softstack

Smart Contract Code Review And Security Analysis Report

CUSTOMER	Minted
SCOPE	Core Contracts
DATE	28.02.2026

Made in Germany by softstack GmbH

Table of Contents

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of the client. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Version / Date	Description
0.1 (15.02.2026)	Layout
0.5 (20.02.2026)	Manual + Automated Security Testing
1.0 (28.02.2026)	Final document
1.1 (28.02.2026)	Re-check

2. About the Project and Company

Company: MintedAssociates Corp

Address: 511 South DuPont Highway Ste 10, Dover, Delaware 19901, USA

Website: https://www.minted.app/

Twitter (X): https://x.com/tryMinted

LinkedIn: https://www.linkedin.com/company/mintedmarketplace

Instagram: https://www.instagram.com/tryminted/

Minted is a stablecoin protocol spanning the **Canton Network** (DAML) and ( ). mUSD is minted 1:1 against stablecoins, with Canton serving as the institutional-grade accounting, compliance, and settlement layer, while handles yield generation and DeFi execution.

The Canton-side DAML templates under audit represent the protocol's privacy-enabled distributed ledger layer, providing:

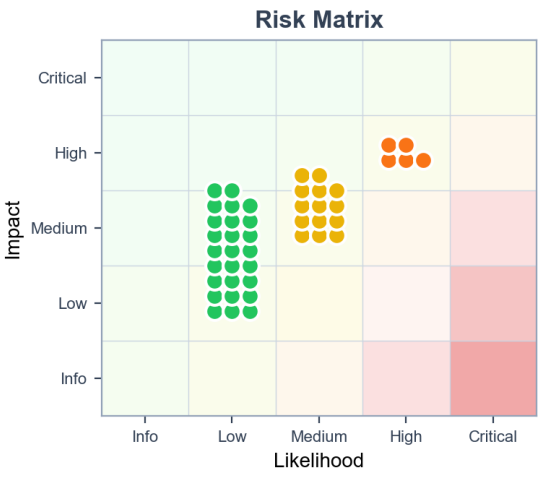
- **CantonDirectMint** — Canton-side thin accounting for mUSD minting with 1:1 USDC backing, rate limiting, and compliance hooks.
- **CantonSMUSD** — Yield vault with share-price model for staked mUSD, including cooldown enforcement and yield synchronization from .
- **CantonLending** — Collateralized borrowing with escrow-based collateral management, interest accrual, and liquidation mechanics.
- **CantonBoostPool** — Boost pool with deposit caps, LP issuance, validator reward distribution, and price synchronization.
- **Compliance** — Blacklist/freeze registry with Set-based O(log n) lookups, pre-transaction validation hooks, and bulk import capabilities.
- **Governance** — Multi-sig governance framework with proposal lifecycle, approval thresholds, emergency rollback, and upgrade coordination.
- **Protocol** — Canton-to- flow with validator signatures, nonce tracking, and supply cap conservation.
- **MintedMUSD / InstitutionalAssetV4** — Core mUSD token with Master Participation Agreement (MPA) embedding, split/merge/transfer, and observer privacy controls.

All backing reserves flow to 's Treasury for yield generation. The Canton side is a thin accounting layer tracking ownership, staking positions, vault collateral, and compliance state.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.1.

Level	CVSS Score	Vulnerability	Required Action
CRITICAL	9.0 – 10.0	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
HIGH	7.0 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
MEDIUM	4.0 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
LOW	0.1 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
INFO	0.0	A vulnerability that has informational character but is not affecting any of the code.	An observation that does not determine a level of risk.



4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

The auditing process follows a routine series of steps:

Code review that includes the following:

- Review of the specifications, sources, and instructions provided to softstack to make sure we understand the size, scope, and functionality of the smart contract.
- Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
- Comparison to specification, which is the process of checking wh er the code does what the specifications, sources, and instructions provided to softstack describe.

Testing and automated analysis that includes the following:

- Test coverage analysis, which is the process of determining wh er the test cases are actually covering the code and how much code is ex ised when we run those test cases.
- Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.

Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.

Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

5. Metrics

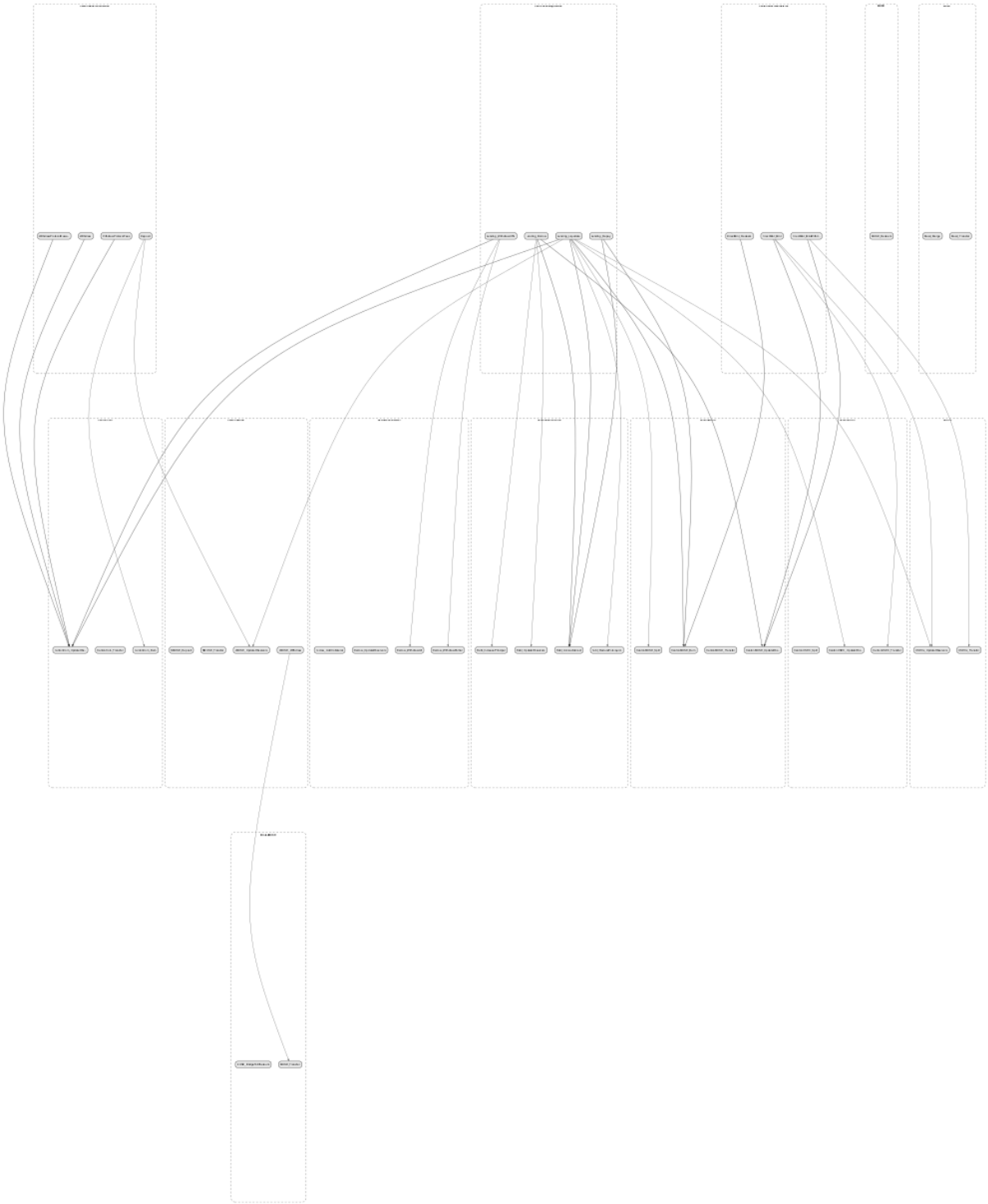
The metrics section gives the reader an overview of the size, quality, flows and capabilities of the codebase.

5.1 Tested Contract Files

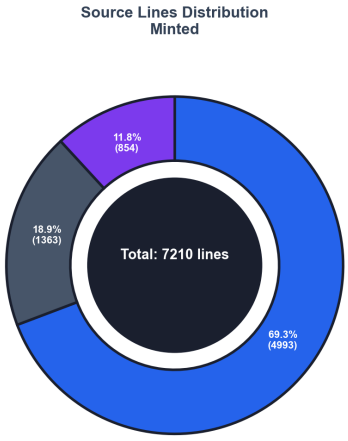
Source: https://github.com/luthatdude/Minted-mUSD-Canton
Commit: `393fd96f322674b25dd4b0c21f593f0874b86211`

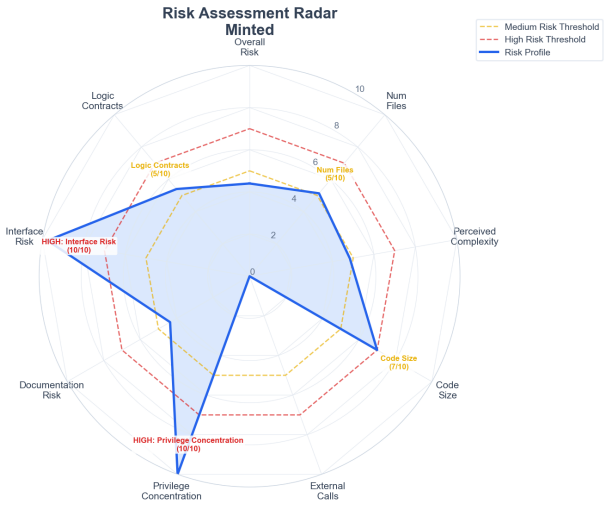
File	Fingerprint (MD5)
./contracts/BLE Protocol.daml	`d9f27e9aabfd95b8090441e7bdaeecaf`
./contracts/BLEProtocol.daml	`990d1d25c2285bf6960e7c52fa50b530`
./contracts/CantonBoostPool.daml	`5f397c26f24f5162f2f6e487de414a05`
./contracts/CantonDirectMint.daml	`f5629539057f6214e49a1eb767615e62`
./contracts/CantonLending.daml	`4451589bf58474bdaaf6fadeadf25fa5`
./contracts/CantonSMUSD.daml	`e08a89f6b052e0b84608dec50238324f`
./contracts/Compliance.daml	`7a73d0b65f872f1bf35b943eab014f45`
./contracts/Governance.daml	`ccec20db5beff27d1bf4c2104f61e0f`
./contracts/InstitutionalAssetV4.daml	`58a0dddb50bd45023808aa1e71340959`
./contracts/InterestRateService.daml	`2eaa6c9bdcd9da5260164c4a2c7cfc35`
./contracts/MUSD_Protocol.daml	`91a9caf41ab1c74cbe3d676eae647be3`
./contracts/Minted/Protocol/V3.daml	`855b499e7fa90d143e14640e0208a451`
./contracts/MintedMUSD.daml	`5bc89d278942df0b7b5c47552a4a767d`
./contracts/TokenInterface.daml	`d8d747a52fedd74de9bb6187b28dc1fc`
./contracts/Upgrade.daml	`679a08299f5bb31babb4d4d261bafc6d`
./contracts/UserPrivacySettings.daml	`3a480d2c4231d6908969a5f21dbbf50a`

5.2 Call Graph



5.3 Source Lines & Risk





5.4 Capabilities

Module	Functions	Public	Restricted	Key Roles
BLE Protocol. In	2	0	2	validator, aggregator
BLE Protocol. Out	2	0	2	validator, aggregator
BLE Protocol.InstitutionalEquityPosition	1	0	1	bank
BLE Protocol.SupplyCap	2	0	2	validator, aggregator
BLE Protocol.Yield	2	0	2	validator, aggregator
BLEProtocol. Request	2	0	2	validator, aggregator
BLEProtocol.InstitutionalEquityPosition	1	0	1	bank
CantonBoostPool.BoostPoolDepositRecord	2	0	2	operator
CantonBoostPool.BoostPoolLP	3	0	3	owner
CantonBoostPool.BoostPoolLPTransferProposal	3	0	3	newOwner, lp
CantonBoostPool.CantonBoostPoolService	11	0	11	operator, user
CantonBoostPool.CantonCoin	4	0	4	owner, issuer
CantonBoostPool.CantonCoinTransferProposal	3	0	3	newOwner, coin
CantonDirectMint. OutRequest	2	0	2	operator

Module	Functions	Public	Restricted	Key Roles
CantonDirectMint.CantonDirectMintService	8	0	8	operator, user
CantonDirectMint.CantonMUSD	5	0	5	owner, issuer
CantonDirectMint.CantonMUSDTransferProposal	3	0	3	newOwner, musd
CantonDirectMint.CantonUSDC	3	0	3	owner
CantonDirectMint.CantonUSDCTransferProposal	2	0	2	newOwner
CantonDirectMint.RedemptionRequest	1	0	1	operator
CantonDirectMint.ReserveTracker	4	0	4	operator
CantonDirectMint.USDCx	4	0	4	owner, issuer
CantonDirectMint.USDCxTransferProposal	2	0	2	newOwner
CantonLending.CantonDebtPosition	6	0	6	operator, borrower
CantonLending.CantonLendingService	18	0	18	user, operator, liquidator
CantonLending.CantonPriceFeed	2	0	2	operator
CantonLending.EscrowedCollateral	5	0	5	operator, owner
CantonSMUSD.CantonSMUSD	4	0	4	owner
CantonSMUSD.CantonSMUSDTransferProposal	3	0	3	newOwner, smusd
CantonSMUSD.CantonStakingService	5	0	5	operator, user
Compliance.ComplianceRegistry	5	0	5	regulator
Governance.EmergencyPauseState	2	0	2	guardian, operator
Governance.MinterRegistry	4	0	4	operator, minter
Governance.MultiSigProposal	4	0	4	approver, rejector, executor
InstitutionalAssetV4.Asset	6	0	6	owner, issuer
InstitutionalAssetV4.TransferProposal	3	0	3	newOwner, asset
InterestRateService.InterestRateService	2	0	2	operator, governance
MUSD_Protocol. Claim	2	0	2	user
MUSD_Protocol. Lock	1	0	1	user

Module	Functions	Public	Restricted	Key Roles
MUSD_Protocol.MintingService	2	0	2	user
MUSD_Protocol.Musd	2	0	2	owner
MUSD_Protocol.MusdTransferProposal	3	0	3	newOwner, musd
MUSD_Protocol.Usdc	2	0	2	owner
MUSD_Protocol.UsdcTransferProposal	3	0	3	newOwner, usdc
MintedMUSD.IssuerRole	4	0	4	issuer
MintedMUSD.MUSD	5	0	5	owner, provider
MintedMUSD.MUSD_Locked	1	0	1	usd
MintedMUSD.MUSD_MintProposal	2	0	2	owner
MintedMUSD.MUSD_RedemptionRequest	2	0	2	usd
MintedMUSD.MUSD_TransferProposal	3	0	3	newOwner, usd
MintedMUSD.MintProposal	3	0	3	owner, issuer
MintedMUSD.MintRequest	3	0	3	issuer, owner
Upgrade.MigrationTicket	2	0	2	operator, holder
Upgrade.UpgradeProposal	3	0	3	operator, approver
Upgrade.UpgradeRegistry	3	0	3	operator, authorizer
UserPrivacySettings.UserPrivacySettings	4	0	4	user
V3. Request	3	0	3	aggregator, validator
V3. InRequest	2	0	2	operator
V3. OutRequest	4	0	4	operator, user
V3. Service	6	0	6	operator
V3.CantonDirectMint	3	0	3	user, operator
V3.CantonSMUSD	4	0	4	user, operator
V3.CollateralDepositProof	1	0	1	verifier
V3.CooldownTicket	2	0	2	owner
V3.LiquidationOrder	3	0	3	keeper, operator
V3.LiquidityPool	2	0	2	receiver, operator

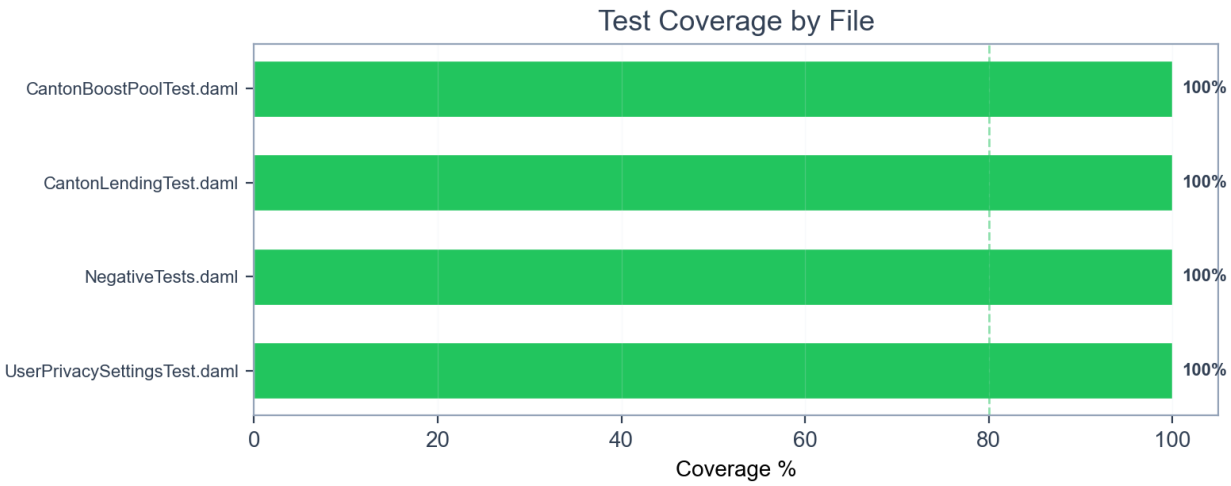
Module	Functions	Public	Restricted	Key Roles
V3.MUSDSupplyService	5	0	5	governance, operator
V3.MUSDTransferProposal	3	0	3	receiver, sender, issuer
V3.MintedMUSD	5	0	5	owner, issuer
V3.PriceOracle	1	0	1	provider
V3.TicketTransferProposal	2	0	2	receiver, sender
V3.Vault	6	0	6	owner, operator, liquidator
V3.VaultManager	3	0	3	operator, owner

5.5 Source Units in Scope

File	Instructions	Functions	Lines	nSLOC	Comments
CantonDirectMint.daml	34	34	722	506	124
UserPrivacySettings.daml	4	5	153	93	45
CantonSMUSD.daml	12	12	280	181	60
InstitutionalAssetV4.daml	9	9	189	142	32
MintedMUSD.daml	23	23	331	249	66
BLE Protocol.daml	9	9	433	279	91
Upgrade.daml	8	8	281	184	62
TokenInterface.daml	0	0	11	2	6
BLEProtocol.daml	3	4	189	115	41
Governance.daml	10	10	399	293	56
Compliance.daml	5	5	156	104	39
MUSD_Protocol.daml	15	17	535	375	107
CantonLending.daml	31	39	1235	896	207
InterestRateService.daml	2	3	210	141	44
CantonBoostPool.daml	26	26	541	343	119

File	Instructions	Functions	Lines	nSLOC	Comments
V3.daml	55	55	1545	1090	264
Total	**246**	**259**	**7210**	**4993**	**1363**

5.6 Test Coverage

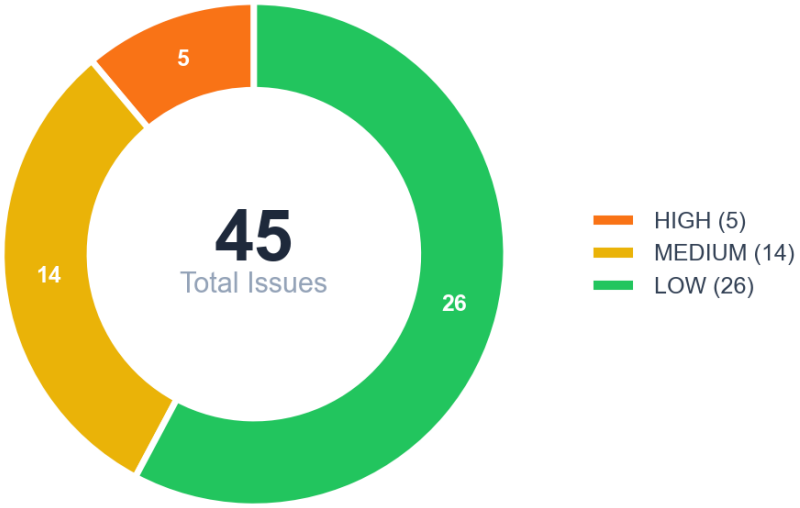


File	Lines Total	Lines Covered	Coverage %
CantonBoostPoolTest.daml	25	25	100.0%
CantonLendingTest.daml	30	30	100.0%
NegativeTests.daml	14	14	100.0%
UserPrivacySettingsTest.daml	24	24	100.0%

6. Scope of Work

The Minted team has provided the DAML Canton smart contract templates for their stablecoin protocol. The audit focuses on validating security, compliance, and correctness of the Canton-side accounting, minting, staking, lending, governance, and logic.

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	Duplicate EscrowCids Allow Inflated Collateral Valuation	HIGH	FIXED
6.2.2	Omitting Borrower Collateral from EscrowCids Forces Liquidation of Healthy Positions	HIGH	FIXED
6.2.3	targetEscrowCid Not Validated Against Borrower Allows Seizing Third-Party Collateral	HIGH	FIXED
6.2.4	GovernanceActionLog Replay Allows Single Approval to Authorize Unlimited Operations	HIGH	FIXED
6.2.5	Share Price Inconsistency Between Deposit and Withdraw in CantonSMUSD	HIGH	FIXED

No	Title	Severity	Status
6.2.6	Legacy SyncYield Missing All Security Guards	MEDIUM	ACKNOWLEDGED
6.2.7	Phantom Migration Attack via Arbitrary Text Strings	MEDIUM	FIXED
6.2.8	Incomplete State Machine in UpgradeRegistry	MEDIUM	FIXED
6.2.9	Fixed Exchange Rate Pool Drain	MEDIUM	FIXED
6.2.10	Vault_Repay Does Not Burn mUSD	MEDIUM	FIXED
6.2.11	Liquidate Does Not Require mUSD Repayment Token	MEDIUM	FIXED
6.2.12	Precision Validation Offset Silently Passes Invalid Amounts	MEDIUM	ACKNOWLEDGED
6.2.13	No Validator Whitelist in Claim	MEDIUM	ACKNOWLEDGED
6.2.14	Supply Tracking Inconsistent Across Lifecycle	MEDIUM	ACKNOWLEDGED
6.2.15	Claim and Lock Not Linked Enables Phantom Claims	MEDIUM	ACKNOWLEDGED
6.2.16	MUSD_Locked Signatory Downgrade Enables Permanent Lock	MEDIUM	ACKNOWLEDGED
6.2.17	IssuerRole currentSupply Never Decremented on Redemption	MEDIUM	ACKNOWLEDGED
6.2.18	No Compliance Integration in MintedMUSD	MEDIUM	FIXED
6.2.19	Single Member Emergency Rollback Without Multi-Sig	MEDIUM	FIXED
6.2.20	No Compliance Checks in CantonLending	LOW	FIXED
6.2.21	Missing Validation on Market State Values	LOW	FIXED
6.2.22	Integer Truncation in Supply Rate Calculation	LOW	FIXED
6.2.23	No Time-Based Staleness Check on Market Data	LOW	FIXED
6.2.24	Unbounded Observer List Growth in IssuerRole	LOW	FIXED
6.2.25	MintingService Missing Ensure Clause	LOW	FIXED
6.2.26	Service Nonce Not Validated for Uniqueness	LOW	FIXED
6.2.27	FulfillRedemption Destroys Tokens Without Settlement Proof	LOW	FIXED
6.2.28	Duplicate Party in Governance List	LOW	FIXED
6.2.29	BatchSize Not Enforced in MigrationTicket	LOW	FIXED

No	Title	Severity	Status
6.2.30	Zero or Negative Count in RecordMigration	LOW	FIXED
6.2.31	No Global Supply Cap on mUSD Minting Through Borrows	LOW	ACKNOWLEDGED
6.2.32	Cancel Archives Without Recording State	LOW	FIXED
6.2.33	UpdateObservers Bypasses UserPrivacySettings	LOW	FIXED
6.2.34	AdjustLeverage borrowAmount Creates Phantom Debt	LOW	ACKNOWLEDGED
6.2.35	Stale Interest in Vault_WithdrawCollateral	LOW	FIXED
6.2.36	AdjustLeverage Phantom Collateral	LOW	FIXED
6.2.37	Burn_Musd Supply Underflow Silently Floors to Zero	LOW	FIXED
6.2.38	Non-Sequential Nonce in	LOW	FIXED
6.2.39	Compliance Test Tests Wrong Failure Path	LOW	FIXED
6.2.40	No Contract Key on ComplianceRegistry	LOW	ACKNOWLEDGED
6.2.41	No Expiry Check on Proposal Execution	LOW	FIXED
6.2.42	Guardian Pause Griefing Loop	LOW	ACKNOWLEDGED
6.2.43	activeProposalCount Is Dead Code	LOW	FIXED
6.2.44	MinterRegistry_UseMintQuota Missing Amount Greater Than Zero Check	LOW	FIXED
6.2.45	Unbounded Observers in Asset_Merge	LOW	FIXED

6.2 Manual and Automated Vulnerability Test

6.2.1 Duplicate EscrowCids Allow Inflated Collateral Valuation

HIGH FIXED

DESCRIPTION

The CantonLendingService template accepts a user-supplied list of escrow contract IDs in all borrowing, withdrawal, and liquidation choices. The helper function computeWeightedCollateralValue iterates this list using DAML fetch, which is a non-consuming read. The same contract ID can be fetched multiple times in the same transaction without error.

No deduplication is performed on the input list. There is no use of DA.List.unique, DA.Set.fromList, or any other uniqueness check. If an attacker passes the same escrow contract ID N times, the computed collateral value is inflated by Nx.

An attacker can deposit a small amount of collateral, then call Lending_Borrow with the same escrow ID repeated many times. Each repetition adds the full collateral value to the running total. This passes the health check and lets the attacker borrow far more mUSD than their real collateral supports. The borrowed mUSD can be d to and sold, leaving the protocol permanently und ollateralized.

CODE LOCATION

File: CantonLending.daml

CantonLending.daml

```
computeWeightedCollateralValue : Party -> Party -> [CollateralConfig] ->
[ContractId EscrowedCollateral] -> [ContractId CantonPriceFeed] -> Bool -> Update
Money
computeWeightedCollateralValue _ _ _ [] _ _ = return 0.0
computeWeightedCollateralValue operator expectedOwner cfgs (eCid :: eRest) feeds
safe = do
escrow <- fetch eCid
assertMsg "ESCROW_OWNER_MISMATCH" (escrow.owner == expectedOwner)
let cfg = getConfig escrow.collateralType cfgs
price <- getCtxPriceByType operator escrow.collateralType feeds safe
let value = if cfg.enabled
then escrow.amount * price * intToNumeric cfg.collateralFactorBps / 10000.0
else 0.0
rest <- computeWeightedCollateralValue operator expectedOwner cfgs eRest feeds
safe
return (value + rest)
choice Lending_Borrow : (ContractId CantonLendingService, ContractId
CantonDebtPosition, ContractId CantonMUSD)
with
user : Party
borrowAmount : Money
escrowCids : [ContractId EscrowedCollateral] -- user-supplied, no dedup
priceFeedCids : [ContractId CantonPriceFeed]
controller user
```

RECOMMENDATION

Deduplicate the escrowCids list before iterating. Add a check at the top of each affected choice:

Alternatively, convert to DA.Set.Set (ContractId EscrowedCollateral) to structurally prevent duplicates.

```
let uniqueEscrows = DA.List.unique escrowCids
assertMsg "DUPLICATE_ESCROW_IDS" (length uniqueEscrows == length escrowCids)
```

6.2.2 Omitting Borrower Collateral from EscrowCids Forces Liquidation of Healthy Positions

HIGH FIXED

DESCRIPTION

In Lending_Liquidate, the liquidator supplies the borrower's escrowCids list as a function argument. The contract does not verify that this list is complete. A liquidator can intentionally omit some of the borrower's collateral positions from the list, making a healthy position appear und ollateralized.

The health check computes collateral value only from the supplied escrowCids. If the borrower has 3 escrow positions totaling 300 USD but the liquidator only passes 1 worth 100 USD, the health factor is computed against 100 USD instead of 300 USD. This causes the health check to fail, allowing the liquidator to seize collateral from a position that should not be liquidatable.

The liquidator profits by seizing collateral at a penalty discount from positions that are actually healthy.

CODE LOCATION

File: CantonLending.daml

CantonLending.daml

```
choice Lending_Liquidate : (ContractId CantonLendingService, ContractId
CantonLiquidationReceipt)
with
liquidator : Party
borrower : Party
repayAmount : Money
targetEscrowCid : ContractId EscrowedCollateral
debtCid : ContractId CantonDebtPosition
musdCid : ContractId CantonMUSD
escrowCids : [ContractId EscrowedCollateral] -- liquidator-supplied, no
completeness check
priceFeedCids : [ContractId CantonPriceFeed]
controller liquidator
totalLiqValue <- computeLiquidationThresholdValue operator borrower configs
escrowCids priceFeedCids False
let healthFactor = if totalDebt > 0.0 then totalLiqValue / totalDebt else 999.0
assertMsg "POSITION_HEALTHY" (healthFactor < 1.0)
```

RECOMMENDATION

Do not rely on the liquidator to supply the borrower's collateral list. Store escrow contract IDs in theCantonDebtPosition template or maintain a registry so the protocol can look up the complete set of collateral for any borrower.

```
-- In CantonDebtPosition, add:
escrowCids : [ContractId EscrowedCollateral]
-- In Lending_Liquidate, use the borrower's on-chain list:
debt <- fetch debtCid
let borrowerEscrows = debt.escrowCids
totalLiqValue <- computeLiquidationThresholdValue operator borrower configs
borrowerEscrows priceFeedCids False
```

6.2.3 targetEscrowCid Not Validated Against Borrower Allows Seizing Third-Party Collateral

HIGH FIXED

DESCRIPTION

In Lending_Liquidate, the targetEscrowCid parameter specifies which collateral to seize. The contract fetches the target escrow and ex ises Escrow_Seize on it, but never checks that the escrow belongs to the borrower being liquidated.

A liquidator can pass any active EscrowedCollateral contract ID as targetEscrowCid, including one owned by an unrelated third party. The ownership assertion only checks the escrowCids list (the health check inputs), not the target being seized. This lets a liquidator call Lending_Liquidate on a legitimately und ollateralized borrower but seize collateral from a completely different user who has nothing to do with the liquidation.

CODE LOCATION

File: CantonLending.daml

CantonLending.daml

```
choice Lending_Liquidate : (ContractId CantonLendingService, ContractId
CantonLiquidationReceipt)
with
liquidator : Party
borrower : Party
repayAmount : Money
targetEscrowCid : ContractId EscrowedCollateral -- no ownership check
debtCid : ContractId CantonDebtPosition
musdCid : ContractId CantonMUSD
escrowCids : [ContractId EscrowedCollateral]
priceFeedCids : [ContractId CantonPriceFeed]
controller liquidator
do
...
-- targetEscrowCid is fetched but owner is never checked against borrower
```

```
targetEscrow <- fetch targetEscrowCid
let cfg = getConfig targetEscrow.collateralType configs
colPrice <- getCtxPriceByType operator targetEscrow.collateralType priceFeedCids
False
let seizeValueUsd = actualRepay * (10000.0 + intToNumeric
cfg.liquidationPenaltyBps) / 10000.0
let seizeAmount = seizeValueUsd / colPrice
let actualSeize = min seizeAmount targetEscrow.amount
-- Seizes collateral without verifying targetEscrow.owner == borrower
(remainingEscrow, seized) <- exercise targetEscrowCid Escrow_Seize with
seizeAmount = actualSeize
```

RECOMMENDATION

Add an ownership check on the target escrow immediately after fetching it:

Also verify that targetEscrowCid is a member of the borrower's escrowCids list.

```
targetEscrow <- fetch targetEscrowCid
assertMsg "TARGET_ESCROW_NOT_OWNED_BY_BORROWER" (targetEscrow.owner == borrower)
```

6.2.4 GovernanceActionLog Replay Allows Single Approval to Authorize Unlimited Operations

HIGH FIXED

DESCRIPTION

When a MultiSigProposal is executed via Proposal_Execute, it creates a GovernanceActionLog. This log is an immutable record with no choices (it cannot be archived or consumed). Downstream templates like MinterRegistry fetch the log to verify governance approval before executing sensitive operations.

The problem is that the GovernanceActionLog is never consumed or archived after use. Because the log persists indefinitely, it can be fetched repeatedly. A single governance approval can authorize unlimited downstream operations. For example, one approval to add a minter can be used to add many different minters by calling MinterRegistry_AddMinter multiple times with the same governanceProofCid.

The log template has a comment "No choices - immutable audit record" confirming it was designed to be permanent, but this makes it replayable.

CODE LOCATION

File: Governance.daml

Governance.daml

```
template GovernanceActionLog
with
operator : Party
proposalId : Text
actionType : ActionType
description : Text
payload : Text
payloadHash : Text
approvers : [Party]
executedBy : Party
executedAt : Time
where
signatory operator, executedBy
-- No choices - immutable audit record
choice MinterRegistry_AddMinter : ContractId MinterRegistry
with
newMinter : Party
quota : Decimal
governanceProofCid : ContractId GovernanceActionLog
controller operator
do
proof <- fetch governanceProofCid -- non-consuming read, can be reused
assertMsg "WRONG_ACTION_TYPE" (proof.actionType == MinterAuthorization)
```

RECOMMENDATION

Add a consuming choice to GovernanceActionLog that archives it after use, or change the downstream templates to archive the log after verifying it:

```
-- Option 1: Add a consume choice to GovernanceActionLog
choice ConsumeProof : ()
controller operator
do return ()
-- Option 2: Archive in the consuming template
archive governanceProofCid
```

6.2.5 Share Price Inconsistency Between Deposit and Withdraw in CantonSMUSD

HIGH FIXED

DESCRIPTION

The SMUSD_Deposit and SMUSD_Withdraw choices in the V3 CantonSMUSD template use different formulas to calculate the share price. Deposit uses a virtual offset formula (virtualAssets / virtualShares with +1000 offsets) to prevent donation attacks. Withdraw uses the raw formula (totalAssets / totalShares) without the virtual offset.

This means a user deposits at one price and withdraws at a different price. For small pool sizes, the difference between the virtual and raw formulas is significant. An attacker can exploit this by:

Depositing when the pool is small to get shares at the virtual-offset priceWaiting for the pool to grow (diluting the virtual offset)Withdrawing at the raw price which is now higher

The reverse is also exploitable depending on the totalAssets/totalShares ratio.

CODE LOCATION

File: Minted/Protocol/V3.daml

Minted/Protocol/V3.daml

```
choice SMUSD_Deposit : (ContractId CantonSMUSD, ContractId CooldownTicket)
with
user : Party
musdCid : ContractId MintedMUSD
controller user
do
...
let virtualShares = totalShares + 1000.0
let virtualAssets = totalAssets + 1000.0
let sharePrice = virtualAssets / virtualShares
let newShares = musd.amount / sharePrice
choice SMUSD_Withdraw : (ContractId CantonSMUSD, ContractId MintedMUSD)
with
user : Party
ticketCid : ContractId CooldownTicket
controller user
do
...
let sharePrice = if totalShares == 0.0 then 1.0 else totalAssets / totalShares
let musdAmount = ticket.shares * sharePrice
```

RECOMMENDATION

Use the same share price formula for both deposit and withdraw. Apply the virtual offset consistently:

```
-- In SMUSD_Withdraw, use the same virtual offset:
let virtualShares = totalShares + 1000.0
let virtualAssets = totalAssets + 1000.0
let sharePrice = virtualAssets / virtualShares
let musdAmount = ticket.shares * sharePrice
```

6.2.6 Legacy SyncYield Missing All Security Guards

MEDIUM ACKNOWLEDGED

DESCRIPTION

The CantonStakingService template has a legacy SyncYield choice that was kept for backwards compatibility. This choice updates the global share price by computing newTotalTreasuryAssets / globalTotalShares. Unlike the newer SyncGlobalSharePrice choice, SyncYield has no security guards.

Missing protections include:

No pause check (SERVICE_PAUSED) - works even when service is pausedNo share price decrease limit - can crash the share price to any valueNo yieldAccrued validation - accepts negative yieldNo minimum share price floor

The newer SyncGlobalSharePrice has a 10% maximum decrease guard (newGlobalSharePrice >= globalSharePrice * 0.9). SyncYield has no such protection. A malicious or compromised operator can call SyncYield with newTotalTreasuryAssets = 1 to crash the share price to near zero, causing massive losses for all stakers.

CODE LOCATION

File: CantonSMUSD.daml

CantonSMUSD.daml

```
choice SyncYield : ContractId CantonStakingService
with
newTotalTreasuryAssets : Money
yieldAccrued : Money
epochNumber : Int
controller operator
do
assertMsg "EPOCH_NOT_SEQUENTIAL" (epochNumber > lastSyncEpoch)
-- No pause check
-- No price decrease limit
-- No yieldAccrued >= 0 check
let newSharePrice = if globalTotalShares > 0.0
then newTotalTreasuryAssets / globalTotalShares
else globalSharePrice
archive self
create this with
globalSharePrice = newSharePrice
globalTotalAssets = newTotalTreasuryAssets
lastSyncEpoch = epochNumber
assertMsg "SERVICE_PAUSED" (not paused)
assertMsg "YIELD_MUST_BE_NONNEG" (yieldAccrued >= 0.0)
let minAllowedPrice = globalSharePrice * 0.9
```

```
assertMsg "SHARE_PRICE_DECREASE_TOO_LARGE" (newGlobalSharePrice >=
minAllowedPrice)
```

RECOMMENDATION

Either remove the legacy SyncYield choice entirely, or add all the same guards that SyncGlobalSharePrice has:

```
choice SyncYield : ContractId CantonStakingService
with ...
controller operator
do
assertMsg "SERVICE_PAUSED" (not paused)
assertMsg "EPOCH_NOT_SEQUENTIAL" (epochNumber > lastSyncEpoch)
assertMsg "YIELD_MUST_BE_NONNEG" (yieldAccrued >= 0.0)
let newSharePrice = if globalTotalShares > 0.0
then newTotalTreasuryAssets / globalTotalShares
else globalSharePrice
let minAllowedPrice = globalSharePrice * 0.9
assertMsg "SHARE_PRICE_DECREASE_TOO_LARGE" (newSharePrice >= minAllowedPrice)
...
```

6.2.7 Phantom Migration Attack via Arbitrary Text Strings

MEDIUM FIXED

DESCRIPTION

The MigrationTicket template stores contracts to migrate as a list of Text strings (contractsToMigrate : [Text]). These are "serialized" contract IDs, but the template never validates them against actual on-chain contracts. The MigrationTicket_Execute choice does not fetch or verify any of the listed contracts.

An attacker can create a MigrationTicket with fabricated contract ID strings. When executed, UpgradeRegistry_RecordMigration is called with the count, incrementing totalMigrated. The systemrecords a migration that never actually happened.

This corrupts migration tracking and can be used to artificially meet migration thresholds, or to prevent legitimate migrations by exhausting quotas.

CODE LOCATION

File: Upgrade.daml

Upgrade.daml

```
template MigrationTicket
with
```

```
  operator : Party
  holder : Party
  upgradeKey : (Party, Text, Text)
  contractsToMigrate : [Text] -- arbitrary strings, never validated
  requestedAt : Time
  batchSize : Int
  direction : MigrationDirection
  where
  signatory operator, holder
  ensure length contractsToMigrate > 0
  && batchSize > 0
  && batchSize <= 100
  choice MigrationTicket_Execute : ContractId UpgradeMigrationLog
  controller operator
  do
  now <- getTime
  (registryCid, registry) <- fetchByKey @UpgradeRegistry upgradeKey
  assertMsg "UPGRADE_NOT_ACTIVE" (registry.status == Active)
  -- Records count based on contractsToMigrate length, never verifies the contracts
  exist
```

RECOMMENDATION

Use actual ContractId types instead of Text strings, and fetch each contract during execution to verify it exists:

```
contractsToMigrate : [ContractId SomeMigratableContract]
-- In MigrationTicket_Execute:
forA_ contractsToMigrate $ \cid -> do
_ <- fetch cid -- verify contract exists
return ()
```

6.2.8 Incomplete State Machine in UpgradeRegistry

MEDIUM FIXED

DESCRIPTION

The UpgradeRegistry template has a status field that can be Active, Completed, or RolledBack. However, the state transitions are not properly guarded. The UpgradeRegistry_RecordMigration choice does not check that the status is Active before recording a forward migration (it only checks the migration window time). The UpgradeRegistry_Complete choice does not check that the status is Active before marking it Completed.

This means migrations can be recorded on a RolledBack registry (if within the time window), and a RolledBack registry can be marked Complete. The status field does not reliably represent the actual state of the upgrade.

CODE LOCATION

File: Upgrade.daml

Upgrade.daml

```
choice UpgradeRegistry_RecordMigration : ContractId UpgradeRegistry
with
count : Int
direction : MigrationDirection
controller operator
do
now <- getTime
case direction of
Forward -> do
assertMsg "MIGRATION_WINDOW_CLOSED" (now <= migrationDeadline)
-- No check: status == Active
create this with totalMigrated = totalMigrated + count
Backward -> do
assertMsg "ROLLBACK_WINDOW_CLOSED" (now <= rollbackDeadline)
create this with totalRolledBack = totalRolledBack + count
choice UpgradeRegistry_Complete : ContractId UpgradeRegistry
controller operator
do
now <- getTime
assertMsg "MIGRATION_WINDOW_NOT_ENDED" (now > migrationDeadline)
-- No check: status == Active (can complete a RolledBack registry)
create this with status = Completed
```

RECOMMENDATION

Add status checks to all state-transitioning choices:

```
-- In UpgradeRegistry_RecordMigration:
assertMsg "UPGRADE_NOT_ACTIVE" (status == Active)
-- In UpgradeRegistry_Complete:
assertMsg "UPGRADE_NOT_ACTIVE" (status == Active)
```

6.2.9 Fixed Exchange Rate Pool Drain

MEDIUM FIXED

DESCRIPTION

The LiquidityPool template in V3.daml uses a fixed exchangeRate field for swaps instead of a constant-product AMM formula. The exchange rate is set at pool creation and only changes when the pool is recreated.

This means swaps do not move the price. An attacker can repeatedly swap in one direction at the fixed rate, draining one side of the pool. In a real AMM, each swap moves the price to make further swaps less favorable. With a fixed rate, the attacker gets the same rate on every swap until the pool is empty.

For example, if exchangeRate = 50000 (1 BTC = 50000 mUSD) and the pool has 100 BTC inquoteReserve, an attacker can swap 5,000,000 mUSD for all 100 BTC at the fixed rate. The rate does not adjust as liquidity is removed.

CODE LOCATION

File: Minted/Protocol/V3.daml

Minted/Protocol/V3.daml

```
template LiquidityPool
with
operator : Party
baseSymbol : Text
quoteSymbol : Text
baseReserve : Money
quoteReserve : Money
exchangeRate : Money -- fixed rate, never adjusts on swap
choice Swap_mUSD_For_Collateral : (ContractId LiquidityPool, Money)
with
musdCid : ContractId MintedMUSD
receiver : Party
controller receiver
do
...
let collateralOut = musdAmount / exchangeRate -- fixed rate
assertMsg "INSUFFICIENT_LIQUIDITY" (quoteReserve >= collateralOut)
archive musdCid
newPool <- create this with
baseReserve = baseReserve + musdAmount
quoteReserve = quoteReserve - collateralOut -- drains without price impact
```

RECOMMENDATION

Replace the fixed exchange rate with a constant-product AMM formula (x * y = k):

```
let k = baseReserve * quoteReserve
let newBaseReserve = baseReserve + musdAmount
let newQuoteReserve = k / newBaseReserve
let collateralOut = quoteReserve - newQuoteReserve
```

6.2.10 Vault_Repay Does Not Burn mUSD

MEDIUM FIXED

DESCRIPTION

The Vault_Repay choice reduces the vault's debt (principalDebt and accruedInterest) but does not require or burn any mUSD token. The repayment is purely an accounting operation that decrements debt fields without consuming actual tokens.

A user can call Vault_Repay with any repayAmount to reduce their debt to zero without spending any mUSD. The mUSD remains in the user's possession while the debt is erased. This makes the vault's mUSD effectively free money.

In contrast, the side BorrowModule requires the borrower to transfer mUSD tokens during repayment. The DAML side has no token consumption requirement.

CODE LOCATION

File: Minted/Protocol/V3.daml

Minted/Protocol/V3.daml

```
choice Vault_Repay : ContractId Vault
with
repayAmount : Money
controller owner
do
now <- getTime
let elapsed = convertRelTimeToMicroseconds (subTime now lastInterestUpdate)
let secondsElapsed = intToNumeric (elapsed / 1000000)
let yearSeconds = 31536000.0 : Money
let newInterest = principalDebt * intToNumeric config.interestRateBps *
secondsElapsed / (10000.0 * yearSeconds)
let totalInterest = accruedInterest + newInterest
let (remainingRepay, newAccrued) =
if repayAmount >= totalInterest
then (repayAmount - totalInterest, 0.0)
else (0.0, totalInterest - repayAmount)
let newPrincipal = if remainingRepay > principalDebt
then 0.0
else principalDebt - remainingRepay
create this with
principalDebt = newPrincipal
accruedInterest = newAccrued
lastInterestUpdate = now
-- No mUSD token is consumed or burned
```

RECOMMENDATION

Require a mUSD token contract ID and burn it during repayment:

```
choice Vault_Repay : ContractId Vault
with
repayAmount : Money
musdCid : ContractId MintedMUSD
controller owner
do
musd <- fetch musdCid
assertMsg "OWNER_MISMATCH" (musd.owner == owner)
assertMsg "INSUFFICIENT_REPAY" (musd.amount >= repayAmount)
archive musdCid -- burn the repayment token
...
```

6.2.11 Liquidate Does Not Require mUSD Repayment Token

MEDIUM FIXED

DESCRIPTION

The Vault Liquidate choice requires the liquidator to specify a repayAmount, but it does not require or consume any mUSD token as repayment. Similar to V3_DAML_H04 (Vault_Repay), the debt reduction is purely an accounting operation.

A liquidator can call Liquidate with any repayAmount without actually providing mUSD. They receive real collateral (via the liquidation penalty mechanism) while not spending any tokens. This makesliquidation a pure profit extraction mechanism rather than a debt repayment service.

CODE LOCATION

File: Minted/Protocol/V3.daml

Minted/Protocol/V3.daml

```
choice Liquidate : (ContractId Vault, ContractId LiquidationReceipt)
with
liquidator : Party
repayAmount : Money
oracleCid : ContractId PriceOracle
controller liquidator
do
...
-- Calculates debt, health factor, seizure amounts
-- Updates vault with reduced debt and collateral
-- Creates LiquidationReceipt
```

```
-- No mUSD token is required, consumed, or burned
-- repayAmount is just a number, not backed by actual tokens
```

RECOMMENDATION

Require the liquidator to provide mUSD tokens for repayment:

```
choice Liquidate : (ContractId Vault, ContractId LiquidationReceipt)
with
liquidator : Party
repayAmount : Money
musdCid : ContractId MintedMUSD
oracleCid : ContractId PriceOracle
controller liquidator
do
musd <- fetch musdCid
assertMsg "NOT_OWNER" (musd.owner == liquidator)
assertMsg "INSUFFICIENT_MUSD" (musd.amount >= repayAmount)
archive musdCid -- burn the repayment
...
```

6.2.12 Precision Validation Offset Silently Passes Invalid Amounts

MEDIUM ACKNOWLEDGED

DESCRIPTION

The Asset_Split choice validates that the split amount respects the instrument's decimal precision. The check uses floor(splitAmount * factor + 0.1) == floor(splitAmount * factor) to verify precision. The + 0.1 offset is intended to handle floating-point rounding, but it is too large.

For an instrument with decimalScale = 2 (cents precision), the value 1.009 should be invalid because it has 3 decimal places. However:

splitAmount * factor = 1.009 * 100 = 100.9floor(100.9 + 0.1) = floor(101.0) = 101floor(100.9) = 100101 != 100: this correctly rejects

But for a value like 1.005:

splitAmount * factor = 1.005 * 100 = 100.5floor(100.5 + 0.1) = floor(100.6) = 100floor(100.5) = 100100 == 100: this incorrectly accepts

The + 0.1 offset means any amount within 0.1/factor of a valid precision boundary passes the check. For decimalScale = 2, amounts that are off by up to 0.001 are silently accepted.

CODE LOCATION

File: InstitutionalAssetV4.daml

InstitutionalAssetV4.daml

```
choice Asset_Split : (ContractId Asset, ContractId Asset)
with splitAmount : Decimal
controller owner
do
assertMsg "Cannot split a locked asset" (isNone lock)
assertMsg "Split amount must be positive" (splitAmount > 0.0)
(_, instrument) <- fetchByKey @Instrument (depository, instrumentId)
let decimalScale = getField @"decimalScale" instrument
let factor = intToDecimal (10 ^ decimalScale)
assertMsg "Split amount violates instrument precision scale"
(floor (splitAmount * factor + 0.1) == floor (splitAmount * factor))
assertMsg "Split amount must be less than total amount" (splitAmount < amount)
a1 <- create this with amount = splitAmount
a2 <- create this with amount = amount - splitAmount
return (a1, a2)
```

RECOMMENDATION

Remove the + 0.1 offset and use a cleaner precision check:

```
let scaled = splitAmount * factor
assertMsg "Split amount violates instrument precision scale"
(scaled == intToDecimal (round scaled))
```

6.2.13 No Validator Whitelist in Claim

MEDIUM ACKNOWLEDGED

DESCRIPTION

The Claim Add_ choice accepts s from any party that creates a contract. There is no whitelist of authorized validators checked during collection. The only check is that the same validator has not already been added (duplicate prevention).

An attacker can create contracts signed by their own parties and submit them to Claim. With enough attacker-controlled "validators", the threshold is met and Finalize_ _Mint can be called to mint mUSD without any legitimate operation.

The Service template has a validators list, but Claim does not reference it and does not verify that signers are in the authorized set.

CODE LOCATION

File: MUSD_Protocol.daml

MUSD_Protocol.daml

```
template [redacted]Claim
with
operator : Party
user : Party
amount : Decimal
nonce : Int
requiredSignatures : Int
collectedValidators : [Party]
where
signatory user, operator
choice Add_[redacted] : ContractId [redacted]Claim
with
[redacted]Cid : ContractId [redacted]
controller user
do
att <- fetch [redacted]Cid
assertMsg "Claim: Invalid Nonce" (att.nonce == nonce)
assertMsg "Claim: Invalid User" (att.user == user)
assertMsg "Claim: Amount Mismatch" (att.amount == amount)
assertMsg "Claim: Validator already signed" (not (elem att.validator
collectedValidators))
-- No check: att.validator `elem` authorizedValidators
```

RECOMMENDATION

Add an authorizedValidators field to Claim and verify each signer:

```
template [redacted]Claim
with
...
authorizedValidators : [Party]
-- In Add_[redacted]:
assertMsg "UNAUTHORIZED_VALIDATOR" (att.validator `elem` authorizedValidators)
```

6.2.14 Supply Tracking Inconsistent Across Lifecycle

MEDIUM ACKNOWLEDGED

DESCRIPTION

The mUSD supply tracking in MUSD_Protocol.daml is inconsistent across the lifecycle. When a user locks mUSD for via Lock_Musd_For_ , the token is archived (removed from circulation) but MintingService.currentSupply is not decremented. When the claim finalizes on the target side via Finalize_ _Mint, currentSupply is incremented again.

This means the same mUSD amount is counted twice in the supply: once from the original mint, and once from the mint. Over multiple operations, currentSupply drifts upward and eventually hits supplyCap, blocking all new minting even though the actual circulating supply has not increased.

Additionally, Cancel_ Lock creates new Musd tokens to refund the user, but the original tokens were already counted in supply. If the cancellation mints fresh tokens while the original supply count was never decremented, the supply counter is now higher than actual circulation.

CODE LOCATION

File: MUSD_Protocol.daml

MUSD_Protocol.daml

```
nonconsuming choice Lock_Musd_For_ : ContractId Lock
with
user : Party
musdCid : ContractId Musd
...
controller user
do
musd <- fetch musdCid
archive musdCid -- removes from circulation
-- MintingService.currentSupply is NOT decremented
svc <- fetch mintingServiceCid
assertMsg "_MINT_EXCEEDS_SUPPLY_CAP" (svc.currentSupply + amount <=
svc.supplyCap)
archive mintingServiceCid
newSvc <- create svc with currentSupply = svc.currentSupply + amount
choice Cancel_Lock : ContractId Musd
controller user, operator
do
...
create Musd with
issuer = operator
owner = user
amount = amount -- new token, supply not updated
```

RECOMMENDATION

Decrement currentSupply when locking mUSD for . Do not increment on the target domain if the same supply tracker is shared. If supply is tracked per domain, clarify the accounting.

```
-- In Lock_Musd_For_: decrement supply
archive mintingServiceCid
newSvc <- create svc with currentSupply = svc.currentSupply - musd.amount
```

6.2.15 Claim and Lock Not Linked Enables Phantom Claims

MEDIUM ACKNOWLEDGED

DESCRIPTION

Claim and Lock are independent templates with no on-chain link between them. A Claim can be created without a corresponding Lock. There is no verification that a lock actually exists for the claimed nonce and amount.

An attacker can create a Claim directly (both user and operator are signatories, and if the operator key is compromised or the operator is malicious) without ever locking mUSD. The claim then collects s and mints new mUSD on the target domain, creating tokens backed by nothing.

The nonce in Claim is not validated against any Lock. The amount is checked against s, but s themselves are not linked to a Lock either.

CODE LOCATION

File: MUSD_Protocol.daml

MUSD_Protocol.daml

```
template Claim
with
operator : Party
user : Party
amount : Decimal
nonce : Int
requiredSignatures : Int
collectedValidators : [Party]
where
signatory user, operator
-- No lockCid or lockReference field
choice Finalize_Mint : (ContractId MintingService, ContractId Musd)
with
mintingServiceCid : ContractId MintingService
controller user
do
assertMsg "Claim: Not enough signatures" (length collectedValidators >=
requiredSignatures)
-- No verification that a Lock with this nonce exists
```

RECOMMENDATION

Add a lockCid field to Claim and verify the lock exists and matches:

```
template Claim
with
```

```
...
lockCid : ContractId [redacted]Lock
-- In Finalize_[redacted]_Mint or at creation:
lock <- fetch lockCid
assertMsg "LOCK_NONCE_MISMATCH" (lock.nonce == nonce)
assertMsg "LOCK_AMOUNT_MISMATCH" (lock.amount == amount)
assertMsg "LOCK_USER_MISMATCH" (lock.user == user)
```

6.2.16 MUSD_Locked Signatory Downgrade Enables Permanent Lock

MEDIUM ACKNOWLEDGED

DESCRIPTION

The MUSD template requires both provider and owner as signatories (dual-signatory). When MUSD_Lock is ex ised, it creates a MUSD_Locked contract where the signatory is only the provider. The owner is downgraded to an observer.

This means the provider can unilaterally control the locked tokens. The owner cannot archive, unlock, or interact with their locked tokens without the provider's cooperation. There is no timeout or expiry on the lock. If the provider loses their key, becomes unresponsive, or acts maliciously, the owner's tokens are permanently locked with no recourse.

CODE LOCATION

File: MintedMUSD.daml

MintedMUSD.daml

```
choice MUSD_Lock : ContractId MUSD_Locked
with
reason : Text
controller provider
do
create MUSD_Locked with usd = this; reason
template MUSD_Locked
with
usd : MUSD
reason : Text
where
signatory usd.provider -- only provider is signatory
observer usd.owner -- owner is demoted to observer
choice MUSD_Unlock : ContractId MUSD
controller usd.provider -- only provider can unlock
do create usd
```

RECOMMENDATION

Add the owner as a signatory on MUSD_Locked, or add a timeout-based self-unlock choice:

```
template MUSD_Locked
with
usd : MUSD
reason : Text
lockedAt : Time
where
signatory usd.provider, usd.owner
choice MUSD_Unlock : ContractId MUSD
controller usd.provider
do create usd
-- Emergency unlock after 30 days
choice MUSD_EmergencyUnlock : ContractId MUSD
controller usd.owner
do
now <- getTime
assertMsg "LOCK_PERIOD_NOT_EXPIRED" (subTime now lockedAt >= days 30)
create usd
```

6.2.17 IssuerRole currentSupply Never Decremented on Redemption

MEDIUM ACKNOWLEDGED

DESCRIPTION

The IssuerRole template tracks currentSupply which is incremented when minting via IssuerRole_Mint. However, when tokens are redeemed through MUSD_Redeem and MUSD_FulfillRedemption, the IssuerRole is not involved and currentSupply is never decremented.

Over time, currentSupply only grows. Once it reaches supplyCap, no new minting is possible even though tokens have been redeemed and removed from circulation. The supply cap becomes a lifetime minting limit rather than a circulating supply cap.

CODE LOCATION

File: MintedMUSD.daml

MintedMUSD.daml

```
choice IssuerRole_Mint : (ContractId IssuerRole, ContractId MUSD)
with
mintOwner : Party
mintAmount : Decimal
mintCurrency : Text
mintObservers : [Party]
```

```
controller issuer, mintOwner
do
assertMsg "Mint amount must be positive." (mintAmount > 0.0)
assertMsg "EXCEEDS_SUPPLY_CAP" (currentSupply + mintAmount <= supplyCap)
musdCid <- create MUSD with ...
newRole <- create this with
currentSupply = currentSupply + mintAmount
choice MUSD_FulfillRedemption : ()
controller usd.provider
do return () -- archives the redemption request (and its embedded token) but does
not decrement supply
```

RECOMMENDATION

Add a burn/redeem path that decrements currentSupply on the IssuerRole:

Call IssuerRole_Burn from MUSD_FulfillRedemption.

```
choice IssuerRole_Burn : ContractId IssuerRole
with
burnAmount : Decimal
controller issuer
do
assertMsg "BURN_AMOUNT_POSITIVE" (burnAmount > 0.0)
assertMsg "BURN_EXCEEDS_SUPPLY" (burnAmount <= currentSupply)
create this with currentSupply = currentSupply - burnAmount
```

6.2.18 No Compliance Integration in MintedMUSD

MEDIUM FIXED

DESCRIPTION

The MintedMUSD.daml module defines the core mUSD token (MUSD template) with mint, transfer, split, merge, redeem, and lock operations. None of these operations check the ComplianceRegistry for blacklisted or frozen users.

A blacklisted party can receive mUSD via MUSD_AcceptTransfer, split tokens, merge tokens, and redeem them. A frozen party can still transfer tokens out. The Compliance.daml module exists in the codebase but is never imported or referenced by MintedMUSD.daml.

This undermines the entire compliance framework and creates regulatory risk since the protocol claims to support compliance but does not enforce it at the token level.

CODE LOCATION

File: MintedMUSD.daml

MintedMUSD.daml

```
template MUSD
with
provider : Party
owner : Party
amount : Decimal
currency : Text
observers : [Party]
where
signatory provider, owner
observer observers
ensure amount >= minimumAmount
choice MUSD_Redeem : ContractId MUSD_RedemptionRequest ...
choice MUSD_Lock : ContractId MUSD_Locked ...
choice MUSD_Split : (ContractId MUSD, ContractId MUSD) ...
choice MUSD_Merge : ContractId MUSD ...
choice MUSD_Transfer : ContractId MUSD_TransferProposal ...
```

RECOMMENDATION

Import the Compliance module and add blacklist/freeze checks to MUSD_Transfer, MUSD_AcceptTransfer, MUSD_Redeem, and IssuerRole_Mint:

```
import Compliance (ComplianceRegistry(..))
-- In MUSD_Transfer:
(_, registry) <- fetchByKey @ComplianceRegistry regulator
assertMsg "SENDER_BLACKLISTED" (not (Set.member owner registry.blacklisted))
assertMsg "SENDER_FROZEN" (not (Set.member owner registry.frozen))
assertMsg "RECEIVER_BLACKLISTED" (not (Set.member newOwner registry.blacklisted))
```

6.2.19 Single Member Emergency Rollback Without Multi-Sig

MEDIUM FIXED

DESCRIPTION

The UpgradeRegistry_EmergencyRollback choice allows any single governance member to roll back an active upgrade. The choice only requires that the authorizer is in the governance list. There is no multi-signature requirement, no quorum, and no timelock.

A single compromised or malicious governance member can roll back a production upgrade, reverting the protocol to a previous version. This can disrupt active migrations, break users who have already migrated, and cause data inconsistency between migrated and non-migrated state.

The normal proposal flow requires M-of-N approval. Emergency rollback should require at least the elevated threshold, not a single member.

CODE LOCATION

File: Upgrade.daml

Upgrade.daml

```
choice UpgradeRegistry_EmergencyRollback : ContractId UpgradeRegistry
with
reason : Text
authorizer : Party
controller authorizer
do
assertMsg "NOT_GOVERNANCE" (authorizer `elem` governance)
assertMsg "REASON_REQUIRED" (T.length reason > 0)
now <- getTime
assertMsg "ROLLBACK_WINDOW_CLOSED" (now <= rollbackDeadline)
create this with status = RolledBack
```

RECOMMENDATION

Require a governance proposal for emergency rollback, or at minimum require multiple governance members:

```
choice UpgradeRegistry_EmergencyRollback : ContractId UpgradeRegistry
with
reason : Text
governanceProofCid : ContractId GovernanceActionLog
controller operator
do
proof <- fetch governanceProofCid
assertMsg "WRONG_ACTION_TYPE" (proof.actionType == ContractUpgrade)
...
```

6.2.20 No Compliance Checks in CantonLending

LOW FIXED

DESCRIPTION

None of the 12 user-facing choices in CantonLendingService check the ComplianceRegistry for blacklisted or frozen users. The Compliance.daml module defines a ComplianceRegistry with blacklist and freeze functionality, along with nonconsuming validation choices. However, CantonLending.daml never imports or references the compliance module.

A blacklisted or frozen user can freely deposit collateral, borrow mUSD, withdraw collateral, repay debt, and participate as a liquidator. This defeats the purpose of the compliance system and creates regulatory risk for the protocol.

CODE LOCATION

File: Compliance.daml

Compliance.daml

```
template ComplianceRegistry
with
regulator : Party
operator : Party
blacklisted : Set.Set Party
frozen : Set.Set Party
lastUpdated : Time
```

RECOMMENDATION

Add a compliance check at the start of each user-facing choice by fetching the ComplianceRegistry and calling ValidateTransfer or an equivalent check:

```
-- At the start of each user-facing choice:
(_, registry) <- fetchByKey @ComplianceRegistry (regulator)
assertMsg "USER_BLACKLISTED" (not (Set.member user registry.blacklisted))
assertMsg "USER_FROZEN" (not (Set.member user registry.frozen))
```

6.2.21 Missing Validation on Market State Values

LOW FIXED

DESCRIPTION

The RateService_SyncMarketState choice accepts newTotalBorrows and newTotalSupply from the operator without validating that they are non-negative or reasonable. Negative values would corrupt the utilization rate calculation and all downstream interest rate computations.

The InterestRateService template ensure clause validates the rate parameters (baseRateBps >= 0, etc.) but the MarketState record has no validation. The SyncMarketState choice only checks that BlockNumber is newer than the previous one.

CODE LOCATION

File: InterestRateService.daml

InterestRateService.daml

```
data MarketState = MarketState with
totalBorrows : Decimal
totalSupply : Decimal
lastUpdateTime : Time
    BlockNumber : Int
```

```
choice RateService_SyncMarketState : ContractId InterestRateService
with
newTotalBorrows : Decimal
newTotalSupply : Decimal
   BlockNumber : Int
controller operator
do
assertMsg "STALE_DATA" (   BlockNumber > marketState.   BlockNumber)
now <- getTime
create this with
marketState = MarketState with
totalBorrows = newTotalBorrows
totalSupply = newTotalSupply
lastUpdateTime = now
   BlockNumber =    BlockNumber
```

RECOMMENDATION

Add validation on the market state values:

```
assertMsg "BORROWS_MUST_BE_NON_NEGATIVE" (newTotalBorrows >= 0.0)
assertMsg "SUPPLY_MUST_BE_NON_NEGATIVE" (newTotalSupply >= 0.0)
assertMsg "BORROWS_CANNOT_EXCEED_SUPPLY" (newTotalBorrows <= newTotalSupply)
```

6.2.22 Integer Truncation in Supply Rate Calculation

LOW FIXED

DESCRIPTION

The RateService_GetSupplyRate choice computes the supply rate as (borrowRate * util * oneMinusReserve) / (10000 * 10000). All values are Int, so the entire calculation uses integer arithmetic with truncation.

The three-term multiplication in the numerator can produce very large intermediate values that overflow, or the final division by 100,000,000 (10000 * 10000) can truncate a meaningful result to zero for moderate inputs.

For example, if borrowRate = 500 (5%), util = 3000 (30%), oneMinusReserve = 9000, the correct supply rate is 500 * 3000 * 9000 / 100000000 = 135. But for lower values like borrowRate = 50, util = 500, oneMinusReserve = 9500, the result is 50 * 500 * 9500 / 100000000 = 2.375, which truncates to 2. The true value is nearly 20% larger.

CODE LOCATION

File: InterestRateService.daml

InterestRateService.daml

```
nonconsuming choice RateService_GetSupplyRate : Int
with
requester : Party
controller requester
do
borrowRate <- exercise self RateService_GetBorrowRate with requester
util <- exercise self RateService_GetUtilization with requester
let oneMinusReserve = 10000 - params.reserveFactorBps
return ((borrowRate * util * oneMinusReserve) / (10000 * 10000))
```

RECOMMENDATION

Use Decimal arithmetic for intermediate calculations:

Alternatively, split the division into two steps to reduce truncation error:

```
let supplyRate = (intToDecimal borrowRate * intToDecimal util * intToDecimal
oneMinusReserve) / 100000000.0
return (truncate supplyRate)
return ((borrowRate * util / 10000) * oneMinusReserve / 10000)
```

6.2.23 No Time-Based Staleness Check on Market Data

LOW FIXED

DESCRIPTION

The InterestRateService uses market data (totalBorrows and totalSupply) synced from via RateService_SyncMarketState. The sync validates that the BlockNumber is newer than theprevious one, but there is no time-based staleness check.

If the operator stops syncing market data (due to failure, operator downtime, or intentional neglect), the interest rate calculations continue using arbitrarily old data. The lastUpdateTime field is stored in MarketState but never checked by any rate calculation choice. Rates could be computed against market conditions from hours or days ago.

CODE LOCATION

File: InterestRateService.daml

InterestRateService.daml

```
nonconsuming choice RateService_GetUtilization : Int
with
requester : Party
controller requester
do
```

```
if marketState.totalSupply == 0.0
then return 0
else do
let util = (marketState.totalBorrows * 10000.0) / marketState.totalSupply
return (min 10000 (truncate util))
create this with
marketState = MarketState with
totalBorrows = newTotalBorrows
totalSupply = newTotalSupply
lastUpdateTime = now
[redacted]BlockNumber = [redacted]BlockNumber
```

RECOMMENDATION

Add a staleness check in rate calculation choices:

```
now <- getTime
let staleness = subTime now marketState.lastUpdateTime
assertMsg "MARKET_DATA_STALE" (staleness <= hours 6)
```

6.2.24 Unbounded Observer List Growth in IssuerRole

LOW FIXED

DESCRIPTION

The IssuerRole_Mint choice appends the mint recipient to the observers list on every mint. The list grows with each unique mint recipient and is never pruned. Over time, this list can become very large.

A large observer list increases the cost of every operation on the IssuerRole contract, slows ledger synchronization for all observers, and leaks information (every observer can see all future minting operations).

CODE LOCATION

File: MintedMUSD.daml

MintedMUSD.daml

```
newRole <- create this with
currentSupply = currentSupply + mintAmount
observers = mintOwner :: observers -- appends every new mint recipient
```

RECOMMENDATION

Do not append mint recipients to the IssuerRole observer list. Use a separate MintProposal pattern where the recipient only observes the proposal, not the entire IssuerRole:

```
newRole <- create this with
currentSupply = currentSupply + mintAmount
-- Do not modify observers
```

6.2.25 MintingService Missing Ensure Clause

LOW FIXED

DESCRIPTION

The MintingService template has no ensure clause. The supplyCap can be set to zero or negative, and currentSupply can be initialized to a value greater than supplyCap. These invariants are not enforced at the contract level.

If MintingService is created with supplyCap = 0, all minting is blocked. If created with currentSupply > supplyCap, the supply cap check in Mint_Musd will always fail, also blocking minting. If supplyCap is negative, the assertion currentSupply + amount <= supplyCap will cause unexpected behavior.

CODE LOCATION

File: MUSD_Protocol.daml

MUSD_Protocol.daml

```
template MintingService
with
operator : Party
usdcIssuer : Party
supplyCap : Decimal
currentSupply : Decimal
observers : [Party]
where
signatory operator
observer usdcIssuer, observers
-- No ensure clause
```

RECOMMENDATION

Add an ensure clause:

```
ensure supplyCap > 0.0
&& currentSupply >= 0.0
&& currentSupply <= supplyCap
```

6.2.26 Service Nonce Not Validated for Uniqueness

LOW FIXED

DESCRIPTION

The Lock_Musd_For_ choice accepts a nonce parameter from the user. There is no validation that this nonce is unique or sequential. The Service template does not track previously used nonces. A user can submit multiple locks with the same nonce, and the same nonce can be used by different users.

If nonces collide, validators may produce s that match multiple lock operations, creating confusion about which lock corresponds to which claim. This can lead to double-minting on the target domain or orphaned locks.

CODE LOCATION

File: MUSD_Protocol.daml

MUSD_Protocol.daml

```
template Service
with
operator : Party
validators : [Party]
observers : [Party]
where
signatory operator
observer validators, observers
-- No nonce tracker field
nonconsuming choice Lock_Musd_For_ : ContractId Lock
with
user : Party
musdCid : ContractId Musd
targetDomain : Text
targetAddr : Text
nonce : Int -- user-supplied, not validated
controller user
do
...
create Lock with
...
nonce = nonce -- no uniqueness check
```

RECOMMENDATION

Track nonces in the Service and enforce sequential assignment:

```
template Service
with
```

```
...
lastNonce : Int
choice Lock_Musd_For_[redacted] : (ContractId [redacted]Service, ContractId [redacted]Lock)
with ...
controller user
do
let newNonce = lastNonce + 1
...
newService <- create this with lastNonce = newNonce
lock <- create [redacted]Lock with nonce = newNonce; ...
return (newService, lock)
```

6.2.27 FulfillRedemption Destroys Tokens Without Settlement Proof

LOW FIXED

DESCRIPTION

The MUSD_FulfillRedemption choice allows the provider to complete a redemption request. When ex ised, it archives the MUSD_RedemptionRequest contract (which embeds the MUSD token data). The choice body is simply "return ()", meaning it archives without any proof that the off-chain settlement (wire transfer, IBAN payment, etc.) was completed.

The provider can call FulfillRedemption before actually sending the fiat payment. The user's mUSD is destroyed, but they may never receive the corresponding fiat. There is no on-chain reference to a settlement confirmation, transaction hash, or settlement timestamp.

CODE LOCATION

File: MintedMUSD.daml

MintedMUSD.daml

```
template MUSD_RedemptionRequest
with
usd : MUSD
requester : Party
instructions : Text
where
signatory usd.provider, requester
choice MUSD_FulfillRedemption : ()
controller usd.provider
do return () -- token is archived (destroyed) with no settlement proof
```

RECOMMENDATION

Require settlement evidence when fulfilling a redemption. Create a receipt or require a settlement reference:

```
choice MUSD_FulfillRedemption : ContractId RedemptionReceipt
with
settlementRef : Text
settledAt : Time
controller usd.provider
do
assertMsg "SETTLEMENT_REF_REQUIRED" (T.length settlementRef > 0)
create RedemptionReceipt with
redeemer = requester
amount = usd.amount
settlementRef
settledAt
```

6.2.28 Duplicate Party in Governance List

LOW FIXED

DESCRIPTION

The UpgradeProposal template accepts a governance list as a parameter but does not check for duplicate parties. If the same party appears multiple times, that party effectively has extra voting weight in the approval process.

For example, if governance = [alice, bob, alice] and approvalThreshold = 2, alice alone can approve the proposal because she counts as 2 distinct governors.

The ensure clause checks that approvalThreshold <= length governance, but length counts duplicates. If governance has 5 entries but only 3 unique parties, the threshold could be 5 but only 3 actual participants exist.

CODE LOCATION

File: Upgrade.daml

Upgrade.daml

```
template UpgradeProposal
with
operator : Party
governance : [Party] -- no uniqueness check
approvalThreshold : Int
approvals : Set.Set Party
...
where
signatory operator
observer governance
ensure approvalThreshold > 0
```

```
  && approvalThreshold <= length governance -- length includes duplicates
  && T.length sourceVersion > 0
  && T.length targetVersion > 0
  && T.length migrationScriptHash == 64
  && migrationWindowDays > 0
  && rollbackWindowDays >= 0
```

RECOMMENDATION

Add a uniqueness check in the ensure clause:

```
ensure ...
  && length governance == length (DA.List.unique governance)
```

6.2.29 BatchSize Not Enforced in MigrationTicket

LOW FIXED

DESCRIPTION

The MigrationTicket template has a batchSize field that is validated in the ensure clause (batchSize > 0 && batchSize <= 100). However, this field is never used during MigrationTicket_Execute. The execution processes all contractsToMigrate regardless of batchSize.

The batchSize was presumably intended to limit the number of contracts migrated in a single transaction to prevent excessive gas/resource usage. Since it is not enforced, a ticket with 1000 contracts and batchSize = 10 will attempt to migrate all 1000 in one transaction.

CODE LOCATION

File: Upgrade.daml

Upgrade.daml

```
template MigrationTicket
with
  operator : Party
  holder : Party
  upgradeKey : (Party, Text, Text)
  contractsToMigrate : [Text]
  requestedAt : Time
  batchSize : Int -- validated in ensure, never used in Execute
  direction : MigrationDirection
where
  signatory operator, holder
  ensure length contractsToMigrate > 0
```

```
&& batchSize > 0
&& batchSize <= 100
```

RECOMMENDATION

Enforce batchSize in MigrationTicket_Execute:

Or process only batchSize contracts at a time and create a new ticket for the remainder.

```
assertMsg "BATCH_TOO_LARGE" (length contractsToMigrate <= batchSize)
```

6.2.30 Zero or Negative Count in RecordMigration

LOW FIXED

DESCRIPTION

The UpgradeRegistry_RecordMigration choice accepts a count parameter that represents the number of contracts migrated. There is no validation that count is positive. A zero or negative count would corrupt the totalMigrated or totalRolledBack counters.

A negative count would decrement the migration counter, potentially allowing previously recorded migrations to be "undone" in the accounting. A zero count is a no-op that wastes resources.

CODE LOCATION

File: Upgrade.daml

Upgrade.daml

```
choice UpgradeRegistry_RecordMigration : ContractId UpgradeRegistry
with
count : Int
direction : MigrationDirection
controller operator
do
now <- getTime
case direction of
Forward -> do
assertMsg "MIGRATION_WINDOW_CLOSED" (now <= migrationDeadline)
create this with totalMigrated = totalMigrated + count -- no check: count > 0
Backward -> do
assertMsg "ROLLBACK_WINDOW_CLOSED" (now <= rollbackDeadline)
create this with totalRolledBack = totalRolledBack + count -- no check: count > 0
```

RECOMMENDATION

Add a positive count check:

```
assertMsg "COUNT_MUST_BE_POSITIVE" (count > 0)
```

6.2.31 No Global Supply Cap on mUSD Minting Through Borrows

LOW ACKNOWLEDGED

DESCRIPTION

The Lending_Borrow choice mints new CantonMUSD tokens when a user borrows against collateral. The totalBorrows field is incremented but never checked against any ceiling. There is no maxTotalBorrows or supply cap enforced in the CantonLendingService template.

If market conditions allow excessive borrowing (for example, during a collateral price spike), the total mUSD supply can grow without bound. On the side, the BorrowModule has a supply cap. The Canton side has no equivalent protection, creating an asymmetry that could lead to unbacked mUSD issuance and depeg.

CODE LOCATION

File: CantonLending.daml

CantonLending.daml

```
template CantonLendingService
with
operator : Party
configs : [CollateralConfig]
totalBorrows : Money -- tracked but never checked against a cap
interestRateBps : Bps
reserveFactorBps : Bps
protocolReserves : Money
minBorrow : Money
closeFactorBps : Bps
paused : Bool
mpaHash : Text
mpaUri : Text
observers : [Party]
newService <- create this with
totalBorrows = totalBorrows + borrowAmount
return (newService, debtCid, musdCid)
```

RECOMMENDATION

Add a maxTotalBorrows field to CantonLendingService and enforce it in Lending_Borrow:

```
-- In template fields:
maxTotalBorrows : Money
```

```
-- In Lending_Borrow:
assertMsg "GLOBAL_BORROW_CAP_EXCEEDED" (totalBorrows + borrowAmount <=
maxTotalBorrows)
```

6.2.32 Cancel Archives Without Recording State

LOW FIXED

DESCRIPTION

The UpgradeProposal_Cancel choice allows the operator to cancel a proposal. The choice body is "return ()", which archives the contract without creating any record. There is no cancellation log, no event emitted, and no state change recorded. The proposal simply disappears from the ledger.

This makes it impossible to audit why a proposal was cancelled or even prove that a proposal existed and was cancelled. For a governance system that requires compliance and auditability, silent removal of proposals is a gap.

Additionally, the activeProposalCount (if it were functional) would not be decremented, since the cancel choice does not interact with GovernanceConfig.

CODE LOCATION

File: Upgrade.daml

Upgrade.daml

```
choice UpgradeProposal_Cancel : ()
with
reason : Text
controller operator
do
assertMsg "CANNOT_CANCEL_ACTIVE" (status == Proposed || status == Approved)
assertMsg "REASON_REQUIRED" (T.length reason > 0)
return () -- archives with no record
```

RECOMMENDATION

Create a cancellation log:

```
choice UpgradeProposal_Cancel : ContractId UpgradeCancellationLog
with
reason : Text
controller operator
do
assertMsg "CANNOT_CANCEL_ACTIVE" (status == Proposed || status == Approved)
assertMsg "REASON_REQUIRED" (T.length reason > 0)
now <- getTime
```

```
create UpgradeCancellationLog with
operator
proposalKey = (operator, sourceVersion, targetVersion)
cancelledBy = operator
reason
cancelledAt = now
```

6.2.33 UpdateObservers Bypasses UserPrivacySettings

LOW FIXED

DESCRIPTION

Multiple templates have direct UpdateObservers or SMUSD_UpdateObservers choices that allow the owner to set arbitrary observers without going through the UserPrivacySettings module. This bypasses the privacy controls, audit trail, and self-observation prevention built into the privacysettings system.

For example, CantonSMUSD has SMUSD_UpdateObservers that accepts a newObservers list directly from the user. The user can set any parties as observers, including themselves or the operator, which the privacy settings module explicitly prevents.

This means the privacy settings system can be circumvented entirely by calling the template-level observer update choices instead of going through the controlled Privacy_AddObserver and Privacy_RemoveObserver paths.

CODE LOCATION

File: CantonSMUSD.daml - SMUSD_UpdateObservers:

CantonSMUSD.daml

```
choice SMUSD_UpdateObservers : ContractId CantonSMUSD
with
newObservers : [Party]
controller owner
do create this with privacyObservers = newObservers -- no validation
```

RECOMMENDATION

Remove or deprecate direct UpdateObservers choices. Route all observer updates through UserPrivacySettings:

```
choice SMUSD_UpdateObservers : ContractId CantonSMUSD
controller owner
do
```

```
newObs <- lookupUserObservers issuer owner
create this with privacyObservers = newObs
```

6.2.34 AdjustLeverage borrowAmount Creates Phantom Debt

LOW ACKNOWLEDGED

DESCRIPTION

The AdjustLeverage choice adds borrowAmount to principalDebt at the start of the leverage loop, but borrowAmount itself does not trigger a mUSD mint at the top level. The variable borrowAmount is combined with the existing debt to compute finalDebt before the loop, and the loop then borrows additional amounts based on the collateral ratio.

If borrowAmount > 0 but the leverage loop runs 0 iterations (loops = 0), the vault's principalDebt increases by borrowAmount without any corresponding mUSD being minted. The user takes on debt without receiving tokens.

Even with loops > 0, the initial borrowAmount inflates the debt calculation used for the health check without that specific amount being minted through the supply-tracked SupplyService_VaultMint path.

CODE LOCATION

File: Minted/Protocol/V3.daml

Minted/Protocol/V3.daml

```
choice AdjustLeverage : (ContractId Vault, ContractId MUSDSupplyService)
with
depositAmount : Money
borrowAmount : Money
poolCid : ContractId LiquidityPool
oracleCid : ContractId PriceOracle
supplyServiceCid : ContractId MUSDSupplyService
...
loops : Int
controller owner
do
assertMsg "MAX_LOOPS_10" (loops <= 10 && loops >= 0)
let newCollateral = collateralAmount + depositAmount
...
(finalCol, finalDebt, _, finalSvcCid) <- foldlA loopFn
(newCollateral, principalDebt + borrowAmount, poolCid, supplyServiceCid) --
borrowAmount added to debt
[1..loops]
-- If loops = 0, foldlA does nothing, finalDebt = principalDebt + borrowAmount
-- but no mUSD was minted for borrowAmount
```

RECOMMENDATION

Either mint mUSD for borrowAmount before the loop starts, or remove borrowAmount from the initial debt accumulator and let the loop handle all borrowing:

```
-- Option: Start loop with existing debt only
(finalCol, finalDebt, _, finalSvcCid) <- foldlA loopFn
(newCollateral, principalDebt, poolCid, supplyServiceCid)
[1..loops]
```

6.2.35 Stale Interest in Vault_WithdrawCollateral

LOW FIXED

DESCRIPTION

The Vault_WithdrawCollateral choice performs a health check using principalDebt + accruedInterest as the total debt. However, it does not accrue interest to the current timestamp before computing the health factor. The accruedInterest field reflects the last time interest was accrued (via Vault_Repay or Liquidate), not the current time.

If significant time has passed since the last interest accrual, the actual debt is higher than what the health check uses. A user could withdraw collateral that would leave the vault und ollateralized if current interest were factored in.

CODE LOCATION

File: Minted/Protocol/V3.daml

Minted/Protocol/V3.daml

```
choice Vault_WithdrawCollateral : ContractId Vault
with
withdrawAmount : Money
oracleCid : ContractId PriceOracle
controller owner
do
assertMsg "INSUFFICIENT_COLLATERAL" (collateralAmount >= withdrawAmount)
let newCollateral = collateralAmount - withdrawAmount
price <- ex ise oracleCid Oracle_GetPrice with
requester = owner
maxStaleness = hours 1
let newValue = newCollateral * price
let totalDebt = principalDebt + accruedInterest -- stale interest, not current
assertMsg "UNHEALTHY_AFTER_WITHDRAW"
(totalDebt == 0.0 || newValue >= totalDebt * config.liquidationThreshold)
create this with collateralAmount = newCollateral
```

RECOMMENDATION

Calculate current interest before the health check:

```
now <- getTime
let elapsed = convertRelTimeToMicroseconds (subTime now lastInterestUpdate)
let secondsElapsed = intToNumeric (elapsed / 1000000)
let yearSeconds = 31536000.0 : Money
let newInterest = principalDebt * intToNumeric config.interestRateBps *
secondsElapsed / (10000.0 * yearSeconds)
let totalDebt = principalDebt + accruedInterest + newInterest
```

6.2.36 AdjustLeverage Phantom Collateral

LOW FIXED

DESCRIPTION

The AdjustLeverage choice adds depositAmount to collateralAmount at the start of the operation, but the deposit is not verified against any actual token transfer. The user specifies depositAmount as a number, and the vault's collateral is credited without consuming a token contract.

A user can call AdjustLeverage with a large depositAmount without actually depositing any collateral. The vault records the inflated collateral, passes the health check, and executes the leverage loop. The borrowed mUSD through the loop is real (minted via SupplyService_VaultMint), but the collateral backing it is phantom.

CODE LOCATION

File: Minted/Protocol/V3.daml

Minted/Protocol/V3.daml

```
choice AdjustLeverage : (ContractId Vault, ContractId MUSDSupplyService)
with
depositAmount : Money
borrowAmount : Money
...
controller owner
do
...
let newCollateral = collateralAmount + depositAmount -- no token consumed
price <- ex[redacted]ise oracleCid Oracle_GetPrice with ...
...
-- Loop borrows and swaps based on inflated collateral value
(finalCol, finalDebt, _, finalSvcCid) <- foldlA loopFn
(newCollateral, principalDebt + borrowAmount, poolCid, supplyServiceCid)
[1..loops]
```

```
...
vaultCid <- create this with
collateralAmount = finalCol -- records phantom collateral
```

RECOMMENDATION

Require a collateral token contract ID and consume it during the deposit:

```
choice AdjustLeverage : (ContractId Vault, ContractId MUSDSupplyService)
with
collateralCid : ContractId CollateralToken -- require actual token
...
controller owner
do
collateral <- fetch collateralCid
assertMsg "OWNER_MISMATCH" (collateral.owner == owner)
archive collateralCid -- consume the deposit
let newCollateral = collateralAmount + collateral.amount
```

6.2.37 Burn_Musd Supply Underflow Silently Floors to Zero

LOW FIXED

DESCRIPTION

The Burn_Musd choice decrements currentSupply when burning mUSD. The code uses a conditional that floors the result to zero if the burn amount exceeds the tracked supply:

This means if currentSupply is out of sync (due to bugs in other supply-tracking paths), a burn operation will silently set the supply to 0 instead of failing. After this, the supply counter no longer reflects reality. Future mints will be allowed up to the full supplyCap even though tokens are in circulation.

The correct behavior is to fail if the burn would underflow, because an underflow indicates a supply tracking bug that needs to be investigated.

CODE LOCATION

File: MUSD_Protocol.daml

MUSD_Protocol.daml

```
let newSupply = if currentSupply > burnAmount then currentSupply - burnAmount else
0.0
```

MUSD_Protocol.daml

```
choice Burn_Musd : (ContractId MintingService, ContractId Usdc)
with
user : Party
musdCid : ContractId Musd
usdcReserveCid : ContractId Usdc
controller user, operator
do
musd <- fetch musdCid
...
archive musdCid
let burnAmount = musd.amount
let newSupply = if currentSupply > burnAmount then currentSupply - burnAmount else
0.0
newService <- create this with currentSupply = newSupply
```

RECOMMENDATION

Fail on underflow instead of silently flooring:

```
assertMsg "SUPPLY_UNDERFLOW" (currentSupply >= burnAmount)
let newSupply = currentSupply - burnAmount
newService <- create this with currentSupply = newSupply
```

6.2.38 Non-Sequential Nonce in

LOW FIXED

DESCRIPTION

The Service in V3.daml tracks lastNonce and uses it for operations. The _Complete In choice validates that the nonce matches the expected pattern, but the nonce assignment in _AssignNonce simply increments lastNonce by 1 without checking for gaps.

When _Complete In processes an , it updates lastNonce to the 's nonce value. If s arrive out of order, a lower nonce could be processed after a higher one, allowing the lastNonce to move backwards. This could enable replay of previously processed s.

Additionally, there is no explicit check that the incoming nonce is exactly lastNonce + 1, only that the is valid. Nonce gaps are silently accepted.

CODE LOCATION

File: Minted/Protocol/V3.daml

Minted/Protocol/V3.daml

```
choice [redacted]_Complete[redacted]In : (ContractId [redacted]Service, ContractId MintedMUSD)
with
recipient : Party
amount : Money
[redacted]Cid : ContractId [redacted]
controller operator
do
...
[redacted] <- fetch [redacted]Cid
...
archive [redacted]Cid
musdCid <- create MintedMUSD with ...
newService <- create this with
total[redacted]dIn = total[redacted]dIn + amount
lastNonce = [redacted].payload.nonce -- accepts any nonce, not sequential
return (newService, musdCid)
choice [redacted]_AssignNonce : (ContractId [redacted]Service, Int)
controller operator
do
let newNonce = lastNonce + 1
newService <- create this with lastNonce = newNonce
return (newService, newNonce)
```

RECOMMENDATION

Enforce sequential nonce processing in [redacted]_Complete[redacted]In:

```
assertMsg "NONCE_NOT_SEQUENTIAL" ([redacted].payload.nonce == lastNonce + 1)
```

6.2.39 Compliance Test Tests Wrong Failure Path

LOW FIXED

DESCRIPTION

The compliance negative test suite for ValidateMint tests that a blacklisted user cannot mint.However, the test asserts on the wrong error message. The test expects the transaction to fail because the user is blacklisted, but the actual failure occurs earlier in the call chain for a different reason.

In Compliance.daml at Line 117, ValidateMint checks the blacklist. However, the test at NegativeTests.daml Lines 148-174 sets up the scenario in a way that the transaction fails before the blacklist check is reached. The test passes (the transaction does abort), but it passes for the wrong reason. This means the blacklist check itself is not actually being tested.

CODE LOCATION

File: NegativeTests.daml (Lines 148-174) - the test scenario fails before the blacklist check is ex ised.

NegativeTests.daml (Lines 148-174)

```
nonconsuming choice ValidateMint : ()
with
minter : Party
amount : Decimal
requester : Party
controller requester
do
assertMsg "MINTER_BLACKLISTED" (not (Set.member minter blacklisted))
assertMsg "MINT_AMOUNT_POSITIVE" (amount > 0.0)
```

RECOMMENDATION

Restructure the negative test so that the only reason the transaction can fail is the blacklist check. Ensure the user is properly set up with all other preconditions met, then verify the specific error message "MINTER_BLACKLISTED" is the cause of failure.

6.2.40 No Contract Key on ComplianceRegistry

LOW ACKNOWLEDGED

DESCRIPTION

The ComplianceRegistry template has no contract key defined. Without a key, other templates cannot use fetchByKey or lookupByKey to find the active registry. They must instead be passed the contract ID as an argument to every choice that needs compliance checks.

This has two consequences. First, it makes it harder to integrate compliance checks into other templates since every choice needs an extra complianceRegistryCid parameter. Second, a caller could pass a stale or attacker-controlled contract ID instead of the current active registry.

With a contract key, any template can look up the unique active registry by the regulator party, preventing stale references and simplifying integration.

CODE LOCATION

File: Compliance.daml

Compliance.daml

```
template ComplianceRegistry
with
regulator : Party
operator : Party
blacklisted : Set.Set Party
frozen : Set.Set Party
```

```
lastUpdated : Time
where
signatory regulator
observer operator
ensure regulator /= operator
-- No contract key defined
```

RECOMMENDATION

Add a contract key to ComplianceRegistry keyed on the regulator:

This allows other templates to use fetchByKey @ComplianceRegistry regulator to get the current active registry without needing a contract ID parameter.

```
key regulator : Party
maintainer key
```

6.2.41 No Expiry Check on Proposal Execution

LOW FIXED

DESCRIPTION

The Proposal_Execute choice checks that the proposal status is Approved and that the timelock has ended, but it does not check wh er the proposal has expired. The MultiSigProposal template has an expiresAt field that is checked during Proposal_Approve, but Proposal_Execute ignores it.

A proposal that was approved before its expiry but not executed promptly can be executed long after it has expired. This is a problem because conditions may have changed since the proposal was created. For example, a proposal to add a specific minter may no longer be appropriate months later.

CODE LOCATION

File: Governance.daml

Governance.daml

```
choice Proposal_Execute : ContractId GovernanceActionLog
with
executor : Party
controller executor
do
assertMsg "NOT_GOVERNOR" (executor `elem` governors || executor == operator)
assertMsg "NOT_APPROVED" (status == Approved)
now <- getTime
case timelockEndsAt of
None -> abort "TIMELOCK_NOT_SET"
```

```
Some timelockEnd -> do
assertMsg "TIMELOCK_NOT_ENDED" (now >= timelockEnd)
-- No check: now < expiresAt
create GovernanceActionLog with ...
```

RECOMMENDATION

Add an expiry check in Proposal_Execute:

```
assertMsg "PROPOSAL_EXPIRED" (now < expiresAt)
```

6.2.42 Guardian Pause Griefing Loop

LOW ACKNOWLEDGED

DESCRIPTION

Any single guardian can trigger EmergencyPause_Trigger to pause the entire protocol. Unpausing requires a full governance proposal through Proposal_Execute, which needs M-of-N approval and a 24-hour timelock. After unpausing, the same guardian can immediately pause again.

This creates an asymmetry: pausing takes one transaction from one party, while unpausing takes multiple approvals plus a 24-hour delay. A malicious or compromised guardian can keep the protocol paused indefinitely by re-pausing immediately after each unpause, at minimal cost. Each pause/unpause cycle consumes days of governance overhead while the protocol remains non-functional.

CODE LOCATION

File: Governance.daml

Governance.daml

```
choice EmergencyPause_Trigger : ContractId EmergencyPauseState
with
guardian : Party
reason : Text
controller guardian
do
assertMsg "NOT_GUARDIAN" (guardian `elem` guardians)
assertMsg "ALREADY_PAUSED" (not isPaused)
assertMsg "REASON_REQUIRED" (T.length reason > 0)
now <- getTime
create this with
isPaused = True
pausedAt = Some now
pausedBy = Some guardian
pauseReason = Some reason
lastUpdated = now
```

```
choice EmergencyPause_Resume : ContractId EmergencyPauseState
with
governanceProofCid : ContractId GovernanceActionLog
controller operator
do
proof <- fetch governanceProofCid
assertMsg "WRONG_ACTION_TYPE" (proof.actionType == EmergencyPause)
assertMsg "NOT_PAUSED" isPaused
```

RECOMMENDATION

Add a cooldown period after unpausing during which the same guardian cannot re-pause. Alternatively, require multi-guardian agreement for pause, or allow the operator to remove a griefing guardian without a full governance proposal.

```
-- Add to EmergencyPauseState:
lastUnpausedAt : Optional Time
-- In EmergencyPause_Trigger, enforce cooldown:
case lastUnpausedAt of
Some t -> assertMsg "PAUSE_COOLDOWN" (subTime now t >= hours 48)
None -> return ()
```

6.2.43 activeProposalCount Is Dead Code

LOW FIXED

DESCRIPTION

The GovernanceConfig template has an activeProposalCount field and a maxActiveProposals field. The maxActiveProposals is validated in the ensure clause (maxActiveProposals > 0), but activeProposalCount is never incremented when a proposal is created and never decremented when one is cancelled or executed.

No choice in the codebase modifies activeProposalCount. It remains at its initial value forever. The anti-spam protection it was meant to provide does not work. Any party can create an unlimited number of proposals.

CODE LOCATION

File: Governance.daml

Governance.daml

```
template GovernanceConfig
with
operator : Party
governors : [(Party, GovernorRole)]
```

```
standardThreshold : Int
elevatedThreshold : Int
timelockDuration : RelTime
proposalExpiry : RelTime
maxActiveProposals : Int -- never checked against activeProposalCount
activeProposalCount : Int -- never incremented or decremented
observers : [Party]
```

RECOMMENDATION

Either implement the proposal counting logic or remove the dead fields. To implement:

```
-- When creating a proposal, increment and check:
assertMsg "TOO_MANY_ACTIVE_PROPOSALS" (activeProposalCount < maxActiveProposals)
-- Then: create this with activeProposalCount = activeProposalCount + 1
-- When a proposal is cancelled/executed/expired, decrement:
-- create this with activeProposalCount = activeProposalCount - 1
```

6.2.44 MinterRegistry_UseMintQuota Missing Amount Greater Than Zero Check

LOW FIXED

DESCRIPTION

The MinterRegistry_UseMintQuota choice allows a minter to use their minting quota. It checks that the amount does not exceed the quota, but it does not check that the amount is greater than zero. A minter can call this choice with amount = 0 and it will succeed, incrementing totalMinted by 0. Whil is is not directly exploitable, it allows quota usage tracking to be polluted and is inconsistent with the positive-amount checks in other minting choices.

More importantly, a negative amount could potentially be passed (DAML Decimal allows negatives) which would increase the quota instead of decreasing it, since the update is q - amount. If amount is negative, the subtraction becomes addition.

CODE LOCATION

File: Governance.daml

Governance.daml

```
choice MinterRegistry_UseMintQuota : ContractId MinterRegistry
with
minter : Party
amount : Decimal
controller minter
do
```

```
assertMsg "NOT_AUTHORIZED_MINTER" (minter `elem` map fst minters)
let currentQuota = case lookup minter minters of
None -> 0.0
Some q -> q
assertMsg "EXCEEDS_MINT_QUOTA" (amount <= currentQuota)
-- No check: amount > 0
let updateQuota (p, q) = if p == minter then (p, q - amount) else (p, q)
create this with
minters = map updateQuota minters
totalMinted = totalMinted + amount
```

RECOMMENDATION

Add a positive amount check:

```
assertMsg "AMOUNT_MUST_BE_POSITIVE" (amount > 0.0)
```

6.2.45 Unbounded Observers in Asset_Merge

LOW FIXED

DESCRIPTION

The Asset_Merge choice concatenates the observer lists of both assets being merged. Over many merge operations, the observers list grows without bound. Each merge appends the other asset's observers using dedup (sort (observers ++ other.observers)), but dedup only removes exact duplicates. If each merged asset has unique observers, the list keeps growing.

A large observer list increases transaction size, slows down ledger operations, and can eventually hit DAML's transaction size limits. It also creates a privacy issue: observers from old asset positions accumulate and can see the merged asset even though they may no longer have a legitimate reason to observe it.

CODE LOCATION

File: InstitutionalAssetV4.daml

InstitutionalAssetV4.daml

```
choice Asset_Merge : ContractId Asset
with otherId : ContractId Asset
controller owner
do
assertMsg "Base asset is locked" (isNone lock)
other <- fetch otherId
assertMsg "Merge Fail: Instrument mismatch" (other.instrumentId == instrumentId)
assertMsg "Merge Fail: Issuer mismatch" (other.issuer == issuer)
```

```
assertMsg "Merge Fail: Depository mismatch" (other.depository == depository)
assertMsg "Merge Fail: Owner mismatch" (other.owner == owner)
assertMsg "Merge Fail: Other asset is locked" (isNone other.lock)
archive oth[redacted]id
create this with
amount = amount + other.amount
observers = dedup (sort (observers ++ other.observers))
```

RECOMMENDATION

Cap the observer list size or allow the owner to reset observers during merge:

Alternatively, use only the base asset's observers and drop the merged asset's observers.

```
let mergedObservers = dedup (sort (observers ++ other.observers))
assertMsg "TOO_MANY_OBSERVERS" (length mergedObservers <= 50)
create this with
amount = amount + other.amount
observers = mergedObservers
```

7. Executive Summary

Two independent softstack experts performed an unbiased and isolated audit of the smart contract provided by the Minted team. The main objective of the audit was to verify the security and functionality claims of the smart contract. The audit process involved a thorough manual code review and automated security testing.

Overall, the audit identified a total of 45 issues, classified as follows:

- No critical issues were found
- 5 high severity issues were found
- 14 medium severity issues were found
- 26 low severity issues were found
- No informational issues were found

The Minted team has successfully addressed all identified issues from the audit. All vulnerabilities have been mitigated based on the recommendations provided in the report. A follow-up review confirms that the fixes have been implemented effectively, ensuring the security and functionality of the smart contract. 11 findings were acknowledged by the team.

The audited codebase comprises 4,993 normalized source lines of code across 16 files.

8. About the Auditor

Established in 2017 under the name Chainsulting, and rebranded as softstack GmbH in 2023, softstack has been a trusted name in Web3 Security space. Within the rapidly growing Web3 industry, softstack provides a comprehensive range of offerings that include software development, cybersecurity, and consulting services. Softstack's competency extends across the security landscape of prominent blockchains like Solana, Tezos, TON, and Polygon. The company is widely recognized for conducting thorough code audits aimed at mitigating risk and promoting transparency.

The firm's proficiency lies particularly in assessing and fortifying smart contracts of leading DeFi projects, a testament to their commitment to maintaining the integrity of these innovative financial platforms. To date, softstack plays a crucial role in safeguarding over $100 billion worth of user funds in various DeFi protocols.

Underpinned by a team of industry veterans possessing robust technical knowledge in the Web3 domain, softstack offers industry-leading smart contract audit services. Committed to evolving with their clients' ever-changing business needs, softstack's approach is as dynamic and innovative as the industry it serves.

Check our website for further information: https://softstack.io

9. Glossary

Term	Definition
CDP	Collateralized Debt Position
CVSS	Common Vulnerability Scoring System
Canton	Privacy-enabled distributed ledger based on DAML
DAML	Digital Asset Modeling Language
DeFi	Decentralized Finance
RFC	Request for Comments
MPA	Master Participation Agreement
mUSD	Minted USD stablecoin
smUSD	Staked mUSD yield token