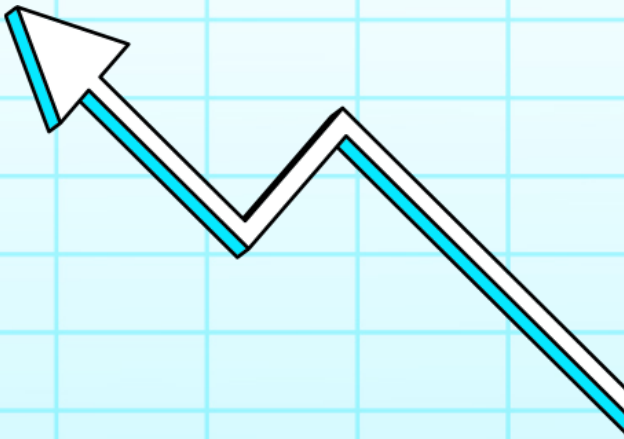
shieldify



Shiny

SECURITY REVIEW

Date: 14 January 2026

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Shiny

Shiny is an all-in-one experience for collectors, making luxury assets as easy to access as digital ones through verifiable mystery packs. Launching with Pokémon, the world's most valuable IP, as its wedge into a broader collectibles market. With first-party supply, guaranteed liquidity, and a founder-led network, the company is built to scale fast and capture the future of collecting, all on shiny.com.

Learn more about Shiny's concept and the technicalities behind it: here

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 6 days with a total of 96 hours dedicated to the audit by the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying two High, two Medium and five Low severity findings. They're mainly related to the lack of an on-chain oracle, blacklisted operators can not be revoked from being an operator and other centralization and admin trust assumptions.

The Shiny team has done a great job with their test suite and provided support and responses to all of the questions that the Shieldify researchers had.

5.1 Protocol Summary

Project Name	**Shiny**
Repository	ShinyUrban
Type of Project	RWA, Treasury, EIP-712
Security Review Timeline	6 days
Review Commit Hash	f49b5db73b297552666783ed587cf0818ef86b75
Fixes Review Commit Hash	55b012146abda6786dc23e53732c09a653ddad34

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
Pawn.sol	190
Treasure.sol	149
sRWA.sol	215
Total	554

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **High** issues: **2**
- **Medium** issues: **2**
- **Low** issues: **5**
- **Info** issues: **4**

ID	Title	Severity	Status
[H-01]	Protocol Insolvency Risk Due to Lack of On-Chain Oracle	High	Acknowledged
[H-02]	Blacklisted Operators Can Not Be Revoked from Being an Operator	High	Fixed
[M-01]	Admin Can Seize Pawn Collateral via `emergencyWithdrawNFT()`	Medium	Fixed
[M-02]	Liquidation Can Be Blocked By Pausing or Blacklisting the NFT Contract, Permanently Trapping Expired Loans	Medium	Fixed
[L-01]	Pawn Duration Not Bound to Signed Quote Allows 730-day Term Tampering	Low	Fixed
[L-02]	Hardcoded 6-decimal Stablecoin Assumptions Brick Protocol on 18-decimal Deployments	Low	Fixed
[L-03]	`permit()` Can Approve a Blacklisted Spender (Missing Blacklist Check) But `approve()` Does Not Allow	Low	Fixed
[L-04]	Admin Can Drain Treasury Reserves via `withdrawUSDC()`	Low	Fixed
[L-05]	Unbounded Fee Parameters Allow Abusive Fees and Redemption DoS	Low	Fixed
[I-01]	Burned UUIDs Can Be Re-Minted	Info	Acknowledged
[I-02]	Redundant `usedSignatures` Tracking Alongside Nonce-Based Replay Protection	Info	Fixed
[I-03]	Configuration Mismatch Risk Across `backendSigner` and `treasury` Addresses	Info	Acknowledged
[I-04]	PawnShop `MAX_LOAN_AMOUNT` Can Exceed Treasury	Info	Fixed

7. Findings

[H-01] Protocol Insolvency Risk Due to Lack of On-Chain Oracle

Severity

High Risk

Description

The protocol's solvency relies on an off-chain `backendSigner` to determine the loan principal (`offerAmount`) without any on-chain verification of collateral value or an on-chain "unhealthy position" liquidation path. As a result, the protocol can become undercollateralized during the loan term (or even at origination if the signed offer is incorrect), while liquidation is only possible after a time-based deadline.

In volatile markets, this design can systematically create bad debt and drain protocol reserves.

Technical Details

1) `offerAmount` is trusted without on-chain valuation checks

`Pawn.pawn()` accepts `offerAmount` from a signature and stores it as the pawn principal. There is no on-chain mechanism to validate that the loan-to-value (LTV) is within safe limits at the time of origination. Protocol takes care of it off-chain.

File: Pawn.sol

```
function pawn(uint256 tokenId, uint8 durationDays, uint256 offerAmount,
    uint256 validUntil, bytes calldata pawnSignature, bytes calldata
    treasurySignature) external nonReentrant whenNotPaused {
     // code
     (bytes32 digest, uint256 currentNonce) = _validatePawnSignature(
         tokenId,
         offerAmount,
         validUntil,
         pawnSignature
     );
     // code
     pawns[tokenId] = Pawn({
         borrower: msg.sender,
         amount: offerAmount,
         deadline: deadline,
         feeBasisPoints: fee,
         active: true
     });
     // code
}
```

2) Liquidation is strictly time-based (no health-factor liquidation)

`Pawn.liquidate()` can only be executed after the pawn deadline. Even if the collateral value collapses during the term, the protocol cannot liquidate early to preserve solvency.

```
function liquidate(uint256 tokenId) external nonReentrant whenNotPaused
    onlyRole(MANAGER_ROLE) {
     Pawn memory pawning = pawns[tokenId];
     if (!pawning.active) revert TokenNotPawned();

     // Liquidation is only possible after the time-based deadline
     if (block.timestamp < pawning.deadline) revert DeadlineNotPassed();

     pawns[tokenId].active = false;

     IERC721Burnable(address(nftToken)).burn(tokenId);
     emit TokenLiquidated(tokenId, msg.sender);
}
```

Location of Affected Code

File: Pawn.sol

Impact

If the collateral value drops below the loan principal during the loan term, rational borrowers are incentivized to default. The protocol then realizes a loss equal to:

`Bad Debt = (Principal + Accrued Fees) - Collateral Value at recovery`

Practical consequences:

- **Protocol insolvency / reserve depletion:** USDC can be paid out against collateral that later becomes insufficient.
- **Systemic bad debt:** The issue is not limited to one position; it is a structural failure mode whenever the signed `offerAmount` exceeds safe LTV or market prices fall during the term.
- **Limited mitigation after the fact:** Because liquidation burns the token (claiming the underlying asset), the protocol is effectively forced to "take delivery" at an unfavourable valuation, crystallising losses.

Proof of Concept

1. An RWA token represents a physical gold bar worth 2,000 USDC.
2. The backend signs a loan offer for 1,800 USDC (90% LTV) due to error, stale pricing, or sudden market movement.
3. The next day, gold drops 20% and the collateral value becomes 1,600 USDC.
4. The protocol cannot liquidate because `block.timestamp < deadline`.
5. The borrower rationally defaults because repayment exceeds collateral value.
6. At `deadline`, the protocol liquidates and ends up recovering an asset worth ~1,600 USDC after having lent 1,800 USDC (excluding fees).
7. Net loss is ~200 USDC (plus any additional loss due to fees/realization costs).

Recommendations

A) Enforce origination LTV on-chain

Integrate an on-chain pricing source for the underlying collateral and enforce a maximum LTV at pawn creation.

```
uint256 collateralValue = oracle.getPrice(tokenId); // must define units
    + decimals
require(offerAmount <= (collateralValue * MAX_LTV_BPS) / 10_000, "LTV too
     high");
```

Minimum requirements for a robust Oracle integration:

- Staleness checks (e.g., revert if price is older than a configured threshold).
- Consistent units/decimals normalization.
- Clear fallback behavior when the oracle is unavailable.

B) Add health-factor-based liquidation during the term

Implement a liquidation condition that triggers when the position becomes undercollateralized (e.g., collateral value falls below a liquidation threshold). This can allow liquidation before the `deadline` and materially reduce bad debt.

C) If an oracle is not feasible, reduce trust in the backend signer

If on-chain pricing is not currently possible, consider compensating controls (still weaker than an oracle), such as:

- Very conservative LTV caps hard-coded on-chain per asset class.
- Shorter maximum loan durations.
- Emergency controls to halt new pawns when pricing is uncertain.

Team Response

Acknowledged.

[H-02] Blacklisted Operators Can Not Be Revoked from Being an Operator

Severity

High Risk

Description

NFTs can be stolen by the blacklisted operators. They can then burn/sell afterwards. Impact is high, as the victim NFT owner cannot revoke the approval from this balcaklisted operator because of this line below

Root cause : - `sRWA::setApprovalForAll(account, false)` reverts if account is blacklisted - `sRWA::approve()` can be called by blacklisted operators.

File: sRWA.sol#L280

```
  function setApprovalForAll(address operator, bool approved) public
     virtual override {
>>>   if (_isBlacklisted[operator]) revert Blacklisted();
      super.setApprovalForAll(operator, approved);
  }
```

In `sRWA.sol` :

1. A router exists, and a user who owns 10 NFTs sets it as his operator by calling `setApprovalForAll(router, true)` .
2. The admin decides to blacklist this operator address (router) by calling `blacklistContract(router)` (e.g., due to OFAC/rogue).
3. The user attempts to revoke the operator by calling `setApprovalForAll(router, false)` , but it reverts in line 301 below. (Issue 1)
4. The rogue (now blacklisted) operator calls `approve(rogue operator's new unblacklisted account, victim owner's tokenId)` .
5. Once approved, the attacker pulls those tokens via `transferFrom(victim owner, new unblacklisted account, victim owner's tokenId)` , and now the attacker owns them.
6. Check line 389 of ERC721 below. It allows operators to assign new approvals for each token id

Location of Affected Code

File: sRWA.sol

```
 function setApprovalForAll(address operator, bool approved) public
    virtual override {
     if (_isBlacklisted[operator]) revert Blacklisted();
     super.setApprovalForAll(operator, approved);
 }

 function approve(address to, uint256 tokenId) public virtual override {
     if (_isBlacklisted[to]) revert Blacklisted();
     super.approve(to, tokenId);
 }
```

File: contracts/token/ERC721/ERC721.sol

```
function _approve(address to, uint256 tokenId, address auth, bool
   emitEvent) internal virtual {
   // Avoid reading the owner unless necessary
   if (emitEvent || auth != address(0)) {
       address owner = _requireOwned(tokenId);

       // We do not use _isAuthorized because single-token approvals
          should not be able to call approve
>>>    if (auth != address(0) && owner != auth && !isApprovedForAll(owner
   , auth)) {
           revert ERC721InvalidApprover(auth);
       }

       if (emitEvent) {
           emit Approval(owner, to, tokenId);
       }
   }

   _tokenApprovals[tokenId] = to;
}
```

Impact

Users who previously granted `setApprovalForAll(router, true)` can have their NFTs stolen after the router is blacklisted, because: - The user cannot revoke the operator approval. - The blacklisted operator can still delegate approvals to an unblacklisted attacker, who can then transfer the NFTs.

Proof of Concept

Run it in `test/System.t.sol`

```
function
    test_POC_BlacklistedOperator_CannotBeRevoked_AndCanStealViaApprove()
    public {
     address router = makeAddr("router");
     address attacker = makeAddr("attacker");

     // Mint 9 more tokens so `user` owns 10 NFTs total (tokenIds 1..10)
     vm.startPrank(owner);
     rwa.mint(user, "UUID-124");
     rwa.mint(user, "UUID-125");
     rwa.mint(user, "UUID-126");
     rwa.mint(user, "UUID-127");
     rwa.mint(user, "UUID-128");
     rwa.mint(user, "UUID-129");
     rwa.mint(user, "UUID-130");
     rwa.mint(user, "UUID-131");
     rwa.mint(user, "UUID-132");
     vm.stopPrank();

     // 1) user makes router an operator
     vm.prank(user);
     rwa.setApprovalForAll(router, true);
     assertTrue(rwa.isApprovedForAll(user, router));

     // 2) admin blacklists the operator/router address
     vm.prank(owner);
     rwa.blacklistContract(router);
     assertTrue(rwa.isBlacklisted(router));

     // 3) user attempts to revoke operator approval, but it reverts
     vm.prank(user);
     vm.expectRevert(RWA.Blacklisted.selector);
     rwa.setApprovalForAll(router, false);

     // 4) blacklisted router approves an unblacklisted attacker for each
        tokenId
     for (uint256 tokenId = 1; tokenId <= 10; tokenId++) {
         vm.prank(router);
         rwa.approve(attacker, tokenId);
         assertEq(rwa.getApproved(tokenId), attacker);
     }

     // 5) attacker pulls the tokens and becomes the owner
     for (uint256 tokenId = 1; tokenId <= 10; tokenId++) {
         vm.prank(attacker);
         rwa.transferFrom(user, attacker, tokenId);
         assertEq(rwa.ownerOf(tokenId), attacker);
     }
}
```

Recommendation

1. Allow revocation even if the operator is blacklisted (only block when setting `approved == true`).
2. Block blacklisted callers from managing approvals (e.g., revert in `approve()` and `setApprovalForAll()` when `msg.sender` is blacklisted), so a blacklisted operator cannot delegate approvals to an unblacklisted attacker.

File: sRWA.sol

```
function approve(address to, uint256 tokenId) public virtual override {
+   if (_isBlacklisted[msg.sender]) revert Blacklisted();
    if (_isBlacklisted[to]) revert Blacklisted();
        super.approve(to, tokenId);
    }
}

function setApprovalForAll(address operator, bool approved) public
    virtual override {
-       if (_isBlacklisted[operator]) revert Blacklisted();
+       if (_isBlacklisted[operator] && approved == true) revert
    Blacklisted();
            super.setApprovalForAll(operator, approved);
        }
}
```

Team Response

Fixed.

[M-01] Admin Can Seize Pawn Collateral via `emergencyWithdrawNFT()`

Severity

Medium Risk

Description

The `pawn(...)` flow escrows a user's NFT inside `PawnShop` as collateral until the user repays via `redeem(...)` or the position is closed via `liquidate(...)` (context).

However, `PawnShop` also exposes an `emergencyWithdrawNFT(...)` function that lets `DEFAULT_ADMIN_ROLE` transfer any NFT held by the contract to an arbitrary address, with no restriction that the loan is inactive and no state cleanup (problem).

This means an admin can directly steal user collateral (even for active loans) and can also permanently break the loan's lifecycle because `redeem(...)` / `liquidate(...)` expect the contract to still own the NFT (impact).

Location of Affected Code

File: Pawn.sol

```
function emergencyWithdrawNFT(uint256 tokenId, address to) external
    nonReentrant onlyRole(DEFAULT_ADMIN_ROLE) {
     // @audit Can transfer active collateral to an arbitrary address
     if (to == address(0)) revert InvalidAddress();
     nftToken.safeTransferFrom(address(this), to, tokenId);
     emit EmergencyNFTWithdrawn(tokenId, to);
}
```

Impact

- **User collateral theft**: An admin can move escrowed NFTs to themselves (or any address), bypassing borrower repayment rules.
- **Protocol/loan state corruption**: Once the NFT is moved out, `redeem(...)` cannot return collateral and `liquidate(...)` cannot burn it, leaving the `pawns[tokenId]` state effectively unresolvable.

Proof of Concept

1. Alice calls `pawn(...)` and `PawnShop` escrows Alice's NFT.
2. Admin calls `emergencyWithdrawNFT(tokenId, admin)`.
3. Alice can no longer redeem her NFT, and the protocol cannot liquidate it on-chain because the NFT is no longer owned by `PawnShop`.

Recommendation

Consider applying the following changes: - **Restrict scope**: Only allow emergency withdrawal for NFTs that are not backing an active pawn: - Require `!pawns[tokenId].active`, or - Only allow withdrawal to `pawns[tokenId].borrower`. - **Add safeguards**: Use a timelock + multisig for `DEFAULT_ADMIN_ROLE`, and consider an on-chain guardian/emergency procedure that cannot seize active collateral. - **Maintain invariants**: If a forced withdrawal is ever allowed, update or clear the pawn state in a way that preserves a consistent resolution path.

Team Response

Fixed.

[M-02] Liquidation Can Be Blocked By Pausing or Blacklisting the NFT Contract, Permanently Trapping Expired Loans

Severity

Medium Risk

Description

When a pawn expires, `PawnShop.liquidate(...)` closes the position by burning the escrowed NFT collateral through the NFT's `burn(...)` function (context).

However, in this system, the NFT is `RWA`, whose burn path is gated by `whenNotPaused`, and whose transfer/burn internals revert if the caller (`auth`) is blacklisted. This means an admin action on `RWA` (pause or blacklist) can cause `PawnShop.liquidate(...)` to revert (problem).

This traps the loan in an unrecoverable state: the borrower cannot redeem after the `deadline`, and the manager cannot liquidate, so the collateral remains stuck in `PawnShop` indefinitely (impact).

Location of Affected Code

File: Pawn.sol#L262-L271

```
function liquidate(uint256 tokenId) external nonReentrant whenNotPaused
    onlyRole(MANAGER_ROLE) {
     Pawn memory pawning = pawns[tokenId];
     if (!pawning.active) revert TokenNotPawned();
     if (block.timestamp < pawning.deadline) revert DeadlineNotPassed();

     pawns[tokenId].active = false;

     // @audit Depends on external NFT burn rules
     IERC721Burnable(address(nftToken)).burn(tokenId);
}
```

File: sRWA.sol#L189-L197

```
function burn(uint256 tokenId) external onlyTokenOwner(tokenId)
    whenNotPaused {
     // code
     _burn(tokenId);
}
```

File: sRWA.sol#L270-L277

```
function _update(address to, uint256 tokenId, address auth) internal
    virtual override returns (address) {
     // code
     if (auth != address(0) && _isBlacklisted[auth]) revert Blacklisted();
     // code
}
```

Impact

- Expired loans may become impossible to close if `RWA` is paused or if `PawnShop` is blacklisted.
- Borrowers cannot redeem after `deadline`, so collateral can remain trapped without a clean resolution path.

Proof of Concept

1. User opens a pawn via `PawnShop.pawn(...)`.
2. Time passes beyond `deadline`.
3. Admin pauses `RWA` (or blacklists `PawnShop` in `RWA`).
4. Manager calls `PawnShop.liquidate(tokenId)`.
5. The call reverts when it reaches `RWA.burn(...)`, leaving the loan stuck.

Recommendation

- **Decouple liquidation from `RWA` pause/blacklist**:
  - Allow `PawnShop` to liquidate by transferring the NFT to a protocol-controlled address if burning is blocked, or
  - Give `PawnShop` a special role in `RWA` that can burn even when paused / not subject to blacklist, or
  - Modify `RWA` so that pausing does not block burns initiated by the protocol liquidator.
- Consider allowing **borrower redemption at/after deadline until liquidation occurs** to avoid "stuck forever" states.

Team Response

Fixed.

[L-01] Pawn Duration Not Bound to Signed Quote Allows 730-day Term Tampering

Severity

Low Risk

Description

`PawnShop` verifies an EIP-712 signature over `(tokenId, offerAmount, validUntil, nonce)` but `pawn()` accepts a user-supplied `durationDays` that is **not signed**. This lets a borrower reuse the same backend+treasury signatures while swapping duration terms:

(Case 1): The backend intended a 7-day loan, but the borrower executed it as a 30-day loan.

Impact (max severity scenario): - If your backend/risk engine prices or approves offers differently per duration (common), a borrower can bypass that policy by choosing the more favourable term on-chain.

(Case 2): The backend intended a 30-day loan, but the borrower executed it as a 7-day loan.

Impact: - Direct protocol revenue loss: borrower pays the 7-day fee schedule instead of the 30-day fee schedule. - If your backend applies different approvals/limits per duration, this is also a term-tampering bypass.

Location of Affected Code

File: Audit_Submission/src/Pawn.sol

```
// durationDays is NOT part of the signed struct
bytes32 private constant PAWN_TYPEHASH =
    keccak256("Pawn(uint256 tokenId,uint256 offerAmount,uint256
       validUntil,uint256 nonce)");

bytes32 structHash = keccak256(
   abi.encode(
       PAWN_TYPEHASH,
       tokenId,
       offerAmount,
       validUntil,
       usedNonce
   )
);

function pawn(uint256 tokenId, uint8 durationDays, uint256 offerAmount,
   uint256 validUntil, bytes calldata, bytes calldata) external {
    // durationDays controls fee + deadline, but is not authenticated by
       signatures
    uint16 fee = durationDays == 7 ? fee7Days : fee30Days;
    uint256 deadline = block.timestamp + (durationDays * 1 days);

    // Treasury signature is also not bound to durationDays
    bytes32 operationHash = keccak256(abi.encode("PAWN", tokenId,
       offerAmount, currentNonce, msg.sender));
}
```

Impact

- **7 → 30**: borrower can extend repayment window / delay liquidation from 7 to 30 days while using a quote intended for 7 days (bypasses duration-based risk approval; increases exposure + liquidity risk).
- **30 → 7**: borrower can execute as 7 days to pay the lower fee schedule (protocol revenue loss + bypasses duration-based pricing).

Proof of Concept

Run in `test/System.t.sol`

```
function test_POC_Pawn_DurationNotSigned_AllowsExtending7DayQuoteTo30Days
    () public {
     /**
      * POC (Case 1): The backend intended a 7-day loan, but the borrower
         executed it as a 30-day loan.
      *
      * Root cause:
      * - `PawnShop`'s EIP-712 signature (`PAWN_TYPEHASH`) does NOT commit
          to `durationDays` (or `deadline`).
      * - The Treasury `operationHash` also does NOT commit to `
         durationDays`.
      * Therefore, the exact same signatures are valid for BOTH `
         durationDays=7` and `durationDays=30`.
      *
      * Impact (max severity scenario):
      * - If your backend/risk engine prices or approves offers
         differently per duration (common),
      *   A borrower can bypass that policy by choosing the more
         favourable term on-chain.
      * - Here we demonstrate the most dangerous direction: a borrower can
          EXTEND the repayment window
      *   (and liquidation delay) from 7 days to 30 days while using the
         same signed quote.
      */
     uint256 tokenId = 1;
     uint256 amount = 500 * 10**6;
     uint256 validUntil = block.timestamp + 1 hours;

     uint256 pawnTime = block.timestamp;

     // "Backend quote" is represented by building signatures while
        expecting `durationDays=7`.
     PawnCall memory c = _buildPawnCall(user, tokenId, 7, amount,
        validUntil);
     bytes32 digest = _pawnDigest(tokenId, amount, validUntil, c.nonce);

     // Borrower flips ONLY the duration parameter (not covered by the
        signature) to extend the loan term.
     c.durationDays = 30;

     _pawnWithCall(user, c, true);

     // The loan is now a 30-day loan on-chain, despite being signed as a
        "7-day quote" off-chain.
     (,, uint256 deadline, uint16 feeBP, bool active) = pawnShop.pawns(
        tokenId);
     assertTrue(active);
     assertEq(deadline, pawnTime + 30 days);
     assertEq(feeBP, pawnShop.fee30Days());
     assertTrue(pawnShop.usedSignatures(digest));
```

```
    // Demonstrate the practical consequence: borrower can wait past 7
       days and still redeem.
    // A correctly-bound 7-day loan would have expired at this point (
       redeem would revert).
    vm.warp(pawnTime + 8 days);
    (uint256 totalDue, uint256 fee) = pawnShop.calculateRepayment(tokenId
       );

    // Borrower already received `amount` during `pawn()`. Mint only the
       fee so the redemption can succeed.
    usdc.mint(user, fee);

    vm.startPrank(user);
    usdc.approve(address(pawnShop), totalDue);
    pawnShop.redeem(tokenId);
    vm.stopPrank();

    assertEq(rwa.ownerOf(tokenId), user);
}

function
 test_POC_Pawn_DurationNotSigned_AllowsUsing30DQuoteAs7Days_ToPayLwrFee()
     public {
    /**
     * POC (Case 2): backend intended a 30-day loan, but borrower
        executes it as a 7-day loan.
     *
     * Impact:
     * - Direct protocol revenue loss: borrower pays the 7-day fee
        schedule instead of the 30-day fee schedule.
     * - If your backend applies different approvals/limits per duration,
         this is also a term-tampering bypass.
     */
    uint256 tokenId = 1;
    uint256 amount = 500 * 10**6;
    uint256 pawnTime = block.timestamp;
    uint256 validUntil = pawnTime + 1 hours;
    uint256 treasuryBal0 = usdc.balanceOf(address(treasury));
```

```
    // Build signatures while the backend expects `durationDays=30`.
    PawnCall memory c = _buildPawnCall(user, tokenId, 30, amount,
       validUntil);
    // Borrower flips ONLY the duration parameter to reduce the fee
       schedule to the 7-day rate.
    c.durationDays = 7;

    _pawnWithCall(user, c, true);

    {
        (,, uint256 deadline, uint16 feeBP, bool active) = pawnShop.pawns
           (tokenId);
        assertTrue(active);
        assertEq(deadline, pawnTime + 7 days);
        assertEq(feeBP, pawnShop.fee7Days());
    }
    assertTrue(pawnShop.usedSignatures(_pawnDigest(tokenId, amount,
       validUntil, c.nonce)));

    // Fee difference: 30-day fee basis points are higher than 7-day fee
       basis points.
    vm.warp(pawnTime + 1 days);
    {
        (uint256 totalDue, uint256 fee) = pawnShop.calculateRepayment(
           tokenId);

        uint256 expectedFee7 = (amount * pawnShop.fee7Days()) / 10_000;
        assertEq(fee, expectedFee7);
        assertTrue(((amount * pawnShop.fee30Days()) / 10_000) >
           expectedFee7);

        // Redeem and verify Treasury only collected the lower (7-day)
           fee.
        usdc.mint(user, fee);

        vm.startPrank(user);
        usdc.approve(address(pawnShop), totalDue);
        pawnShop.redeem(tokenId);
        vm.stopPrank();

        assertEq(usdc.balanceOf(address(treasury)), treasuryBal0 +
           expectedFee7);
    }
    assertEq(rwa.ownerOf(tokenId), user);
}
```

Recommendation

Bind the loan term to signatures by including `durationDays` (or the derived `deadline`) in the `pawn()` EIP-712 signed struct.

Team Response

Fixed.

[L-02] Hardcoded 6-decimal Stablecoin Assumptions Brick Protocol on 18-decimal Deployments

Severity

Low Risk

Description

Note: this issue is valid if the protocol deploys in chains like BSC 1. In the BSC chain, a stable coin with $8B in circulation has 18 decimals: address 2. Search other stablecoins, here at https://bscscan.com/tokens. They are mostly of 18 decimals

Multiple core parameters are hardcoded as if the payment token ("USDC") always uses **6 decimals** (multiplying by `10**6`). If the deployed payment token uses **18 decimals** (common for stables on some chains), these constants become off by 10^12, causing critical functionality to revert or enforce meaningless limits.

Specifically: - `PawnShop.MAX_LOAN_AMOUNT` is expressed in 6-decimal units, so any realistic 18-decimal loan amount becomes "too high" and reverts. - `RWA.MIN_SELLBACK_AMOUNT` is intended to represent "1 USDC", but on 18 decimals, it becomes dust, so the minimum sellback policy is not enforced. - `Treasury.maxTransferPerCall` defaults to a 6-decimal unit limit, so `transferWithSignature()` reverts for normal 18-decimal amounts until an admin updates the value.

Location of Affected Code

File: Audit_Submission/src/Pawn.sol

```
uint256 public constant MAX_LOAN_AMOUNT = 1_000_000 * 10**6;

if (offerAmount == 0 || offerAmount > MAX_LOAN_AMOUNT) revert
    InvalidAmount();
```

File: Audit_Submission/src/sRWA.sol

```
uint256 public constant MIN_SELLBACK_AMOUNT = 1 * 10**6;

if (usdcAmount < MIN_SELLBACK_AMOUNT) revert InvalidAmount();
```

File: Audit_Submission/src/Treasury.sol

```
uint256 public maxTransferPerCall = 100_000 * 10**6;

if (limit == type(uint256).max) limit = maxTransferPerCall;
if (amount > limit) revert AmountExceedsMaxTransfer();
```

Impact

- **Protocol bricking on 18-decimals deployments**: borrowers cannot take "normal" loans because `pawn()` reverts with `InvalidAmount()` for realistic values.
- **Broken economic constraint**: `MIN_SELLBACK_AMOUNT` no longer enforces "minimum 1 token"; users can sell back for dust amounts if the backend signs it.
- **Operational failure risk**: Treasury payouts revert with `AmountExceedsMaxTransfer()` until `maxTransferPerCall` is manually updated.

Proof of Concept

Run it in `test/System.t.sol`

```
function
    test_POC_18DecimalsStable_Hardcoded1e6_Assumptions_BrickOrBreakLimits
    () public {
     /**
      * This POC demonstrates a real deployment hazard:
      * the protocol hardcodes several "USDC has 6 decimals" constants (x
         * 10**6),
      * but many chains/tokens (e.g. BSC stables) use 18 decimals.
      *
      * In THIS test suite, `MockUSDC` inherits OpenZeppelin `ERC20`,
         which uses 18 decimals by default.
      * So amounts like 1e18 represent "1 whole token", while 1e6
         represents "0.000000000001 token".
      *
      * We prove all 3 issues in one flow:
      *  (1) PawnShop.MAX_LOAN_AMOUNT is set to 1_000_000 * 1e6 (6-decimal
          units).
      *      With an 18-decimal token, even a 1-token loan (1e18) is >
         MAX_LOAN_AMOUNT and reverts.
      *  (2) RWA.MIN_SELLBACK_AMOUNT is set to 1 * 1e6, intended to mean
         "1 USDC".
      *      With an 18-decimal token, this is dust (1e-12 token), so the
         "min 1 token" policy is not enforced.
      *  (3) Treasury.maxTransferPerCall defaults to 100_000 * 1e6, also
         6-decimals units.
      *      With an 18-decimal token, even a 1-token payout exceeds the
         limit until the admin updates it.
      */
     assertEq(usdc.decimals(), 18);

     // 1 whole token in base units for an 18-decimal ERC20.
     uint256 oneToken = 1e18;
```

```
/* ============================================================
 * (1) PawnShop MAX_LOAN_AMOUNT bricks normal loans on 18-decimals.
 * ============================================================ */
{
    uint256 tokenId = 1;

    // MAX_LOAN_AMOUNT is 1_000_000 * 1e6 = 1e12 base units.
    // On an 18-decimal token, 1e12 base units is only 0.000001 token
       .
    assertEq(pawnShop.MAX_LOAN_AMOUNT(), 1_000_000 * 10**6);
    assertGt(oneToken, pawnShop.MAX_LOAN_AMOUNT());

    // Even with valid signatures, the call reverts before signature
       verification due to the hardcoded max amount.
    PawnCall memory c = _buildPawnCall(user, tokenId, 7, oneToken,
       block.timestamp + 1 hours);

    // Approve first (this should succeed). Then prove the pawn()
       itself is bricked by the hardcoded MAX_LOAN_AMOUNT.
    vm.prank(user);
    rwa.approve(address(pawnShop), tokenId);

    vm.expectRevert(PawnShop.InvalidAmount.selector);
    _pawnWithCall(user, c, false);

    // Sanity: pawn reverted, so the NFT was not transferred.
    assertEq(rwa.ownerOf(tokenId), user);
}

/* ============================================================
 * (2) RWA MIN_SELLBACK_AMOUNT becomes dust on 18-decimal tokens.
 * ============================================================ */
uint256 userBalBeforeSell = usdc.balanceOf(user);
{
    uint256 tokenId = 1;
    uint256 sellAmount = rwa.MIN_SELLBACK_AMOUNT(); // 1 * 1e6 base
       units
    assertEq(sellAmount, 1 * 10**6);

    // On 18 decimals, this is far less than 1 whole token, so the
       intended "min 1 token" policy is not enforced.
    assertLt(sellAmount, oneToken);
```

```
        uint256 validUntil = block.timestamp + 1 hours;
        uint256 nonce = rwa.nonces(user);
        bytes memory sellSig = _signSellback(backendPk, tokenId,
            sellAmount, validUntil, nonce);
        bytes32 opHash = keccak256(abi.encode("SELLBACK", tokenId,
            sellAmount, nonce, user));
        bytes memory treasSig = _signTreasury(backendPk, user, sellAmount
            , validUntil, opHash);

        // This succeeds because `sellAmount` >= MIN_SELLBACK_AMOUNT (
            even though it's dust in 18-decimal terms).
        vm.prank(user);
        rwa.burnForSellback(tokenId, sellAmount, validUntil, sellSig,
            treasSig);

        assertEq(usdc.balanceOf(user), userBalBeforeSell + sellAmount);
        vm.expectRevert();
        rwa.ownerOf(tokenId); // tokenId 1 is burned
    }

    /* ============================================================
     * (3) Treasury maxTransferPerCall is also 6-decimals by default.
     * ============================================================ */
    {
        // Create a new approved contract so we can call `
            transferWithSignature` from an authorized sender.
        TreasurySigReplayer replayer;
        vm.startPrank(owner);
        replayer = new TreasurySigReplayer(treasury);
        treasury.approveContractForTransfers(address(replayer));
        vm.stopPrank();
```

```
        // Try a "normal" 1-token payout.
        uint256 validUntil = block.timestamp + 1 hours;
        bytes32 opHash = keccak256("POC_TREASURY_LIMIT_18_DECIMALS");
        bytes memory sig = _signTreasury(backendPk, user, oneToken,
            validUntil, opHash);

        // This fails because Treasury defaults to maxTransferPerCall =
            100_000 * 1e6 (6-decimals units).
        // In 18-decimal terms, that limit is tiny, so even 1 token
            exceeds it.
        assertEq(treasury.maxTransferPerCall(), 100_000 * 10**6);
        assertGt(oneToken, treasury.maxTransferPerCall());

        vm.expectRevert(Treasury.AmountExceedsMaxTransfer.selector);
        replayer.replayTreasurySignature(user, oneToken, validUntil,
            opHash, sig);

        // Admin can "fix" this after deployment by updating the limit to
             18-decimal units,
        // e.g. 100_000 tokens = 100_000 * 1e18.
        vm.startPrank(owner);
        treasury.setMaxTransferPerCall(100_000 * oneToken);
        // Fund Treasury so the post-fix payout doesn't fail the balance
            check.
        usdc.mint(address(treasury), oneToken);
        vm.stopPrank();

        uint256 userBalBeforePayout = usdc.balanceOf(user);
        replayer.replayTreasurySignature(user, oneToken, validUntil,
            opHash, sig);
        assertEq(usdc.balanceOf(user), userBalBeforePayout + oneToken);
    }
}
```

Recommendation

- Remove `10**6` hardcoding for the payment token.
- Make thresholds **decimals-aware** by :
  - **Passing `tokenDecimals` / `unit` as a constructor parameter**

Team Response

Fixed.

[L-03] `permit()` Can Approve a Blacklisted Spender (Missing Blacklist Check) But approve() Does Not Allow

Severity

Low Risk

Description

`RWA` enforces blacklist restrictions in `approve()` / `setApprovalForAll()` by reverting when the target operator/spender is blacklisted.

However, the gasless approval path `permit()` does **not** validate `_isBlacklisted[spender]` and directly calls `_approve(spender, ...)`. As a result, a blacklisted address can become the current `getApproved(tokenId)` via `permit()`, even though `approve()` would revert.

Note: the blacklisted spender still cannot execute `transferFrom()` while blacklisted because `_update()` reverts when `auth` (caller) is blacklisted. So this is primarily a **policy bypass / inconsistency** (and potential UX/integration/compliance confusion), not an immediate theft vector by itself.

Location of Affected Code

File: Audit_Submission/src/sRWA.sol

```
function permit(address spender, uint256 tokenId, uint256 deadline, uint8
    v, bytes32 r, bytes32 s) external {
  // code
   _approve(spender, tokenId, owner); // missing: if (_isBlacklisted[
      spender]) revert Blacklisted();
}

function approve(address to, uint256 tokenId) public virtual override {
   if (_isBlacklisted[to]) revert Blacklisted(); // blacklist enforced
      here
   super.approve(to, tokenId);
}
```

Impact

A blacklisted address can still appear as the approved spender for a token (`getApproved(tokenId)`), despite the blacklist policy.

Proof of Concept

Run in `test/System.t.sol`

```
function test_POC_Permit_Allows_BlacklistedSpender_ToBeApproved() public
    {
     /**
      * POC: `RWA.permit()` does NOT check `_isBlacklisted[spender]`,
         unlike `approve()`.
      *
      * Expected blacklist policy (as implemented in approve()):
      * - You should NOT be able to approve a blacklisted spender.
      *
      * Actual behavior:
      * - `approve(blacklisted, tokenId)` reverts,
      * - but `permit(blacklisted, tokenId, ...)` succeeds and sets `
         getApproved(tokenId) == blacklisted`.
      *
      * Note: the blacklisted spender still cannot transfer (blocked by `
         _update()`'s `auth` blacklist check),
      * so the impact is mainly "policy bypass / inconsistent approval
         state".
      */
     address blacklistedSpender = makeAddr("blacklistedSpender");
     address relayer = makeAddr("relayer");
     address receiver = makeAddr("receiver");

     // Mint a fresh token to `owner` (we have ownerPk so we can sign
        permits).
     vm.startPrank(owner);
     rwa.mint(owner, "PERMIT-BLACKLIST");
     uint256 tokenId = rwa.nextTokenId() - 1;
```

```
    // Admin blacklists the spender.
    rwa.blacklistContract(blacklistedSpender);
    assertTrue(rwa.isBlacklisted(blacklistedSpender));

    // Baseline: normal approve() respects blacklist and reverts.
    vm.expectRevert(RWA.Blacklisted.selector);
    rwa.approve(blacklistedSpender, tokenId);
    vm.stopPrank();

    // Build an EIP-712 permit approving the blacklisted spender.
    uint256 nonce = rwa.nonces(owner);
    uint256 deadline = block.timestamp + 1 hours;
    (uint8 v, bytes32 r, bytes32 s) = _signPermit(ownerPk,
       blacklistedSpender, tokenId, nonce, deadline);

    // Anyone can submit the permit (gasless approval). Use a relayer for
        clarity.
    vm.prank(relayer);
    rwa.permit(blacklistedSpender, tokenId, deadline, v, r, s);

    // Approval state is now set to a blacklisted address (policy bypass)
       .
    assertEq(rwa.getApproved(tokenId), blacklistedSpender);

    // Even though approved, the blacklisted spender cannot transfer due
       to `_update()` reverting on `auth` blacklist.
    vm.prank(blacklistedSpender);
    vm.expectRevert(RWA.Blacklisted.selector);
    rwa.transferFrom(owner, receiver, tokenId);
}
```

Recommendation

Consider adding the same blacklist enforcement to `permit()` as `approve()` :- `if (spender != address(0) && _isBlacklisted[spender]) revert Blacklist`

Team Response

Fixed.

[L-04] Admin Can Drain Treasury Reserves via `withdrawUSDC()`

Severity

Low Risk

Description

The `Treasury` contract is the central pool of USDC that the system uses to fund `PawnShop.pawn(...)` loans and `RWA.burnForSellback(...)` buybacks (context).

However, `Treasury.withdrawUSDC(...)` allows `DEFAULT_ADMIN_ROLE` to transfer arbitrary amounts of USDC to an arbitrary address, with no on-chain constraints.

This means an admin can fully drain user/protocol reserves and render core flows insolvent, effectively enabling a rug pull where the admin can steal all Treasury funds while users have NFTs locked in active pawn positions.

Location of Affected Code

File: Treasure.sol#L167-L174

```
function withdrawUSDC(address to, uint256 amount) external nonReentrant
    onlyRole(DEFAULT_ADMIN_ROLE) {
     // @audit Admin can withdraw the entire USDC reserve at any time
     if (to == address(0)) revert InvalidAddress();
     if (amount == 0) revert InvalidAmount();
     if (amount > usdc.balanceOf(address(this))) revert
        InsufficientBalance();

     usdc.safeTransfer(to, amount);
     emit USDCWithdrawn(to, amount);
}
```

Impact

The vulnerability enables direct fund loss as the admin can transfer all USDC out of the Treasury, leading to protocol insolvency where drained reserves cause signature-authorized payouts to fail due to insufficient balance, halting both `pawn(...)` and sellback flows. This effectively creates a rug pull scenario where the admin can drain all Treasury reserves while users have NFTs locked in active pawn positions, leaving users unable to access new loans and completely compromising the protocol's solvency.

Proof of Concept

1. `Treasury` holds USDC reserves for loans and buybacks.
2. Admin calls `withdrawUSDC(admin, usdc.balanceOf(address(treasury)))`.
3. `Treasury` becomes empty; future payouts revert due to `InsufficientBalance()`.

Recommendation

Allow a rescue token functionality with no USDC withdrawal or keep a track of the protocol's funds and only allow to sweep the dust.

Team Response

Fixed.

[L-05] Unbounded Fee Parameters Allow Abusive Fees and Redemption DoS

Severity

Low Risk

Description

`PawnShop` charges a fee (in basis points) on top of the borrowed principal when a borrower redeems their NFT via `redeem(...)` (context).

However, `setFees(...)` allows `MANAGER_ROLE` to set `fee7Days` and `fee30Days` to any `uint16` value, with no upper bound (e.g., `<= BASIS_POINTS`) and no sanity checks (problem).

This means a manager can set fees to values that make redemption economically impossible (or unexpectedly expensive), effectively trapping users or extracting arbitrary value at repayment time (impact).

Location of Affected Code

File: Pawn.sol#L111-L115

```
function setFees(uint16 _fee7Days, uint16 _fee30Days) external onlyRole(
    MANAGER_ROLE) {
     // @audit No bounds checks (e.g., <= BASIS_POINTS)
     fee7Days = _fee7Days;
     fee30Days = _fee30Days;
     emit FeesUpdated(_fee7Days, _fee30Days);
}
```

Impact

- **Redemption DoS**: Fees can be set so high that borrowers cannot (or will not) repay.
- **Unexpected user loss**: Borrowers who expected fixed or bounded fees can be forced into paying excessive fees to redeem collateral.

Proof of Concept

1. User pawns an NFT.
2. Manager calls `setFees(65_535, 65_535)`.
3. `redeem(...)` now requires paying principal + ~655.35% fee, which will likely be infeasible for most borrowers.

Recommendation

Enforce bounds (example): - `require(_fee7Days <= BASIS_POINTS && _fee30Days <= BASIS_POINTS)`

Team Response

Fixed.

[L-05] Unbounded Fee Parameters Allow Abusive Fees and Redemption DoS

Severity

Low Risk

Description

`PawnShop` charges a fee (in basis points) on top of the borrowed principal when a borrower redeems their NFT via `redeem(...)` (context).

However, `setFees(...)` allows `MANAGER_ROLE` to set `fee7Days` and `fee30Days` to any `uint16` value, with no upper bound (e.g., `<= BASIS_POINTS`) and no sanity checks (problem).

This means a manager can set fees to values that make redemption economically impossible (or unexpectedly expensive), effectively trapping users or extracting arbitrary value at repayment time (impact).

Location of Affected Code

File: Pawn.sol#L111-L115

```
function setFees(uint16 _fee7Days, uint16 _fee30Days) external onlyRole(
    MANAGER_ROLE) {
     // @audit No bounds checks (e.g., <= BASIS_POINTS)
     fee7Days = _fee7Days;
     fee30Days = _fee30Days;
     emit FeesUpdated(_fee7Days, _fee30Days);
}
```

Impact

- **Redemption DoS**: Fees can be set so high that borrowers cannot (or will not) repay.
- **Unexpected user loss**: Borrowers who expected fixed or bounded fees can be forced into paying excessive fees to redeem collateral.

Proof of Concept

1. User pawns an NFT.
2. Manager calls `setFees(65_535, 65_535)`.
3. `redeem(...)` now requires paying principal + ~655.35% fee, which will likely be infeasible for most borrowers.

Recommendation

Enforce bounds (example): - `require(_fee7Days <= BASIS_POINTS && _fee30Days <= BASIS_POINTS)`

Team Response

Fixed.

[I-01] Burned UUIDs Can Be Re-Minted

Severity

Informational Risk

Description

The `RWA.mint(...)` assigns a `uuid` to each token and enforces uniqueness by tracking `_usedUUIDs[uuid]` (context).

However, all burn paths (`burn(...)`, `burnForRedemption(...)`, `burnForSellback(...)`) delete `_usedUUIDs[uuid]`, making the UUID available for reuse (problem).

This means the on-chain guarantee is "no two currently existing tokens share a UUID", not "a UUID is unique forever". If off-chain systems treat `uuid` as a permanent physical-asset identifier, UUID reuse can lead to confusion or allow re-issuance of previously redeemed assets without an explicit re-deposit workflow (impact).

Location of Affected Code

File: sRWA.sol#L140-L152

```
function mint(address to, string calldata uuid) external onlyRole(
    MINTER_ROLE) nonReentrant whenNotPaused {
     // code
     if (_usedUUIDs[uuid]) revert UUIDAlreadyExists();
     // code
     _tokenUUID[tokenId] = uuid;
     _usedUUIDs[uuid] = true;
     _safeMint(to, tokenId);
}
```

File: sRWA.sol#L200-L207

```
function burnForRedemption(uint256 tokenId) external onlyTokenOwner(
    tokenId) whenNotPaused {
     string memory uuid = _tokenUUID[tokenId];
     // @audit UUID becomes reusable after burn
     delete _usedUUIDs[uuid];
     delete _tokenUUID[tokenId];
     _burn(tokenId);
}
```

Impact

UUID uniqueness is not permanent.

Proof of Concept

1. Admin mints a token with `uuid = "UUID-123"`.

2. Owner calls `burnForRedemption(tokenId)`.
3. Admin mints a new token again with `uuid = "UUID-123"`; the mint succeeds because `_usedUUIDs["UUID-123"]` was deleted.

Recommendation

Decide which uniqueness guarantee is required:

- If UUID must be globally unique forever, **do not delete** `_usedUUIDs[uuid]` on burn.

Team Response

Acknowledged.

[I-02] Redundant `usedSignatures` Tracking Alongside Nonce-Based Replay Protection

Severity

Informational Risk

Description

`PawnShop` and `RWA` apply two replay protections to EIP-712 signatures:

- Per-user `nonces[user]` included in the signed payload and incremented on success.
- A global `usedSignatures[digest]` mapping.

For these flows, the nonce already prevents replay for the same sender: after a successful call, the contract computes the digest using `nonce + 1`, so the old signature no longer verifies. As a result, `usedSignatures` is largely redundant and adds an extra storage write.

Note: `Treasury.transferWithSignature(...)` does not include a nonce, so `usedSignatures` (or adding a nonce/salt) is required there to prevent replay until `validUntil`.

Location of Affected Code

File: Pawn.sol

File: sRWA.sol

- `usedSignatures[digest]` check + write in signature validation paths
- `nonces[msg.sender]++` performed after successful validation

File: Treasury.sol

- Signature payload lacks a nonce; `usedSignatures` provides replay protection until `validUntil`

Impact

Each successful signature-based call in `PawnShop` / `RWA` incurs an extra `SSTORE` for `usedSignatures[digest]`.

Recommendation

In `PawnShop` and `RWA`, remove `usedSignatures` and rely on nonces for replay protection.

Team Response

Fixed.

[I-03] Configuration Mismatch Risk Across `backendSigner` and `treasury` Addresses

Severity

Informational Risk

Description

The system relies on coordinated configuration across three separate contracts:

- `PawnShop` validates a backend signature (`pawnSignature`) using its own `backendSigner`.
- `RWA` validates a backend signature (`sellbackSignature`) using its own `backendSigner`.
- `Treasury` validates a backend signature (`treasurySignature`) using its `backendSigner`, and also requires the caller to be allowlisted.

These settings are each managed independently via admin setters (context).

However, there is no on-chain enforcement that these configuration values remain consistent (problem).

This means a simple operational misconfiguration (e.g., updating `PawnShop.backendSigner` but not `Treasury.backendSigner`, or pointing `PawnShop.treasury` at the wrong address) can halt core flows like `pawn(...)` and `burnForSellback(...)` (impact).

Location of Affected Code

File: Pawn.sol

```
address public treasury;
address public backendSigner;

function setTreasury(address _treasury) external onlyRole(
    DEFAULT_ADMIN_ROLE) {
     // code
     treasury = _treasury;
}

function setBackendSigner(address _signer) external onlyRole(
    DEFAULT_ADMIN_ROLE) {
     // code
     backendSigner = _signer;
}
```

File: Treasure.sol

```
address public backendSigner;

function setBackendSigner(address _signer) external onlyRole(
    DEFAULT_ADMIN_ROLE) {
     // code
     backendSigner = _signer;
}
```

File: sRWA.sol

```
address public treasury;
address public backendSigner;

function setTreasury(address _treasury) external onlyRole(
    DEFAULT_ADMIN_ROLE) {
     // code
     treasury = _treasury;
}

function setBackendSigner(address _signer) external onlyRole(
    DEFAULT_ADMIN_ROLE) {
     // code
     backendSigner = _signer;
}
```

Impact

Mismatched signers or wrong treasury addresses cause signature checks or external calls to revert, halting user actions.

Proof of Concept

1. Admin updates `PawnShop.backendSigner` to a new key.
2. Admin forgets to update `Treasury.backendSigner`.

3. Users can no longer successfully call `pawn(...)` because the pawn signature and treasury signature are validated against different signer keys across contracts.

Recommendation

Use a **single configuration source of truth**, e.g. a registry contract that stores `backendSigner` / `treasury` addresses read by all modules.

Team Response

Acknowledged.

[I-04] PawnShop `MAX_LOAN_AMOUNT` Can Exceed Treasury

Severity

Informational Risk

Description

`PawnShop` validates offers up to its hard cap `MAX_LOAN_AMOUNT = 1_000_000 * 10**6`, but the funding path `Treasury.transferWithSignature(...)` applies a per-call limit from `contractTransferLimits[msg.sender]` (defaulting to `maxTransferPerCall = 100_000 * 10**6` when unset).

As a result, `PawnShop` can accept and sign offers up to 1,000,000 USDC that will deterministically revert in Treasury for amounts above 100,000 USDC unless the transfer limit is explicitly raised.

Location of Affected Code

File: Pawn.sol#L62

```
// max loan is 1 million usdc
uint256 public constant MAX_LOAN_AMOUNT = 1_000_000 * 10**6;
```

File: Treasure.sol

```
mapping(address => uint256) public contractTransferLimits;
uint256 public maxTransferPerCall = 100_000 * 10**6;

function transferWithSignature(address recipient, uint256 amount, uint256
    validUntil, bytes32 operationHash, bytes calldata signature) external
    nonReentrant whenNotPaused {
    // code
    uint256 limit = contractTransferLimits[msg.sender];
    if (limit == type(uint256).max) limit = maxTransferPerCall;

    if (amount > limit) revert AmountExceedsMaxTransfer();
    if (amount > usdc.balanceOf(address(this))) revert
       InsufficientBalance();
    // code
}
```

Impact

Users can waste gas on offers that pass `PawnShop` checks but fail funding.

Recommendation

Align the constraints:

- Either lower `PawnShop.MAX_LOAN_AMOUNT` to match the effective Treasury limit, or
- Raise/configure `Treasury.maxTransferPerCall` / `contractTransferLimits[PawnShop]` to support the intended max loan.

Team Response

Fixed.

shieldify

Thank you!