PsiFi NFT Checkout Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Kage

SBSecurity (Blckhv, Slavcheww)

April 7, 2026

Contents

1 About Cyfrin **2**

2 Disclaimer **2**

3 Risk Classification **2**

4 Protocol Summary **2**

5 Audit Scope **2**

6 Executive Summary **3**

7 Findings **5**

7.1 Low Risk . . . 5

7.1.1 Missing `.gitignore` exposes deployment secrets to accidental git commit . . . 5

7.1.2 `MINTER_ROLE` grant to sales contract is non-atomic with deployment . . . 5

7.1.3 Deployment scripts use plaintext private keys via environment variables instead of Foundry encrypted keystore . . . 5

7.2 Informational . . . 7

7.2.1 Upgradeable contracts use non-upgradeable OpenZeppelin base contracts with sequential storage and no gap reservations . . . 7

7.2.2 No post-deployment verification in deployment script . . . 8

7.2.3 Missing events for `collection` and `paymentToken` initialization . . . 8

7.2.4 Inconsistent pragma version in `PsiFiTokenId` library . . . 9

7.2.5 All privileged roles are granted to a single address at initialization . . . 9

7.3 Gas Optimization . . . 11

7.3.1 Use transient storage for the reentrancy guard . . . 11

7.3.2 Cache `paymentToken` storage variable to avoid redundant SLOADs . . . 11

7.3.3 Revert directly inside `mintBatchTo` validation loop instead of using boolean flags . . . 12

7.3.4 Do not initialize loop counter to default value . . . 12

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the in-scope code being audited.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

PsiFi is a provider-neutral NFT checkout system deployed on Polygon. It allows end users to purchase ERC-1155 NFTs using USDC through third-party payment providers (Transak, Crossmint, or PsiFi ops).

The protocol consists of two UUPS-upgradeable proxy contracts:

- **PsiFiPolygonUSDCSalesUpgradeable** — validates EIP-712 signed purchase quotes, pulls USDC from payers, routes payments to merchant settlement addresses, and triggers minting. Supports single and batch purchases with replay protection via unique order IDs.
- **PsiFiSharedERC1155Upgradeable** — an upgradeable ERC-1155 collection that mints NFTs to buyer-specified recipient addresses, decoupled from payout wallets.

Access control is role-based (OpenZeppelin AccessControl) with roles for admin, upgrader, minter, pauser, signer management, and URI management. The sales contract starts paused post-deployment and requires explicit MINTER_ROLE grant and unpause before operation.

Token IDs use a deterministic 256-bit encoding scheme where the lower 192 bits encode merchant and product identifiers, with metadata interpretation handled off-chain.

The audit focused on the diff between the initial commit and the latest changes, covering both core contracts, the deployment script, and associated role/upgrade logic.

5 Audit Scope

The audit scope was limited to:

```
src/libraries/PsiFiTokenId.sol
src/PsiFiPolygonUSDCSalesUpgradeable.sol
src/PsiFiSharedERC1155.sol
src/PsiFiSharedERC1155Upgradeable.sol
```

6 Executive Summary

Over the course of 3 days, the Cyfrin team conducted an audit on the NFT Checkout code provided by PsiFi. The findings consist of 3 Low severity issues with the remainder being informational and gas optimizations.

Summary

Project Name	NFT Checkout
Repository	psifi-shared-erc1155
Commit	b4ef4287ee42...
Fix Commit	5b706d1a0ebc...
Audit Timeline	Apr 1st - Apr 3rd, 2026
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	0
Low Risk	3
Informational	5
Gas Optimizations	4
Total Issues	12

Summary of Findings

[L-1] Missing `.gitignore` exposes deployment secrets to accidental git commit	Acknowledged
[L-2] `MINTER_ROLE` grant to sales contract is non-atomic with deployment	Resolved
[L-3] Deployment scripts use plaintext private keys via environment variables instead of Foundry encrypted keystore	Resolved
[I-1] Upgradeable contracts use non-upgradeable OpenZeppelin base contracts with sequential storage and no gap reservations	Resolved
[I-2] No post-deployment verification in deployment script	Resolved
[I-3] Missing events for `collection` and `paymentToken` initialization	Resolved
[I-4] Inconsistent pragma version in `PsiFiTokenId` library	Resolved
[I-5] All privileged roles are granted to a single address at initialization	Resolved
[G-1] Use transient storage for the reentrancy guard	Resolved

[G-2] Cache `paymentToken` storage variable to avoid redundant SLOADs	Resolved
[G-3] Revert directly inside `mintBatchTo` validation loop instead of using boolean flags	Resolved
[G-4] Do not initialize loop counter to default value	Resolved

7 Findings

7.1 Low Risk

7.1.1 Missing `.gitignore` exposes deployment secrets to accidental git commit

Description: The project root has no `.gitignore` file. The deployment scripts read sensitive values from environment variables (`DEPLOYER_PRIVATE_KEY`, `ADMIN_PRIVATE_KEY`, `POLYGONSCAN_API_KEY`) which are typically stored in a `.env` file following standard Foundry workflow. Without a `.gitignore`, a `.env` file created in the project root will be tracked by git.

```
// DeployPsiFiCheckoutV1.s.sol:20
uint256 deployerPrivateKey = vm.envUint("DEPLOYER_PRIVATE_KEY");

// foundry.toml:24
polygon_amoy = { key = "${POLYGONSCAN_API_KEY}" }
```

Impact: If the deployer private key is leaked via an accidental commit, an attacker could impersonate the deployer. While the deployer does not retain admin roles after deployment, the deployer wallet likely holds funds for gas and future deployments.

Recommended Mitigation: Add a `.gitignore` at the project root:

```
.env
.env.*
out/
cache/
broadcast/
```

PsiFi: Acknowledged.

7.1.2 `MINTER_ROLE` grant to sales contract is non-atomic with deployment

Description: The `MINTER_ROLE` grant to the sales proxy is performed in a separate broadcast block in `DeployPsiFiCheckoutV1::run` (lines 58-61), requiring `ADMIN_PRIVATE_KEY` to be provided. If this env var is not set, the grant is skipped with only a console log warning (lines 63-65). This creates a window after deployment where the sales contract is fully configured and unpaused, but `PsiFiPolygonUSDCSalesUpgradeable::purchaseWithUSDC` and `purchaseBatchWithUSDC` will revert when calling `PsiFiSharedERC1155Upgradeable::mintTo` or `mintBatchTo` because the sales contract lacks `MINTER_ROLE`.

Impact: Temporary DoS on the sales contract between deployment and role grant. No funds at risk due to atomic transaction revert — if the mint fails, the USDC transfer is also reverted. The severity is bounded because the protocol controls when to publicize the sales address.

Recommended Mitigation: Either grant `MINTER_ROLE` within the deployer's broadcast block by having the deployer be the `initialDefaultAdmin`, or deploy the sales contract in a paused state and only unpause after the role grant is confirmed.

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.1.3 Deployment scripts use plaintext private keys via environment variables instead of Foundry encrypted keystore

Description: `DeployPsiFiCheckoutV1` loads deployer and admin private keys as plaintext environment variables via `vm.envUint("DEPLOYER_PRIVATE_KEY")` and `vm.envOr("ADMIN_PRIVATE_KEY", ...)`. Even when stored in a gitignored `.env` file, plaintext private keys on disk are vulnerable to: (a) accidental exposure via shell history, process environment inspection, or log files; (b) compromise if the developer machine is accessed; (c) leaking through CI/CD logs or environment dumps.

```
// DeployPsiFiCheckoutV1.s.sol:20-21
uint256 deployerPrivateKey = vm.envUint("DEPLOYER_PRIVATE_KEY");
uint256 adminPrivateKey = vm.envOr("ADMIN_PRIVATE_KEY", uint256(0));
```

Foundry provides an encrypted keystore via `cast wallet import` that encrypts the key at rest with a password. The key is never exposed in plaintext files, environment variables, or process memory beyond the broadcast window.

Impact: If the deployer private key is compromised, an attacker gains control of the deployer address. While the deployer does not retain admin roles after deployment, the address likely holds funds for gas and future deployments and could be reused across chains.

Recommended Mitigation: Deployment scripts currently assume env-var based private key loading. This is a common Foundry workflow but offers weaker secret-at-rest protections than encrypted keystore usage. Consider migrating operational deployment workflows to Foundry keystores :

```
# One-time setup: import key into encrypted keystore
cast wallet import deployer --interactive

# Deploy using the named account - prompts for password, key never in plaintext
forge script script/DeployPsiFiCheckoutV1.s.sol --account deployer --broadcast
```

Update the deployment script to use `vm.startBroadcast()` (no private key argument) instead of `vm.startBroadcast(deployerPrivateKey)`:

```
- uint256 deployerPrivateKey = vm.envUint("DEPLOYER_PRIVATE_KEY");
- vm.startBroadcast(deployerPrivateKey);
+ vm.startBroadcast();
```

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.2 Informational

7.2.1 Upgradeable contracts use non-upgradeable OpenZeppelin base contracts with sequential storage and no gap reservations

Description: Both UUPS-upgradeable proxy contracts inherit from non-upgradeable OZ v5 contracts (`AccessControl`, `Pausable`, `EIP712`, `ERC1155`, `ERC1155Burnable`, `ERC1155Pausable`, `ERC1155Supply`) which use sequential storage slots starting at slot 0 with no `__gap` reservations or ERC-7201 namespaced storage. If a V2 implementation changes inheritance order, switches to upgradeable OZ variants, or inserts a new base contract with storage, all slot assignments shift and proxy state is silently corrupted.

Collection proxy (`PsiFiSharedERC1155Upgradeable`) storage layout:

- slot 0: `AccessControl._roles`
- slot 1: `ERC1155._balances`
- slot 2: `ERC1155._operatorApprovals`
- slot 3: `ERC1155._uri`
- slot 4: `Pausable._paused`
- slot 5: `ERC1155Supply._totalSupply`
- slot 6: `ERC1155Supply._totalSupplyAll`
- slot 7: `_baseMetadataUri`

Sales proxy (`PsiFiPolygonUSDCSalesUpgradeable`) storage layout:

- slot 0: `AccessControl._roles`
- slot 1: `Pausable._paused`
- slot 2: `EIP712._nameFallback`
- slot 3: `EIP712._versionFallback`
- slot 4: `collection`
- slot 5: `paymentToken`
- slot 6: `quoteSigner`
- slot 7: `usedOrderIds`
- slot 8: `_reentrancyStatus`

Impact: A storage-incompatible upgrade could corrupt persistent proxy state. Examples of possible consequences include:

- loss or corruption of ERC-1155 balances, approvals, or supply tracking
- corruption of access control state
- corruption of sales contract configuration such as collection, paymentToken, or quoteSigner
- corruption of usedOrderIds, which could invalidate replay protection for still-valid signed quotes

Because this requires a privileged upgrade action, the issue is best understood as upgrade-safety risk rather than a present user-triggerable vulnerability.

Proof of Concept:

1. At block N, user calls `purchaseWithUSDC` with a valid quote. `usedOrderIds[orderId]` set to `true` at `keccak256(orderId . 7)`
2. Admin deploys V2 that shifts storage layout (e.g., switching `AccessControl` to `AccessControlUpgradeable` which uses ERC-7201), calls `upgradeToAndCall`
3. `usedOrderIds` mapping silently moves to a different slot. All entries from old slot read as `false`

4. Attacker extracts signature from step 1 calldata and calls `purchaseWithUSDC` with identical parameters. `usedOrderIds[orderId]` returns `false` (reading from new empty slot). Signature still verifies. `expiresAt` check passes if within window
5. Duplicate purchase succeeds — attacker pays USDC and recipient receives duplicate NFTs

Recommended Mitigation: Consider replacing the non-upgradeable OpenZeppelin imports with their `@openzeppelin/contracts-upgradeable` equivalents which use ERC-7201 namespaced storage:

```
- import {AccessControl} from "@openzeppelin/contracts/access/AccessControl.sol";
- import {Pausable} from "@openzeppelin/contracts/utils/Pausable.sol";
- import {EIP712} from "@openzeppelin/contracts/utils/cryptography/EIP712.sol";
+ import {AccessControlUpgradeable} from
↪  "@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol";
+ import {PausableUpgradeable} from "@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol";
+ import {EIP712Upgradeable} from
↪  "@openzeppelin/contracts-upgradeable/utils/cryptography/EIP712Upgradeable.sol";
```

These provide isolated storage namespaces per contract and are designed for proxy patterns. Alternatively, document the exact inheritance order and slot map as a binding constraint for all future upgrades and add storage layout verification tests to CI.

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.2.2 No post-deployment verification in deployment script

Description: `DeployPsiFiCheckoutV1::run` logs proxy addresses via `console2.log` but does not read back on-chain state to verify the deployment was correct. There are no assertions that the proxy's implementation slot points to the correct implementation, that admin roles are assigned, that `quoteSigner` was set, that the `collection` address in the sales contract matches the deployed collection proxy, or that `paymentToken` was set correctly.

Impact: Deployment errors (wrong token address, wrong signer, wrong admin) go undetected until a user interaction fails. A wrong `paymentTokenAddress` would cause the sales contract to pull the wrong ERC20 from buyers. A wrong `quoteSigner` would make all purchases fail (DoS) until corrected.

Recommended Mitigation: Add verification reads after deployment:

```
// After vm.stopBroadcast()
require(collection.hasRole(collection.DEFAULT_ADMIN_ROLE(), initialDefaultAdmin), "admin role missing");
require(address(sales.collection()) == address(collectionProxy), "collection mismatch");
require(address(sales.paymentToken()) == paymentTokenAddress, "payment token mismatch");
require(sales.quoteSigner() == initialQuoteSigner, "signer mismatch");
```

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.2.3 Missing events for `collection` and `paymentToken` initialization

Description: `PsiFiPolygonUSDCSalesUpgradeable::initialize` sets `collection` and `paymentToken` storage variables but does not emit events for these state changes. While `quoteSigner` correctly emits `QuoteSignerSet`, no analogous events exist for the other two critical configuration parameters. This makes it harder for off-chain systems to index and verify the initial configuration of deployed proxies.

```
// PsiFiPolygonUSDCSalesUpgradeable.sol:118-127
collection = IPsiFiSharedERC1155(collectionAddress);
paymentToken = IERC20(paymentTokenAddress);
quoteSigner = initialQuoteSigner;
// ...
emit QuoteSignerSet(initialQuoteSigner);  // no events for collection or paymentToken
```

Recommended Mitigation: Add and emit `CollectionSet` and `PaymentTokenSet` events in `initialize`.

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.2.4 Inconsistent pragma version in `PsiFiTokenId` library

Description: `PsiFiTokenId.sol` uses a floating pragma `^0.8.20` while all other in-scope contracts use a pinned pragma `0.8.30`. This inconsistency could lead to the library being compiled with a different compiler version than the contracts that consume it in certain build configurations.

```
// PsiFiTokenId.sol:2
pragma solidity ^0.8.20;

// All other contracts:
pragma solidity 0.8.30;
```

Recommended Mitigation: Pin the library pragma to match the rest of the codebase:

```
- pragma solidity ^0.8.20;
+ pragma solidity 0.8.30;
```

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.2.5 All privileged roles are granted to a single address at initialization

Description: Across all three contracts, every privileged role is granted to the same `initialDefaultAdmin` address during construction or initialization.

In `PsiFiSharedERC1155Upgradeable`, the `initialize` function grants `DEFAULT_ADMIN_ROLE`, `MINTER_ROLE`, `PAUSER_ROLE`, `URI_MANAGER_ROLE`, and `UPGRADER_ROLE` to `initialDefaultAdmin`:

PsiFiSharedERC1155Upgradeable.sol#L64-L68

```
_grantRequiredRole(DEFAULT_ADMIN_ROLE, initialDefaultAdmin);
_grantRequiredRole(MINTER_ROLE, initialDefaultAdmin);
_grantRequiredRole(PAUSER_ROLE, initialDefaultAdmin);
_grantRequiredRole(URI_MANAGER_ROLE, initialDefaultAdmin);
_grantRequiredRole(UPGRADER_ROLE, initialDefaultAdmin);
```

In `PsiFiPolygonUSDCSalesUpgradeable`, the `initialize` function grants `DEFAULT_ADMIN_ROLE`, `PAUSER_ROLE`, `SIGNER_MANAGER_ROLE`, and `UPGRADER_ROLE` to `initialDefaultAdmin`:

PsiFiPolygonUSDCSalesUpgradeable.sol#L122-L125

```
_grantRequiredRole(DEFAULT_ADMIN_ROLE, initialDefaultAdmin);
_grantRequiredRole(PAUSER_ROLE, initialDefaultAdmin);
_grantRequiredRole(SIGNER_MANAGER_ROLE, initialDefaultAdmin);
_grantRequiredRole(UPGRADER_ROLE, initialDefaultAdmin);
```

In `PsiFiSharedERC1155`, the constructor grants `MINTER_ROLE`, `PAUSER_ROLE`, and `URI_MANAGER_ROLE` to `initialDefaultAdmin` (who also holds `DEFAULT_ADMIN_ROLE` via `AccessControlDefaultAdminRules`):

PsiFiSharedERC1155.sol#L39-L41

```
_grantRequiredRole(MINTER_ROLE, initialDefaultAdmin);
_grantRequiredRole(PAUSER_ROLE, initialDefaultAdmin);
_grantRequiredRole(URI_MANAGER_ROLE, initialDefaultAdmin);
```

Since `DEFAULT_ADMIN_ROLE` is the admin role for all other roles, this single address can also grant and revoke any role. If this address is compromised, an attacker gains full control over the protocol: minting arbitrary tokens, pausing/unpausing contracts, changing the metadata URI, rotating the quote signer, upgrading both contracts to malicious implementations, and sweeping native balances.

Impact: Compromise of the single admin address gives an attacker complete control over the protocol, including the ability to mint unlimited tokens, upgrade contracts to arbitrary implementations, redirect sales payout wallets (via quote signer rotation), and drain native balances.

Recommended Mitigation: Distribute roles across separate addresses at initialization. At minimum, separate the `UPGRADER_ROLE` and `DEFAULT_ADMIN_ROLE` from the operational roles (`MINTER_ROLE`, `PAUSER_ROLE`, `URI_MANAGER_ROLE`, `SIGNER_MANAGER_ROLE`). Use a multisig or timelock for the `DEFAULT_ADMIN_ROLE` and `UPGRADER_ROLE`.

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.3 Gas Optimization

7.3.1 Use transient storage for the reentrancy guard

Description: `PsiFiPolygonUSDCSalesUpgradeable` implements a custom storage-based reentrancy guard (`_reentrancyStatus`) that costs ~5,000 gas on the warm SSTORE reset path per transaction. Since the contract targets Solidity 0.8.30 and deploys on Polygon (which supports EIP-1153 transient storage via the Dencun upgrade), the guard can use the `transient` keyword instead, saving ~2,900 gas per guarded call. Transient storage automatically resets to zero at the end of each transaction at near-zero cost.

```
// PsiFiPolygonUSDCSalesUpgradeable.sol
uint256 private _reentrancyStatus;       // line 94
_reentrancyStatus = 1;                   // line 116
_nonReentrantBefore();                   // lines 186, 211
_nonReentrantAfter();                    // lines 193, 213
```

Recommended Mitigation: Add the `transient` keyword to the variable declaration. Adjust the guard logic to use 0 as "unlocked" (the transient default) and 1 as "locked". Remove the `_reentrancyStatus = 1` initialization from `initialize` and the body of `_nonReentrantAfter` since transient storage auto-resets each transaction:

```
- uint256 private _reentrancyStatus;
+ uint256 private transient _reentrancyStatus;

  // In initialize():
- _reentrancyStatus = 1;

  function _nonReentrantBefore() internal {
-     if (_reentrancyStatus == 2) {
+     if (_reentrancyStatus == 1) {
          revert ReentrantCall();
      }
-     _reentrancyStatus = 2;
+     _reentrancyStatus = 1;
  }

// consider removing the `_nonReentrantAfter` function since
// transient storage auto-resets to 0 at end of transaction. Otherwise
// if still needed, just set the flag to zero instead of 1
  function _nonReentrantAfter() internal {
-     _reentrancyStatus = 1;
+     _reentrancyStatus = 0;
  }
```

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.3.2 Cache `paymentToken` storage variable to avoid redundant SLOADs

Description: In both `PsiFiPolygonUSDCSalesUpgradeable::purchaseWithUSDC` and the batch purchase flow, `paymentToken` is read from storage twice on the success path — once for its address in signature verification and once for the token transfer. Each additional warm SLOAD costs 100 gas.

```
// PsiFiPolygonUSDCSalesUpgradeable.sol
paymentTokenAddress: address(paymentToken)  // line 178 - 1st SLOAD
paymentToken.safeTransferFrom(...)          // line 190 - 2nd SLOAD
```

Recommended Mitigation: Cache `paymentToken` into a local variable at the start of each external function:

```
  function purchaseWithUSDC(...) external whenNotPaused {
+     IERC20 paymentToken_ = paymentToken;
      // use paymentToken_ throughout
```

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.3.3 Revert directly inside `mintBatchTo` validation loop instead of using boolean flags

Description: `PsiFiSharedERC1155Upgradeable::mintBatchTo` uses three local variables (`hasInvalidTokenId`, `invalidTokenId`, `hasInvalidQuantity`) as flags during the validation loop, then checks them after the loop. Since each branch already `break`s immediately, the function can revert directly inside the loop, eliminating the flag variables and the post-loop conditional checks.

```
// PsiFiSharedERC1155Upgradeable.sol:93-114
bool hasInvalidTokenId = false;
uint256 invalidTokenId = 0;
bool hasInvalidQuantity = false;
for (uint256 i = 0; i < tokenIds.length; ++i) {
    if (!PsiFiTokenId.isValidV1(tokenIds[i])) {
        hasInvalidTokenId = true;
        invalidTokenId = tokenIds[i];
        break;
    }
    // ...
}
```

Recommended Mitigation: Replace the flag pattern with direct reverts inside the loop:

```
  for (uint256 i = 0; i < tokenIds.length; ++i) {
      if (!PsiFiTokenId.isValidV1(tokenIds[i])) {
-         hasInvalidTokenId = true;
-         invalidTokenId = tokenIds[i];
-         break;
+         revert InvalidTokenId(tokenIds[i]);
      }
      if (quantities[i] == 0) {
-         hasInvalidQuantity = true;
-         break;
+         revert InvalidQuantity();
      }
  }
```

PsiFi: Fixed in 11599db.

Cyfrin: Verified.

7.3.4 Do not initialize loop counter to default value

Description: Solidity defaults `uint256` to `0`, so `uint256 i = 0` in for-loop initializers is redundant.

Affected locations:

- `PsiFiPolygonUSDCSalesUpgradeable.sol`: lines 340, 385, 424
- `PsiFiSharedERC1155Upgradeable.sol`: line 97

Recommended Mitigation: Remove the explicit `= 0` initialization:

```
- for (uint256 i = 0; i < lineItems.length; ++i) {
+ for (uint256 i; i < lineItems.length; ++i) {
```

PsiFi: Fixed in 11599db.

Cyfrin: Verified.