Securitize Bridge CCTPv2 Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

MrPotatoMagic

October 7, 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 Securitize Bridge protocol enables cross-chain interoperability for tokenized real-world assets (RWAs) and USDC stablecoin transfers via Circle's Cross-Chain Transfer Protocol (CCTP) v1 & v2. The system consists of two primary bridge contracts: `SecuritizeBridge` for bridging DS Tokens (digitally securitized tokens representing traditional securities) and `USDCBridge` (with its upgraded V2 variant) for USDC transfers using Wormhole integration with CCTP.

`SecuritizeBridge` facilitates the movement of DS Tokens between chains while maintaining regulatory compliance through integration with Securitize's registry service. When users bridge their tokens, the source chain burns the tokens and transmits both the transfer details and the investor's compliance attributes (KYC/AML data, accreditation status, and jurisdiction information) to the destination chain via Wormhole's message-passing infrastructure. Upon receipt, the destination chain mints equivalent tokens to the recipient while simultaneously updating the local registry with the investor's compliance profile, ensuring seamless regulatory adherence across jurisdictions.

The USDC bridge components implement a dual-protocol architecture, combining Wormhole's cross-chain messaging with Circle's native CCTP for secure stablecoin transfers. The V2 implementation attempts to integrate with Circle's latest CCTP v2 features, including fast finality options and automated attestation handling. The bridge burns USDC on the source chain through Circle's `TokenMessenger` contract, while Wormhole relays the transfer metadata and CCTP attestation to enable minting on the destination chain. This design aims to provide users with automated, single-transaction cross-chain transfers without requiring manual attestation retrieval or separate destination chain transactions.

Both bridges employ access control with designated bridge callers authorized to initiate transfers, upgradeable proxy patterns for future improvements, and configurable parameters such as gas limits and chain-specific addresses. The protocol's architecture prioritizes security through established infrastructure providers (Wormhole and Circle) while maintaining the regulatory compliance requirements essential for tokenized securities through Securitize's identity and registry systems.

5 Audit Scope

The focus of this audit was `USDCBridgeV2` integration between Wormhole & Circle CCTPv2, though other contracts were also reviewed:

```
contracts/bridge/SecuritizeBridge.sol
contracts/bridge/USDCBridgeV2.sol
contracts/utils/BaseContract.sol
contracts/utils/BaseRBACContract.sol
contracts/utils/Proxies.sol
contracts/wormhole/WormholeCCTPUpgradeable.sol
```

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Securitize Bridge CCTPv2 smart contracts provided by Securitize. In this period, a total of 32 issues were found.

The findings consist of 4 Medium and 4 Low severity issues with the remainder being informational and gas optimizations.

Integration Testing

One difficulty during this audit was a lack of officially available documentation and code examples from Wormhole on Solidity integration with CCTPv2; at the time of audit:

- Wormhole's official documentation on Solidity Wormhole <-> CCTP integration related to v1
- Wormhole's official code examples similarly related to Wormhole integration with CCTPv1
- Wormhole's publicly available off-chain code only integrated with CCTPv1

We overcame this limitation by two ways:

1) both Cyfrin and Securitize reached out to our contacts at Wormhole to get integration Wormhole <-> CCTPv2 specifications: [1, 2, 3]
2) we simplified the `USDCBridgeV2` contract while keeping the same core logic, wrote some Foundry scripts and performed live integration testing successfully bridging 10 USDC from Ethereum Sepolia to Base Sepolia using Wormhole with CCTPv2

We provided our integration test code and scripts as an additional audit deliverable and list relevant accounts and transactions here:

- User Ethereum `https://sepolia.etherscan.io/address/0x3088066410183071099832a606f30729e9fe7891`
- Bridge Ethereum `https://sepolia.etherscan.io/address/0x702E25b16425854F425606dF59404e74E176c940`
- User Base `https://sepolia.basescan.org/address/0x3088066410183071099832a606f30729e9fe7891`
- Bridge Base `https://sepolia.basescan.org/address/0x702E25b16425854F425606dF59404e74E176c940`
- Can see that the Base user token transfers have one single successful delivery of 10 USDC `https://sepolia.basescan.org/address/0x3088066410183071099832a606f30729e9fe7891#tokentxns`
- The transaction in wormholescan `https://wormholescan.io/#/tx/0xcb06a6ac71d10584f960036cb25c8d62f1507416812` `network=Testnet&view=overview`

Rebasing Multipler Cross-Chain Risks

The latest version of `DSToken` has a rebasing multiplier feature that can increase or decrease. If the multiplier is about to increase, investors can unintentionally lose shares if they bridge tokens before the multiplier increases. On chains where front-running is possible, investors can take advantage of this knowledge to prevent their share count from being reduced by negative rebasing:

- Alice an investor knows the rebasing multiplier is about to decrease on Ethereum (for example by seeing the incoming negative rebase transaction in the mempool)
- Alice front-runs the negative rebase transaction to bridge her tokens from Ethereum to Arbitrum
- the negative rebase transaction goes through on Ethereum, Arbitrum etc
- Alice's bridging transaction finishes on Arbitrum; Alice gets more shares than she should

Possible operational mitigations include:

- pause new bridging transactions prior to updating rebasing multipler
- verify there are no in-transit bridging transactions or wait until all complete
- update rebasing multiplier for specific `DSToken` on all chains (ideally using private mempools on chains where front-running is possible)
- unpause allowing new bridging transactions
- using shares instead of tokens as the cross-chain message parameter

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 Bridge CCTPv2
Repository	bc-securitize-bridge-sc
Commit	fbbbc11339b5...
Fix Commit	10c9d4a896e4...
Audit Timeline	Sep 22nd - Sep 26th, 2025
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	4
Low Risk	4
Informational	15
Gas Optimizations	9
Total Issues	32

Summary of Findings

[M-1] Uninitialized CCTP domain mapping can send USDC to the incorrect blockchain	Resolved
[M-2] Hard-coding 0 max fee with fast finality is incompatible as this combination commonly has minimum fees of 1	Resolved
[M-3] Investors using smart contract wallets may have their destination chain tokens issued to an address they don't control	Acknowledged
[M-4] Bridging `DSToken` back-and-forth between chains causes `totalIssuance` cap to be reached, preventing further issuances and cross-chain transfers	Resolved
[L-1] Upgradeable contracts which are inherited from should use ERC7201 namespaced storage layouts or storage gaps to prevent storage collision	Acknowledged
[L-2] No way to retrieve ETH sent with call to `SecuritizeBridge::receiveWormholeMessages`	Resolved
[L-3] Don't use `transfer` to send ETH	Resolved
[L-4] Inconsistent usage of `whenNotPaused` modifier for bridging fulfillment in `SecuritizeBridge` and `USDCBridgeV2`	Acknowledged
[I-01] Use named imports	Resolved
[I-02] Use named mapping parameters to make explicit the purpose of keys and values	Resolved
[I-03] Emit missing events on important parameter changes	Resolved
[I-04] Use `SafeERC20` approval and transfer functions instead of standard IERC20 functions	Resolved
[I-05] Prefer `Ownable2StepUpgradeable` instead of `OwnableUpgradeable` in `BaseContract`	Acknowledged
[I-06] Rename `SecuritizeBridge::whChainId` to `whRefundChainId`	Acknowledged
[I-07] Upgradeable contracts should call `_disableInitializers` in constructor	Resolved
[I-08] Use `addressNotZero` modifier on `USDCBridgeV2::setBridgeAddress`	Resolved
[I-09] Misleading event emission in `USDCBridgeV2::addBridgeCaller, removeBridgeCaller` when role was not granted or revoked	Acknowledged
[I-10] Remove unused function `USDCBridgeV2::_redeemUSDC`	Resolved
[I-11] Uninitialized country for valid investor wallets allows bypassing US compliance lockup period	Acknowledged
[I-12] `USDCBridgeV2` can't bridge to non-EVM chains even though Wormhole and Circle CCTP support this	Acknowledged
[I-13] Remove unused imports	Resolved
[I-14] Follow function declaration solidity style guide in `BaseContract`	Resolved
[I-15] Pending/re-executable messages sourced from old bridge addresses will not be executable if bridge address is updated	Acknowledged
[G-1] Cache storage to prevent identical storage reads	Acknowledged

[G-2] Fail fast without doing unnecessary work	Resolved
[G-3] Use `targetAddress` instead of `bridgeAddresses[targetChain]` for check in `SecuritizeBridge::bridgeDSTokens`	Resolved
[G-4] Refactor `SecuritizeBridge::bridgeDSTokens` and `quoteBridge` to use `internal` function saves 2 storage reads per bridging transaction	Resolved
[G-5] Use named return values where this can eliminate local variables	Acknowledged
[G-6] Refactor `SecuritizeBridge::validateLockedTokens` to take `dsServiceConsumer` as input parameter	Resolved
[G-7] Use `ReentrancyGuardTransientUpgradeable` for faster `nonReentrant` modifiers	Acknowledged
[G-8] Remove unused return from `USDCBridgeV2:_sendUSDCWithPayloadToEvm`	Acknowledged
[G-9] Pass `_refundChain` as input to `USDCBridgeV2::_buildCCTPKey` saves 1 storage read and external call	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 Uninitialized CCTP domain mapping can send USDC to the incorrect blockchain

Description: `USDCBridgeV2::chainIdToCCTPDomain` maps Wormhole chain IDs to Circle's CCTP domain IDs:

```
mapping(uint16 => uint32) public chainIdToCCTPDomain;

function getCCTPDomain(uint16 _chain) internal view returns (uint32) {
    return chainIdToCCTPDomain[_chain];  // @audit returns 0 if not set!
}
```

Impact: When this mapping isn't initialized for a wormhole chain id, it returns 0 by default (Solidity's default value for `uint32`). However Circle's CCTP domain 0 is Ethereum mainnet. So if a mapping has not been configured for a given wormhole chain id eg (6 for Avalanche),`USDCBridgeV2::_transferUSDC` will happily send USDC to Ethereum instead of Avalanche:

```
        circleTokenMessenger.depositForBurn(
            _amount,
            getCCTPDomain(_targetChain), // @audit 0 by default = Ethereum mainnet
            targetAddressBytes32,        // mintRecipient on destination
            USDC,           // burnToken
            destinationCallerBytes32,        // destinationCaller (restrict who can mint)
            0,
            1000
        );
```

Recommended Mitigation: The simplest option is to change `USDCBridgeV2::getCCTPDomain` to only allow domain 0 for wormhole's Ethereum chain id:

```
function getCCTPDomain(uint16 _chain) internal view returns (uint32 domain) {
    domain = chainIdToCCTPDomain[_chain];
    // Wormhole ChainID 2 = Ethereum https://wormhole.com/docs/products/reference/chain-ids/
    // Only allow CCTP Domain 0 for Ethereum
    ↪  https://developers.circle.com/cctp/cctp-supported-blockchains#cctp-v2-supported-domains
    require(domain != 0 || _chain == 2, "CCTP domain not configured");
}
```

Another potential solution is to change `setBridgeAddress` such that it always sets the CCTP domain as well eg:

```
    function setBridgeAddress(uint16 _chainId, address _bridgeAddress, uint32 _cctpDomain) external
    ↪  override onlyRole(DEFAULT_ADMIN_ROLE) {
        bridgeAddresses[_chainId] = _bridgeAddress;
        chainIdToCCTPDomain[_chain] = _cctpDomain;
        emit BridgeAddressAdd(_chainId, _bridgeAddress, _cctpDomain);
    }
```

Also consider changing `removeBridgeAddress` to delete from `chainIdToCCTPDomain` eg:

```
    function removeBridgeAddress(uint16 _chainId) external override onlyRole(DEFAULT_ADMIN_ROLE) {
        delete bridgeAddresses[_chainId];
        delete chainIdToCCTPDomain[_chainId];
        emit BridgeAddressRemove(_chainId);
    }
```

Securitize: Fixed in commit d750854 by removing `setBridgeAddress` and adding a new function `setCCTPBridgeAddress` which enforces that CCTP Domain is configured at the same time as target bridge address for the same wormhole chain id. Also changed `removeBridgeAddress` to clear both mappings together as well.

Cyfrin: Verified.

7.1.2 Hard-coding 0 max fee with fast finality is incompatible as this combination commonly has minimum fees of 1

Description: Circle's CCTPv2 Technical Guide provides the following relevant information:

- Messages with `minFinalityThreshold` of 1000 or lower are considered Fast messages
- Messages with `minFinalityThreshold` of 2000 are considered Standard messages (in practice everything > 1000 is considered Standard)
- The applicable fee should be retrieved every time before executing a transaction using this API

The provided API requires specifying the CCTP input and output domains. Using the `wget` form of the API, the minimum fees for fast finality (1000) are typically 1:

- Ethereum -> Avalanche

```
$ wget --quiet \
  --method GET \
  --header 'Content-Type: application/json' \
  --output-document \
  - https://iris-api-sandbox.circle.com/v2/burn/USDC/fees/0/1
[{"finalityThreshold":1000,"minimumFee":1},{"finalityThreshold":2000,"minimumFee":0}]%
```

- Ethereum -> Solana

```
$ wget --quiet \
  --method GET \
  --header 'Content-Type: application/json' \
  --output-document \
  - https://iris-api-sandbox.circle.com/v2/burn/USDC/fees/0/5
[{"finalityThreshold":1000,"minimumFee":1},{"finalityThreshold":2000,"minimumFee":0}]%
```

Impact: Many CCTP cross-domain transfers have a minimum fee of 1 for fast finality, but `USDCBridgeV2::_transferUSDC` hard-codes a maximum fee of 0 with fast finality 1000:

```
circleTokenMessenger.depositForBurn(
    _amount,
    getCCTPDomain(_targetChain),
    targetAddressBytes32,        // mintRecipient on destination
    USDC,           // burnToken
    destinationCallerBytes32,        // destinationCaller (restrict who can mint)
    0,   // @audit maximum fee
    1000 // @audit fast finality
);
```

This combination is incompatible and will result in many cross-domain transfers unable to use fast finality, reverting to standard finality. If the minimum fee for standard finality ever becomes > 0, this would cause all attempted cross-domain transfers to revert since the automatic downgrade to standard finality would no longer be possible.

Recommended Mitigation: Ideally the maximum fee and finality should be provided as inputs:

- current fee bps should be retrieved off-chain using the provided API for the desired domain combination
- multiply fee bps by the amount to be transferred to calculate the maximum fee
- pass maximum fee and desired finality as inputs when calling `circleTokenMessenger.depositForBurn`

At least there should be a way to change the maximum fee, it shouldn't be hard-coded to zero as this causes the protocol to become unusable if standard finality fees become non-zero.

Securitize: Fixed in commit 0d3e50d by:

- always using standard finality
- max fee is now a variable so we can change it if Circle increases standard finality fees in the future

Cyfrin: Verified.

7.1.3 Investors using smart contract wallets may have their destination chain tokens issued to an address they don't control

Description: `SecuritizeBridge::bridgeDSTokens` encodes `_msgSender()` in the payload message to be destination address for bridged tokens delivered on the destination chain; this is expected to represent the investor's wallet address:

```
// Send Relayer message
        wormholeRelayer.sendPayloadToEvm{value: msg.value} (
            targetChain,
            targetAddress,
            abi.encode(
                investorDetail.investorId,
                value,
                _msgSender(), // @audit destination address of bridged tokens
                investorDetail.country,
                investorDetail.attributeValues,
                investorDetail.attributeExpirations
            ), // payload
            0, // no receiver value needed since we"re just passing a message
            gasLimit,
            whChainId,
            _msgSender()
        );
```

`SecuritizeBridge::receiveWormholeMessages` executed on the destination chain adds the address to the investor's account and issues tokens to it:

```
        address[] memory investorWallets = new address[](1);
        investorWallets[0] = investorWallet;

        // @audit assumes investor controls same wallet address
        // on destination chain - not necessarily true for smart contract wallets
        registryService.updateInvestor(investorId, investorId, country, investorWallets, attributeIds,
        ↪  attributeValues, attributeExpirations);
        dsToken.issueTokens(investorWallet, value);
```

Impact: If the investor's wallet is a smart contract wallet (multisig/AA wallets - which can have different addresses on different chains), there is no guarantee that the investor will have their smart wallet contract deployed at the same address on the destination chain.

In an unlikely worst-case scenario, the address on the destination chain is owned by a different EOA user who is not an investor. This causes `SecuritizeBridge::receiveWormholeMessages` to give a foreign user control over the original investor's account because it registers the foreign user's address as belonging to the investor on the destination chain:

- Alice uses a Gnosis Safe at 0xAAA... on Ethereum
- Alice bridges her DSTokens to Arbitrum
- Bob (malicious) controls EOA 0xAAA... on Arbitrum (same address, different controller)
- The bridge registers 0xAAA... on Arbitrum as Alice's wallet
- Bob now controls all of Alice's tokens on Arbitrum

Recommended Mitigation: A simple solution is to allow investors to provide valid destination chain wallets and encode it in the payload, however this may also be ripe for abuse by allowing investors to claim wallets they don't actually control.

In another cross-chain regulated TradFi protocol we've audited, the protocol had functionality to bridge investor addresses and credentials cross-chain, and this occurred separately to token bridging.

The cross-chain token bridging could only work if the destination address and its associated credentials had already been bridged; this ensured that the destination address was always valid.

Securitize: Acknowledged; we are aware of this edge case as it was raised in a previous audit. If the destination address is not an EOA and the investor has no control of tokens in destination chain, tokens remain locked due to lock-up period and Securitize has enough time to seize/burn the tokens.

7.1.4 Bridging `DSToken` back-and-forth between chains causes `totalIssuance` cap to be reached, preventing further issuances and cross-chain transfers

Description: `StandardToken::totalIssuance` is not decreased by burns but is used to enforce maximum cap, since `totalIssuance` is supposed to track the total number of tokens ever issued, not the current "supply".

However there is an interesting consequence to this when considering cross-chain bridging via `SecuritizeBridge`; `receiveWormholeMessages` calls `DSToken::issueTokens` on the destination chain which increases `StandardToken::totalIssuance`.

Impact: Consider this scenario:

- Alice bridges from Ethereum -> Arbitrum with 1000 `DSToken`
- Alice bridges back from Arbitrum -> Ethereum with the "same" 1000 `DSToken`
- Alice keeps doing this over and over again

This process continually increases the `totalIssuance` on both chains, even though it is just the same tokens going back and forth; at some point this will cause the cap to be hit on one of the chains. This doesn't even require malicious investors, just investors who bridge back-and-forth frequently.

Once the cap is hit further issuances and cross-chain transfers will revert on that chain.

Recommended Mitigation: Potential mitigations include:

- have bridging actually decrement `totalIssuance` on the source chain
- have `SecuritizeBridge::receiveWormholeMessages` call `DSToken::issueTokensCustom` passing a `reason == "BRIDGING"` then in `TokenLibrary::issueTokensCustom` don't increment `totalIssuance` for `"BRIDGING"` reason
- track the number of bridged tokens separately and modify the cap check to account for this

Securitize: Fixed in commit c2e62c9; the cap was deprecated and associated checks removed. There is a similar compliance-related check that uses `totalSupply` so correctly accounts for burns.

Cyfrin: Verified.

7.2 Low Risk

7.2.1 Upgradeable contracts which are inherited from should use ERC7201 namespaced storage layouts or storage gaps to prevent storage collision

Description: The protocol has upgradeable contracts which other contracts inherit from. These contracts should either use:

- ERC7201 namespaced storage layouts - example
- storage gaps (though this is an older and no longer preferred method)

The ideal mitigation is that all upgradeable contracts use ERC7201 namespaced storage layouts; without using one of the above two techniques storage collision can occur during upgrades. The affected contracts are:

- `CCTPBase`
- `CCTPBase`

Securitize: Acknowledged; `CCTPBase` and `CCTPBase` will be removed later as they will become deprecated.

7.2.2 No way to retrieve ETH sent with call to `SecuritizeBridge::receiveWormholeMessages`

Description: `SecuritizeBridge::receiveWormholeMessages` is marked as `payable` however:

- it does nothing with `msg.value`
- there is no function in `SecuritizeBridge` to withdraw ETH

Impact: If ETH should be sent along with the call to `SecuritizeBridge::receiveWormholeMessages`, it will be stuck in the contract unable to be retrieved.

Recommended Mitigation: Add a function `withdrawETH` that allows the contract owner to withdraw the contract's ETH balance.

Securitize: Fixed in commits 923e50e, 2b18646 by adding a `withdrawETH` function the owner can call.

Cyfrin: Verified.

7.2.3 Don't use `transfer` to send ETH

Description: Using `transfer` to send ETH hasn't been recommended since the Istanbul hard fork in December 2019 which increased the gas cost of some operations; `transfer` hard-codes gas to 2300 which can cause receiving functions to revert hence is not future-proof.

The recommended way to send eth is to use `call` and Solady has an optimized way of doing this in SafeTransferLib::safeTransferETH.

`transfer` also may not work as expected on L2s, for example there was this incident which resulted in 921 ETH being stuck on zksync Era due to the smart contract using transfer to send eth, though eventually zksync developed a solution to rescue the stuck eth.

Affected code in `USDCBridgeV2::withdrawETH`:

```
bridge/USDCBridgeV2.sol
202:        _to.transfer(amount);
```

Securitize: Fixed in commit 2b18646.

Cyfrin: Verified.

7.2.4 Inconsistent usage of `whenNotPaused` modifier for bridging fulfillment in `SecuritizeBridge` and `USDCBridgeV2`

Description: `USDCBridgeV2` inherits pause functionality from `BaseRBACContract`. Currently `USDCBridgeV2::receiveWormholeMessages` does not apply the `whenNotPaused` modifier allowing receipt of tokens to occur on a paused destination chain contract.

In contrast `SecuritizeBridge::receiveWormholeMessages` does have the `whenNotPaused` preventing receipt of tokens on a paused destination chain contract.

Recommended Mitigation: There is an inconsistent usage of `whenNotPaused` modifier for bridging fulfillment; if there is no good reason for this difference then it should be made consistent. Either:

- don't allow bridging fulfillment when destination contracts are paused
- allow bridging fulfillment when destination contracts are paused but don't allow new bridging requests when paused

Securitize: Acknowledged; for now we prefer to leave the modifiers unchanged. We can prevent issuances on flying bridges for dsTokens, as we control them and we can issue, burn, etc. We do not want to pause USDC flying bridges, as we do not have control over usdc circle stable coin.

7.3 Informational

7.3.1 Use named imports

Description: Use named imports consistently throughout the codebase.

Securitize: Fixed in commit 85ca7bb.

Cyfrin: Verified.

7.3.2 Use named mapping parameters to make explicit the purpose of keys and values

Description: Use named mapping parameters to make explicit the purpose of keys and values:

```
wormhole/WormholeCCTPUpgradeable.sol
79:    mapping(uint16 => uint32) public chainIdToCCTPDomain;

bridge/USDCBridgeV2.sol
63:    mapping(uint16 => address) public bridgeAddresses;
64:    mapping(uint16 => uint32) public chainIdToCCTPDomain;

bridge/SecuritizeBridge.sol
40:    mapping(uint16 => address) public bridgeAddresses;
```

Securitize: Fixed in commit 40f4db0.

Cyfrin: Verified.

7.3.3 Emit missing events on important parameter changes

Description: Emit missing events on important parameter changes:

- `CCTPSender::setCCTPDomain`
- `USDCBridgeV2::setCCTPDomain`

Securitize: Fixed in commit cd8c8ad for `USDCBridgeV2`, leaving the other as it will be deprecated.

Cyfrin: Verified.

7.3.4 Use `SafeERC20` approval and transfer functions instead of standard IERC20 functions

Description: Use SafeERC20::forceApprove and `safeTransfer` functions instead of standard IERC20 functions:

```
wormhole/WormholeCCTPUpgradeable.sol
121:        IERC20(USDC).approve(address(circleTokenMessenger), amount);

bridge/USDCBridgeV2.sol
150:        IERC20(USDC).transferFrom(_msgSender(), address(this), _amount);
264:        IERC20(USDC).approve(address(circleTokenMessenger), _amount);
```

Securitize: Fixed in commit d75ac6f.

Cyfrin: Verified.

7.3.5 Prefer `Ownable2StepUpgradeable` instead of `OwnableUpgradeable` in `BaseContract`

Description: In `BaseContract` prefer Ownable2StepUpgradeable instead of `OwnableUpgradeable`.

Securitize: Acknowledged.

7.3.6 Rename `SecuritizeBridge::whChainId` to `whRefundChainId`

Description: The only purpose of `SecuritizeBridge::whChainId` is to be the wormhole refund chain id; rename it to something like `whRefundChainId` which accurately describes its purpose.

Also there are no functions to change this value; consider adding one if this may be required.

Securitize: Acknowledged.

7.3.7 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:

- `SecuritizeBridge`
- `USDCBridgeV2`

Securitize: Fixed in commit 4b7f654.

Cyfrin: Verified.

7.3.8 Use `addressNotZero` modifier on `USDCBridgeV2::setBridgeAddress`

Description: Use `addressNotZero` modifier on `USDCBridgeV2::setBridgeAddress`:

```
-   function setBridgeAddress(uint16 _chainId, address _bridgeAddress) external override
↪   onlyRole(DEFAULT_ADMIN_ROLE) {
+   function setBridgeAddress(uint16 _chainId, address _bridgeAddress) external override
↪   addressNotZero(_bridgeAddress) onlyRole(DEFAULT_ADMIN_ROLE) {
```

Securitize: Fixed in commit f51d885.

Cyfrin: Verified.

7.3.9 Misleading event emission in `USDCBridgeV2::addBridgeCaller, removeBridgeCaller` when role was not granted or revoked

Description: `AccessControlUpgradeable::_grantRole,_revokeRole` return `bool` to indicate whether the role has been granted or revoked.

But `USDCBridgeV2::removeBridgeCaller, addBridgeCaller` ignore the returned `bool` since they call the `public` functions and always emits an event even if no role was granted or revoked.

Recommended Mitigation:

```
    function addBridgeCaller(address _account) external override addressNotZero(_account)
    ↪   onlyRole(DEFAULT_ADMIN_ROLE) {
-       grantRole(BRIDGE_CALLER, _account);
-       emit BridgeCallerAdded(_account);
+       if(_grantRole(BRIDGE_CALLER, _account)) emit BridgeCallerAdded(_account);
    }

    function removeBridgeCaller(address _account) external override addressNotZero(_account)
    ↪   onlyRole(DEFAULT_ADMIN_ROLE) {
-       revokeRole(BRIDGE_CALLER, _account);
-       emit BridgeCallerRemoved(_account);
+       if(_revokeRole(BRIDGE_CALLER, _account)) emit BridgeCallerRemoved(_account);
```

```
    }
```

Securitize: Acknowledged.

7.3.10 Remove unused function `USDCBridgeV2::_redeemUSDC`

Description: The `private` function `USDCBridgeV2::_redeemUSDC` is not used anywhere; remove it.

Securitize: Fixed in commit 07a872e.

Cyfrin: Verified.

7.3.11 Uninitialized country for valid investor wallets allows bypassing US compliance lockup period

Description: One way that wallets can be registered in the protocol is via `RegistryService.sol::addWallet`, meanwhile the country field for the registered wallet can be set separately using a separate function `setCountry`.

Hence there can be a small window during these two transactions which leaves the country field as an empty string (default value) for a valid wallet .

Impact: Since during bridging of DS tokens the `country` would be an empty string, the `region` memory variable will also default to 0. This would mean the `lockPeriod` would store and use the non-US lock period:

```
string memory country = registryService.getCountry(investorId);
uint256 region = complianceConfigurationService.getCountryCompliance(country);

uint256 lockPeriod = (region == US) ? complianceConfigurationService.getUSLockPeriod() :
↪  complianceConfigurationService.getNonUSLockPeriod();
        uint256 availableBalanceForTransfer =
        ↪  complianceService.getComplianceTransferableTokens(_msgSender(), block.timestamp,
        ↪  uint64(lockPeriod));
```

This is problematic since:

1. Uninitialized country fields for valid wallets are allowed to perform bridging
2. If the intended country = US, the lockPeriod uses the non-US lock period instead.

This small period of time can be used by malicious wallet owners to use a lower `lockPeriod` (if non-US lock time is smaller than US lock time).

Recommended Mitigation: Check if country is an empty string in `validateLockedTokens` and revert if true.

Securitize: Acknowledged; If the investor is bridging tokens, they were already minted/issued. Hence compliance rules were validated including country and region at the time of token minting/issuance.

7.3.12 `USDCBridgeV2` can't bridge to non-EVM chains even though Wormhole and Circle CCTP support this

Description: Both Wormhole and Circle CCTP support bridging between EVM and non-EVM chains, however `USCBridgeV2` prevents bridging to non-EVM chains since:

1) `_sendUSDCWithPayloadToEvm` always calls `wormholeRelayer.sendToEvm`
2) `bridgeAddresses[_targetChain]` stores the target bridges using `address`, but this is not compatible with target bridges on non-EVM chains such as Solana

Impact: Bridging to non-EVM chains is not supported.

Recommended Mitigation: If bridging to non-EVM chains should be supported:

- use `wormholeRelayer.send` instead of `sendToEvm`
- `bridgeAddresses[_targetChain]` should store target bridges as `bytes32` then cast them to `address` when bridging to EVM chains

- generally addresses for remote chains should be passed as input, stored and used using `bytes32` not `address`

Securitize: Acknowledged; by design for now.

7.3.13 Remove unused imports

Description: Remove unused imports:

- `USDCBridgeV2.sol`

```
28:import {IBridge} from "./IBridge.sol";
```

Securitize: Fixed in commit a45cb7e.

Cyfrin: Verified.

7.3.14 Follow function declaration solidity style guide in `BaseContract`

Description: Functions `pause` and `unpause` defines the visibility `public` after the modifier `onlyOwner`. As per the Solidity Style Guide, the order expects modifiers to be placed after visibility declarations.

```
function pause() onlyOwner external {
        _pause();
    }

    function unpause() onlyOwner external {
        _unpause();
    }
```

Recommended Mitigation: Update the code in the following way:

```
function pause() external onlyOwner {
        _pause();
    }

    function unpause() external onlyOwner {
        _unpause();
    }
```

Securitize: Fixed in commit 61fb6a6.

Cyfrin: Verified.

7.3.15 Pending/re-executable messages sourced from old bridge addresses will not be executable if bridge address is updated

Description: SecuritizeBridge and USDCBridgeV2 provide owner with the ability to update bridge addresses. This can be done incase a new bridge address is expected to be used and the previous one is being deprecated.

```
function setBridgeAddress(uint16 chainId, address bridgeAddress) external override onlyOwner {
        bridgeAddresses[chainId] = bridgeAddress;
        emit BridgeAddressAdd(chainId, bridgeAddress);
    }
```

Impact: One important behaviour to be aware of here is that there could be pending or failed destination messages waiting to be delivered. If the bridge address is updated before these are executed, it is possible for them to never be executable again unless the bridge address is updated to the previous one.

Recommended Mitigation: Consider waiting for delivery of pending messages and execute any failed destination messages before updating the bridge address for a chain.

Securitize: Acknowledged.

7.4 Gas Optimization

7.4.1 Cache storage to prevent identical storage reads

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

- `contracts/wormhole/WormholeCCTPUpgradeable.sol`

```
// cache `USDC` in `redeemUSDC`
65:         uint256 beforeBalance = IERC20(USDC).balanceOf(address(this));
67:         return IERC20(USDC).balanceOf(address(this)) - beforeBalance;
```

- `contracts/bridge/SecuritizeBridge.sol`

```
// cache `dsToken` in `bridgeDSTokens`
73:         require(dsToken.balanceOf(_msgSender()) >= value, "Not enough balance in source chain to
↪  bridge");
88:         dsToken.burn(_msgSender(), value, BRIDGE_REASON);
108:         emit DSTokenBridgeSend(targetChain, address(dsToken), _msgSender(), value);

// cache `dsToken` in `receiveWormholeMessages`
144:         dsToken.issueTokens(investorWallet, value);
146:         emit DSTokenBridgeReceive(sourceChain, address(dsToken), investorWallet, value);
```

- `contracts/bridge/USDCBridgeV2.sol`

```
// cache `USDC` in `sendUSDCCrossChainDeposit`
143:         if (IERC20(USDC).balanceOf(_msgSender()) < _amount) {
150:         IERC20(USDC).transferFrom(_msgSender(), address(this), _amount);
// also change `_transferUSDC` to take cached `USDC` as parameter to save
// 2 storage reads inside `_transferUSDC`;
// cache `USDC, `circleTokenMessenger` in `_transferUSDC`
264:         IERC20(USDC).approve(address(circleTokenMessenger), _amount);
268:         circleTokenMessenger.depositForBurn(
272:             USDC,          // burnToken
```

Securitize: Acknowledged.

7.4.2 Fail fast without doing unnecessary work

Description: If a transaction is going to revert, then revert as fast as possible without doing unnecessary work. Strategies to achieve this include:

- perform all input-related validation first
- read only enough storage or make enough external calls to perform the next validation step

For example in `SecuritizeBridge::bridgeDSTokens`:

```
    function bridgeDSTokens(uint16 targetChain, uint256 value) external override payable whenNotPaused {
        // @audit why do all this work...
        uint256 cost = quoteBridge(targetChain);
        require(msg.value >= cost, "Transaction value should be equal or greater than quoteBridge
        ↪  response");
        require(dsToken.balanceOf(_msgSender()) >= value, "Not enough balance in source chain to
        ↪  bridge");
        address targetAddress = bridgeAddresses[targetChain];
        require(bridgeAddresses[targetChain] != address(0), "No bridge address available");

        IDSRegistryService registryService =
        ↪  IDSRegistryService(dsServiceConsumer.getDSService(dsServiceConsumer.REGISTRY_SERVICE()));
        require(registryService.isWallet(_msgSender()), "Investor not registered");

        // @audit ...if txn will revert here due to invalid input?
        require(value > 0, "DSToken value must be greater than 0");
```

And in `USDCBridgeV2::sendUSDCCrossChainDeposit`

```
    function sendUSDCCrossChainDeposit(
        uint16 _targetChain,
        address _recipient,
        uint256 _amount
    ) external override whenNotPaused nonReentrant onlyRole(BRIDGE_CALLER) {
        uint256 deliveryCost = quoteBridge(_targetChain);
        // @audit why perform this storage read...
        address targetBridge = bridgeAddresses[_targetChain];
        // @audit ...if this check will just revert? Perform this check immediately
        // after `uint256 deliveryCost = quoteBridge(_targetChain);`
        if (address(this).balance < deliveryCost) {
            revert InsufficientContractBalance();
        }
        // @audit why perform this check here?
        if (IERC20(USDC).balanceOf(_msgSender()) < _amount) {
            revert NotEnoughBalance();
        }
        // @audit if it is going to revert from this? Perform this check immediately
        // after ` address targetBridge = bridgeAddresses[_targetChain];`
        if (targetBridge == address(0)) {
            revert BridgeAddressUndefined();
        }
```

Securitize: Fixed in commit 2ad89cf.

Cyfrin: Verified.

7.4.3 Use `targetAddress` instead of `bridgeAddresses[targetChain]` for check in `SecuritizeBridge::bridgeDSTokens`

Description: Use `targetAddress` instead of `bridgeAddresses[targetChain]` for check in `SecuritizeBridge::bridgeDSTokens`:

```
        address targetAddress = bridgeAddresses[targetChain];
-       require(bridgeAddresses[targetChain] != address(0), "No bridge address available");
+       require(targetAddress != address(0), "No bridge address available");
```

Securitize: Fixed in commit 9081b85.

Cyfrin: Verified.

7.4.4 Refactor `SecuritizeBridge::bridgeDSTokens` and `quoteBridge` to use `internal` function saves 2 storage reads per bridging transaction

Description: `SecuritizeBridge::bridgeDSTokens`:

- L71 calls `quoteBridge` which reads `wormholeRelayer` and `gasLimit` from storage
- L91 calls `wormholeRelayer.sendPayloadToEvm` which reads `wormholeRelayer` from storage again
- L108 reads `gasLimit` from storage again

Storage reads are expensive; refactor like this to avoid identical storage reads here:

```
// new internal function
    function _quoteBridge(IWormholeRelayer relayer, uint256 _gasLimit, uint16 targetChain) internal
    ↪  view returns (uint256 cost) {
        (cost, ) = relayer.quoteEVMDeliveryPrice(targetChain, 0, _gasLimit);
    }
```

```
// modify `quoteBridge` to use new internal function
    function quoteBridge(uint16 targetChain) public override view returns (uint256 cost) {
        (cost, ) = _quoteBridge(wormholeRelayer, gasLimit, targetChain);
    }

// in `bridgeDSTokens` to cache `wormholeRelayer` and `gasLimit`
// then pass them to `_quoteBridge` and use them at L91 & L108
```

The same optimization should also be applied to `USDCBridgeV2::sendUSDCCrossChainDeposit`, `quoteBridge` and `_sendUSDCWithPayloadToEvm`.

Securitize: Fixed in commit 47c1ad0.

Cyfrin: Verified.

7.4.5 Use named return values where this can eliminate local variables

Description: Use named return values where this can eliminate local variables:

- `SecuritizeBridge::getInvestorData`

Securitize: Acknowledged.

7.4.6 Refactor `SecuritizeBridge::validateLockedTokens` to take `dsServiceConsumer` as input parameter

Description: `SecuritizeBridge::bridgeDSTokens` at L77 reads `dsServiceConsumer` from storage then at L83 calls `validateLockedTokens`.

The internal function `validateLockedTokens` itself re-reads `dsServiceConsumer` from storage multiple times.

Reading from storage is expensive; instead:

- cache `dsServiceConsumer` once in `bridgeDSTokens`
- refactor `validateLockedTokens` to take `dsServiceConsumer` as an input parameter
- in `bridgeDSTokens` when calling `validateLockedTokens` pass the cached `dsServiceConsumer` as an input parameter

Securitize: Fixed in commits bcc83e0, 0cf5dd9.

Cyfrin: Verified.

7.4.7 Use `ReentrancyGuardTransientUpgradeable` for faster `nonReentrant` modifiers

Description: Use ReentrancyGuardTransientUpgradeable for faster `nonReentrant` modifiers:

```
bridge/USDCBridgeV2.sol
27:import {ReentrancyGuardUpgradeable} from
↪  "@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol";
51:contract USDCBridgeV2 is IUSDCBridge, IWormholeReceiver, BaseRBACContract,
↪  ReentrancyGuardUpgradeable {
86:        __ReentrancyGuard_init();
```

Securitize: Acknowledged.

7.4.8 Remove unused return from `USDCBridgeV2:_sendUSDCWithPayloadToEvm`

Description: Function `USDCBridgeV2::_sendUSDCWithPayloadToEvm()` returns the sequence number from the `wormholeRelayer.sendToEvm` external call. However, this value is never utilized thereafter.

Recommended Mitigation: Consider removing this variable.

Securitize: Acknowledged.

7.4.9 Pass `_refundChain` as input to `USDCBridgeV2::_buildCCTPKey` saves 1 storage read and external call

Description: `USDCBridgeV2::sendUSDCCrossChainDeposit` passes `wormhole.chainId()` as the third-to-last parameter:

```
        _sendUSDCWithPayloadToEvm(
            _targetChain,
            targetBridge, // address (on targetChain) to send token and payload to
            payload,
            0, // receiver value
            gasLimit,
            _amount,
            wormhole.chainId(), // @audit `_refundChain`
            address(this),
            deliveryCost
        );
```

But then `_sendUSDCWithPayloadToEvm` calls `_buildCCTPKey` which performs the same work again:

```
    function _buildCCTPKey() private view returns (MessageKey memory) {
        return MessageKey(CCTP_KEY_TYPE, abi.encodePacked(getCCTPDomain(wormhole.chainId()),
        ↪  uint64(0)));
    }
```

Refactor `_buildCCTPKey` to take `uint16 _whSourceChain` as an input parameter and use it like this:

```
    function _buildCCTPKey(uint16 _whSourceChain) private view returns (MessageKey memory) {
        return MessageKey(CCTP_KEY_TYPE, abi.encodePacked(getCCTPDomain(_whSourceChain), uint64(0)));
    }
```

Securitize: Acknowledged.