhere.

Securitize: Fixed in commit 7fc22e2.

Cyfrin: Verified.

7.4.2 Upgradeable contracts should call `_disableInitializers` in constructor

Description: Upgradeable contracts should call `_disableInitializers` in constructor:

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

Affected contracts:

- `contracts/bulk/BulkOperator.sol`
- `contracts/compliance/ComplianceConfigurationService.sol`
- `contracts/compliance/ComplianceServiceNotRegulated.sol`
- `contracts/compliance/ComplianceServiceWhitelisted.sol`
- `contracts/compliance/InvestorLockManager.sol`
- `contracts/compliance/LockManager.sol`
- `contracts/compliance/WalletManager.sol`
- `contracts/issuance/TokenIssuer.sol`
- `contracts/multicall/IssuerMulticall.sol`
- `contracts/rebasing/SecuritizeRebasingProvider.sol`
- `contracts/registry/RegistryService.sol`
- `contracts/registry/WalletRegistrar.sol`
- `contracts/swap/SecuritizeSwap.sol`
- `contracts/token/DSToken.sol`
- `contracts/trust/TrustService.sol`
- `contracts/utils/TransactionRelayer.sol`

Securitize: Fixed in commit 094baaf; note some of the listed contracts have been removed as they were obsolete.

Cyfrin: Verified.

7.4.3 Prefer explicit unsigned integer sizes

Description: Prefer explicit unsigned integer sizes:

```
trust/TrustService.sol
218:        for (uint i = 0; i < _addresses.length; i++) {

compliance/ComplianceConfigurationService.sol
34:        for (uint i = 0; i < _countries.length; i++) {

compliance/WalletManager.sol
```

```
75:         for (uint i = 0; i < _wallets.length; i++) {
97:         for (uint i = 0; i < _wallets.length; i++) {
```

Securitize: Fixed in commit 26c5bb0.

Cyfrin: Verified.

7.4.4 Use named mapping parameters to explicitly note the purpose of keys and values

Description: Use named mapping parameters to explicitly note the purpose of keys and values:

```
token/TokenLibrary.sol
36:         mapping(address => uint256) walletsBalances;
37:         mapping(string => uint256) investorsBalances;

mocks/TestToken.sol
39:     mapping(address => uint256) balances;
40:     mapping(address => mapping(address => uint256)) allowed;

data-stores/TokenDataStore.sol
27:     mapping(address => mapping(address => uint256)) internal allowances;
28:     mapping(uint256 => address) internal walletsList;
30:     mapping(address => uint256) internal walletsToIndexes;

swap/SecuritizeSwap.sol
43:     mapping(string => uint256) internal noncePerInvestor;

utils/TransactionRelayer.sol
48:     mapping(bytes32 => uint256) internal noncePerInvestor;

utils/MultiSigWallet.sol
46:     mapping(address => bool) isOwner; // immutable state

data-stores/RegistryServiceDataStore.sol
44:         // Ref:
↪   https://docs.soliditylang.org/en/v0.7.1/070-breaking-changes.html#mappings-outside-storage
45:         // mapping(uint8 => Attribute) attributes;
48:     mapping(string => Investor) internal investors;
49:     mapping(address => Wallet) internal investorsWallets;
52:      * @dev DEPRECATED: This mapping is no longer used but must be kept for storage layout
↪   compatibility in the proxy.
53:      * Do not use this mapping in new code. It will be removed in future non-proxy implementations.
56:     mapping(address => address) internal DEPRECATED_omnibusWalletsControllers;
58:     mapping(string => mapping(uint8 => Attribute)) public attributes;

data-stores/InvestorLockManagerDataStore.sol
24:     mapping(string => mapping(uint256 => Lock)) internal investorsLocks;
25:     mapping(string => uint256) internal investorsLocksCounts;
26:     mapping(string => bool) internal investorsLocked;
27:     mapping(string => mapping(bytes32 => mapping(uint256 => Lock))) internal
↪   investorsPartitionsLocks;
28:     mapping(string => mapping(bytes32 => uint256)) internal investorsPartitionsLocksCounts;
29:     mapping(string => bool) internal investorsLiquidateOnly;

data-stores/TrustServiceDataStore.sol
23:     mapping(address => uint8) internal roles;
24:     mapping(string => address) internal entitiesOwners;
25:     mapping(address => string) internal ownersEntities;
26:     mapping(address => string) internal operatorsEntities;
27:     mapping(address => string) internal resourcesEntities;

data-stores/ComplianceConfigurationDataStore.sol
```

```
24:     mapping(string => uint256) public countriesCompliances;

data-stores/WalletManagerDataStore.sol
24:     mapping(address => uint8) internal walletsTypes;
25:     mapping(address => mapping(string => mapping(uint8 => uint256))) internal walletsSlots;

data-stores/ServiceConsumerDataStore.sol
23:     mapping(uint256 => address) internal services;

data-stores/LockManagerDataStore.sol
24:     mapping(address => uint256) internal locksCounts;
25:     mapping(address => mapping(uint256 => Lock)) internal locks;

data-stores/ComplianceServiceDataStore.sol
29:     mapping(string => uint256) internal euRetailInvestorsCount;
30:     mapping(string => uint256) internal issuancesCounters;
31:     mapping(string => mapping(uint256 => uint256)) issuancesValues;
32:     mapping(string => mapping(uint256 => uint256)) issuancesTimestamps;
```

Securitize: Fixed in commit 6c7bc52.

Cyfrin: Verified.

7.4.5 Emit missing events

Description: Emit missing events:

- `DSToken::setFeature`, `setFeatures`, `setCap`
- `TrustService::addEntity`, `changeEntityOwner`, `addOperator`, `removeOperator`, `addResource`, `removeResource`
- `SecuritizeSwap::updateNavProvider`

Securitize: Most of these were removed as they were deprecated, the `setFeature` was left for now as it is not used at the moment.

Cyfrin: Verified.

7.4.6 Consider reverting in `RebasingLibrary` functions if rounding down to zero occurs

Description: `RebasingLibrary` has two functions `convertTokensToShares` and `convertSharesToTokens`. If rounding down to zero occurs such that the input is non-zero but the output is zero, consider reverting as it makes little sense to continue processing at that point. For example:

- `convertTokensToShares` should revert if `_tokens > 0 && shares == 0`
- `convertSharesToTokens` should revert if `_shares > 0 && tokens == 0`

Securitize: Fixed in commit 60c5f92 by adding this check in `convertTokensToShares`. Note that we didn't add it in `convertSharesToTokens` as that is used by functions such as `StandardToken::balanceOf`.

Cyfrin: Verified.

7.4.7 Use named constants instead of hard-coded literals for important values

Description: Use named constants instead of hard-coded literals for important values:

```
contracts/compliance/ComplianceServiceRegulated.sol
// error codes in `ComplianceServiceRegulated::completeTransferCheck`
217:            return (0, VALID);
221:            return (90, INVESTOR_LIQUIDATE_ONLY);
225:            return (20, WALLET_NOT_IN_REGISTRY_SERVICE);
230:            return (26, DESTINATION_RESTRICTED);
```

```
238:            return (16, TOKENS_LOCKED);
243:                return (32, HOLD_UP);
250:                return (51, AMOUNT_OF_TOKENS_UNDER_MIN);
257:                return (50, ONLY_FULL_TRANSFER);
261:                return (33, HOLD_UP);
269:                return (25, FLOWBACK);
276:                return (50, ONLY_FULL_TRANSFER);
286:                return (51, AMOUNT_OF_TOKENS_UNDER_MIN);
294:            return (61, ONLY_ACCREDITED);
305:                return (40, MAX_INVESTORS_IN_CATEGORY);
316:                return (40, MAX_INVESTORS_IN_CATEGORY);
322:                return (51, AMOUNT_OF_TOKENS_UNDER_MIN);
329:                return (62, ONLY_US_ACCREDITED);
339:                return (40, MAX_INVESTORS_IN_CATEGORY);
350:                return (40, MAX_INVESTORS_IN_CATEGORY);
356:                return (51, AMOUNT_OF_TOKENS_UNDER_MIN);
362:                return (40, MAX_INVESTORS_IN_CATEGORY);
373:            return (40, MAX_INVESTORS_IN_CATEGORY);
382:            return (71, NOT_ENOUGH_INVESTORS);
390:            return (51, AMOUNT_OF_TOKENS_UNDER_MIN);
397:            return (51, AMOUNT_OF_TOKENS_UNDER_MIN);
405:            return (52, AMOUNT_OF_TOKENS_ABOVE_MAX);
408:        return (0, VALID);

// denominator representing 100%
108:            return compConfService.getMaxUSInvestorsPercentage() *
↪  (complianceService.getTotalInvestorsCount()) / 100;
111:        return Math.min(compConfService.getUSInvestorsLimit(),
↪  compConfService.getMaxUSInvestorsPercentage() * (complianceService.getTotalInvestorsCount()) / 100);

compliance/ComplianceConfigurationService.sol
238:        require(_uint_values.length == 16, "Wrong length of parameters");

registry/RegistryService.sol
43:        for (uint8 index = 0; index < 16; index++) {
135:        require(_attributeId < 16, "Unknown attribute");

trust/TrustService.sol
216:        require(_addresses.length <= 30, "Exceeded the maximum number of addresses");

compliance/WalletManager.sol
74:        require(_wallets.length <= 30, "Exceeded the maximum number of wallets");
96:        require(_wallets.length <= 30, "Exceeded the maximum number of wallets");
```

Securitize: Acknowledged.

7.4.8 Remove obsolete `return` statements when already using named return variables

Description: Remove obsolete `return` statements when already using named return variables.

- `contracts/utils/BulkBalanceChecker.sol`

```
43:        return balances;
```

- `contracts/swap/SecuritizeSwap.sol`

```
236:        return (dsTokenAmount, currentNavRate);
```

Securitize: Fixed in commit f3daea2 for `BulkBalanceChecker`; `SecuritizeSwap` was deleted as it is obsolete.

Cyfrin: Verified.

7.4.9 Prefer `ECDSA::tryRecover` to using `ecrecover` directly

Description: `ecrecover` is susceptible to signature malleability so it is not recommended to use it directly.

Prefer ECDSA::tryRecover in `MultiSigWallet` and `TransactionRelayer`.

Securitize: `TransactionRelayer` was significantly refactored and now uses OZ `ECDSA`. `MultiSigWallet` was removed as it was deprecated.

Cyfrin: Verified.

7.4.10 `TransactionRelayer` and `SecuritizeSwap` should use `CommonUtils::encodeString`

Description: `TransactionRelayer::toBytes32` duplicates functionality already available in `CommonUtils::encodeString`; remove the duplicated code and use `CommonUtils::encodeString` instead.

Also this line should use `CommonUtils::encodeString` instead of re-implementing it again:

```
L178:                       keccak256(abi.encodePacked(senderInvestor)),
```

`SecuritizeSwap` also duplicates this functionality:

```
191:                keccak256(abi.encodePacked(_senderInvestorId))
```

Securitize: Fixed in commit 1f69125 for `TransactionRelayer`. `SecurtizeSwap` was removed as it was deprecated.

Cyfrin: Verified.

7.4.11 Not mechanism to automatically cleanup locks that are already unlock-able on LockManagers

Description: `InvestorLockManager` as well as `LockManager` contracts don't have a mechanism to automatically clean up unlock-able locks; it is only possible to remove locks via manual intervention using the function `removeLockRecord` or `removeLockRecordForInvestor`, none of which validates if the locking period has already passed or not.

Impact: The lack of a mechanism to automatically clear up already unlocked locks can cause multiple problems/inefficiencies, such as:

- Wasting gas when transferring tokens by iterating over and over on already unlocked locks
- Admin errors when deleting locks that could unintentionally remove the lock for a lockIndex that is actually locked and should remain as is.

Recommended Mitigation: Consider implementing a mechanism to automatically clean up already unlocked locks, similar to how the investors Issuances are automatically cleaned up by `ComplianceServiceRegulated::cleanUpInvestorIssuances`.

Securitize: Acknowledged.

7.4.12 `SecuritizeSwap::buy` should revert if `stableCoinAmount` is zero

Description: `SecuritizeSwap::buy` should revert if `stableCoinAmount` is zero as otherwise it means that the investor can buy `dsToken` by paying zero stable coins; this could be possible if the investor calls `buy` with a small enough `_dsTokenAmount` to trigger rounding down to zero inside `calculateStableCoinAmount`.

With `DSToken` being configured with 2 decimals this rounding down to zero won't occur in `calculateStableCoinAmount`, but if `DSToken` is ever configured with standard 18 decimals this code would become vulnerable to free minting of tokens.

Recommended Mitigation:

```
    function buy(uint256 _dsTokenAmount, uint256 _maxStableCoinAmount) public override whenNotPaused {
        require(IDSRegistryService(getDSService(REGISTRY_SERVICE)).isWallet(msg.sender), "Investor not
        ↪  registered");
        require(_dsTokenAmount > 0, "DSToken amount must be greater than 0");
        require(navProvider.rate() > 0, "NAV Rate must be greater than 0");

        uint256 stableCoinAmount = calculateStableCoinAmount(_dsTokenAmount);
+       require(stableCoinAmount != 0, "Paying zero not allowed");
```

Securitize: `SecuritizeSwap` was removed as it was deprecated.

Cyfrin: Verified.

7.4.13 Possible to escape burning of DSTokens by removing the wallet from the current investor and adding it as the wallet of another investor

Description: Investors owning multiple investor accounts can split the `balanceOfInvestor()` and `balanceOf()` in their accounts by removing the wallet from the investor when issuing tokens and then using another investor account to add that wallet as its own.

- This makes that the `balanceOfInvestor()` for the issued tokens to remain accounted for the investor1, and the `balanceOf()` pointing to the actual wallet.

Putting the contract in that state allows investors to escape burning operations because the burning execution would result in underflow when attempting to burn the `investorsBalance` of the current investor that owns the wallet from where tokens should be burnt.

```
    function burn(
        ...
    ) public returns (uint256) {
        ...
        //@audit-info => here will occur the underflow because the current investor owning the `who`
        ↪  wallet has not the investorBalance required to decrement `sharesToBurn` from it
        updateInvestorBalance(
            _tokenData,
            IDSRegistryService(_services[REGISTRY_SERVICE]),
            _who,
            sharesToBurn,
            CommonUtils.IncDec.Decrease
        );

        ...
    }
```

Impact: Investors can escape burning of their DSTokens.

Proof of Concept: Add the next PoC in `dstoken-regulated.test.ts`

```
   it.only('escaping burning by forcing an underflow', async function () {
        const [owner, investor1, investor2, secondWalletInvestor1] = await hre.ethers.getSigners();
        const { registryService, dsToken } = await loadFixture(deployDSTokenRegulated);

        // Setup: Register two investors
        await registryService.registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1,
        ↪  INVESTORS.INVESTOR_ID.INVESTOR_COLLISION_HASH_1);
        await registryService.registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2,
        ↪  INVESTORS.INVESTOR_ID.INVESTOR_COLLISION_HASH_2);

        // Setup: Add wallets to investors
        await registryService.addWallet(investor1, INVESTORS.INVESTOR_ID.INVESTOR_ID_1);
        await registryService.addWallet(investor2, INVESTORS.INVESTOR_ID.INVESTOR_ID_2);

        const registryServiceFromInvestor1 = await registryService.connect(investor1);
        await registryServiceFromInvestor1.addWalletByInvestor(secondWalletInvestor1.address);

        // Setup: Issue tokens to secondWalletInvestor1
        const issueTokens = 500;
        await dsToken.setCap(1000);
        await dsToken.issueTokens(secondWalletInvestor1.address, issueTokens);

        expect(await dsToken.balanceOf(secondWalletInvestor1.address)).to.equal(issueTokens);

        // secondWalletInvestor1 is removed as a wallet of investor1
        await registryService.removeWallet(secondWalletInvestor1.address,
        ↪   INVESTORS.INVESTOR_ID.INVESTOR_ID_1);

        // investor2 adds secondWalletInvestor1 wallet as its own
        const registryServiceFromInvestor2 = await registryService.connect(investor2);
        await registryServiceFromInvestor2.addWalletByInvestor(secondWalletInvestor1.address);

        //@audit-info => attempting to burn tokens from secondWalletInvestor1 fails because of underflow
        ↪   when reducing the investorsBalance of the investor2
        //@audit-info => reverts because of underflow
        await expect(dsToken.burn(secondWalletInvestor1.address, issueTokens, "")).to.be.reverted;

        expect(await
        ↪   dsToken.balanceOfInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1)).to.equal(issueTokens);
        expect(await dsToken.balanceOf(secondWalletInvestor1.address)).to.equal(issueTokens);
    });
```

Securitize: Acknowledged but in reality this is not possible since the same entity can't control multiple investor accounts; we prevent this during the KYC/onboarding process.

7.4.14 Refactor duplicated checks into modifiers

Description: Refactor duplicated checks into modifiers:

- `contracts/compliance/LockManager.sol`

```
// 1) invalid address
compliance/InvestorLockManager.sol
60:        require(_to != address(0), "Invalid address");
108:        require(_to != address(0), "Invalid address");
132:        require(_who != address(0), "Invalid address");
152:        require(_who != address(0), "Invalid address");

compliance/LockManager.sol
96:        require(_to != address(0), "Invalid address");
109:        require(_to != address(0), "Invalid address");
144:        require(_who != address(0), "Invalid address");
159:        require(_who != address(0), "Invalid address");

token/TokenLibrary.sol
76:        require(_params._to != address(0), "Invalid address");
142:        require(_from != address(0), "Invalid address");
143:        require(_to != address(0), "Invalid address");

// 2) invalid time
compliance/ComplianceServiceNotRegulated.sol
65:        require(_time > 0, "Time must be greater than zero");

compliance/InvestorLockManager.sol
177:        require(_time > 0, "Time must be greater than zero");

compliance/ComplianceServiceWhitelisted.sol
116:        require(_time > 0, "Time must be greater than zero");

compliance/LockManager.sol
169:        require(_time > 0, "Time must be greater than zero");
```

```
compliance/ComplianceServiceRegulated.sol
706:        require(_time != 0, "Time must be greater than zero");

// 3) max number of wallets
compliance/WalletManager.sol
74:        require(_wallets.length <= 30, "Exceeded the maximum number of wallets");
96:        require(_wallets.length <= 30, "Exceeded the maximum number of wallets");
```

Securitize: Cyfrin:

7.4.15 `InvestorLockManager::createLockForInvestor`, `removeLockRecordForInvestor` should revert for invalid investor id

Description: `InvestorLockManager::createLockForInvestor` should revert for invalid investor id:

```
    function createLockForInvestor(string memory _investor, uint256 _valueLocked, uint256 _reasonCode,
    ↪   string calldata _reasonString, uint256 _releaseTime)
        public
        override
        validLock(_valueLocked, _releaseTime)
        onlyTransferAgentOrAboveOrToken
    {
+    require(!CommonUtils.isEmptyString(_investor), "Unknown investor");
```

The same applies to `removeLockRecordForInvestor` - perhaps create a modifier and use that modifier on both functions.

Securitize: Acknowledged; while this is true, it also allows us to fully lock an investorId BEFORE it actually gets created on chain. There are cases where we know the investor id beforehand and in that case we could use this.

7.4.16 Refactor away duplicated code between `ComplianceService::newPreTransferCheck` and `preTransferCheck`

Description: `ComplianceService::newPreTransferCheck` and `preTransferCheck` do exactly the same thing, the only difference is that:

- `newPreTransferCheck` takes `balanceFrom` and `paused` as input parameters
- `preTransferCheck` doesn't and has to look them up

So to remove code duplication, `preTransferCheck` should call `newPreTransferCheck` with the parameters it looks up eg:

```
    function preTransferCheck(
        address _from,
        address _to,
        uint256 _value
    ) public view virtual override returns (uint256 code, string memory reason) {
        IDSToken token = getToken();
        return newPreTransferCheck(_from, _to, _value, token.balanceOf(_from), token.isPaused());
    }
```

Securitize: Fixed in commit 3b1894a.

Cyfrin: Verified.

7.4.17 Cache compliance service and compliance configuration service for much cleaner code in `ComplianceServiceRegulated::completeTransferCheck`

Description: `ComplianceServiceRegulated::completeTransferCheck` has a lot of `if` statements which are very large and hard to understand as they are full of `ComplianceServiceRegulated(_services[COMPLIANCE_SER-`

VICE]) and IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]).

This can be easily improved by caching those two values into local variables:

```
ComplianceServiceRegulated complServiceReg = ComplianceServiceRegulated(_services[COMPLIANCE_SERVICE]);
IDSComplianceConfigurationService complConfigService
    = IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]);
```

Then in the `if` conditions just use `complServiceReg` and `complConfigService` which massively simplifies them vastly improving readability.

Securitize: Acknowledged; this was analyzed in the past and there is an intentional balance when using local variables. Historically, we've had many issues with errors like "Stack Too Deep" when using too many of them, and they did not improve readability or gas usage too much. We prefer to not go through this process again because it's currently working as intended.

7.4.18 `ComplianceServiceRegulated::getServices` should use constants from `ComplianceServiceLibrary` when setting array indexes

Description: `ComplianceServiceRegulated::getServices` should use constants from `ComplianceServiceLibrary` when setting array indexes:

```
function getServices() internal view returns (address[] memory services) {
    services = new address[](6);
    services[ComplianceServiceLibrary.DS_TOKEN] = getDSService(DS_TOKEN);
    services[ComplianceServiceLibrary.REGISTRY_SERVICE] = getDSService(REGISTRY_SERVICE);
    services[ComplianceServiceLibrary.WALLET_MANAGER] = getDSService(WALLET_MANAGER);
    services[ComplianceServiceLibrary.COMPLIANCE_CONFIGURATION_SERVICE] =
    ↪  getDSService(COMPLIANCE_CONFIGURATION_SERVICE);
    services[ComplianceServiceLibrary.LOCK_MANAGER] = getDSService(LOCK_MANAGER);
    services[ComplianceServiceLibrary.COMPLIANCE_SERVICE] = address(this);
}
```

Securitize: Acknowledged.

7.4.19 Using `block.number` instead of `block.timestamp` as an expiration check to execute signature

Description: On `TransactionRelayer::executeByInvestorWithBlockLimit()`, the 3rd value in the `params` array is compared against the `block.number`, the purpose is to prevent executing old signatures. Using `block.number` is not recommended when enforcing time-dependent operations.

```
    function executeByInvestorWithBlockLimit(
        ...
        uint256[] memory params
    ) public {
        ...
        require(params[2] >= block.number, "Transaction too old");
        ...
    }
```

It is best to use `block.timestamp` because the `block.number` time length varies from chain to chain.

Securitize:

Cyfrin:

7.4.20 Remove useless function `ComplianceServiceRegulated::adjustTransferCounts`

Description: The function `ComplianceServiceRegulated::adjustTransferCounts` is useless as it always just calls `adjustTotalInvestorsCounts` with its two parameters; remove it and simply call `adjustTotalInvestorsCounts` direct:

```diff
    function recordTransfer(
        address _from,
        address _to,
        uint256 _value
    ) internal override returns (bool) {
        if (compareInvestorBalance(_to, _value, 0)) {
-            adjustTransferCounts(_to, CommonUtils.IncDec.Increase);
+            adjustTotalInvestorsCounts(_to, CommonUtils.IncDec.Increase);
        }

        if (compareInvestorBalance(_from, _value, _value)) {
            adjustTotalInvestorsCounts(_from, CommonUtils.IncDec.Decrease);
        }

        cleanupInvestorIssuances(_from);
        cleanupInvestorIssuances(_to);
        return true;
    }

-    function adjustTransferCounts(
-        address _from,
-        CommonUtils.IncDec _increase
-    ) internal {
-        adjustTotalInvestorsCounts(_from, _increase);
-    }
```

Securitize: Fixed in commit 5e524cd.

Cyfrin: Verified.

7.4.21 Protocol classifies retail investors based solely on whether they are qualified investors, without checking if they are accredited investors

Description: The protocol classifies "retail investors" based solely on whether they are qualified investors, without checking if they are accredited investors. Throughout the codebase, retail investors are identified using `!RegistryService::isQualifiedInvestor()`, but for US investors the correct definition would be `!isQualifiedInvestor() && !isAccreditedInvestor()`.

Impact: In the current configuration:

- `isRetail` and the other related checks are only ever performed for EU investors
- non-retail EU investors are marked using the `Qualified` attribute

So while the current code appears to be fine as it is only used for EU investors, a bug could easily be introduced in the future if a developer calls `isRetail` in relation to US investors or adopts the pattern in other places but for US investors.

Recommended Mitigation: Considering either renaming `isRetail` or adding an explanatory comment directly above it to indicate that this function should only be used for EU investors.

Securitize: This is the correct behavior as this function should only be used for EU investors; in commit 6aef381 we've added comments explaining this.

Cyfrin: Verified.

7.4.22 Not possible to rehash `DOMAIN_SEPARATOR` in `MultiSigWallet` to update chainId if is ever required

Description: `MultiSigWallet` does not have a function to rehash the `DOMAIN_SEPARATOR` in case it is required to update the chainId, which was used for the initial DOMAIN_SEPARATOR when the contract was deployed.

Recommended Mitigation: Consider adding a function similar to `TransactionRelayer::updatedomainSeparator()` to allow updating the `DOMAIN_SEPARATOR` in the MultiSigWallet

Securitize: `MultiSigWallet` was removed as it is deprecated.

Cyfrin: Verified.

7.4.23 `ComplianceConfigurationService::getWorldWideForceFullTransfer` is not applied to US investors

Description: `ComplianceConfigurationService::getWorldWideForceFullTransfer` function name implies global application across all regions, but the implementation only applies this setting to non-US investors. In the `completeTransferCheck` function, the worldwide setting is checked exclusively in the non-US investor code path, while US investors are only subject to the region-specific `getForceFullTransfer()` setting:

```
if (_args.fromRegion == US) {
            ...
             }
        } else {
            ...
                IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]).getWorld ⌋
                ↪  WideForceFullTransfer() &&
                _args.fromInvestorBalance > _args.value
            ) { //@audit the getgetWorldWideForceFullTransfer is not being checked for us investor?
                return (50, ONLY_FULL_TRANSFER);
            }
        }
```

This creates a fundamental inconsistency where "worldwide" compliance settings do not actually apply worldwide.

Impact: Us investor can bypass `ComplianceConfigurationService::getWorldWideForceFullTransfer` restrictions.

Proof of Concept: Run the next proof of concept in `dstoken-regulated.test.ts`

```
describe('Worldwide Force Full Transfer Bug POC', function () {
        it('Should demonstrate that getWorldWideForceFullTransfer is not applied to US investors',
        ↪  async function () {
          const [usInvestor, nonUsInvestor, differentInvestor] = await hre.ethers.getSigners();
          const { dsToken, registryService, complianceService, complianceConfigurationService } = await
          ↪  loadFixture(deployDSTokenRegulatedWithRebasingAndEighteenDecimal);

          // Setup: Register investors
          await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, usInvestor, registryService);
          await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, nonUsInvestor, registryService);
          await registerInvestor(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2, differentInvestor,
          ↪  registryService);

          // Set countries
          await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, INVESTORS.Country.USA);
          await registryService.setCountry(INVESTORS.INVESTOR_ID.INVESTOR_ID_2,
          ↪  INVESTORS.Country.GERMANY);
          await registryService.setCountry(INVESTORS.INVESTOR_ID.US_INVESTOR_ID_2,
          ↪  INVESTORS.Country.FRANCE);

          // Set country compliance
          await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.USA, 1); // US =
          ↪  compliant
          await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.GERMANY, 2); //
          ↪  Germany = EU
          await complianceConfigurationService.setCountryCompliance(INVESTORS.Country.FRANCE, 2); //
          ↪  France = EU
          // Set all compliance rules - no limits and short lock period for testing
          await complianceConfigurationService.setAll(
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 150, 1, 1, 0], // No investor limits, 1 second lock
            ↪  period
            [false, false, false, false, false] // Disable all compliance checks initially
          );

          // Issue tokens to both investors
          await dsToken.issueTokens(usInvestor.address, 200);
          await dsToken.issueTokens(nonUsInvestor.address, 200);

          // Wait for lock period to expire
          await time.increase(2); // Wait 2 seconds to ensure lock period expires

          // Configure compliance settings
          await complianceConfigurationService.setForceFullTransfer(false); // Disable US-specific force
          ↪  full transfer
          await complianceConfigurationService.setWorldWideForceFullTransfer(true); // Enable worldwide
          ↪  force full transfer

          // Verify balances
          expect(await dsToken.balanceOf(usInvestor.address)).to.equal(200);
          expect(await dsToken.balanceOf(nonUsInvestor.address)).to.equal(200);

          // Non-US investor partial transfer - should fail (correct behavior)
          await expect(
        dsToken.connect(nonUsInvestor).transfer(differentInvestor.address, 100)
          ).to.be.revertedWith('Only full transfer');

           // US investor partial transfer - should succeed due to bug
           await dsToken.connect(usInvestor).transfer(differentInvestor.address, 100);
           expect(await dsToken.balanceOf(usInvestor.address)).to.equal(100);
           expect(await dsToken.balanceOf(differentInvestor.address)).to.equal(100);

        });
      });
```

Recommended Mitigation: Ensure `ComplianceConfigurationService::getWorldWideForceFullTransfer` truly applies worldwide by modifying the US investor code path to also check the worldwide setting:.

Securitize: Acknowledged; by design.

7.4.24 `ComplianceServiceRegulated` and its parent `ComplianceServiceWhitelisted` uses a chain of `initializer` modifiers when calling the `initialize`

Description: `ComplianceServiceRegulated` is the child contract inheriting from `ComplianceServiceWhitelisted`. The `ComplianceServiceRegulated::initialize()` uses the `initializer modifier` as well as the `ComplianceServiceWhitelisted:initialize()`.

```
contract ComplianceServiceRegulated is ComplianceServiceWhitelisted {
    function initialize() public virtual override onlyProxy initializer {
        super.initialize();
    }
}

contract ComplianceServiceWhitelisted is ComplianceService {
    function initialize() public virtual override onlyProxy initializer {
        ComplianceService.initialize();
    }
}
```

According to OpenZeppelin's documentation and best practices, the initializer modifier should only be used in the final initialization function of an inheritance chain, while initialization functions of parent contracts should use the onlyInitializing modifier. This ensures proper initialization when using inheritance.

Recommended Mitigation: * change `ComplianceServiceWhitelisted` to have this:

```
contract ComplianceServiceWhitelisted is ComplianceService {

    function initialize() public virtual override onlyProxy initializer {
        _initialize();
    }

    function _initialize() internal onlyInitializing {
        ComplianceService.initialize();
    }
```

- change `ComplianceServiceRegulated` to have this:

```
contract ComplianceServiceRegulated is ComplianceServiceWhitelisted {

    function initialize() public virtual override onlyProxy initializer {
        _initialize();
    }
```

Securitize: Fixed in commit b24ecd5.

Cyfrin: Verified.

7.4.25 Feature flags are not being used in the `TokenLibrary`

Description: `TokenLibrary::setFeature` implements a feature flag system that is completely non-functional in the current codebase. While the infrastructure exists to enable/disable features, no business logic actually checks these feature flags; consider removing this code if it is deprecated or obsolete.

```
 function setFeature(SupportedFeatures storage supportedFeatures, uint8 featureIndex, bool enable)
 ↪  public {
        uint256 base = 2;
        uint256 mask = base**featureIndex;

        // Enable only if the feature is turned off and disable only if the feature is turned on
        if (enable && (supportedFeatures.value & mask == 0)) {
            supportedFeatures.value = supportedFeatures.value ^ mask;
        } else if (!enable && (supportedFeatures.value & mask >= 1)) {
            supportedFeatures.value = supportedFeatures.value ^ mask;
        }
    }
```

Securitize: Acknowledged.

7.5 Gas Optimization

7.5.1 Don't perform storage reads unless necessary

Description: Reading from storage is expensive; don't perform storage reads unless necessary.

- `contracts/service/ServiceConsumer.sol`

```
// in `onlyMaster` don't load trust manager unless required
    modifier onlyMaster {
-        IDSTrustService trustManager = getTrustService();
-        require(owner() == msg.sender || trustManager.getRole(msg.sender) == ROLE_MASTER, "Insufficient
↪   trust level");
+        if(owner() != msg.sender) require(getTrustService().getRole(msg.sender) == ROLE_MASTER,
↪   "Insufficient trust level");
        _;
    }
```

Securitize: Fixed in commit 1ab6fd9.

Cyfrin: Verified.

7.5.2 Cache result of identical external calls when the result can't change

Description: Cache result of identical external calls when the result can't change.

- `contracts/service/ServiceConsumer.sol`

```
// in the modifiers, cache `trustManager.getRole(msg.sender)` and
// use the cached value inside the `require` statements
59:        require(trustManager.getRole(msg.sender) == ROLE_TRANSFER_AGENT ||
↪   trustManager.getRole(msg.sender) == ROLE_ISSUER || trustManager.getRole(msg.sender) == ROLE_MASTER,
↪   "Insufficient trust level");
65:        require(trustManager.getRole(msg.sender) == ROLE_ISSUER || trustManager.getRole(msg.sender)
↪   == ROLE_MASTER, "Insufficient trust level");
71:        require(trustManager.getRole(msg.sender) == ROLE_TRANSFER_AGENT ||
↪   trustManager.getRole(msg.sender) == ROLE_MASTER, "Insufficient trust level");
77:        require(
78:            trustManager.getRole(msg.sender) == ROLE_EXCHANGE
79:            || trustManager.getRole(msg.sender) == ROLE_ISSUER
80:            || trustManager.getRole(msg.sender) == ROLE_TRANSFER_AGENT
81:            || trustManager.getRole(msg.sender) == ROLE_MASTER,
100:        require(trustManager.getRole(msg.sender) == ROLE_ISSUER ||
↪   trustManager.getRole(msg.sender) == ROLE_MASTER, "Insufficient trust level");
108:        require(trustManager.getRole(msg.sender) == ROLE_TRANSFER_AGENT ||
↪   trustManager.getRole(msg.sender) == ROLE_MASTER, "Insufficient trust level");
116:        require(trustManager.getRole(msg.sender) == ROLE_ISSUER ||
↪   trustManager.getRole(msg.sender) == ROLE_MASTER, "Insufficient trust level");
```

- `contracts/swap/SecuritizeSwap.sol`

```
// cache `navProvider.rate` in `buy`, also change `calculateStableCoinAmount` to take the rate
// as an input to save another call inside it
L132:        require(navProvider.rate() > 0, "NAV Rate must be greater than 0");
L141:        emit Buy(msg.sender, _dsTokenAmount, stableCoinAmount, navProvider.rate());
L240:        return _dsTokenAmount * navProvider.rate() / (10 ** ERC20(address(dsToken)).decimals());
```

- `contracts/compliance/ComplianceServiceRegulated.sol`

```
// cache these two external calls in `getUSInvestorsLimit`
103:        if (compConfService.getMaxUSInvestorsPercentage() == 0) {
107:        if (compConfService.getUSInvestorsLimit() == 0) {

// cache `getInvestor(_to)` in `preIssuanceCheck`
```

```
420:        string memory toCountry = IDSRegistryService(_services[REGISTRY_SERVICE]).getCountry(IDSReg
↪  istryService(_services[REGISTRY_SERVICE]).getInvestor(_to));
431:        if (IDSLockManager(_services[LOCK_MANAGER]).isInvestorLiquidateOnly(IDSRegistryService(_ser
↪  vices[REGISTRY_SERVICE]).getInvestor(_to))) {
```

Securitize: Acknowledged.

7.5.3 Use `calldata` for read-only external inputs

Description: Use `calldata` for read-only external inputs:

- `BulkOperator::bulkIssuance`, `bulkRegisterAndIssuance`, `bulkBurn`
- `TokenIssuer::issueTokens`, `registerInvestor`
- `SecuritizeSwap::nonceByInvestor`, `swap`, `executePreApprovedTransaction`, `doExecuteByInvestor`, `_registerNewInvestor`
- `WalletManager::addIssuerWallets`, `addPlatformWallets`
- `InvestorLockManagerBase::lockInvestor`, `unlockInvestor`, `isInvestorLocked`, `setInvestorLiquidateOnly`, `isInvestorLiquidateOnly` (and similar functions in related classes `IDSLockManager`, `LockManager`)
- `ComplianceConfigurationService::getCountryCompliance`
- `ComplianceServiceRegulated::doPreTransferCheckRegulated`, `completeTransferCheck` for `_services`,

Securitize: We:

- deleted a number of these contracts as they were obsolete
- used `calldata` in some places where this wouldn't result in a "stack-too-deep" error.

Cyfrin: Verified.

7.5.4 In Solidity don't initialize to default values

Description: In Solidity don't initialize to default values:

```
multicall/IssuerMulticall.sol
31:        for (uint256 i = 0; i < data.length; i++) {

service/ServiceConsumer.sol
38:    uint8 public constant ROLE_NONE = 0;

compliance/ComplianceConfigurationService.sol
34:        for (uint i = 0; i < _countries.length; i++) {

compliance/ComplianceServiceRegulated.sol
27:    uint256 internal constant DS_TOKEN = 0;
34:    uint256 internal constant NONE = 0;
718:        uint256 totalLockedTokens = 0;
720:        for (uint256 i = 0; i < investorIssuancesCount; i++) {
815:        uint256 currentIndex = 0;

trust/TrustService.sol
218:        for (uint i = 0; i < _addresses.length; i++) {

compliance/IDSComplianceService.sol
23:    uint256 internal constant NONE = 0;

trust/IDSTrustService.sol
40:    uint8 public constant NONE = 0;

compliance/WalletManager.sol
```

```
75:        for (uint i = 0; i < _wallets.length; i++) {
97:        for (uint i = 0; i < _wallets.length; i++) {

compliance/InvestorLockManager.sol
190:        uint256 totalLockedTokens = 0;
191:        for (uint256 i = 0; i < investorLockCount; i++) {

registry/WalletRegistrar.sol
48:        for (uint256 i = 0; i < _wallets.length; i++) {
56:        for (uint256 i = 0; i < _attributeIds.length; i++) {

registry/IDSRegistryService.sol
34:    uint8 public constant NONE = 0;
40:    uint8 public constant PENDING = 0;

registry/RegistryService.sol
43:        for (uint8 index = 0; index < 16; index++) {
74:        for (uint256 i = 0; i < _wallets.length; i++) {
82:        for (uint256 i = 0; i < _attributeIds.length; i++) {
99:        for (uint8 i = 0; i < 4; i++) {

token/DSToken.sol
29:    uint256 internal constant DEPRECATED_OMNIBUS_NO_ACTION = 0;  // Deprecated, kept for backward
↪   compatibility

token/TokenLibrary.sol
29:    uint256 internal constant COMPLIANCE_SERVICE = 0;
31:    uint256 internal constant DEPRECATED_OMNIBUS_NO_ACTION = 0; // Deprecated, keep for backwards
↪   compatibility
96:        uint256 totalLocked = 0;
97:        for (uint256 i = 0; i < _params._valuesLocked.length; i++) {

swap/SecuritizeSwap.sol
222:        for (uint256 i = 0; i < _investorAttributeIds.length; i++) {

compliance/ComplianceServiceNotRegulated.sol
48:        code = 0;

utils/BulkBalanceChecker.sol
39:        for (uint256 i = 0; i < length; i++) {

utils/MultiSigWallet.sol
61:        for (uint256 i = 0; i < owners_.length; i++) {
113:        for (uint256 i = 0; i < threshold; i++) {
123:        bool success = false;

compliance/LockManager.sol
180:        uint256 totalLockedTokens = 0;
181:        for (uint256 i = 0; i < investorLockCount; i++) {

compliance/IDSWalletManager.sol
26:    uint8 public constant NONE = 0;

bulk/BulkOperator.sol
54:        for (uint256 i = 0; i < addresses.length; i++) {
61:        for (uint256 i = 0; i < data.length; i++) {
80:        for (uint256 i = 0; i < addresses.length; i++) {

utils/TransactionRelayer.sol
191:        bool success = false;
```

Securitize: Acknowledged.

7.5.5 Cache storage to prevent identical storage reads

Description: Reading from storage is expensive; cache storage to prevent identical storage reads:

- `contracts/registry/RegistryService.sol`:

```
// cache `getInvestor(_address)` in `getInvestorDetails`
206:        return (getInvestor(_address), getCountry(getInvestor(_address)));
```

- `contracts/issuance/TokenIssuer.sol`

```
// cache `getRegistryService()` in `issueTokens` and `registerInvestor`
44:        if (getRegistryService().isWallet(_to)) {
45:            require(CommonUtils.isEqualString(getRegistryService().getInvestor(_to), _id), "Wallet
↪   does not belong to investor");
63:        if (!getRegistryService().isInvestor(_id)) {
64:            getRegistryService().registerInvestor(_id, _collisionHash);
65:            getRegistryService().setCountry(_id, _country);
69:                getRegistryService().setAttribute(_id, KYC_APPROVED, _attributeValues[0],
↪   _attributeExpirations[0], "");
70:                getRegistryService().setAttribute(_id, ACCREDITED, _attributeValues[1],
↪   _attributeExpirations[1], "");
71:                getRegistryService().setAttribute(_id, QUALIFIED, _attributeValues[2],
↪   _attributeExpirations[2], "");
```

- `contracts/token/TokenLibrary.sol`

```
// cache `supportedFeatures.value` in `setFeature`
62:        if (enable && (supportedFeatures.value & mask == 0)) {
63:            supportedFeatures.value = supportedFeatures.value ^ mask;
64:        } else if (!enable && (supportedFeatures.value & mask >= 1)) {
65:            supportedFeatures.value = supportedFeatures.value ^ mask;

// This function can also delete the `base` variable since it is hard-coded and
// only used once so no point to it.
```

- `contracts/token/StandardToken.sol`

```
// cache `allowances[msg.sender][_spender] + _addedValue` then use it
// when writing storage and emitting event. Do something similar in `decreaseApproval`
// to avoid re-reading storage when emitting the event
128:        allowances[msg.sender][_spender] = allowances[msg.sender][_spender] + _addedValue;
129:        emit Approval(msg.sender, _spender, allowances[msg.sender][_spender]);
```

- `contracts/trust/TrustService.sol`

```
// cache `roles[msg.sender]` in modifiers
70:        require(roles[msg.sender] == MASTER || roles[msg.sender] == ISSUER, "Not enough
↪   permissions");
86:        if (roles[msg.sender] != MASTER) {
87:            if (roles[msg.sender] == ISSUER) {
90:                require(roles[msg.sender] == _role, "Not enough permissions. Only same role
↪   allowed");
100:        if (roles[msg.sender] != MASTER) {
102:            if (roles[msg.sender] == ISSUER) {
105:                require(roles[msg.sender] == role, "Not enough permissions. Only same role
↪   allowed");

// cache `roles[msg.sender]` and `ownersEntities[msg.sender]` in `onlyEntityOwnerOrAbove`
// ideally here perform the `roles[msg.sender]` checks first and only perform the
↪   `ownersEntities[msg.sender]`
// afterwards if required
113:            roles[msg.sender] == MASTER ||
114:                roles[msg.sender] == ISSUER ||
115:                (!CommonUtils.isEmptyString(ownersEntities[msg.sender]) &&
116:                  CommonUtils.isEqualString(ownersEntities[msg.sender], _name)),

// cache `ownersEntities[_owner]` in `onlyExistingEntityOwner`
140:            !CommonUtils.isEmptyString(ownersEntities[_owner]) &&
141:            CommonUtils.isEqualString(ownersEntities[_owner], _name),

// cache `operatorsEntities[_operator]` in `onlyExistingOperator`
154:            !CommonUtils.isEmptyString(operatorsEntities[_operator]) &&
155:            CommonUtils.isEqualString(operatorsEntities[_operator], _name),

// cache `resourcesEntities[_resource]` in `onlyExistingResource`
168:            !CommonUtils.isEmptyString(resourcesEntities[_resource]) &&
169:           CommonUtils.isEqualString(resourcesEntities[_resource], _name),
```

- contracts/swap/SecuritizeSwap.sol

```
// cache `IDSRegistryService(getDSService(REGISTRY_SERVICE))` in `swap`
// pass it in as a parameter to `_registerNewInvestor` to save another identical storage read
103:        if (!IDSRegistryService(getDSService(REGISTRY_SERVICE)).isInvestor(_senderInvestorId)) {
114:        string memory investorWithNewWallet =
↪   IDSRegistryService(getDSService(REGISTRY_SERVICE)).getInvestor(_newInvestorWallet);
116:            IDSRegistryService(getDSService(REGISTRY_SERVICE)).addWallet(_newInvestorWallet,
↪   _senderInvestorId);
214:        IDSRegistryService registryService = IDSRegistryService(getDSService(REGISTRY_SERVICE));

// cache `USDCBridge` and `bridgeChainId` in `executeStableCoinTransfer`
// a more optimized implementation looks like this:
function executeStableCoinTransfer(address from, uint256 value) private {
    // 1 SLOAD since both stored in the same slot
    (IUSDCBridge USDCBridgeCache, uint16 bridgeChainIdCache) = (USDCBridge, bridgeChainId);

    if (bridgeChainIdCache != 0 && address(USDCBridgeCache) != address(0)) {
        stableCoinToken.transferFrom(from, address(this), value);
        stableCoinToken.approve(address(USDCBridgeCache), value);
        USDCBridgeCache.sendUSDCCrossChainDeposit(bridgeChainIdCache, issuerWallet, value);
    } else {
        stableCoinToken.transferFrom(from, issuerWallet, value);
    }
}
```

- contracts/compliance/ComplianceServiceWhitelisted.sol

```
// cache `getToken()` in `preTransferCheck`
50:        return doPreTransferCheckWhitelisted(_from, _to, _value, getToken().balanceOf(_from),
↪   getToken().isPaused());
```

- contracts/compliance/ComplianceServiceRegulated.sol

```
// cache `getRegistryService()` and `getComplianceConfigurationService()` in `recordTransfer`
// ideally these would be cached upstream and passed down to functions such as `recordTransfer`
// that need them
800:        string memory investor = getRegistryService().getInvestor(_who);
801:        string memory country = getRegistryService().getCountry(investor);

803:        uint256 region = getComplianceConfigurationService().getCountryCompliance(country);
807:            lockTime = getComplianceConfigurationService().getUSLockPeriod();
809:            lockTime = getComplianceConfigurationService().getNonUSLockPeriod();
```

Securitize: Acknowledged.

7.5.6 Use named return variables where this optimizes away a local variable definition

Description: Use named return variables where this optimizes away a local variable definition, then also remove the final obsolete `return` statement:

- `StandardToken::balanceOf`, `totalSupply`
- `DSToken::totalIssued`, `balanceOfInvestor`, `getCommonServices`
- `TokenLibrary::issueTokensCustom`, `issueTokensWithNoCompliance`, `burn`
- `RegistryService::getInvestorDetailsFull`
- `SecuritizeSwap::calculateDsTokenAmount`
- `MulticallProxy::_slice`, `_callTarget`
- `LockManager::getTransferableTokens`
- `InvestorLockManager::getTransferableTokens`
- `ComplianceConfigurationService::getAll`

Securitize: Acknowledged.

7.5.7 Since attribute expiration is deprecated, remove as input parameters, don't write it to storage and put comment explaining this

Description: `RegistryService::setAttribute` sets a given `_expiry` field for each attribute at which point the attribute should expire.

However this is never checked anywhere, for example these functions which determine whether a user is accredited or qualified never check the expiry:

```
    function isAccreditedInvestor(string calldata _id) external view override returns (bool) {
        return getAttributeValue(_id, ACCREDITED) == APPROVED;
    }

    function isAccreditedInvestor(address _wallet) external view override returns (bool) {
        string memory investor = investorsWallets[_wallet].owner;
        return getAttributeValue(investor, ACCREDITED) == APPROVED;
    }

    function isQualifiedInvestor(address _wallet) external view override returns (bool) {
        string memory investor = investorsWallets[_wallet].owner;
        return getAttributeValue(investor, QUALIFIED) == APPROVED;
    }

    function isQualifiedInvestor(string calldata _id) external view override returns (bool) {
        return getAttributeValue(_id, QUALIFIED) == APPROVED;
    }
```

Asking the client they have said that attribute expiry is deprecated and not used anywhere, that in practice they are passing zeros for the expiry inputs.

Recommended Mitigation: Ideally remove all attribute inputs from functions, however this breaks existing interfaces so is more invasive.

At a minimum change `RegistryService::setAttribute` to never set `attributes[_id][_attributeId].expiry` (leaving it as default zero) and put a comment noting the deprecation in the relevant data store:

```
data-stores/RegistryServiceDataStore.sol
26:        uint256 expiry;
```

Securitize: Acknowledged.

7.5.8 Remove setting deprecated `lastUpdatedBy` in RegistryService

Description: The client has informed us that `lastUpdatedBy` is deprecated so it should not be updated in `RegistryService`:

```
registry/RegistryService.sol
113:        investors[_id].lastUpdatedBy = msg.sender;
140:        investors[_id].lastUpdatedBy = msg.sender;
```

Additionally a comment should be placed to indicate this in the relevant data store:

```
data-stores/RegistryServiceDataStore.sol
33:        address lastUpdatedBy;
40:        address lastUpdatedBy;
```

Or the storage slots should be renamed to `DEPRECATED_lastUpdatedBy` as has been done in other places for deprecated storage slots.

Securitize: Fixed in commit 9a80a47 by no longer writing to `lastUpdatedBy` and in commit e6165e4 by renaming the variable to explicitly indicate it is deprecated.

Cyfrin: Verified.

7.5.9 Cheaper not to cache `calldata` array length

Description: It is cheaper not to cache `calldata` array length:

- `BulkBalanceChecker::getTokenBalances`

Securitize: Fixed in commit fd6eb3b.

Cyfrin: Verified.

7.5.10 More efficient way of checking for empty string in `CommonUtils::isEmptyString`

Description: More efficient way of checking for empty string in `CommonUtils::isEmptyString`:

```
function isEmptyString(string memory _str) internal pure returns (bool) {
    return bytes(_str).length == 0;
}
```

Securitize: Fixed in commit 22b117a.

Cyfrin: Verified.

7.5.11 More efficient way of comparing two strings for equality in `CommonUtils::isEqualString`

Description: More efficient way of comparing two strings for equality in `CommonUtils::isEqualString` from Solady:

```
  function isEqualString(string memory a, string memory b) internal pure returns (bool result) {
      /// @solidity memory-safe-assembly
      assembly {
          result := eq(keccak256(add(a, 0x20), mload(a)), keccak256(add(b, 0x20), mload(b)))
      }
  }
```

Securitize: Acknowledged.

7.5.12 Cache computation results instead of repeatedly performing the same computation

Description: Cache computation results instead of repeatedly performing the same computation.

- `contracts/utils/TransactionRelayer.sol`

```
// cache `toBytes32(investorId)` in `setInvestorNonce`
142:        uint256 investorNonce = noncePerInvestor[toBytes32(investorId)];
144:        noncePerInvestor[toBytes32(investorId)] = newNonce;

// cache `toBytes32(senderInvestor)` in `doExecuteByInvestor`
175:                        noncePerInvestor[toBytes32(senderInvestor)],
178:                        keccak256(abi.encodePacked(senderInvestor)),
190:        noncePerInvestor[toBytes32(senderInvestor)]++;
```

Securitize: Acknowledged.

7.5.13 Remove deprecated collision hash and proof hash from function calls

Description: Colllision hash when registering users and proof hash when registering attributes have been deprecated; remove them from function interfaces.

Also in `RegistryServiceDataStore` add comments alongside `Attribute::proofHash` and `Investor::collisionHash` to note they are deprecated and no longer used.

Securitize: Acknowledged.

7.5.14 Return fast in `ComplianceServiceRegulated::checkHoldUp` if platform wallet

Description: `ComplianceServiceRegulated::checkHoldUp` should return fast if `_isPlatformWalletFrom == true`; there's no reason to do all the processing in that case:

```
    function checkHoldUp(
        address[] memory _services,
        address _from,
        uint256 _value,
        bool _isUSLockPeriod,
        bool _isPlatformWalletFrom
    ) internal view returns (bool hasHoldUp) {
        // platform wallets have no lock period so return false (default)
        // and skip all processing if it is a platform wallet
        if(!_isPlatformWalletFrom) {
            ComplianceServiceRegulated complianceService
                = ComplianceServiceRegulated(_services[COMPLIANCE_SERVICE]);
            uint256 lockPeriod;
            if (_isUSLockPeriod) {
                lockPeriod = IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]).getUSLockPeriod();
            } else {
                lockPeriod = IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]).getNonUSLockPeriod();
            }

            hasHoldUp =
                complianceService.getComplianceTransferableTokens(
                    _from, block.timestamp, uint64(lockPeriod)) < _value;
        }
    }
```

Securitize: Fixed in commit c407d0c.

Cyfrin: Verified.

7.5.15 Perform local variable checks first prior to external calls in composite `if` statement conditions

Description: When an `if` statement condition is joined together using `&&` operators from multiple composite parts, local variable checks should be performed first prior to external calls. This is because if the local variable checks evaluate to `false` there is no need to then perform the external calls.

Three places where this occurs is in `ComplianceServiceRegulated::completeTransferCheck`:

```
            if (IDSComplianceConfigurationService(
                    _services[COMPLIANCE_CONFIGURATION_SERVICE]).getForceFullTransfer() &&
                 _args.fromInvestorBalance > _args.value
            ) {

            if (IDSComplianceConfigurationService(.
                    _services[COMPLIANCE_CONFIGURATION_SERVICE]).getWorldWideForceFullTransfer() &&
                 _args.fromInvestorBalance > _args.value
            ) {
```

Instead perform the local variable checks first:

```
            if (_args.fromInvestorBalance > _args.value &&
                IDSComplianceConfigurationService(
                   _services[COMPLIANCE_CONFIGURATION_SERVICE]).getForceFullTransfer()
            ) {

            if (_args.fromInvestorBalance > _args.value &&
                IDSComplianceConfigurationService(
                   _services[COMPLIANCE_CONFIGURATION_SERVICE]).getWorldWideForceFullTransfer()
            ) {
```

Similar optimization can be applied to:

- L284->287 EU check
- L290->292 accreditation check

Securitize: Cyfrin:

7.5.16 Fast fail without performing unnecessary storage reads or external calls

Description: Fast fail without performing unnecessary storage reads or external calls. For example in `ComplianceServiceRegulated::preIssuanceCheck` the start of the function looks like this:

```
function preIssuanceCheck(
    address[] calldata _services,
    address _to,
    uint256 _value
) public view returns (uint256 code, string memory reason) {
    ComplianceServiceRegulated complianceService =
    ↪   ComplianceServiceRegulated(_services[COMPLIANCE_SERVICE]);
    IDSComplianceConfigurationService complianceConfigurationService =
    ↪   IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]);
    IDSWalletManager walletManager = IDSWalletManager(_services[WALLET_MANAGER]);
    string memory toCountry = IDSRegistryService(_services[REGISTRY_SERVICE]).getCountry(IDSRegistrySer
    ↪   vice(_services[REGISTRY_SERVICE]).getInvestor(_to));
    uint256 toRegion = complianceConfigurationService.getCountryCompliance(toCountry);

    if (toRegion == FORBIDDEN) {
        return (26, DESTINATION_RESTRICTED);
    }

    if (!complianceService.checkWhitelisted(_to)) {
        return (20, WALLET_NOT_IN_REGISTRY_SERVICE);
    }
```

But if the function is going to return because `_to` is not whitelisted, then it makes no sense to spend gas performing all the interim unrelated storage reads and external calls. Instead storage reads and external calls should only be made as they are needed, eg:

```
function preIssuanceCheck(
    address[] calldata _services,
    address _to,
    uint256 _value
) public view returns (uint256 code, string memory reason) {
    ComplianceServiceRegulated complianceService =
    ↪   ComplianceServiceRegulated(_services[COMPLIANCE_SERVICE]);
    // fail fast if not whitelisted
    if (!complianceService.checkWhitelisted(_to)) {
        return (20, WALLET_NOT_IN_REGISTRY_SERVICE);
    }

    IDSComplianceConfigurationService complianceConfigurationService =
    ↪   IDSComplianceConfigurationService(_services[COMPLIANCE_CONFIGURATION_SERVICE]);

     // don't need this until much later in the function so no point doing it here
    // IDSWalletManager walletManager = IDSWalletManager(_services[WALLET_MANAGER]);

    // add this to improve readability as this is used multiple times
    IDSRegistryService regService = IDSRegistryService(_services[REGISTRY_SERVICE]);

    string memory toCountry = regService.getCountry(regService.getInvestor(_to));
    uint256 toRegion = complianceConfigurationService.getCountryCompliance(toCountry);

    if (toRegion == FORBIDDEN) {
        return (26, DESTINATION_RESTRICTED);
    }

    // continue remaining processing, following the principles of failing fast by only
    // perform storage reads and external calls as they are needed
```

Securitize: Fixed in commit 80d536e.

Cyfrin: Verified.

7.5.17 Not possible to send native via the `TransactionRelayer` to the target contract when executing `executeByInvestorWithBlockLimit`

Description: `TransactionRelayer::executeByInvestorWithBlockLimit` makes an external call to a `destination` address, one of the input parameters of this function is `value`, which is encoded in `params[0]`, and this parameter is used to specify the amount of native balance that will be transferred to the `destination` address on the external call made in `doExecuteByInvestor`.

The problem is that the `TransactionRelayer::executeByInvestorWithBlockLimit` is not payable, which means that native can't be sent as part of the txn. Also, `TransactionRelayer` reverts when attempting to fund it by transferring native from one account to another.

```
    function doExecuteByInvestor(
        ...
        uint256[] memory params
    ) private {
       ...
        bool success = false;
        uint256 value = params[0];
        uint256 gasLimit = params[1];
        assembly {
            success := call(
            gasLimit,
            destination,
//@audit => Amount of native to transfer on the external call
            value,
            add(data, 0x20),
            mload(data),
            0,
            0
            )
        }
        require(success, "transaction was not executed");
    }
```

Impact: Signatures including native balance to be sent to the target contract will revert.

Proof of Concept: Add the next foundry PoC to the test suite.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.22;

import "forge-std/Test.sol";
import "../contracts/utils/TransactionRelayer.sol";
import {ERC1967Proxy} from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract MockTransactionRelayer is TransactionRelayer {
    function entryPoint(address target, uint256 _value) public {
        internalCall(target, _value);
    }

    function internalCall(address target, uint256 _value) internal {
        bool success = false;
        assembly {
            success := call(
            gas(),
            target,
            _value,
            0,
            0,
            0,
            0
            )
        }
        require(success, "call to target reverted");
    }
}

contract TransactionRelayerTest is Test {
    address implementation;
    MockTransactionRelayer transactionRelayer;

    function setUp() public {
        implementation = address(new MockTransactionRelayer());
        // bytes memory data = abi.encodeCall(TransactionRelayer.initialize, "");
        address proxy = address(new ERC1967Proxy(implementation, ""));
        transactionRelayer = MockTransactionRelayer(proxy);
    }

    function test_transactionRelayer() public {
        transactionRelayer.initialize();

        //@audit-info => Not possible to fund native to the TransactionRelayer
        vm.expectRevert();
        (bool success, ) = address(transactionRelayer).call{value: 1 ether}("");
        require(success);

        assertEq(address(transactionRelayer).balance, 0);

        //@audit-info => Not possible to transfer out native from the TransactionRelayer
        address user1 = makeAddr("user1");
        vm.expectRevert();
        transactionRelayer.entryPoint(user1, 1 ether);

        //@audit-info => Not possible to send native in the call because not payable modifier
        // transactionRelayer.entryPoint{value: 1 ether}(user1, 1 ether);
    }
}
```

Recommended Mitigation: Either make the function `executeByInvestorWithBlockLimit` payable or add the `receive` to allow the Relayer to be funded separately and allow `executeByInvestorWithBlockLimit` to spend from that balance.

Alternatively, remove the `value` parameter from the signature and from the `params[]` so that the external call explicitly never transfers natively to the destination address.

Securitize: `TransactionRelayer` has been significantly changed such that `executeByInvestorWithBlockLimit` and most other functions now always revert since they were deprecated. The only remaining working function is `executePreApprovedTransaction` which doesn't take a `value` parameter nor send native.

Cyfrin: Verified.

7.5.18 Don't write to the same storage slot multiple times

Description: In EVM writing to storage is expensive; ideally only write to the same storage slot once. For example `ComplianceServiceRegulated::cleanupInvestorIssuances` does this:

```
        uint256 time = block.timestamp;

        uint256 currentIssuancesCount = issuancesCounters[investor];
        uint256 currentIndex = 0;

        if (currentIssuancesCount == 0) {
            return;
        }

        while (currentIndex < currentIssuancesCount) {
            uint256 issuanceTimestamp = issuancesTimestamps[investor][currentIndex];

            bool isNoLongerLocked = issuanceTimestamp <= (time - lockTime);

            if (isNoLongerLocked) {
                if (currentIndex != currentIssuancesCount - 1) {
                    issuancesTimestamps[investor][currentIndex] =
                    ↪  issuancesTimestamps[investor][currentIssuancesCount - 1];
                    issuancesValues[investor][currentIndex] =
                    ↪  issuancesValues[investor][currentIssuancesCount - 1];
                }

                delete issuancesTimestamps[investor][currentIssuancesCount - 1];
                delete issuancesValues[investor][currentIssuancesCount - 1];

                // @audit storage write to decrement
                issuancesCounters[investor]--;
                // @audit storage read of value just written
                currentIssuancesCount = issuancesCounters[investor];
            } else {
                currentIndex++;
            }
        }
```

This is very inefficient as it results in an additional storage write and storage read during every loop iteration when `isNoLongerLocked == true`. Instead just decrement the `currentIssuancesCount` variable then write once to `issuancesCounters[investor]` after the loop:

```
        while (currentIndex < currentIssuancesCount) {
            uint256 issuanceTimestamp = issuancesTimestamps[investor][currentIndex];

            bool isNoLongerLocked = issuanceTimestamp <= (time - lockTime);

            if (isNoLongerLocked) {
                if (currentIndex != currentIssuancesCount - 1) {
                    issuancesTimestamps[investor][currentIndex] =
                    ↪  issuancesTimestamps[investor][currentIssuancesCount - 1];
                    issuancesValues[investor][currentIndex] =
                    ↪  issuancesValues[investor][currentIssuancesCount - 1];
                }

                delete issuancesTimestamps[investor][currentIssuancesCount - 1];
                delete issuancesValues[investor][currentIssuancesCount - 1];

                currentIssuancesCount--;
            } else {
                currentIndex++;
            }
        }

        issuancesCounters[investor] = currentIssuancesCount;
```

Also there don't appear to be any unit tests around this area; I commented out the `while` loop and re-ran the test suite and no tests failed! So ideally before changing anything write some unit tests first to ensure the optimized version doesn't break anything.

Securitize: Fixed in commit 10ac116 where we also added additional unit tests around the contents of the `while` loop.

Cyfrin: Verified.

7.5.19 `ComplianceServiceRegulated::getComplianceTransferableTokens` should call `IDSLockManager::getTransferableTokensForInvestor`

Description: `ComplianceServiceRegulated::getComplianceTransferableTokens` already loads the registry and fetches the investor id, so therefore it should call `IDSLockManager::getTransferableTokensForInvestor` instead of `getTransferableTokens` to save again loading the registry and again fetching the investor id:

```
    function getComplianceTransferableTokens(
        address _who,
        uint256 _time,
        uint64 _lockTime
    ) public view override returns (uint256) {
        require(_time != 0, "Time must be greater than zero");
        string memory investor = getRegistryService().getInvestor(_who);

-       uint256 balanceOfInvestor = getLockManager().getTransferableTokens(_who, _time);
+       uint256 balanceOfInvestor = getLockManager().getTransferableTokensForInvestor(investor, _time);
```

Securitize: Fixed in commit 382eaae.

Cyfrin: Verified.

7.5.20 Remove return value from `DSToken::updateInvestorBalance` as it is never checked

Description: `DSToken::updateInvestorBalance` is an `internal` function which returns `bool` but this return value is never checked anywhere; remove it:

```
token/DSToken.sol
304:        updateInvestorBalance(_from, _value, CommonUtils.IncDec.Decrease);
305:        updateInvestorBalance(_to, _value, CommonUtils.IncDec.Increase);
308:    function updateInvestorBalance(address _wallet, uint256 _value, CommonUtils.IncDec _increase)
↪   internal override returns (bool) {

mocks/StandardTokenMock.sol
72:    function updateInvestorBalance(address, uint256, CommonUtils.IncDec) internal pure override
↪   returns (bool) {

token/TokenLibrary.sol
94:        updateInvestorBalance(_tokenData, IDSRegistryService(_services[REGISTRY_SERVICE]),
↪   _params._to, shares, CommonUtils.IncDec.Increase);
129:        updateInvestorBalance(
165:        updateInvestorBalance(
193:        updateInvestorBalance(_tokenData, registryService, _from, _shares,
↪   CommonUtils.IncDec.Decrease);
194:        updateInvestorBalance(_tokenData, registryService, _to, _shares,
↪   CommonUtils.IncDec.Increase);
197:    function updateInvestorBalance(TokenData storage _tokenData, IDSRegistryService
↪   _registryService, address _wallet, uint256 _shares, CommonUtils.IncDec _increase) internal returns
↪   (bool) {

token/IDSToken.sol
131:    function updateInvestorBalance(address _wallet, uint256 _value, CommonUtils.IncDec _increase)
↪   internal virtual returns (bool);
```

The current return value can also be misleading, for example if `_wallet` doesn't belong to an investor then the update never happens but it still returns `true`:

```
function updateInvestorBalance(address _wallet, uint256 _value, CommonUtils.IncDec _increase) internal
↪   override returns (bool) {
    string memory investor = getRegistryService().getInvestor(_wallet);
    // @audit if `_wallet` doesn't belong to an investor, no update occurs
    if (!CommonUtils.isEmptyString(investor)) {
        uint256 balance = balanceOfInvestor(investor);
        if (_increase == CommonUtils.IncDec.Increase) {
            balance += _value;
        } else {
            balance -= _value;
        }

        ISecuritizeRebasingProvider rebasingProvider = getRebasingProvider();

        uint256 sharesBalance = rebasingProvider.convertTokensToShares(balance);

        tokenData.investorsBalances[investor] = sharesBalance;
    }
    // @audit but the function still returns `true` which is misleading
    return true;
}
```

So it seems simpler to just remove the `bool` return value as it isn't ever read anyway and this is an internal function which doesn't affect public interfaces.

Securitize: Fixed in commit 2219e9a.

Cyfrin: Verified.