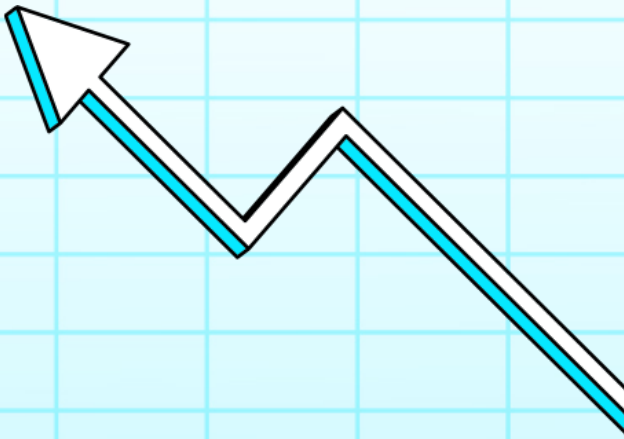
shieldify



Harmonix Finance

SECURITY REVIEW

Date: 17 November 2025

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 has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Rust, Go, Vyper, Move and Cairo.

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 Harmonix Finance

Harmonix Finance is a DeFi platform that combines advanced hedge fund-grade strategies with native blockchain technology to optimize yield generation and liquidity efficiency. Designed for both retail and institutional investors, Harmonix offers tools to generate sustainable returns on assets like ETH, BTC, stablecoins, and DeFi positions such as staked ETH, restaked ETH, PT Pendle, and Hyperliquid tokens.

Harmonix stands apart by merging sophisticated derivative strategies, such as delta-neutral methods and out-of-the-money (OTM) options, with the transparency and accessibility of DeFi, ensuring users can earn more while mitigating risk.

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 3 days, with a total of 48 hours dedicated by the Shieldify team.

Overall, the code is well-written. The audit report identified two High, two Medium and six Low severity issues. They're mainly related to the left testing function that allows anyone to finalize the sale early, zero-address permit signer disables all access controls and possible rewriting of the allocation rules after deposits through the `whitelistReserve()` function.

The Harmonix team has been highly responsive to the Shieldify research team's inquiries and promptly implemented all recommendations.

5.1 Protocol Summary

Project Name	**Harmonix Finance – Token Sale**
Repository	core-contracts
Type of Project	Yield Optimizer, TokenSale
Security Review Timeline	3 days
Review Commit Hash	72db1cc14bd3601a94a97e5096400599f5a72cac
Fixes Review Commit Hash	67d060f57740ec480a9866dc84c2cdb89f17048c

5.2 Scope

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

File	**nSLOC**
contracts/token_sales/HarTokenSale.sol	256
Total	**256**

6. Findings Summary

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

- **High** issues: 2
- **Medium** issues: 2
- **Low** issues: 6

ID	Title	Severity	Status
[H-01]	Anyone Can Finalize the Sale Early via `testFinalizeSettlement()`	High	Fixed
[H-02]	The `finalizeSettlement()` Can Be Dossed Leading to Refund Failures	High	Fixed
[M-01]	Zero-address Permit Signer Disables All Access Controls	Medium	Fixed
[M-02]	Mutable `whitelistReserve()` Allows Rewriting Allocation Rules After Deposits	Medium	Fixed
[L-01]	Off by One Error in `\_validatePermit()` Function	Low	Fixed
[L-02]	Rounding Mismatch in Finalization Can Make Refunds Insolvent	Low	Fixed
[L-03]	Missing Parameter Validations in Initialization	Low	Fixed
[L-04]	Permit Signer Can Be Changed Mid-Sale, Invalidating All Existing Permits	Low	Acknowledged
[L-05]	Initialize Function Can Be Frontrunned to Become the Owner	Low	Fixed
[L-06]	Settlement Will Not Be Finalised Unless Purchase Tokens Are Transferred to the Contract Before Finalisation of the Settlement	Low	Acknowledged

7. Findings

[H-01] Anyone Can Finalize the Sale Early via `testFinalizeSettlement()`

Severity

High Risk

Description

The `testFinalizeSettlement()` is publicly callable and lacks both an owner check and a time check. Any address can flip `settlementFinalized = true` at any time, freezing allocations and permanently blocking further purchases.

Location of Affected Code

File: contracts/token_sales/HarTokenSale.sol#L182-L185

```
function testFinalizeSettlement() external {
    require(!settlementFinalized, "Settlement: finalized");
    _finalizeSettlement();
}
```

Impact

- Permanent DoS of `purchase()` for everyone (`require(!settlementFinalized)` fails).
- Allocations (`totalAccepted` , `priorityRaiseFilled` , `publicRaiseFilled`) get locked at attacker-chosen block, skewing results and preventing intended participation.

Recommendation

Remove `testFinalizeSettlement()` in production.

Team Response

Fixed.

[H-02] The `finalizeSettlement()` Can Be Dossed Leading to Refund Failures

Severity

High Risk

Description

In the fix, the team added `treasury` address and during purchase, all the token goes to the treasury and then later the tokens will be transferred from the treasury to the contract before they call the `finalizeSettlement()` as per the current design to ensure the users get the refunds accordingly.

However, anyone can DoS the `finalizeSettlement()` forever because of the strict check:

```
require(curBalance + safetyBuffer == totalRefund, "Settlement: total
    refund not matched");
```

The `curBalance` is taken as the contract balance, which can be increased by anyone by directly transferring the `purchaseToken` to the contract.

An attacker can simply:

- Monitor the mempool for the finalizeSettlement() transaction
- Front-run it by sending any amount of purchaseToken > 1000 wei directly to the contract
- This makes curBalance > totalRefund - safetyBuffer
- The equality check fails, reverting the transaction

Thus, DoS-ing the transaction forever and thus users can't ever claim their refunds.

Location of Affected Code

File: contracts/token_sales/HarTokenSale.sol#L186-L195

```
function finalizeSettlement() external onlyOwner {
    require(block.timestamp >= saleConfig.settleTime, "Settlement: too
       early");
    require(!settlementFinalized, "Settlement: finalized");

    _finalizeSettlement();
    uint256 curBalance = purchaseToken.balanceOf(address(this));
    uint256 safetyBuffer = 1000;
    uint256 totalRefund = totalCommitted - totalAccepted;
    require(curBalance + safetyBuffer == totalRefund, "Settlement: total
       refund not matched");
}
```

Impact

settlements can't be finalised, leading to refund failures

Recommendation

Change the strict equality :

```
- require(curBalance == totalRefund + safetyBuffer, "Settlement:
   insufficient balance");
+ require(curBalance >= totalRefund + safetyBuffer, "Settlement:
   insufficient balance");
```

Team Response

Fixed.

[M-01] Zero-address Permit Signer Disables All Access Controls

Severity

Medium Risk

Description

When `sonarConfig.permitSigner` is set to `address(0)`, the signature verification is completely bypassed. The only remaining check is `require(signature.length == 0)`, which means anyone can craft their own `PurchasePermitWithAllocation` struct with arbitrary values.

Users can self-whitelist by setting `maxAmount = priorityMaxAmountWithDecimals`, effectively declaring themselves priority participants. They can also set arbitrary public caps and bypass minimum commitment checks. The entire two-tier allocation system collapses.

The code explicitly supports this mode (note the error message "Permit: signatures disabled"), which means it's a valid configuration that the owner can enable via `setPermitSigner(address(0))`. The problem is that this mode has no access controls.

Location of Affected Code

File: contracts/token_sales/HarTokenSale.sol#L327-L335

```
function _validatePermit(PurchasePermitWithAllocation calldata permit,
    bytes calldata signature) internal view {
     // ... saleUUID, expiresAt, wallet checks pass with user-supplied
        values

     address permitSigner_ = sonarConfig.permitSigner;
     if (permitSigner_ != address(0)) {
         address recovered = PurchasePermitWithAllocationLib.recoverSigner
            (permit, signature);
         if (recovered != permitSigner_) {
             revert PurchasePermitUnauthorizedSigner(recovered);
         }
     } else {
         require(signature.length == 0, "Permit: signatures disabled");
         // ← No authentication whatsoever
     }
}
```

File: contracts/token_sales/HarTokenSale.sol#L143

In `purchase()`, the priority check relies entirely on user-supplied data:

```
function purchase( uint256 amount, PurchasePermitWithAllocation calldata
    request, bytes calldata signature ) external {
   // code
   bool isPriorityCommitment = _isPriorityCommitment(request.maxAmount);
   // Returns true if maxAmount == priorityMaxAmountWithDecimals
   // Attacker controls maxAmount when permitSigner is zero
   //code
}
```

Impact

Attack scenario:

1. Owner deploys with `permitSigner = address(0)` or calls `setPermitSigner(address(0))` at any point
2. Attacker crafts a fake permit:

```
PurchasePermitWithAllocation({
    permit: PurchasePermit({
        saleUUID: sonarConfig.saleUUID,          // Public value
        wallet: attackerAddress,                  // Their address
        expiresAt: type(uint64).max,             // Far future
        payload: ""
    }),
    reservedAmount: 0,
    minAmount: 0,                                 // Bypass minimums
    maxAmount: priorityMaxAmountWithDecimals  // Self-whitelist!
});
```

1. Calls `purchase(amount, fakePermit, "")` with empty signature
2. Gets marked as priority: `investors[attacker].isWhitelisted = true` and `priorityCommitted` increases

Result:

- Unlimited Sybil attacks to monopolize the `whitelistReserve`
- Arbitrary cap setting (not limited to priority amount)
- Complete bypass of the whitelist/public tier system
- Per-entity caps become meaningless

Recommendation

Either remove support for `permitSigner == address(0)` entirely, or if this mode is truly needed, add additional validation:

```
function setPermitSigner(address newSigner) external onlyOwner {
    require(newSigner != address(0), "Zero signer not allowed");
    sonarConfig.permitSigner = newSigner;
}
```

Alternatively, if permissionless mode is intentional, add explicit caps in that mode:

```
} else {
    require(signature.length == 0, "Permit: signatures disabled");
    require(request.maxAmount <= someHardcodedPublicCap, "No self-
       whitelisting");
    require(!_isPriorityCommitment(request.maxAmount), "Priority disabled
        without signer");
}
```

Team Response

Fixed.

[M-02] Mutable `whitelistReserve()` Allows Rewriting Allocation Rules After Deposits

Severity

Medium Risk

Description

The `setWhitelistReserve()` function allows the owner to change the whitelist reserve at any time before settlement, including after users have already deposited funds. This directly changes the priority vs public allocation split that users based their decisions on.

Increasing the reserve benefits priority users at the expense of public participants. Decreasing it does the opposite. Either way, users committed funds under one set of rules and may end up with a completely different allocation.

Location of Affected Code

File: contracts/token_sales/HarTokenSale.sol#L240-L245

```
function setWhitelistReserve(uint256 newReserve) external onlyOwner {
    require(!settlementFinalized, "Config: settled");
    require(newReserve <= saleConfig.maxRaise, "Config: reserve too high"
        );
    saleConfig.whitelistReserve = newReserve;  // Can change anytime
        before settlement
    emit WhitelistReserveUpdated(newReserve);
}
```

Impact

Scenario 1 - Reserve decreased:

```
Initial: whitelistReserve = 100k USDC
Priority users deposit 80k, expecting full allocation
Owner changes to whitelistReserve = 20k mid-sale
Priority users now only get 20k allocation, 60k overflows to public pool
Economic expectations broken
```

Scenario 2 - Reserve increased:

```
Initial: whitelistReserve = 20k USDC
Public users deposit expecting 180k capacity (200k total - 20k reserved)
Owner changes to whitelistReserve = 100k mid-sale
Public capacity shrinks to 100k, and many public users face pro-rata cuts
```

- No consent mechanism or warning period
- Users can't withdraw after parameter change (funds locked until settlement)
- Creates trust issues and a potential manipulation vector
- Most token sales freeze parameters once deposits begin

Recommendation

Freeze `whitelistReserve` once the sale opens or after the first deposit:

```
function setWhitelistReserve(uint256 newReserve) external onlyOwner {
    require(!settlementFinalized, "Config: settled");
    require(block.timestamp < saleConfig.saleOpen, "Sale: already started
        ");
    require(newReserve <= saleConfig.maxRaise, "Config: reserve too high"
        );
    saleConfig.whitelistReserve = newReserve;
    emit WhitelistReserveUpdated(newReserve);
}
```

Alternatively, require a time-lock (24-48 hours' notice) before changes take effect, or snapshot the reserve at the first deposit.

Team Response

Fixed.

[L-01] Off by One Error in `_validatePermit()` Function

Severity

Low Risk

Description

The `_validatePermit()` function has a check for `expiresAt`, purchase can't be made if the permit has expired, however, this has an off-by-one error and the call fails when the `block.timestamp == expiresAt`.

Location of Affected Code

File: contracts/token_sales/HarTokenSale.sol#L319

```
function _validatePermit( PurchasePermitWithAllocation calldata permit,
    bytes calldata signature ) internal view {
  // code
  if (permit.permit.expiresAt <= block.timestamp) {
      revert PurchasePermitExpired();
  }
  // code
}
```

Impact

Purchase can't be made when `block.timestamp == expiresAt`.

Recommendation

Consider applying the following mitigation:

```
- if (permit.permit.expiresAt <= block.timestamp) {
+ if (permit.permit.expiresAt < block.timestamp) {

      revert PurchasePermitExpired();
  }
```

Team Response

Fixed.

[L-02] Rounding Mismatch in Finalization Can Make Refunds Insolvent

Severity

Low Risk

Description

The `_finalizeSettlement()` sets global aggregates (`priorityRaiseFilled`, `publicRaiseFilled`, `totalAccepted`) using bucket-level numbers:

- `whitelistAllocated = min(totalPriorityCommitted, whitelistReserve)`
- `publicAllocated = min(remainingCapacity, publicCommitments)`

But per-user acceptance (used by `claimRefund()` and `saleStatus()`) is computed later via `_calculateAcceptedCommitment()` with integer division per user, causing rounding down for each user in both the priority and public pro-rata steps.

As a result:

- `sum(user.accepted)` can be strictly less than `totalAccepted` set at finalize.
- Therefore, `sum(user.refund)` becomes strictly greater than `totalCommitted - totalAccepted`.

If the owner withdraws the full `totalAccepted`, the contract's remaining balance equals `totalCommitted - totalAccepted`, which is not enough to pay all users' refunds (because of the extra rounding dust owed). The last refund(s) will revert due to insufficient balance, creating a refund DoS and a balance shortfall.

Example (base units, no decimals):

- `whitelistReserve = 10`, `maxRaise = 50`
- User A: `priorityCommitted=100`, `committed=100`
- User B: `priorityCommitted=1`, `committed=1`

- Totals: `totalPriorityCommitted=101`, `totalCommitted=101`

Finalize:

- `whitelistAllocated = 10`
- `remainingCapacity = 40`
- `publicCommitments = 101 - 10 = 91`
- `publicAllocated = min(40, 91) = 40`
- `totalAccepted = 10 + 40 = 50`

Per-user acceptance (with integer floors):

- Priority:
  - A: `floor(100*10/101) = 9`
  - B: `floor(1*10/101) = 0`
  - Sum priority accepted = 9 (rounding loss of 1 vs `whitelistAllocated=10`)
- Public scaling factor uses `remainingCapacity/publicCommitments = 40/91`:
  - A public: `floor((100-9)*40/91) = floor(91*40/91) = 40`
  - B public: `floor((1-0)*40/91) = 0`
- User totals:
  - A accepted = `9 + 40 = 49`
  - B accepted = `0`
  - Sum accepted = 49 (but `totalAccepted` is 50)

Refund sums:

- Total refunds by users = `totalCommitted - sumAccepted = 101 - 49 = 52`
- Contract balance left after owner withdraws `totalAccepted=50` is `101 - 50 = 51`
- Shortfall = 1 -> last refund reverts.

This is deterministic whenever per-user rounding creates dust in either (or both) buckets.

Location of Affected Code

- Global accounting in `_finalizeSettlement()`:
  File: contracts/token_sales/HarTokenSale.sol#L198-L210

```
function _finalizeSettlement() internal {
  // code
  // Bucket-level calculation (no per-user rounding)
  uint256 whitelistAllocated = totalPriorityCommitted <=
     whitelistReserve_
      ? totalPriorityCommitted
      : whitelistReserve_;
```

```
  uint256 publicAllocated = publicCommitments <= remainingCapacity
      ? publicCommitments
      : remainingCapacity;

  totalAccepted = whitelistAllocated + publicAllocated;  // Used for
     withdrawals
  // code
}


- Per-user accounting in \codex{_calculateAcceptedCommitment()}: \newline
File: [contracts/token\_sales/HarTokenSale.sol#L346-L367](https://github.
   com/harmonixfi/core-contracts/blob/72
   db1cc14bd3601a94a97e5096400599f5a72cac/contracts/token\_sales/
   HarTokenSale.sol#L346-L367)
\begin{lstlisting}[language=solidity]
function _calculateAcceptedCommitment(address investor) internal view
   returns (uint256) {
  // code
  // Priority pro-rata (rounds down per user)
  if (totalPriorityCommitted <= whitelistReserve_) {
      priorityAccepted = priorityDeposit;
  } else {
      priorityAccepted = (priorityDeposit * whitelistReserve_) /
         totalPriorityCommitted;
      // ↑ Rounds down for EACH user
  }

  // Public pro-rata (rounds down per user)
  if (publicCommitments <= remainingCapacity) {
      publicAccepted = publicDeposit;
  } else {
      publicAccepted = (publicDeposit * remainingCapacity) /
         publicCommitments;
      // ↑ Rounds down for EACH user
  }
  // code
}
```

- Withdrawal uses global accounting:

```
function withdrawRaised(uint256 amount, address to) external onlyOwner {
    uint256 available = totalAccepted - totalWithdrawn;
    require(amount <= available, "Withdraw: exceeds raise");
    // Owner can legally withdraw full totalAccepted
}
```

Impact

- If the owner withdraws `totalAccepted`, the contract may not have enough funds left to pay all refunds. The final `claimRefund()` calls revert.

- Global aggregates (`totalAccepted`) do not equal the sum of per-user accepted amounts actually used for refunds.

Recommendation

Calculate `totalAccepted` by iterating through all investors during settlement, summing their actual individual accepted amounts:

```
function _finalizeSettlement() internal {
    settlementFinalized = true;

    // Calculate bucket allocations first
    uint256 whitelistAllocated = totalPriorityCommitted <=
       whitelistReserve_
        ? totalPriorityCommitted
        : whitelistReserve_;

    uint256 remainingCapacity = saleConfig.maxRaise > whitelistAllocated
        ? saleConfig.maxRaise - whitelistAllocated
        : 0;

    uint256 publicCommitments = totalCommitted - whitelistAllocated;
    uint256 publicAllocated = publicCommitments <= remainingCapacity
        ? publicCommitments
        : remainingCapacity;

    priorityRaiseFilled = whitelistAllocated;
    publicRaiseFilled = publicAllocated;

    // Calculate actual totalAccepted by summing individual allocations
    // (requires tracking investor addresses, or accept the rounding loss
    // by making totalAccepted slightly lower than bucket math suggests)

    // Alternative: Round DOWN the bucket allocations to account for per-
       user rounding
    uint256 expectedRoundingLoss = estimateRoundingLoss();
    totalAccepted = whitelistAllocated + publicAllocated -
       expectedRoundingLoss;
}
```

Or enforce that the owner cannot withdraw the last few tokens to ensure refund solvency:

```
function withdrawRaised(uint256 amount, address to) external onlyOwner {
    require(settlementFinalized, "Withdraw: not settled");

    uint256 available = totalAccepted - totalWithdrawn;
    uint256 safetyBuffer = 1000; // Adjust based on expected user count
    require(amount <= available - safetyBuffer, "Leave buffer for
       rounding");

    totalWithdrawn += amount;
    purchaseToken.safeTransfer(to, amount);
}
```

The cleanest solution is to ensure `totalAccepted` accurately reflects the sum of individual accepted amounts, accounting for rounding losses.

Team Response

Fixed.

[L-03] Missing Parameter Validations in Initialization

Severity

Low Risk

Description

The initialization function does not validate that `pricePerToken > 0` and `maxRaise > 0`. These missing checks allow deployment with invalid configurations that either break view functions or make the entire sale useless.

If `pricePerToken = 0`, the `saleStatus()` view function always reverts due to division by zero, breaking all UIs and frontends.

If `maxRaise = 0`, the sale accepts deposits but allocates nothing. All users get 100% refunds after settlement, making the sale completely pointless.

Location of Affected Code

File: contracts/token_sales/HarTokenSale.sol#L96-L114

```
function initialize(address owner_, address purchaseToken_,
    SaleConfigStruct memory saleConfig_, SonarConfigStruct memory
    sonarConfig_) external initializer {
     require(purchaseToken_ != address(0), "Init: zero token");
     require(saleConfig_.whitelistReserve <= saleConfig_.maxRaise, "Init:
        reserve too high");
     require(saleConfig_.priorityMaxAmount > 0, "Init: zero priority cap")
        ;
     // Missing: require(saleConfig_.pricePerToken > 0, "Init: zero price
        ");
     // Missing: require(saleConfig_.maxRaise > 0, "Init: zero max raise")
        ;

     saleConfig = saleConfig_;
}
```

File: contracts/token_sales/HarTokenSale.sol#L302

```
// pricePerToken = 0 breaks this:
function saleStatus(address investor) external view returns (...) {
    allocated = accepted / saleConfig.pricePerToken;  // ← Division by
        zero
}
```

File: contracts/token_sales/HarTokenSale.sol#L187-L196

```
// maxRaise = 0 makes this useless:
function _finalizeSettlement() internal {
    uint256 remainingCapacity = saleConfig.maxRaise > whitelistAllocated
        ? saleConfig.maxRaise - whitelistAllocated
        : 0;  // Always 0 if maxRaise = 0
    // Result: totalAccepted = 0
}
```

Impact

With `pricePerToken = 0`:

- All `saleStatus()` calls revert
- Frontends can't display user allocations
- Severely degraded UX (core functions still work but no visibility)

With `maxRaise = 0`:

- Sale accepts deposits but allocates zero tokens
- All users waste gas claiming 100% refunds
- Protocol wastes deployment costs and reputation damage

Both are easy deployment mistakes that should be caught at initialization.

Recommendation

Add both validations during initialization:

```
function initialize(...) external initializer {
    require(purchaseToken_ != address(0), "Init: zero token");
    require(saleConfig_.pricePerToken > 0, "Init: zero price");
    require(saleConfig_.maxRaise > 0, "Init: zero max raise");
    require(saleConfig_.whitelistReserve <= saleConfig_.maxRaise, "Init:
        reserve too high");
    require(saleConfig_.priorityMaxAmount > 0, "Init: zero priority cap")
        ;
    require(saleConfig_.saleOpen < saleConfig_.settleTime, "Init: sale
        opens after settlement");
    // code
}
```

Team Response

Fixed.

[L-04] Permit Signer Can Be Changed Mid-Sale, Invalidating All Existing Permits

Severity

Low Risk

Description

The `setPermitSigner()` function can be called at any time before settlement, including while the sale is actively running. Changing the signer immediately invalidates all previously issued permits because their signatures will no longer match the new signer. Users holding valid (not expired) permits will suddenly face transaction reverts until they obtain newly signed permits.

Unlike `setWhitelistReserve()`, which only works before settlement, there's no restriction on when the signer can be changed.

Location of Affected Code

File: contracts/token_sales/HarTokenSale.sol#L247-L250

```
function setPermitSigner(address newSigner) external onlyOwner {
    sonarConfig.permitSigner = newSigner;  // No time or state
        restrictions
    emit PermitSignerUpdated(newSigner);
}
```

File: contracts/token_sales/HarTokenSale.sol#L241

```
// Compare with:
function setWhitelistReserve(uint256 newReserve) external onlyOwner {
    require(!settlementFinalized, "Config: settled");  // ← Has
        restriction
    // code
}
```

Impact

- Instant DoS for all users with existing permits
- If rotation happens close to `settleTime`, users may be completely locked out
- No grace period or dual-signer support
- Users must detect the change and request new permits from the backend

Example scenario:

1. Sale runs for 7 days, 100 users have valid permits
2. Day 6: Admin rotates signer (security incident, operational change, etc.)
3. All 100 users' purchase attempts start failing
4. Users scramble to get new permits in the last 24 hours
5. Some miss the window entirely

Recommendation

Add restrictions to prevent mid-sale changes:

```
function setPermitSigner(address newSigner) external onlyOwner {
    require(!settlementFinalized, "Config: settled");
    require(block.timestamp < saleConfig.saleOpen, "Sale: already started
        ");
    sonarConfig.permitSigner = newSigner;
    emit PermitSignerUpdated(newSigner);
}
```

Alternatively, implement a dual-signer grace period or time-locked rotation to avoid hard breaks.

Team Response

Acknowledged.

[L-05] Initialize Function Can Be Frontrunned to Become the Owner

Severity

Low Risk

Description

By frontrunning the `initialize()` function, the attacker controls: 1. Owner Role 2. `setWhitelistReserve()` 3. `setPermitSigner()` 4. `withdrawRaised()` 5. `_authorizeUpgrade()`

Location of Affected Code

File: contracts/token_sales/HarTokenSale.sol#L96-L114

```
function initialize(
    address owner_,
    address purchaseToken_,
    SaleConfigStruct memory saleConfig_,
    SonarConfigStruct memory sonarConfig_
) external initializer {
    require(purchaseToken_ != address(0), "Init: zero token");
    require(saleConfig_.whitelistReserve <= saleConfig_.maxRaise, "Init:
        reserve too high");
    require(saleConfig_.priorityMaxAmount > 0, "Init: zero priority cap")
        ;
    require(saleConfig_.saleOpen < saleConfig_.settleTime, "Init: sale
        opens after settlement");

    __Ownable_init(owner_);
    __UUPSUpgradeable_init();

    purchaseToken = IERC20Metadata(purchaseToken_);
    priorityMaxAmountWithDecimals = saleConfig_.priorityMaxAmount * (10
        ** purchaseToken.decimals());
    saleConfig = saleConfig_;
    sonarConfig = sonarConfig_;
}
```

Impact

Prevents Initialization

Recommendation

The best practice here and the correct fix is using `_disableInitializers()` + atomic proxy initialization. The Proxy must be deployed with initialization calldata in the same transaction, so the proxy is never left uninitialized and cannot be frontrun.

Team Response

Fixed.

[L-06] Settlement Will Not Be Finalised Unless Purchase Tokens Are Transferred to the Contract Before Finalisation of the Settlement

Severity

Low Risk

Description

In the fix from the previous code, the protocol added a treasury address:

```
address public treasury;
```

During purchase, they are transferring the whole `purchaseToken` from the `msg.sender` to the treasury address:

```
purchaseToken.safeTransferFrom(msg.sender, treasury, amount);
```

This directly transfers all the purchase tokens to the treasury address, which previously was transferred to the sale contract and then was later withdrawn from withdrawRaised().

This change was done to directly transfer the purchase tokens instead of first transferring to the contract and then later withdrawing.

Later, the team intend to transfer those purchase tokens to the contract before calling `finalizeSettlement` and after that, refunds can be called.

However, if the team fails to transfer those purchase tokens from the treasury back to the contract or delays it for any reason, the settlement can't be finalised, and so the refunds can't be claimed.

Location of Affected Code

File: contracts/token_sales/HarTokenSale.sol#L227-L244

```
function claimRefund() external {
    require(settlementFinalized, "Refund: not settled");
    require(!investors[msg.sender].refundClaimed, "Refund: already
        claimed");

    uint256 commitment = investors[msg.sender].committed;
    require(commitment > 0, "Refund: no commitment");

    uint256 accepted = _calculateAcceptedCommitment(msg.sender);
    uint256 refund = commitment - accepted;
    require(refund > 0, "Refund: nothing owed");

    investors[msg.sender].refundClaimed = true;
    investors[msg.sender].refundClaimedAmount = refund;
    totalRefundClaimed += refund;

    purchaseToken.safeTransfer(msg.sender, refund); //@audit this will
        always fail
    emit RefundClaimed(msg.sender, refund);
}
```

Impact

Refunds can't be claimed since the settlement can't be finalised.

Recommendation

Ensure sending enough purchase tokens to the contract to finalise settlements.

Team Response

Acknowledged.

shieldify

Thank you!