PUBLIC

Code Assessment

of the Plusplus Custody Smart Contracts

September 19, 2025

Produced for

Plusplus

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Plusplus Team,

Thank you for trusting us to help Plusplus AG with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Plusplus Custody according to Scope to support you in forming an opinion on their security risks.

Plusplus AG implements an on-chain custody system intended to centrally hold and administer assets, with clearly split accounting per id.

The most critical subjects covered in our audit are asset solvency, functional correctness, accounting, and access control. Security regarding asset solvency and accounting is high. There can be small accounting mismatches. However, these can be fully mitigated through off-chain processes (see Tick-accrual Mismatch on New Deposit). Functional correctness is high after Duplicate Inputs Can Overwrite Storage has been fixed. Security regarding access control is high.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1
• Code Corrected	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Plusplus Custody repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	08 Sept 2025	32ad18086023f87a241f3b4473e8cc5fc35f23a9	Initial Version
2	12 Sept 2025	ceea30074fcf60a6cde40673e77f98ff575c2c3f	Fixes Version

For the solidity smart contracts, the compiler version `0.8.30` was chosen.

As part of version 1, the following contracts are included in scope:

```
src/
    WBTCDepositManager.sol
    ZCHFSavingsManager.sol
```

In Version 2, the following contract was added to the scope of the assessment:

```
src/
    RedemptionLimiter.sol
```

2.1.1 Excluded from scope

Any file not explicitly listed in the Scope section is out of scope. In particular, external libraries (e.g. OpenZeppelin), third-party contracts (e.g. WBTC, ZCHF, and its Savings Module), and tests are excluded.

Additionally, all configurations and operational usage are out of scope, including role assignments, deployment, and migrations.

2.2 System Overview

This system overview describes Version 2 of the contracts, as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes according to the versions.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Plusplus AG implements an on-chain custody system intended to centrally hold and administer assets. The custody system consists of two independent managers with similar operational patterns. The contracts have full custody of the assets and are managed by a set of privileged roles. Users never interact directly with the contracts.

2.2.1 WBTCDepositManager

The contract holds WBTC deposits subject to a linear, non-compounding custody fee applied to the principal.

2.2.1.1 Deposit and Redemption flows

Deposits are initiated by an operator with the `createDeposits` function. It stores per-deposit records. Each deposit is represented by a `bytes32` identifier and contains two fields: the `principal` and `startTime`.

Redemptions are initiated by an operator with the `redeemDeposits` function. It computes the value of each deposit at the time of redemption using `depositValue()`, transfers the total value to a receiver address, and deletes the per-deposit record. The receiver address must have the `RECEIVER_ROLE`.

2.2.1.2 Accounting

From creation onward, a custody fee accrues at a fixed annual rate, currently set to `FEE_ANNUAL_PPM = 9_500` (0.95% p.a.), calculated using a 365-day year. The fee could be configured differently by redeploying the contract with a different fee constant. Any value is theoretically supported. The fee is applied linearly in time and without compounding. The helper `depositValue(id)` returns the amount owed to the depositor by subtracting the elapsed-time fee from the original principal and capping the result to at least zero.

Formally, for principal P, creation time t_0, and time t, the value is:

$$\text{value}(t) \;=\; \max(0,\; P - \text{fee}(t))$$

Where the fee is computed as:

$$\text{fee}(t) \;=\; P \cdot \frac{\text{FEE\_ANNUAL\_PPM}}{1\,000\,000} \cdot \frac{t - t_0}{365}$$

2.2.1.3 Global Accounting

To value the entire system in `O(1)` time, the contract maintains two aggregates that are updated on every create/redeem:

- `totalPrincipal` $= P_{\text{tot}} = \sum_i P_i$. The sum of active principals.
- `principalTimeProductSum` $= S = \sum_i P_i \cdot t_{0,i}$. The sum of each principal multiplied by its start time.

At time t, these let the contract derive the total elapsed principal-time $t \cdot P_{\text{tot}} - S$, which converts directly to total fees at the annual ppm rate, and then to total value owed to depositors:

$$\text{totalFees}(t) \;=\; (t \cdot P_{\text{tot}} - S) \cdot \frac{\text{FEE\_ANNUAL\_PPM}}{10^6} \cdot \frac{1}{365 \text{ days}},$$

$$\text{totalValue}(t) \;=\; \max(0,\; P_{\text{tot}} - \text{totalFees}(t)).$$

The accumulated fees currently held by the contract (i.e., not owed to depositors) are computed as the on-chain WBTC balance minus the total value:

$$\text{accumulatedFees}(t) \;=\; \text{WBTCbalance}(\text{this}) \;-\; \text{totalValue}(t).$$

As the fees are calculated on-chain, the contract can ensure that it always holds enough WBTC to cover the total value owed to depositors. The only exception is if the admin has used `moveWBTC()` to withdraw funds.

2.2.1.4 Additional Management functions

The contract exposes the following additional management functions:

- `collectFees(receiver)`: Transfers `accumulatedFees()` to `receiver`. Callable by `OPERATOR_ROLE`. `receiver` must have `RECEIVER_ROLE`.
- `rescueTokens(token, receiver, amount)`: Recovers ERC-20 tokens (except WBTC) or ETH accidentally sent to the contract. Callable by `OPERATOR_ROLE`. `receiver` must have `RECEIVER_ROLE`.
- `moveWBTC(receiver, amount)`: Raw WBTC transfer that bypasses any accounting (intended for migrations). Callable by `DEFAULT_ADMIN_ROLE` only. `receiver` must have `RECEIVER_ROLE`.

2.2.2 ZCHFSavingsManager

The contract batches ZCHF into the Frankencoin Savings module and tracks each user deposit independently. Interest accrues via a tick-based mechanism after a fixed delay. A yearly fee is applied and is capped by the actual earned interest. Each ID always receives at least their principal back.

2.2.2.1 Deposit and Redemption flows

The `createDeposits(identifiers, amounts, source)` function batch-creates deposits, pulls ZCHF from `source` (single transfer), calls the Savings module once (`save(total)`), and records the deposit for each identifier. All items in the batch share the same `createdAt` and a pre-shifted tick baseline (see below).

The `redeemDeposits(identifiers, receiver)` function computes each deposit's net interest at the current block time, deletes records, and performs a single Savings module withdrawal (`withdraw(receiver, total)`). The `receiver` must have the `RECEIVER_ROLE`.

2.2.2.2 Accounting

The savings module exposes a tick counter measured in `ppm * seconds`. When a deposit is created, the manager records a baseline tick that records the start of interest accrual based on the Savings module's fixed delay. Concretely, it takes the current tick value and adds the product of the module's current rate and the `INTEREST_DELAY`. From that point on, only tick growth after this baseline counts toward interest earned by this id.

After the delay has elapsed, gross interest increases whenever the Savings module's tick counter advances. The amount is proportional to both the deposit's initial principal and the number of ticks accumulated since the baseline.

The management fee is applied to the principal. It accrues linearly with time from the deposit's creation at `FEE_ANNUAL_PPM` , which is currently set to `12_500` (1.25% p.a.). The fee is then capped, so that it can never exceed the gross interest that actually accrued. This cap guarantees that net interest is never negative and that principal is never reduced. The fee could be configured differently by redeploying the contract with a different fee constant. Any value is theoretically supported.

On redemption, the contract returns the sum of the principal plus net interest (i.e., gross interest minus the capped fee). Before the interest delay expires, there is no interest and, due to the cap, no fee.

As the fee to be claimed with `moveZCHF()` is calculated off-chain by the operator, the contract cannot guarantee that it always holds enough ZCHF to cover all deposits. However, it is assumed that the operator will deposit additional ZCHF using `addZCHF()` in case the balance ever becomes insufficient.

2.2.2.3 Administrative Functions

The contract exposes the following additional management functions:

- `addZCHF(source, amount)`: Saves ZCHF in the savings module without creating deposits (e.g., to correct underfunding).

- `moveZCHF(receiver, amount)`: Withdraws exactly `amount` from the module to `receiver`. `receiver` must hold `RECEIVER_ROLE`. This function bypasses any accounting and is intended to collect fees or migrate funds.
- `rescueTokens(token, receiver, amount)`: Recovers ERC-20 tokens or ETH held by the manager contract. `receiver` must hold `RECEIVER_ROLE`.

2.2.2.4 Changes in Version 2

The following changes were made in Version 2 of the codebase:

- To mitigate Operator Could Break Accounting, Version 2 introduces an on-chain, per-operator redemption rate limit. The shared logic is defined in the abstract contract RedemptionLimiter. The contract provides functions to enforce a rolling 24-hour quota for redemptions. The quota is set in asset units and refills linearly with time. It never reaches a value higher than the daily cap.

Both managers extend `RedemptionLimiter` and enforce it. Calls to `redeemDeposits()` use the operator's current allowance. If no limit is set, the call reverts. If the amount exceeds the available allowance, the call reverts. A view helper, `availableRedemptionQuota()`, exposes how much can be redeemed at the current time. Administrators can configure the limit per operator using `setDailyLimit()`. Setting a limit also refills the quota to the new daily limit.

- The `moveZCHF` function previously allowed passing an `amount` greater than the ZCHFSavingsManager's balance in the savings module. In this case, it would just transfer the entire balance. In Version 2, the behavior was changed to revert in case the full `amount` is not available to be transferred.

2.3 Trust Model

2.3.1 Roles and Privileges

2.3.1.1 Admin

Trust Level: Fully trusted.

Capabilities: Grants and revokes roles. In the worst case, it could steal all the funds and transfer them to an arbitrary address.

2.3.1.2 Receiver

Trust Level: Fully trusted.

Capabilities: Receives redemptions, fees, and rescues. In the worst case, it could steal all the funds that the operator decides to send it.

2.3.1.3 Operator

Trust Level: Partially trusted.

Capabilities: Executes all operational flows. It is not able to extract, lock, or lose deposited value. In the worst case, it could break the accounting by withdrawing funds to a Receiver and cause delays in interest accrual. See Operator Could Break Accounting. The operator is assumed to correctly calculate the fees taken via `moveZCHF()`.

2.3.1.4 Frankencoin Governance

Trust Level: Fully trusted (only by ZCHFSavingsManager).

Capabilities: Frankencoin governance is responsible for the parameters of the Frankencoin system, such as available minters and collaterals. In the worst case, it could misconfigure the system in such a way that the ZCHF token loses its peg and becomes worthless (all configuration changes take time, so they can be monitored). Additionally, it could disable the current savings module and replace it with a new one. In this case, a migration of funds to a new ZCHFSavingsManager would be necessary.

2.3.2 Upgradeability

Both contracts are deployed as non-upgradeable implementations. Fee rates and external addresses are fixed at construction. Changes require a new deployment and potentially a migration, which could be done using the provided admin functions.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	1

- Duplicate Inputs Can Overwrite Storage Code Corrected

Informational Findings	3

- Caching Could Save Gas Code Corrected
- Deposit Details Can Return Inconsistent Values Specification Changed
- Unlocked Compiler Version Code Corrected

6.1 Duplicate Inputs Can Overwrite Storage

Correctness Low Version 1 Code Corrected

CS-PPCUS-001

In ZCHFSavingsManager, the `createDeposits` function checks that none of the identifiers used as input are already in use. However, it does not check for duplicate identifiers within the input array itself. This means that if the operator provides the same identifier multiple times in the input array, the last occurrence will overwrite the previous ones, leading to untracked deposits.

The untracked deposits can be withdrawn by an operator using `moveZCHF()`.

As there will be an event emitted for each duplicate identifier used, the events and storage state will be inconsistent. The multiple DepositCreated events will never be matched by multiple DepositRedeemed events, as the deposit can only be redeemed once.

Code correct:

In Version 2, `createDeposits()` now checks each identifier before writing, not just during the pre-validation pass. After the first occurrence of an ID is stored, its `createdAt` is non-zero. A second occurrence in the same batch would therefore pass `if (deposits[id].createdAt != 0)` and the transaction would revert.

6.2 Caching Could Save Gas

Informational Version 1 Code Corrected

CS-PPCUS-002

Some functions read the same storage variable multiple times. Caching these values in memory variables could save gas.

For example:

```
uint64 deltaTicks = currentTicks > deposit.ticksAtDeposit ? currentTicks - deposit.ticksAtDeposit : 0;
```

This could be cached as follows:

```
uint64 ticksAtDeposit = deposit.ticksAtDeposit;
uint64 deltaTicks = currentTicks > ticksAtDeposit ? currentTicks - ticksAtDeposit : 0;
```

By assigning the read storage value to memory, it can be ensured that the storage is only read once.

This optimization could be applied in the following locations:

1. In ZCHFSavingsManager `getDepositDetailsAt()`, `deposit.initialAmount` and `deposit.ticksAtDeposit` are read multiple times.
2. In ZCHFSavingsManager `redeemDeposits()`, `deposit.initialAmount` is read multiple times.
3. In WBTCDepositManager `redeemDeposits()`, `deposit.principal` is read multiple times.

For each optimization made, it is recommended to measure the gas savings with the specific compiler and optimizer settings used in production. This will ensure that the optimization is not redundant due to compiler optimization.

Code corrected:

In Version 2, repeated storage reads are cached in local variables.

6.3 Deposit Details Can Return Inconsistent Values

Informational Version 1 Specification Changed

CS-PPCUS-003

In ZCHFSavingsManager, `getDepositDetailsAt` function can return inconsistent values when used with past or future timestamps. This includes the following behaviors:

1. If the function is queried for a timestamp that was before the last Frankencoin savings rate change, it will revert.
2. If the function is queried for a timestamp that was before the deposit was created, it will return `(initialAmount, 0)`, even though there was no amount deposited at this time.
3. If the function is queried for a timestamp at which an old deposit existed, but the identifier has since been reused for a new deposit, it will return `(initialAmount,0)`, where `initialAmount` is the amount of the new deposit, not the old one that existed at the time.
4. If the function is queried for a timestamp at which the deposit existed, but the deposit has since been redeemed, it will return `(0,0)`.
5. If the function is queried for a timestamp that is in the future, it will return an amount that assumes the Frankencoin savings rate will remain constant until that time, which is not guaranteed.

Specification changed:

In Version 2, `getDepositDetailsAt` was removed. All callers now use `getDepositDetails`, which computes accrual strictly at the current block (`block.timestamp` with `savingsModule.currentTicks()`). As past and future timestamps are no longer supported, the above inconsistent behaviors are no longer present.

6.4 Unlocked Compiler Version

Informational Version 1 Code Corrected

CS-PPCUS-005

The contracts use the `pragma solidity ^0.8.30` directive to specify a minimum compiler version, but do not specify a maximum version. This means that the contracts can be compiled with future compiler versions, which may introduce changes that could affect the contract's behavior. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using a different version than they were tested with.

Code corrected:

The pragma directives in the Solidity files have been fixed to `0.8.30`.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Tick-accrual Mismatch on New Deposit

Informational **Version 1** **Acknowledged**

CS-PPCUS-004

The Savings module is account-based, not position-based. The manager contract (`ZCHFSavingsManager`) has one single savings account in the module: `savings[address(this)]`.

Each time additional funds are deposited, the module advances the account's `ticks` by a weighted delay (`currentRatePPM * INTEREST_DELAY`) so that the new money effectively waits `INTEREST_DELAY` days before earning interest:

```
uint64 weightedAverage = uint64(
    (saved * (balance.ticks - ticks) + uint256(amount) * currentRatePPM * INTEREST_DELAY)
    / (saved + amount)
);
balance.ticks = ticks + weightedAverage; // ticks == currentTicks()
```

Because `balance.ticks` becomes greater than `currentTicks()` right after a deposit, the entire pooled account earns zero new interest until `currentTicks()` catches up again.

The manager's per-deposit view function (`getDepositDetails{,At}`) does not account for this pause and continues to report increasing interest based on `ticks(timestamp)`, even though the module accrues none during the delay:

```
uint64 deltaTicks = currentTicks > deposit.ticksAtDeposit
    ? currentTicks - deposit.ticksAtDeposit
    : 0;
```

If redemptions are executed within this window, the manager can withdraw the overstated amount. If all ids were withdrawn at the same time while this is the case, `withdraw` would not be able to service all the amounts.

Consider the following example:

1. A deposit of 100 ZCHF is made at time 0.
2. Another deposit of 100 ZCHF is made at time 1.5d. The delay set in the savings module will be set to the average of 1.5d and 3d = 2.25d.
3. At time 3.5d, the `getDepositDetails()` function will report that the first deposit has been earning interest for 0.5d. However, there has been no interest as the manager will only start earning interest at 1.5 + 2.25d = 3.75d.
4. If there is no redemption until time 4.5d, (3 days after the last deposit) the interest has equalized and the accounting is now correct.

The small accounting inaccuracy will happen every time there is a new deposit. If there are withdrawals while the accounting is inaccurate, it may stay inaccurate as the principal that earns interest is reduced. However, this effect can be fully mitigated by leaving a buffer of unwithdrawn fees in the manager contract, which generates enough interest to cover any potential over-withdrawals.

Acknowledged:

Plusplus AG states:

We are okay with there being slight mismatches that sync up later. We can handle this by either waiting and/or depositing buffer funds to the savings module (Process Change).

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Operator Could Break Accounting

Note Version 1

The operator role is partially trusted. In the worst case, a malicious operator could intentionally delete all deposits by calling `redeemDeposits()` on all funds. While it would be possible to recreate the state off-chain, this would break the accounting model of the on-chain contract. As the fees charged depend on the principal and deposit time of each deposit, a deleted deposit cannot be simply recreated using `createDeposits()`. Recreating the deposit with the redeemed balance and the current timestamp would lead to different fee calculations than the original deposit.

Note that while this situation would be challenging accounting-wise, it would not lead to a loss of user funds, as the redeemed balance would be forwarded to a fully trusted receiver address. The worst-case financial impact would be the loss of interest that would have accrued on the deposits if they had remained active, as well as the interest of the 3-day delay upon re-depositing to the Frankencoin Savings contract.

Specification changed:

In Version 2, a daily redemption limit was introduced for operators. This reduces the damages a malicious operator can do. The above note still applies, but the maximum withdrawable amount is now the limit configured by the admin instead of all funds.