plainshift

August 25th, 2025

Cashmere CCTP

Comprehensive Security Assessment

contact@plainshift.io
plainshift.io

Table of Contents

Disclaimer

This security assessment represents a time-boxed security review using tooling and manual review methodologies. Our findings reflect our comprehensive evaluation of the materials provided in-scope and are specific to the commit hash referenced in this report.

The scope of this security assessment is strictly limited to the code explicitly specified in the report. External dependencies, integrated third-party services, libraries, and any other code components not explicitly listed in the scope have not been reviewed and are excluded from this assessment.

Any modifications to the reviewed codebase, including but not limited to smart contract upgrades, protocol changes, or external dependency updates will require a new security assessment, as they may introduce considerations not covered in the current review.

In no event shall Plainshift's aggregate liability for all claims, whether in contract or any other theory of liability, exceed the Services Fee paid for this assessment. The client agrees to hold Plainshift harmless against any and all claims for loss, liability, damages, judgments and/or civil charges arising out of exploitation of security vulnerabilities in the contracts reviewed.

By accepting this report, you acknowledge that deployment and implementation decisions rest solely with the client. Any reliance upon the information in this report is at your own discretion and risk. This disclaimer is governed by and construed in accordance with the laws specified in the engagement agreement between Plainshift and the client.

1. Introduction

Plainshift is a full-stack security firm built on the “shift left” security philosophy. We often work with teams early in the product development process to bring security to a greater organizational range than just smart contracts. From the web app, to fuzzing/formal verification, to a team’s operational security, full-stack security can only be achieved by first understanding there is no “scope” to fully protect the users that trust you.

We’re here to meaningfully revolutionize how teams approach security and guide them towards a holistic approach rather than the single sided approach so prevalent today.

Learn more about us at https://plainshift.io.

1.1. Executive Summary

Plainshift was tasked with reviewing the Cashmere CCTP contracts for Cashmere Labs from August 17th, 2025 to August 25th, 2025. We ultimately found and confirmed 4 medium, 4 low severity issues and 5 informational findings.

1.2. Project Timeline

Date	Phase
August 17th, 2025	Kickoff
August 25th, 2025	Audit End
August 25th, 2025	Delivery

1.3. Scope

Repositories	https://github.com/cashmere-prod/contracts-prod
Version	`5426c1dd037d5725f6b976b8828b314c566f83e6`
Contracts	Multi-chain implementation across Solana, EVM, Sui, and Aptos
Type	Rust (Solana), Solidity (EVM), Move (Sui/Aptos)
Platform	Multi-chain (Solana, Ethereum, Sui, Aptos)

1.4. Overview of Findings

Our comprehensive review yielded 4 medium, 4 low severity issues alongside 5 informational findings.

Severity/Impact Level	Count
High	0
Medium	4
Low	4
Informational	5

2. Findings

2.1. [M1] Gas drop limit of zero incorrectly treated as “no limit”

Target	`instructions/transfer/common.rs` (Solana), `CashmereCCTP.sol` (EVM), `transfer.move` (Aptos)
Severity	Medium
Category	Vulnerability

2.1.1. Description

On Solana, EVM, and Aptos, the validation logic for native gas drops checks if the configured limit for a destination domain is zero. If it is, the logic bypasses the limit check entirely, treating a zero-limit as an infinite limit. This is counter intuitive, as an administrator would likely set a limit to zero with the intention of disabling gas drops.

The vulnerable code pattern is as follows (example from Solana’s `pre_transfer`):

```
let native_gas_drop_limit = config.max_native_gas_drop[destination_domain as usize];
require!(native_gas_drop_limit == 0 || gas_drop_amount <= native_gas_drop_limit,
TransferError::GasDropLimitExceeded);
```

If `native_gas_drop_limit` is `0`, the first part of the `||` condition is `true`, so the entire check passes regardless of the `gas_drop_amount`.

2.1.2. Impact

An administrator misconfiguration, intended to disable gas drops by setting a limit to `0`, would enable the “no limit” scenario. This would allow users to request arbitrarily large gas drop amounts, leading to an uncapped drain of funds from the protocol’s gas collector wallets.

2.1.3. Recommendations

We recommend modifying the validation logic to treat a limit of zero as disallowing gas drops. This can be achieved by removing the bypass and simply checking if the requested amount is less than or equal to the limit, making the default “zero” state secure.

```
- require!(native_gas_drop_limit == 0 || gas_drop_amount <= native_gas_drop_limit,
  TransferError::GasDropLimitExceeded);
+ require!(gas_drop_amount <= native_gas_drop_limit, TransferError::GasDropLimitExceeded);
```

2.1.4. Remediation

Acknowledged.

2.2. [M2] Paused flag cannot be managed by Admin after deployment

Target	`instructions/admin.rs` (Solana), `transfer.move` (Sui)
Severity	Medium
Category	Vulnerability

2.2.1. Description

Both the Sui and Solana contracts contain a `paused` boolean flag in their global configuration state, which is correctly checked by `transfer` functions to halt operations. However, neither contract's administrative module exposes a function to modify this state after initialization. The `initialize_ix` function in Solana sets `paused = false`, but no function in `admin.rs` can change it.

This means the `paused` flag is permanently set to `false` upon deployment, making the emergency-stop functionality useless.

2.2.2. Impact

This prevents administrators from being able to halt the protocol in the event of a security incident. Without a "pause" switch, user funds would remain at risk until a full contract upgrade can be deployed.

2.2.3. Recommendations

We recommend implementing a new administrative function on both chains that provides a permissioned mechanism for an authorized admin to set the `paused` flag to `true` or `false`.

2.2.4. Remediation

Fixed.

2.3. [M3] Incorrect native token refund recipient

Target	`transfer.move` (Sui)
Severity	Medium
Category	Vulnerability

2.3.1. Description

In the Sui `prepare_deposit_for_burn_ticket` function, logic exists to calculate and return leftover native tokens (SUI) to the transaction's initiator as change. The implementation incorrectly uses the cross-chain `recipient` address as the destination for this refund instead of the onchain sender.

The flawed line of code is:

```
// return native change
transfer::public_transfer(native_fee_coin, recipient);
```

Here, `recipient` is the intended final owner of the assets on the destination chain, not the sender who initiated the transaction on Sui and is owed the change.

2.3.2. Impact

This results in a direct loss of funds for the user. Their native token change is sent to the cross-chain `recipient` address, an entity they do not control, instead of being returned to their own wallet.

2.3.3. Recommendations

The `public_transfer` call for the native token refund must be directed to the `ctx.sender` address to ensure the original payer receives their correct change.

```
- // return native change
- transfer::public_transfer(native_fee_coin, recipient);
+ // return native change
+ transfer::public_transfer(native_fee_coin, ctx.sender());
```

2.3.4. Remediation

Fixed.

2.4. [M4] ED25519 verifier ignores message size parameter

Target	`solana/src/utils/ed25519.rs`
Severity	Medium
Category	Vulnerability

2.4.1. Description

The `verify_ed25519_ix` helper function is responsible for validating offchain signatures. The function correctly deserializes the `message_data_size` field from the signature verification instruction but then fails to use this value. Instead, it creates a slice from the message offset to the very end of the instruction data, potentially creating a buffer of the wrong size for comparison.

```
// Deserialize the Ed25519 signature offsets
    let ed25519_offsets = Ed25519SignatureOffsets {
        signature_offset: u16::from_le_bytes([data[2], data[3]]),
        signature_instruction_index: u16::from_le_bytes([data[4], data[5]]),
        public_key_offset: u16::from_le_bytes([data[6], data[7]]),
        public_key_instruction_index: u16::from_le_bytes([data[8], data[9]]),
        message_data_offset: u16::from_le_bytes([data[10], data[11]]),
        message_data_size: u16::from_le_bytes([data[12], data[13]]),
        message_instruction_index: u16::from_le_bytes([data[14], data[15]]),
     };

 [..]

 let message_data = &verify_instruction.data[ed25519_offsets.message_data_offset as
 usize..];
```

2.4.2. Impact

Issue can cause valid transactions with correctly formatted instructions to fail. In a worst case, it compromises the integrity of the signature check, as the comparison happens against an incorrectly sized buffer, which could be exploited.

2.4.3. Recommendations

We recommend slicing the message data using both the provided offset and the `message_data_size` to ensure the exact intended message is used in the comparison. A bounds check should also be added for safety.

2.4.4. Remediation

Fixed.

2.5. [L1] Inconsistent event data for transferred amount

Target	`transfer.move` (Aptos)
Severity	Low
Category	Vulnerability

2.5.1. Description

The `CashmereTransfer` event on Aptos logs the `usdc_amount` parameter in its `amount` field. This value represents the gross amount provided by the user before any protocol fees are subtracted. This is inconsistent with the EVM and Solana implementations, which log the net amount that is actually burned and bridged via CCTP after all deductions.

2.5.2. Impact

This data discrepancy complicates offchain data aggregation, monitoring, and analytics. It can lead to incorrect accounting or reconciliation for services and dApps that rely on this event data to track cross-chain volumes, as the Aptos originated transfers will appear larger than they are.

2.5.3. Recommendations

We recommend modifying the event on Aptos to emit the net `amount` after all fees and any USDC-based gas drops have been deducted.

2.5.4. Remediation

Acknowledged.

2.6. [L2] Undefined error code literal used in assertion

Target	`transfer.move` (Aptos)
Severity	Low
Category	Vulnerability

2.6.1. Description

The `set_fee_bp` function in the Aptos contract contains an assertion to validate the fee parameter against a maximum value. In case of failure, this assertion uses the hardcoded integer literal `2` as the error code. This is against Move best practices, which favor the use of named constants for error codes to improve clarity.

```
public fun set_fee_bp(sender: &signer, auth: Object<AdminCap>, fee_bp: u64) acquires
Config {
        assert!(object::is_owner(auth, signer::address_of(sender)), E_NOT_AN_ADMIN);
        assert!(fee_bp <= MAX_FEE_BP, 2);
        let config: &mut Config = borrow_global_mut(get_object_address());
        config.fee_bp = fee_bp;
    }
```

2.6.2. Impact

Using a “magic number” as an error code makes it difficult for developers, tools, and users to debug failed transactions. The raw code `2` provides no context for the error’s meaning without inspecting the source code.

2.6.3. Recommendations

We recommend defining a named constant for the error code and use this constant in the assertion.

2.6.4. Remediation

Fixed.

2.7. [L3] Unchecked return value in multiple functions

Target	`CashmereCCTP.sol`
Severity	Low
Category	Vulnerability

2.7.1. Description

The `withdrawFee` function initiates a USDC transfer via the standard `IERC20(usdc).transfer()` interface but fails to check the boolean value returned by this call. Although the official USDC contract implementation is known to revert on failure, the ERC20 specification permits compliant tokens to return `false` on failure without reverting. The contract's code does not account for this possibility.

```
function withdrawFee(
       uint256 _usdcAmount,
       uint256 _nativeAmount,
       address _destination
   ) external onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {
       if (block.timestamp - state.lastFeeWithdrawTimestamp < FEE_WITHDRAW_COOLDOWN) {
           revert WithdrawCooldownNotPassed();
       }

        if (_usdcAmount > FEE_WITHDRAW_LIMIT) {
            revert WithdrawLimitExceeded();
        }

        IERC20(usdc).transfer(_destination, _usdcAmount);
        (bool success, ) = payable(_destination).call{value: _nativeAmount}("");
        if (!success) {
            revert NativeTransferFailed();
        }
        state.lastFeeWithdrawTimestamp = block.timestamp;
        emit FeeWithdraw(_destination, _usdcAmount, _nativeAmount);
    }
```

The same issue exists also in `_transferFrom` function, as well as in `resetApprove`.

2.7.2. Impact

If this contract's logic were ever used with a non-reverting ERC20 token, a failed token transfer would go undetected. The function would proceed to transfer native assets and emit a success event, leading the protocol to send out funds while having received no tokens in return.

2.7.3. Recommendations

We recommend wrapping the vulnerable functions calls with a `require` statement to ensure its return value is `true`. For even greater robustness, use a well-vetted library like OpenZeppelin's `SafeERC20`.

2.7.4. Remediation

Fixed by adding appropriate comments informing that this mechanism is safe only for the USDC ERC20 token.

2.8. [L4] Events emit hardcoded placeholder for CCTP nonce

Target	`solana/cashmere_cctp/src/instructions/transfer/transfer_ix.rs`, `solana/cashmere_cctp/src/instructions/transfer/transfer_v2_ix.rs`
Severity	Low
Category	Vulnerability

2.8.1. Description

When a cross-chain transfer is initiated, the Solana program emits a `TransferEvent` which should include the CCTP message nonce, an important identifier for tracking the transfer. The implementation instead emits a hardcoded placeholder value of `-1` or `-2` in the `cctp_nonce` field.

Code from `transfer_ix.rs`:

```
emit!(TransferEvent {
        destination_domain,
        nonce: ctx.accounts.config.nonce,
        recipient,
        solana_owner,
        user: ctx.accounts.owner.key(),
        amount,
        gas_drop_amount,
        cctp_nonce: -1,
         fee_is_native,
         cctp_message: ctx.accounts.message_sent_event_data.key(),
    });
```

2.8.2. Impact

The absence of the real CCTP nonce in event logs makes it impossible for off-chain services, indexers, or end users to track the status of their transfers across chains.

2.8.3. Recommendations

We recommend updating the implementation to extract the actual CCTP nonce generated during the CPI call to the Circle program.

2.8.4. Remediation

Acknowledged.

2.9. [G1] Uncached storage reads and redundant function calls

Target	`CashmereCCTP.sol`
Severity	Informational
Category	Gas Optimization

2.9.1. Description

The internal `_beforeTransfer` function makes a call to the `getFee` view function, which in turn performs a storage read (`SLOAD`) for `state.feeBP` . Additionally, the function accesses `msg.sender` multiple times. These storage reads and external state accesses are not cached within the function's execution frame.

2.9.2. Impact

Each uncached storage read (`SLOAD`) and external state access (`CALLER`) adds unnecessary gas overhead to every transfer transaction, making them more expensive than necessary.

2.9.3. Recommendations

We recommend caching `state.feeBP` and `msg.sender` in local stack variables at the beginning of the function. Perform the fee calculation inline using the cached variable to avoid the external call and the redundant storage reads.

2.9.4. Remediation

Fixed.

2.10. [G2] Redundant overflow checks on nonce increment

Target	`CashmereCCTP.sol`
Severity	Informational
Category	Gas Optimization

2.10.1. Description

The contract increments a `uint32 nonce` on every successful transfer using `state.nonce++`. With default Solidity compiler settings (version >=0.8.0), this simple increment operation includes an overflow check, which consumes a small amount of gas. An overflow of a `uint32` is a theoretical impossibility for this contract, as it would require over 4.3 billion transactions.

2.10.2. Impact

A minor but frequent and unnecessary gas cost is incurred on every successful transfer executed by the contract.

2.10.3. Recommendations

We recommend wrapping the nonce increment operation in an `unchecked` block. This will instruct the compiler to remove the redundant overflow check, saving a small amount of gas on each transaction without introducing any practical risk.

```
- state.nonce++;
+ unchecked {
+   state.nonce++;
+ }
```

2.10.4. Remediation

Fixed.

2.11. [G3] Use of `require` strings instead of custom errors

Target	`CashmereCCTP.sol`
Severity	Informational
Category	Gas Optimization

2.11.1. Description

Several administrative functions, such as `setFeeBP`, use `require` statements with string messages to handle failed validation checks. Since Solidity 0.8.4, custom errors have been available and are significantly more gas-efficient than revert strings.

Example pattern:

```
function setFeeBP(uint16 _feeBP) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(_feeBP <= MAX_FEE_BP, "fee too high");
}
```

2.11.2. Impact

Using `require` with strings increases both the contract's deployed bytecode size and the gas cost of runtime reversions for these specific checks compared to using custom errors.

2.11.3. Recommendations

We recommend replacing all `require` statements that use reason strings with custom error definitions. This will optimize gas usage for reversions and reduce the overall deployment cost of the contract.

```
+ error FeeTooHigh();
  function setFeeBP(uint16 _feeBP) external onlyRole(DEFAULT_ADMIN_ROLE) {
-     require(_feeBP <= MAX_FEE_BP, "fee too high");
+     if (_feeBP > MAX_FEE_BP) revert FeeTooHigh();
      state.feeBP = _feeBP;
      emit FeeBPUpdated(_feeBP);
  }
```

2.11.4. Remediation

Fixed.

2.12. [G4] Inefficient storage struct packing

Target	`CashmereCCTP.sol`
Severity	Informational
Category	Gas Optimization

2.12.1. Description

The `State` struct contains several members that are smaller than 32 bytes, but their ordering prevents optimal packing by the Solidity compiler. A full `uint256` field (`lastFeeWithdrawTimestamp`) is placed between smaller fields, forcing the variables that follow it into a new 32-byte storage slot.

Current layout uses 3 storage slots:

```
struct State {
    address signer;           // Slot 1 (20 bytes)
    uint64 maxUSDCGasDrop;    // Slot 1 (8 bytes)
    uint32 nonce;             // Slot 1 (4 bytes) -> Fills Slot 1
    uint256 lastFeeWithdrawTimestamp; // Slot 2 (32 bytes)
    uint16 feeBP;             // Slot 3 (2 bytes)
    bool reentrancyLock;      // Slot 3 (1 byte)
    bool paused;              // Slot 3 (1 byte)
}
```

2.12.2. Impact

The contract unnecessarily uses an extra storage slot. This significantly increases the gas cost of every transaction that modifies state and increases the cost of reads.

2.12.3. Recommendations

We recommend resizing `lastFeeWithdrawTimestamp` to a smaller `uint40` , which is safe for Unix timestamps until the year 5554. This allows all state variables to be packed into just two storage slots.

```
  struct State {
-     address signer;
-     uint64 maxUSDCGasDrop;
-     uint32 nonce;
-     uint256 lastFeeWithdrawTimestamp;
-     uint16 feeBP;
-     bool reentrancyLock;
-     bool paused;
+     address signer; // 20
+      uint64 maxUSDCGasDrop; // 8
+      uint32 nonce; // 4
+      // Slot 1 is now full (32 bytes)
+      uint40 lastFeeWithdrawTimestamp; // 5 (fits in Slot 2)
+      uint16 feeBP; // 2 (fits in Slot 2)
+      bool reentrancyLock; // 1 (fits in Slot 2)
+      bool paused; // 1 (fits in Slot 2)
  }
```

2.12.4. Remediation

Fixed.

2.13. [I1] Missing `entry` decorator makes Aptos functions externally unavailable

Target	`transfer.move` (Aptos)
Severity	Informational
Category	Vulnerability

2.13.1. Description

In Aptos Move, a function must be marked with the `#[entry]` decorator to be directly callable by an external account via a transaction. In the `cashmere_cctp::transfer` module, the majority of the administrative functions including `set_paused`, `set_fee_bp`, and `set_fee_collector` are declared as `public` but are missing this decorator.

While these functions are visible to other modules, they cannot be invoked as the entry point of a transaction, which is the only way for the protocol's administrator to call them. Only `set_signer_key` and `transfer_outer` are correctly marked as entry functions.

```
1  public fun set_paused(sender: &signer, auth: Object<AdminCap>, paused: bool) acquires
   Config {
2          assert!(object::is_owner(auth, signer::address_of(sender)), E_NOT_AN_ADMIN);
3          let config: &mut Config = borrow_global_mut(get_object_address());
4          config.paused = paused;
5      }
```

2.13.2. Impact

The administrator of the protocol is unable to execute any of the essential management functions after deployment. The protocol's parameters are effectively frozen and unchangeable.

2.13.3. Recommendations

We recommend adding the `#[entry]` decorator to all public facing administrative functions that are intended to be called directly by the protocol's administrator.

2.13.4. Remediation

Acknowledged. The team clarified this is intentional, as they plan to use external scripts for administrative functions rather than calling the module directly.