Global Registry Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

Jorge

November 6, 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

`DSToken` has been previously audited by Cyfrin. This audit was concerned with the following incremental changes:

- creation of a new "Global Registry" service
- two new compliance contracts `BlackListManager` and `ComplianceServiceGlobalWhitelisted`
- implementation of Permit interface (ERC-2612) in `DSToken`
- ability to change `DSToken` name and symbol

The audit scope was limited to the following files:

```
// https://github.com/securitize-io/bc-global-registry-service-sc/tree/dev
contracts/data-stores/GlobalRegistryServiceDataStore.sol
contracts/registry/GlobalRegistryService.sol
contracts/utils/BaseRBACContract.sol

// https://github.com/securitize-io/dstoken/tree/dev
contracts/compliance/BlackListManager.sol
contracts/compliance/ComplianceServiceGlobalWhitelisted.sol
contracts/data-stores/BlackListManagerDataStore.sol
contracts/token/ERC20PermitMixin.sol
contracts/token/StandardToken.sol
```

6 Executive Summary

Over the course of 3 days, the Cyfrin team conducted an audit on the Global Registry smart contracts provided by Securitize. In this period, a total of 16 issues were found.

The findings consist of 3 Medium and 3 Low severity issues with the remainder being informational and gas optimizations. Of the 3 Mediums:

- 7.1.1 - bug that when triggered inflated a counter leading to subsequent invalid DoS reverts
- 7.1.2 - compliance gap that allowed blacklisted users to transfer tokens to non-blacklisted users
- 7.1.3 - DSToken permit functionality broken once token name changed

Finding 7.1.3 was the most interesting since the implementation of Permit interface (ERC-2612) in `DSToken` and the ability to change `DSToken` name were two new features introduced as part of this audit. Individually the new features worked fine but the error manifested once both new features were used together.

Summary

Project Name	Global Registry
Repository	bc-global-registry-service-sc
Commit	e36348808186...
Fix Commit	22545b3cd240...
Repository 2	dstoken
Commit	7f24fcabd13b...
Fix Commit	a616d398add9...
Audit Timeline	Oct 27th - Oct 29th, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	3
Informational	6
Gas Optimizations	4
Total Issues	16

Summary of Findings

[M-1] Same wallet can be added multiple times to an investor, artificially increasing their wallet count causing adding new wallets to revert	Resolved
[M-2] `ComplianceServiceGlobalWhitelisted::newPreTransferCheck` and `preTransferCheck` allow blacklisted users to transfer tokens	Resolved
[M-3] Token name update breaks EIP-712 Domain Separator for permit functionality	Resolved
[L-1] `StandardToken::transferWithPermit` can be DoS attacked by front-running to directly call `ERC20PermitMixin::permit`	Resolved
[L-2] Cross-chain incompatibility with `block.number` based timeout mechanism	Resolved
[L-3] Missing signature deadline for `GlobalRegistryService::executePreApprovedTransaction`	Resolved
[I-1] Don't emit misleading events when roles haven't been added or revoked	Resolved
[I-2] Don't initialize to default values	Resolved
[I-3] `GlobalRegistryService::executePreApprovedTransaction` is incompatible with smart wallet operators	Acknowledged

[I-4] Use event indexing for faster off-chain parameter lookup	Resolved
[I-5] Inherited `AccessControlUpgradeable` functions bypass some validation checks	Acknowledged
[I-6] `ComplianceServiceGlobalWhitelisted::getComplianceTransferableTokens` returns positive token amount for blacklisted users	Resolved
[G-1] Remove redundant calls to `EnumerableSet::contains`	Resolved
[G-2] In `StandardToken::updateNameAndSymbol` cache existing name and symbol then pass them to child functions	Acknowledged
[G-3] Inline small `private` functions only called once	Acknowledged
[G-4] Remove unused `ExecutePreApprovedTransaction::nonce`	Resolved

7 Findings

7.1 Medium Risk

7.1.1 Same wallet can be added multiple times to an investor, artificially increasing their wallet count causing adding new wallets to revert

Description: `GlobalRegistryService::_updateInvestor` calls `_addWallet` when the wallet being added is already registered to this investor:

```
for (uint8 i = 0; i < walletAddresses.length; i++) {
    // @audit if it is a wallet and it doesn't belong to this investor, revert
    if (isWallet(walletAddresses[i]) && !CommonUtils.isEqualString(getInvestor(walletAddresses[i]),
    ↪  id)) {
        revert WalletBelongsToAnotherInvestor();
    }
    // @audit otherwise add it - even if it is a wallet that already belongs to this investor!
    else {
        _addWallet(walletAddresses[i], id);
    }
}
```

`GlobalRegistryService::_addWallet` in turn increments the investor's `walletCount` and reverts once the max are reached:

```
function _addWallet(address walletAddress, string memory id) internal addressNotZero(walletAddress)
↪  returns (bool) {
    if (investors[id].walletCount >= MAX_WALLETS_PER_INVESTOR) {
        revert MaxWalletsReached();
    }
    address sender = _msgSender();
    investorsWallets[walletAddress] = Wallet(id, sender);
    investors[id].walletCount++;

    emit GlobalWalletAdded(walletAddress, id, sender);

    return true;
}
```

Impact: An investor's wallet count can be artificially inflated when updating that investor's details especially via `GlobalRegistryService::updateInvestor` which can be called with all of an investor's existing data and only some modified fields. Once `MAX_WALLETS_PER_INVESTOR` is reached no further updates are possible.

There are also other impacts such as never being able to remove an investor since `GlobalRegistryService::removeWallet` won't be able to decrement the investor's `walletCount` back to 0 hence `removeInvestor` will always revert.

Proof Of Concept: First add this `view` function to `GlobalRegistryService.sol`:

```
    function walletCountByInvestor(string calldata investorId) public view returns (uint256) {
        return investors[investorId].walletCount;
    }
```

Then add the PoC to `global-registry-service.tests.ts`:

```
  it('Bug - adding same wallet for same investor inflates wallet count', async function () {
    const [, investor] = await hre.ethers.getSigners();
    const { globalRegistryService } = await loadFixture(deployGRS);
    await globalRegistryService.updateInvestor(
      INVESTORS.INVESTOR_ID.INVESTOR_ID_1,
      INVESTORS.INVESTOR_ID.INVESTOR_COLLISION_HASH_1,
      US,
      [investor],
      [1, 2, 4],
      [1, 1, 1],
      [0, 0, 0],
    );

    await globalRegistryService.updateInvestor(
      INVESTORS.INVESTOR_ID.INVESTOR_ID_1,
      INVESTORS.INVESTOR_ID.INVESTOR_COLLISION_HASH_1,
      US,
      [investor],
      [1, 2, 4],
      [1, 1, 1],
      [0, 0, 0],
    );

    expect(await
    ↪  globalRegistryService.walletCountByInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1)).to.equal(2);

  });
```

Run with `npx hardhat test --grep "adding same wallet for same investor inflates wallet count"`.

Recommended Mitigation: In `GlobalRegistryService::_updateInvestor` if the wallet being added is a wallet and already belongs to the same investor, don't do anything. Here is a more efficient implementation of `_updateInvestor` that avoids duplicate work done by the `isWallet` and `isInvestor` functions while also fixing this bug:

```
function _updateInvestor(string calldata id, address[] memory walletAddresses) internal returns (bool) {
    // revert if max wallet would be breached
    uint256 walletAddressesLen = walletAddresses.length;
    if (walletAddressesLen > MAX_WALLETS_PER_INVESTOR) {
        revert TooManyWallets();
    }

    // register investor if they don't already exist
    if (!isInvestor(id)) {
        _registerInvestor(id);
    }

    for (uint8 i; i < walletAddressesLen; i++) {
        address newWallet = walletAddresses[i];

        // is the wallet already registered to an investor?
        string memory walletExistingInvestor = getInvestor(newWallet);

        // if not then add it
        if(!isInvestor(walletExistingInvestor)) {
            _addWallet(newWallet, id);
        }
        // otherwise revert if it is registered to another investor
        else if(!CommonUtils.isEqualString(walletExistingInvestor, id)) {
            revert WalletBelongsToAnotherInvestor();
        }
        // if it is already registered to this investor, do nothing
    }

    return true;
}
```

Securitize: Fixed in commit 5713fd2.

Cyfrin: Verified.

7.1.2 `ComplianceServiceGlobalWhitelisted::newPreTransferCheck` and `preTransferCheck` allow blacklisted users to transfer tokens

Description: `ComplianceServiceGlobalWhitelisted::newPreTransferCheck` and `preTransferCheck` only enforce that the recipient of tokens is not blacklisted, but they don't check that the sender is also not blacklisted:

```
function newPreTransferCheck(
    address _from,
    address _to,
    uint256 _value,
    uint256 _balanceFrom,
    bool _pausedToken
) public view virtual override returns (uint256 code, string memory reason) {
    // First check if recipient is blacklisted
    if (getBlackListManager().isBlacklisted(_to)) {
        return (100, WALLET_BLACKLISTED);
    }

    // Then perform the standard whitelist check
    return super.newPreTransferCheck(_from, _to, _value, _balanceFrom, _pausedToken);
}

function preTransferCheck(address _from, address _to, uint256 _value) public view virtual override
↪   returns (uint256 code, string memory reason) {
    // First check if recipient is blacklisted
    if (getBlackListManager().isBlacklisted(_to)) {
        return (100, WALLET_BLACKLISTED);
    }

    // Then perform the standard whitelist check
    return super.preTransferCheck(_from, _to, _value);
}
```

Impact: A blacklisted user can transfer their tokens to a non-blacklisted user, effectively evading the blacklist.

Recommended Mitigation: `ComplianceServiceGlobalWhitelisted::newPreTransferCheck` and `preTransferCheck` should return correct error codes if `from` address is blacklisted.

Securitize: Fixed in commits 32d1a02, a616d39.

Cyfrin: Verified.

7.1.3 Token name update breaks EIP-712 Domain Separator for permit functionality

Description: The EIP-712 domain separator is initialized once during contract deployment in `__ERC20PermitMixin_init()` with the initial token name:

```
function __ERC20PermitMixin_init(string memory name_) internal onlyInitializing {
    __EIP712_init(name_, "1");  // Domain separator set with initial name
    __Nonces_init();
}
```

However, `StandardToken` allows the Master role to update the token name via `updateNameAndSymbol`:

```
function updateNameAndSymbol(string calldata _name, string calldata _symbol) external onlyMaster {
    // ...
    name = _name;  // Name updated but EIP-712 domain separator NOT updated
    // ...
}
```

Impact: When the token name is changed, the EIP-712 domain separator remains unchanged. This creates a mismatch between what wallets use to generate permit signatures (the current token name) and what the contract uses to validate them (the original deployment name). Potential impact:

1. **Complete permit functionality breakage**: After any name change, 100% of newly generated permit signatures will fail validation with "Permit: invalid signature"
2. **Silent failure mode**: Users and integrations have no programmatic way to detect this mismatch; the error message doesn't indicate it's a domain separator issue
3. **All dynamic integrations break**: Any dApp that fetches `token.name()` to generate permit signatures will automatically break after a name update
4. **Inconsistent behavior**: Permits signed before the name change continue to work, while new ones fail, creating a confusing split state
5. **No easy recovery path**: Fixing this requires either a contract upgrade or instructing all users/integrations to use the deprecated name (breaking EIP-2612 expectations)

Proof of Concept: Add a new file `test/change.name.permit.test.ts`:

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

describe('M-01: Token Name Update Breaks Permit Functionality - Proof of Concept', function() {

  describe('Demonstrating the Vulnerability', function() {

    it('CRITICAL: Permit fails after name update - All new permits become invalid', async function() {
      const [owner, spender] = await hre.ethers.getSigners();
      const { dsToken } = await loadFixture(deployDSTokenRegulated);

      // Initial state: Token name is "Token Example 1"
      const originalName = await dsToken.name();
      expect(originalName).to.equal('Token Example 1');

      //  STEP 1: Permit works BEFORE name change
      console.log('\n--- BEFORE NAME CHANGE ---');
      const deadline1 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const value = 100;
      const message1 = {
        owner: owner.address,
        spender: spender.address,
        value,
        nonce: await dsToken.nonces(owner.address),
        deadline: deadline1,
      };

      // User signs with original name "Token Example 1"
      const sig1 = await buildPermitSignature(
        owner,
        message1,
        originalName,  // Uses "Token Example 1"
        await dsToken.getAddress()
      );

      // Permit succeeds with original name
      await dsToken.permit(owner.address, spender.address, value, deadline1, sig1.v, sig1.r, sig1.s);
      console.log(' Permit with original name: SUCCESS');
      expect(await dsToken.allowance(owner.address, spender.address)).to.equal(value);

      //  STEP 2: Master updates token name
      console.log('\n--- NAME CHANGE ---');
      const newName = 'Token Example 2 - Updated';
      await dsToken.updateNameAndSymbol(newName, 'TX2');
      expect(await dsToken.name()).to.equal(newName);
      console.log(`Token name updated: "${originalName}" → "${newName}"`);

      //  STEP 3: Permit FAILS after name change
      console.log('\n--- AFTER NAME CHANGE ---');
      const deadline2 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const message2 = {
        owner: owner.address,
        spender: spender.address,
        value: 200,
        nonce: await dsToken.nonces(owner.address),
        deadline: deadline2,
      };

      // User's wallet fetches current name and generates signature
      const currentName = await dsToken.name(); // Returns "Token Example 2 - Updated"
      console.log(`User's wallet uses current name: "${currentName}"`);

      const sig2 = await buildPermitSignature(
        owner,
        message2,
        currentName,  // Uses NEW name "Token Example 2 - Updated"
        await dsToken.getAddress()
      );

      //  PERMIT FAILS - Domain separator mismatch!
      await expect(
        dsToken.permit(owner.address, spender.address, 200, deadline2, sig2.v, sig2.r, sig2.s)
      ).to.be.revertedWith('Permit: invalid signature');

      console.log(' Permit with new name: FAILED - "Permit: invalid signature"');
      console.log('\n VULNERABILITY CONFIRMED: All new permits fail after name change!');
    });

    it('IMPACT: Old permits continue working while new ones fail - Inconsistent behavior', async
        ↪   function() {
      const [owner, spender] = await hre.ethers.getSigners();
      const { dsToken } = await loadFixture(deployDSTokenRegulated);

      const originalName = await dsToken.name();

      // Generate permit signature BEFORE name change
      const deadline1 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const message1 = {
        owner: owner.address,
        spender: spender.address,
        value: 100,
        nonce: await dsToken.nonces(owner.address),
        deadline: deadline1,
      };

      const oldSignature = await buildPermitSignature(
        owner,
        message1,
        originalName,
        await dsToken.getAddress()
      );

      // Master changes the name
      await dsToken.updateNameAndSymbol('Token Example 2', 'TX2');
      const newName = await dsToken.name();

      //  OLD permit (signed before name change) still works!
      await dsToken.permit(
        owner.address,
        spender.address,
        100,
        deadline1,
        oldSignature.v,
        oldSignature.r,
        oldSignature.s
      );
      console.log(' Old permit (signed before name change): SUCCESS');

      //  NEW permit (signed after name change) fails!
      const deadline2 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const message2 = {
        owner: owner.address,
        spender: spender.address,
        value: 200,
        nonce: await dsToken.nonces(owner.address),
        deadline: deadline2,
      };

      const newSignature = await buildPermitSignature(
        owner,
        message2,
        newName,  // Uses new name
        await dsToken.getAddress()
      );

      await expect(
        dsToken.permit(owner.address, spender.address, 200, deadline2, newSignature.v, newSignature.r,
            ↪  newSignature.s)
      ).to.be.revertedWith('Permit: invalid signature');

      console.log(' New permit (signed after name change): FAILED');
      console.log('\n INCONSISTENT STATE: Split behavior based on signature timing!');
    });

    it('IMPACT: DApp integrations break silently', async function() {
      const [owner, spender] = await hre.ethers.getSigners();
      const { dsToken } = await loadFixture(deployDSTokenRegulated);

      // Simulate a DApp that dynamically fetches token name
      async function dAppGeneratePermitSignature(tokenContract, ownerSigner, spenderAddress, value,
          ↪  deadline) {
        // Standard DApp implementation: fetch name dynamically
        const tokenName = await tokenContract.name();
        const tokenAddress = await tokenContract.getAddress();

        const message = {
          owner: ownerSigner.address,
          spender: spenderAddress,
          value,
          nonce: await tokenContract.nonces(ownerSigner.address),
          deadline,
        };

        return await buildPermitSignature(ownerSigner, message, tokenName, tokenAddress);
      }

      //  DApp works fine initially
      const deadline1 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const sig1 = await dAppGeneratePermitSignature(dsToken, owner, spender.address, 100, deadline1);

      await dsToken.permit(owner.address, spender.address, 100, deadline1, sig1.v, sig1.r, sig1.s);
      console.log(' DApp integration BEFORE name change: SUCCESS');

      // Master updates name
      await dsToken.updateNameAndSymbol('Token Example 2', 'TX2');
      console.log('\n  Token name updated to "Token Example 2"');

      //  DApp breaks - it fetches the NEW name but contract validates against OLD name
      const deadline2 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const sig2 = await dAppGeneratePermitSignature(dsToken, owner, spender.address, 200, deadline2);

      await expect(
        dsToken.permit(owner.address, spender.address, 200, deadline2, sig2.v, sig2.r, sig2.s)
      ).to.be.revertedWith('Permit: invalid signature');

      console.log(' DApp integration AFTER name change: FAILED');
      console.log(' DApp has NO way to detect this issue programmatically!');
    });

    it('WORKAROUND: Permit succeeds if user manually uses ORIGINAL name (terrible UX)', async
        ↪   function() {
      const [owner, spender] = await hre.ethers.getSigners();
      const { dsToken } = await loadFixture(deployDSTokenRegulated);

      const originalName = await dsToken.name(); // "Token Example 1"

      // Master updates name
      await dsToken.updateNameAndSymbol('Token Example 2', 'TX2');

      //  Using current name fails
      const deadline1 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const message1 = {
        owner: owner.address,
        spender: spender.address,
        value: 100,
        nonce: await dsToken.nonces(owner.address),
        deadline: deadline1,
      };

      const sigWithNewName = await buildPermitSignature(
        owner,
        message1,
        await dsToken.name(),  // "Token Example 2" (current)
        await dsToken.getAddress()
      );

      await expect(
        dsToken.permit(owner.address, spender.address, 100, deadline1, sigWithNewName.v,
            ↪  sigWithNewName.r, sigWithNewName.s)
      ).to.be.revertedWith('Permit: invalid signature');
      console.log(' Permit with current name "Token Example 2": FAILED');

      //  Using ORIGINAL name works (but terrible UX)
      const deadline2 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const message2 = {
        owner: owner.address,
        spender: spender.address,
        value: 200,
        nonce: await dsToken.nonces(owner.address),
        deadline: deadline2,
      };

      const sigWithOriginalName = await buildPermitSignature(
        owner,
        message2,
        originalName,  // "Token Example 1" (original)
        await dsToken.getAddress()
      );

      await dsToken.permit(
        owner.address,
        spender.address,
        200,
        deadline2,
        sigWithOriginalName.v,
        sigWithOriginalName.r,
        sigWithOriginalName.s
      );
      console.log(' Permit with ORIGINAL name "Token Example 1": SUCCESS');
      console.log('\n WORKAROUND: Users must use deprecated name - breaks EIP-2612 expectations!');
    });

    it('VERIFICATION: DOMAIN_SEPARATOR remains unchanged after name update', async function() {
      const { dsToken } = await loadFixture(deployDSTokenRegulated);

      const originalName = 'Token Example 1';

      // Get domain separator before name change
      const domainSeparatorBefore = await dsToken.DOMAIN_SEPARATOR();
      console.log('Domain separator BEFORE name change:', domainSeparatorBefore);

      // Compute expected domain separator with original name
      const expectedDomainBefore = hre.ethers.TypedDataEncoder.hashDomain({
        version: '1',
        name: originalName,
        verifyingContract: await dsToken.getAddress(),
        chainId: (await hre.ethers.provider.getNetwork()).chainId,
      });

      expect(domainSeparatorBefore).to.equal(expectedDomainBefore);

      // Update name
      await dsToken.updateNameAndSymbol('Token Example 2', 'TX2');
      const newName = await dsToken.name();
      console.log(`\nName updated: "${originalName}" → "${newName}"`);

      // Get domain separator after name change
      const domainSeparatorAfter = await dsToken.DOMAIN_SEPARATOR();
      console.log('Domain separator AFTER name change:', domainSeparatorAfter);

      //  DOMAIN SEPARATOR UNCHANGED!
      expect(domainSeparatorAfter).to.equal(domainSeparatorBefore);
      console.log('\n VERIFIED: Domain separator did NOT update with new name!');

      // Compute what the domain separator SHOULD be with new name
      const expectedDomainWithNewName = hre.ethers.TypedDataEncoder.hashDomain({
        version: '1',
        name: newName,
        verifyingContract: await dsToken.getAddress(),
        chainId: (await hre.ethers.provider.getNetwork()).chainId,
      });

      console.log('\nExpected domain with NEW name:', expectedDomainWithNewName);
      console.log('Actual domain separator:      ', domainSeparatorAfter);
      console.log('Match:', domainSeparatorAfter === expectedDomainWithNewName ? '' : '');

      // They don't match - this is the root cause
      expect(domainSeparatorAfter).to.not.equal(expectedDomainWithNewName);
    });
  });

  describe('Real-World Attack Scenarios', function() {

    it('SCENARIO 1: Protocol rebranding breaks all user permits', async function() {
      const [owner, user1, user2, dex] = await hre.ethers.getSigners();
      const { dsToken, registryService } = await loadFixture(deployDSTokenRegulated);

      // Setup investors
      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, user1, registryService);
      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, user2, registryService);
      await dsToken.issueTokens(user1, 1000);

      console.log('\n SCENARIO: Token rebranding from "Token Example 1" to "Acme Securities Token"');

      // Before rebrand: User1 can use permit to approve DEX
      const deadline1 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const message1 = {
        owner: user1.address,
        spender: dex.address,
        value: 500,
        nonce: await dsToken.nonces(user1.address),
        deadline: deadline1,
      };

      const sig1 = await buildPermitSignature(
        user1,
        message1,
        await dsToken.name(),
        await dsToken.getAddress()
      );

      await dsToken.permit(user1.address, dex.address, 500, deadline1, sig1.v, sig1.r, sig1.s);
      console.log(' User1 successfully approved DEX using permit (before rebrand)');

      //  PROTOCOL REBRANDS
      await dsToken.updateNameAndSymbol('Acme Securities Token', 'AST');
      console.log('\n Protocol rebrands to "Acme Securities Token"');

      // After rebrand: All new permits fail
      const deadline2 = BigInt(Math.floor(Date.now() / 1000) + 3600);
      const message2 = {
        owner: user2.address,
        spender: dex.address,
        value: 500,
        nonce: await dsToken.nonces(user2.address),
        deadline: deadline2,
      };

      await dsToken.issueTokens(user2, 1000);

      const sig2 = await buildPermitSignature(
        user2,
        message2,
        await dsToken.name(), // Uses new name
        await dsToken.getAddress()
      );
      await expect(
        dsToken.permit(user2.address, dex.address, 500, deadline2, sig2.v, sig2.r, sig2.s)
      ).to.be.revertedWith('Permit: invalid signature');

      console.log(' User2 permit FAILS after rebrand');
      console.log(' Impact: 100% of new users cannot use gasless approvals!');
      console.log(' Result: Support tickets flood in, users confused');
    });

    it('SCENARIO 2: Front-end integration breaks without warning', async function() {
      const [owner, user, spender] = await hre.ethers.getSigners();
      const { dsToken, registryService } = await loadFixture(deployDSTokenRegulated);

      await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, user, registryService);
      await dsToken.issueTokens(user, 1000);

      console.log('\n SCENARIO: Frontend dApp integration');

      // Simulate frontend code
      const frontendPermitFlow = async (token, fromUser, toSpender, amount) => {
        // Standard EIP-2612 implementation in frontend
        const name = await token.name(); // Fetch current name dynamically
        const deadline = BigInt(Math.floor(Date.now() / 1000) + 3600);
        const nonce = await token.nonces(fromUser.address);

        const message = {
          owner: fromUser.address,
          spender: toSpender,
          value: amount,
          nonce,
          deadline,
        };

        const signature = await buildPermitSignature(
          fromUser,
          message,
          name,
          await token.getAddress()
        );

        return { deadline, signature };
      };

      //  Frontend works initially
      const { deadline: d1, signature: s1 } = await frontendPermitFlow(dsToken, user, spender.address,
      ↪  100);
      await dsToken.permit(user.address, spender.address, 100, d1, s1.v, s1.r, s1.s);
      console.log(' Frontend permit flow: SUCCESS (initial deployment)');

      // Master updates name (e.g., for compliance reasons)
      await dsToken.updateNameAndSymbol('Compliant Token v2', 'CTv2');
      console.log('\n  Master updates name for compliance');

      //  Frontend breaks silently
      const { deadline: d2, signature: s2 } = await frontendPermitFlow(dsToken, user, spender.address,
      ↪  200);
      await expect(
        dsToken.permit(user.address, spender.address, 200, d2, s2.v, s2.r, s2.s)
      ).to.be.revertedWith('Permit: invalid signature');

      console.log(' Frontend permit flow: BROKEN (after name update)');
      console.log(' Error message gives NO hint about name mismatch');
```

```
      console.log(' Users see "Invalid signature" and blame wallet/frontend');
    });
  });
});
```

Run with: `npx hardhat test --grep "Token Name Update Breaks Permit Functionality"`

Recommended Mitigation: The most elegant solution appears to be:

- `StandardToken` defines a `_name` function that just returns `name`:

```
    function _name() internal view virtual override returns (string memory) {
        return name;  // Returns current storage variable
    }
```

- `ERC20PermitMixin` overrides `EIP712Upgradeable::_EIP712Name` to call this function:

```
    //  Override to return dynamic name instead of cached name
    function _EIP712Name() internal view virtual override returns (string memory) {
        return _name();  // Calls abstract function implemented by StandardToken
    }

    // Abstract function for StandardToken to implement
    function _name() internal view virtual returns (string memory);
```

Securitize: Fixed in commit 4ebb9b7.

Cyfrin: Verified.

7.2 Low Risk

7.2.1 `StandardToken::transferWithPermit` can be DoS attacked by front-running to directly call `ERC20PermitMixin::permit`

Description: `StandardToken::transferWithPermit` contains two calls:

- first to `ERC20PermitMixin::permit`
- second to the `StandardToken::transferFrom`

Impact: Since the permit signature and parameters are visible in the mempool before execution, an attacker can extract these values and front-run the transaction by directly calling `StandardToken::permit`. This consumes the user's nonce causing the original call `StandardToken::transferWithPermit` to revert, making it impossible to atomically grant the approval and transfer the tokens.

Proof of concept Run the PoC in `dstoken-regulated.test.ts` inside the `describe('Permit transfer', async function () {`:

```
it('front-running attack on transferWithPermit()', async () => {
        const [owner, spender, recipient, attacker] = await hre.ethers.getSigners();
        const { dsToken, registryService } = await loadFixture(deployDSTokenRegulated);
        const value = 100;
        const deadline = BigInt(Math.floor(Date.now() / 1000) + 3600);

        // Owner creates a signature to allow spender to transfer tokens to recipient
        const message = {
          owner: owner.address,
          spender: spender.address,
          value,
          nonce: await dsToken.nonces(owner.address),
          deadline,
        };
        const { v, r, s } = await buildPermitSignature(owner, message, await dsToken.name(), await
        ↪  dsToken.getAddress());

        // Register investors and issue tokens to owner; see that the attacker is not even an ibnvestor
        ↪  so it could be any address
        await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_1, owner, registryService);
        await registerInvestor(INVESTORS.INVESTOR_ID.INVESTOR_ID_2, recipient, registryService);

        await dsToken.issueTokens(owner, value);

        // ATTACK SCENARIO 1: Attacker front-runs by calling permit() directly
        await dsToken.connect(attacker).permit(owner.address, spender.address, value, deadline, v, r,
        ↪  s);


        // When the original transferWithPermit() executes, it FAILS
        // because the nonce has already been used
        await expect(
          dsToken.connect(spender).transferWithPermit(owner.address, recipient.address, value,
          ↪  deadline, v, r, s)
        ).to.be.revertedWith('Permit: invalid signature');
      });
```

Recommended Mitigation: Use the try and catch pattern:

```
function transferWithPermit(
    address from,
    address to,
    uint256 value,
    uint256 deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external returns (bool) {
    // Try to execute permit, but don't revert if it fails
    try this.permit(from, msg.sender, value, deadline, v, r, s) {
        // Permit succeeded
    } catch {
        // Permit failed (possibly due to front-running or already executed)
        // Verify we have sufficient allowance to proceed
        require(allowance(from, msg.sender) >= value, "Insufficient allowance");
    }

    // Perform the actual transferFrom
    return transferFrom(from, to, value);
}
```

Securitize: Fixed in commit d7cf385.

Cyfrin: Verified.

7.2.2 Cross-chain incompatibility with `block.number` based timeout mechanism

Description: `GlobalRegistryService::addGlobalInvestorWallet` function uses `block.number` to validate transaction freshness through a `blockLimit parameter`:

```
function addGlobalInvestorWallet(
    string calldata id,
    address walletAddress,
    uint256 blockLimit
) external override whenNotPaused onlySelf newWallet(walletAddress) returns (bool) {
    if (blockLimit < block.number) {
        revert TransactionTooOld();
    }
    // ...
}
```

According to the hardhat configuration, this contract is designed to be deployed on multiple chains with vastly different block production rates:

- Ethereum Mainnet - Block time: ~12-14 seconds
- Sepolia (Ethereum testnet) - Block time: ~12-14 seconds
- Arbitrum (chainId: 421614) - Block time: ~0.25 seconds (48-56x faster)
- Optimism (chainId: 11155420) - Block time: ~2 seconds (6-7x faster)
- Avalanche/Fuji (chainId: 43113) - Block time: ~2 seconds (6-7x faster)

The problem is that block production rates vary dramatically across these chains, making block-based time validation inconsistent and unreliable. I an operator signs a pre-approved transaction with `blockLimit = currentBlock + 100`:

- On Ethereum: Valid for ~100 × 13 seconds = ~21 minutes
- On Arbitrum: Valid for ~100 × 0.25 seconds = ~25 seconds
- On Optimism/Avalanche: Valid for ~100 × 2 seconds = ~3.3 minutes

Impact: Transactions could revert in one chain and be added in other chains if an investor is adding the same wallet among different chains.

Recommended Mitigation: Replace `block.number` with `block.timestamp` for consistent cross-chain behavior.

Securitize: Fixed in commits 8f92757, e99c56f, 920e496.

Cyfrin: Verified.

7.2.3 Missing signature deadline for `GlobalRegistryService::executePreApprovedTransaction`

Description: `GlobalRegistryService::executePreApprovedTransaction` allows an `Operator` to call an arbitrary contract and function, though the current intent is for it to call `addGlobalInvestorWallet`.

Impact: While `addGlobalInvestorWallet` implements a deadline using `block.number` (see other issue about using timestamp), if `executePreApprovedTransaction` is used to call other functions then no deadline check may be implemented or deadline checks will need to be duplicated in many other places.

Recommended Mitigation: Implement a timestamp-based deadline check in `GlobalRegistryService::executePreApprovedTransaction`. Also consider adding a way for admin or operators to increase `noncePerInvestor[txData.senderInvestor]` so that nonces can be invalidated.

Securitize: Fixed in commits 8f92757, e99c56f, 920e496

Cyfrin: Verified.

7.3 Informational

7.3.1 Don't emit misleading events when roles haven't been added or revoked

Description: `AccessControlUpgradeable::_grantRole` and `_revokeRole` return `bool` to indicate whether the role was actually granted or revoked.

Impact: Some functions using these don't check the `bool` return then emit events; such events will be misleading if the roles were not actually granted or revoked.

Recommended Mitigation: The affected functions are:

- `GlobalRegistryService::changeAdmin, addOperator, revokeOperator`

In these functions check the return of `_grantRole` and `_revokeRole` and only emit events if the roles were actually granted or revoked.

Securitize: Fixed in commit c7d50ac.

Cyfrin: Verified.

7.3.2 Don't initialize to default values

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

```
registry/GlobalRegistryService.sol
        for (uint8 i = 0; i < walletAddresses.length; i++) {
```

Securitize: Fixed in commit 5713fd2.

Cyfrin: Verified.

7.3.3 `GlobalRegistryService::executePreApprovedTransaction` is incompatible with smart wallet operators

Description: `GlobalRegistryService::executePreApprovedTransaction` allows an operator to execute pre-approved transactions using signatures. However it always calls `ECDSA::recover` which means it won't work for operators who are smart wallets.

This appears to be the intention but if smart wallet support is desired consider using the SignatureChecker library.

Securitize: Acknowledged; we know the signer (who is always an "Operator") does not use smart wallets.

7.3.4 Use event indexing for faster off-chain parameter lookup

Description: Events in `IGlobalRegistryService` should use `indexed` keywords on the 3 most important parameters per event to enable faster lookup by those parameters off-chain.

Securitize: Fixed in commit 0c2321a.

Cyfrin: Verified.

7.3.5 Inherited `AccessControlUpgradeable` functions bypass some validation checks

Description: The `GlobalRegistryService` contract implements custom wrapper functions (`addOperator()`, `revokeOperator()`, `changeAdmin()`) with additional safety checks and event emissions on top of OpenZeppelin's `AccessControlUpgradeable`. However, the contract fails to override the underlying public functions from `AccessControlUpgradeable`, allowing admins and operator to bypass some validation checks.

The contract creates wrapper functions with validation:

```
function addOperator(address operator)
    external virtual override
    onlyRole(DEFAULT_ADMIN_ROLE)
    addressNotZero(operator)  // Safety check
{
    _grantRole(OPERATOR_ROLE, operator);
    emit OperatorAdded(operator);
}

function changeAdmin(address newAdmin)
    external virtual override
    onlyRole(DEFAULT_ADMIN_ROLE)
    addressNotZero(newAdmin)  // Safety check
{
    _grantRole(DEFAULT_ADMIN_ROLE, newAdmin);
    _revokeRole(DEFAULT_ADMIN_ROLE, _msgSender());
    emit AdminChanged(newAdmin);
}
```

But the underlying OpenZeppelin functions remain publicly accessible without overrides:

```
function grantRole(bytes32 role, address account) public virtual onlyRole(getRoleAdmin(role)) {
    _grantRole(role, account);
}

function revokeRole(bytes32 role, address account) public virtual onlyRole(getRoleAdmin(role)) {
    _revokeRole(role, account);
}

function renounceRole(bytes32 role, address callerConfirmation) public virtual {
    if (callerConfirmation != _msgSender()) {
        revert AccessControlBadConfirmation();
    }
    _revokeRole(role, callerConfirmation);
}
```

Impact: Two unlikely problems could occur:

- Admin can renounce their role without assigning a successor, permanently bricking the contract (This can be done through `AccessControlUpgradeable::renounceRole`)
- Operator adding directly via `AccessControlUpgradeable::grantRole` won't emit the `OperatorAdded` event

Recommended Mitigation: Consider overriding default functions to revert which not going to be used in the inherited `AccessControlUpgradeable` contract.

Securitize: Cyfrin:

7.3.6 `ComplianceServiceGlobalWhitelisted::getComplianceTransferableTokens` returns positive token amount for blacklisted users

Description: `ComplianceServiceGlobalWhitelisted::getComplianceTransferableTokens` inherited from `ComplianceServiceWhitelisted` will return positive amount of transferable tokens even if the user is blacklisted; this is misleading because they actually have ZERO transferable tokens due to being blacklisted.

```
  function getComplianceTransferableTokens(
        address _who,
        uint256 _time,
        uint64 /*_lockTime*/
    ) public view virtual override returns (uint256) {
        require(_time > 0, "Time must be greater than zero");
        return getLockManager().getTransferableTokens(_who, _time);
    }
```

Recommended Mitigation: The function should return 0 for blacklisted addresses.

Securitize: Fixed in commit dc11a37.

Cyfrin: Verified.

7.4 Gas Optimization

7.4.1 Remove redundant calls to `EnumerableSet::contains`

Description: `EnumerableSet::_add` and `_remove` already call `_contains` (or perform similar logic), hence there is no need to call this prior to adding or removing elements in:

- `BlackListManager::_addToBlacklist, _removeFromBlacklist`

Recommended Mitigation: Call `EnumerableSet::add` or `remove` directly and revert if they return `false`.

Securitize: Fixed in commit a878a41.

Cyfrin: Verified.

7.4.2 In `StandardToken::updateNameAndSymbol` cache existing name and symbol then pass them to child functions

Description: `StandardToken::updateNameAndSymbol`:

- reads the existing `name` and `symbol`
- calls `CommonUtils.isEqualString` to compare existing to proposed new
- if different, calls `_updateName` and `_updateSymbol` which re-read the identical existing values from storage to emit the event

Recommended Mitigation: Since reading from storage is expensive, ideally `updateNameAndSymbol` would

- cache the existing `name` and `symbol`
- use the cached values for the call to `CommonUtils.isEqualString`
- pass the cached values to `_updateName` and `_updateSymbol` which can use them to emit the events

Securitize: Acknowledged.

7.4.3 Inline small `private` functions only called once

Description: `StandardToken::_updateName` and `_updateSymbol` are very small `private` functions that are only ever called once by `updateNameAndSymbol`.

Hence it is more gas efficient to inline them; here is an implementation that also caches the identical storage reads so `name` and `symbol` are only read once from storage:

```
function updateNameAndSymbol(string calldata _name, string calldata _symbol) external onlyMaster {
    require(!CommonUtils.isEmptyString(_name), "Name cannot be empty");
    require(!CommonUtils.isEmptyString(_symbol), "Symbol cannot be empty");

    string memory nameCache = name;
    if (!CommonUtils.isEqualString(_name, nameCache)) {
        emit NameUpdated(nameCache, _name);
        name = _name;
    }

    string memory symbolCache = symbol;
    if (!CommonUtils.isEqualString(_symbol, symbolCache)) {
        emit SymbolUpdated(symbolCache, _symbol);
        symbol = _symbol;
    }
}
```

Securitize: Acknowledged.

7.4.4 Remove unused `ExecutePreApprovedTransaction::nonce`

Description: `ExecutePreApprovedTransaction::nonce` is never actually used, since:

- `GlobalRegistryService::hashTx` always reads the current nonce from storage `noncePerInvestor[txData.senderInvestor]`
- the caller must have used the current nonce to sign - otherwise the signature will fail validation
- when validation succeeds, `executePreApprovedTransaction` always increments the current nonce by 1 so it can never be re-used

Hence the above mechanics correctly validate the nonce and `ExecutePreApprovedTransaction::nonce` can be safely removed as it is never used.

Securitize: Fixed in commit c841572.

Cyfrin: Verified.