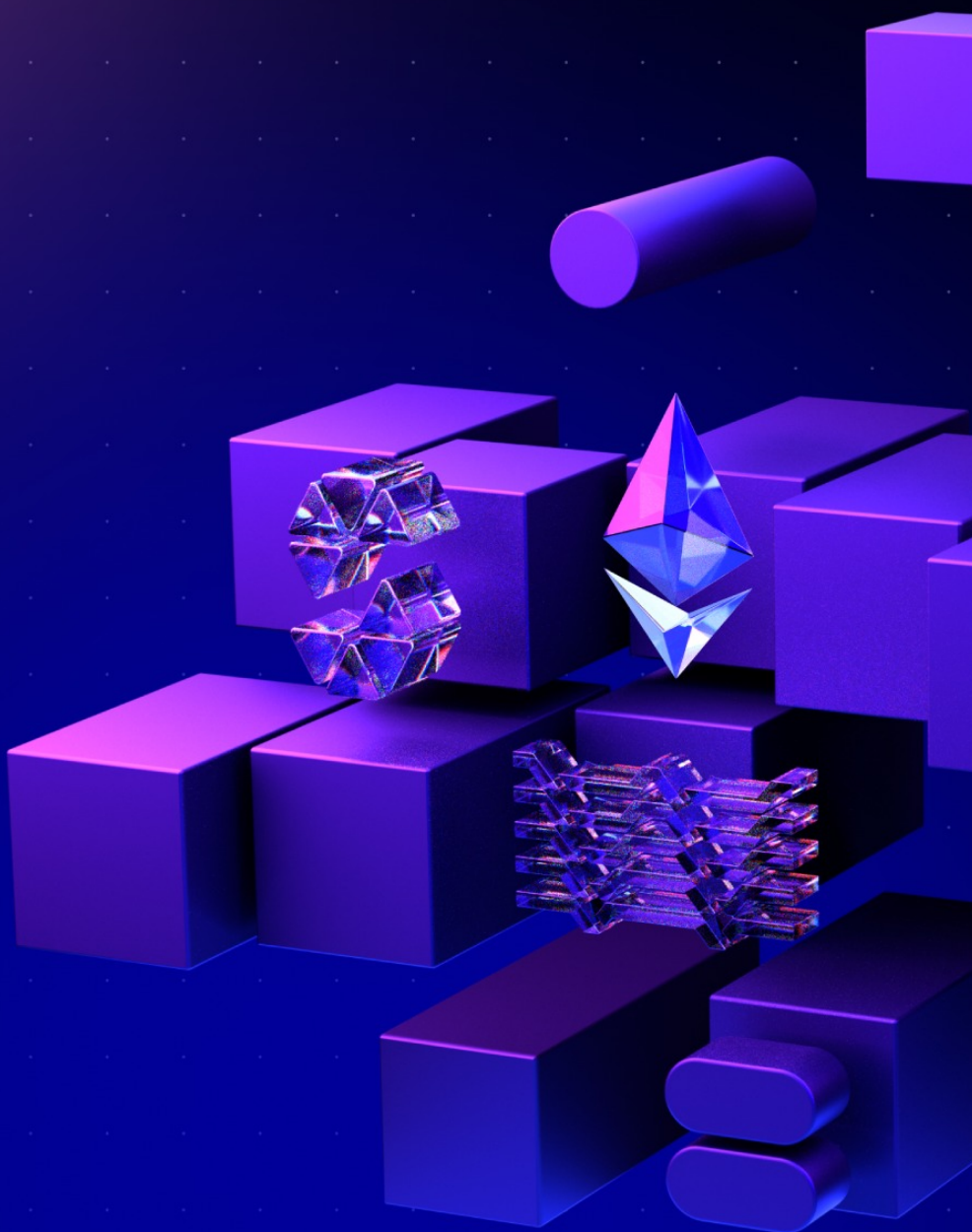
ackee
blockchain security

Printr

Protocol

8.12.2025



Audit Report

Contents

1. Document Revisions

1.0	Draft Report	20.09.2025
1.1	Final Report	14.10.2025
2.0-draft	Draft Report	17.10.2025
2.1-draft	Draft Report	24.10.2025
2.1	Final Report	11.11.2025
2.2	Final Report	08.12.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**
   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**
   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**
   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**
   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**
   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

6. **Wake-AI assisted vulnerability discovery**
   As the last step, the scope is checked against Wake AI, an LLM-powered audit tool, to identify potentially missed vulnerabilities. This step is executed at the end of the audit process to avoid distracting auditors from manual review.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Dmytro Khimchenko	Lead Auditor
Naoki Yoshida	Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Revision 1.0

Printr engaged Ackee Blockchain Security to perform a security review of Printr Protocol with a total time donation of 23 engineering days in a period between August 14 and October 1, 2025, with Dmytro Khimchenko as the lead auditor.

The audit was performed on the commit `98127e5`[1] in the evm repository and the scope included all Solidity files in the `src` directory, except for:

- `src/interfaces/*`,
- `src/libs/*`,
- `src/liquidity/WAGMILiquidity.sol`,
- `src/liquidity/LynexLiquidity.sol`,
- `src/liquidity/TraderjoeLiquidity.sol`.

We began our review by implementing and executing manually-guided differential fuzz tests in Wake testing framework to verify the correctness of the system. Our workflow consisted of the following steps:

1. We implemented fuzzing of the local bonding curve of the telecoin before its graduation. The fuzz test yielded the H1 issue.
2. We continued with fuzzing graduation of the telecoin and deploying it via using different liquidity modules and discovered the C1 issue.
3. We fuzzed integration with Axelar Interchain Token Service and LayerZero and discovered the other severe findings C2, C4, C5 C7.

More details about the fuzzing process can be found in the Fuzzing section of Report Revision 1.0. The list of the implemented flows and invariants is

available in the Appendix B.

In parallel, we performed a thorough manual review of the code, especially focusing on integration with other systems such as Axelar Interchain Token Service and LayerZero and discovered C3, C6, C7 issues.

During the review, we focused on the following aspects:

- correctness of the arithmetic of the system, specifically buying and selling of the telecoins;
- graduation of the telecoin and interaction with different liquidity modules;
- bridging of the telecoins to other chains via Axelar Interchain Token Service and LayerZero;
- ensuring there are no griefing attacks possible;
- ensuring there are no common issues such as data validation;
- ensuring approvals of the tokens are correctly consumed;
- detecting possible reentrancies in the code;
- ensuring access controls are not too relaxed or too strict; and
- ensuring interactions with other systems are correctly implemented.

All issues of possible high or critical severity were immediately reported to the Printr team.

Our review resulted in 37 findings, ranging from Info to Critical severity. The most severe findings are connected to integration with other projects such as Axelar Interchain Token Service, LayerZero and Uniswap. Remaining critical findings are connected to the access controls.

The code documentation has many discrepancies with the real codebase and should be rewritten as mentioned in the W4 issue. The issue C2 was discovered by the Printr team during the audit.

Ackee Blockchain Security recommends Printr:

- write integration tests for the communication with other projects such as Axelar Interchain Token Service, LayerZero and Uniswap;
- write consistent documentation for the codebase;
- implement a comprehensive test suite for single-chain and cross-chain scenarios;
- thoroughly validate the access controls for all externally accessible functions;
- read and review the complete audit report; and
- perform a new re-audit of the whole protocol.

The project is not ready for deployment in the current state.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

Printr engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision.

Printr provided a pull request with the fixes and provided comments to most of the acknowledged findings.

The review was performed between October 6 and October 7, 2025 on the commit `47667e5`[2].

From the reported 37 findings:

- 21 issues were fixed;
- 9 issues were acknowledged;
- 6 minor issues were fixed partially; and

- 1 minor issue remained unresolved.

All fixes were verified in isolation, focusing on whether the reported vulnerabilities and issues were properly addressed. We did not analyze the overall impact of the fixes on the system.

The fixes addressed the critical and major vulnerabilities reported in the previous audit, and the protocol demonstrates improved safety.

Due to the severity of the issues and the overall number of them, we cannot guarantee that the provided fixes did not introduce new vulnerabilities.

We have requested an additional time donation from Printr of 7 MDs to perform an additional review of the protocol.

We do not suggest deploying the protocol in the current state before the additional review.

Revision 2.0

Printr engaged Ackee Blockchain Security to perform a security review of Printr Protocol with a total time donation of 7 engineering days in a period between October 7 and October 21, 2025, with Dmytro Khimchenko as the lead auditor.

The audit was performed on the commit `47667e5`[3] in the contracts repository and the scope was the same as in the previous revision.

We began our review by rewriting the fuzzing test suites to reflect the fixes made. Additionally, we enriched the manually guided fuzzing test of the buy, sell and spend operations for the ERC20 tokens, which yilded the H4 issue and the C8 issue.

In parallel, we performed a manual review of the codebase and discovered the C9 finding.

The issue C8 has been since the revision 1.0. The issue C9 has appeared due to the changes in the LayerZero integration design.

We took a deep dive into the logic of the contracts. For testing and fuzzing, we have involved Wake testing framework. During the review, we paid special attention to:

- ensuring the integration with liquidity modules is correct;
- ensuring bridging between chains is correct;
- ensuring access controls are not too relaxed or too strict; and
- looking for common issues such as data validation.

After the discussion with the client, we decided to audit the commit hash with fixes and additional improvements of the codebase, as the Printr team has been developing the integration tests in parallel with the audit.

The protocol is not ready for the deployment yet. This revision can be considered as an intermediate.

Our review resulted in 10 findings, ranging from Info to Critical severity. The most severe one C8 and C9.

Ackee Blockchain Security recommends Printr:

- read and review the complete audit report; and
- address all identified issues.

The protocol is not ready for the deployment yet. This revision can be considered as an intermediate See Report Revision 2.0 for the system overview and trust model.

Revision 2.1

Printr engaged Ackee Blockchain Security to perform a fix review of the

findings from the previous revision with a total time donation of 2 engineering days.

Printr provided a pull request with the fixes and provided comments to most of the acknowledged findings.

The review was performed between October 20 and October 24, 2025 on the commit `1e735e5`[4].

In addition to the review of fixes of initial findings, we performed additional review of the protocol overall. This resulted in new findings in the revision with the most severe one being C10. Most of the findings were reported to the client during the audit.

The critical issue has appeared due to the changes in the protocol design from revision 2.0.

Additionally, we updated the fuzzing test suites to reflect the fixes made and design changes to the protocol.

In the process, we took a deep dive into the logic of the contracts and their integrations with other protocols. For testing and fuzzing, we involved Wake testing framework. During the review, we paid special attention to:

- ensuring the integration with liquidity modules is correct;
- ensuring bridging between chains is correct;
- ensuring access controls are not too relaxed or too strict; and
- looking for common issues such as data validation.

At the end of the review, we engaged the Wake AI tool, which discovered the following issues: L10, W15, I14, I15, I13. Table with the discovered issues can be found in the Appendix C.

From the reported 10 findings:

- 6 issues were fixed; and
- 4 issues were acknowledged;

Our review resulted in 13 findings, ranging from Info to Critical severity. The most severe one was C10.

Ackee Blockchain Security recommends Printr:

- read and review the complete audit report; and
- address all identified issues.

Do not deploy the protocol in the current state without additional fix review.

See Report Revision 2.1 for the update of the system overview and trust model.

Revision 2.2

Printr engaged Ackee Blockchain Security to perform a fix review of the findings from the previous revision.

Printr provided a pull request with the fixes and provided comments to most of the acknowledged findings.

The review was performed between November 21 and October 24, 2025 on the commit `8973230`[5].

From the reported 13 findings:

- 10 issues were fixed;
- 1 issue was partially fixed; and
- 2 issues were acknowledged.

No new findings were reported.

There is nothing more we could find as a blocker for deployment.

However, due to the number and severity of findings across all revisions, we recommend considering an additional audit by one more independent team.

[1] full commit hash: `98127e5da3df1d3c3f0912ae79d539058012bcb5`

[2] full commit hash: `47667e52c7e3396a2458687fab5daa3b94177c5f`

[3] full commit hash: `47667e52c7e3396a2458687fab5daa3b94177c5f`

[4] full commit hash: `1e735e551f621849e02b2d46c672331e85cafd87`

[5] full commit hash: `89732301ae5e20c0e4e3d0495af3ac9b89a6e120`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
10	4	5	10	15	16	**60**

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
C1: Native Token is not refunded after Uniswap trade	**Critical**	1.0	Fixed
C2: Incorrect fee calculation for bridging via ITS	**Critical**	1.0	Fixed
C3: `teleport` function always fails for the ITS transactions as all fees are transferred to the Printr	**Critical**	1.0	Fixed
C4: Inverted condition for creation of `PrintrTeleportingTelecoin`	**Critical**	1.0	Fixed

Finding title	Severity	Reported	Status
C5: Only sending to Solana peer address even for EVMs at LayerZero causes wrong receiver address	**Critical**	1.0	Fixed
C6: Missing access control in `witnessTeleport`	**Critical**	1.0	Fixed
C7: Recipient address is left-padded during encoding and right-padded during decoding for LayerZero	**Critical**	1.0	Fixed
H1: Loss of precision during conversion of `uint256` to `uint80`	**High**	1.0	Partially fixed
H2: `SELF` immutable address pointing to implementation contract	**High**	1.0	Fixed
H3: Lack of destination address verification	**High**	1.0	Acknowledged
M1: Unchecked call in `_refund` function	Medium	1.0	Fixed
M2: Variable `priceLimit` is ignored for specific cases	Medium	1.0	Fixed
M3: `teleportFrom` function does not work due to insufficient approval for `TokenManager` to burn tokens	Medium	1.0	Acknowledged
M4: Potential overflow on teleport amount conversion	Medium	1.0	Fixed

Finding title	Severity	Reported	Status
L1: Everyone can link telecoins to other chains	Low	1.0	Acknowledged
L2: Missing refund in `teleport` function	Low	1.0	Acknowledged
L3: Limited gas for ETH transfers causes withdrawal failures for smart contract wallets	Low	1.0	Fixed
L4: Possible overflow in `print` function	Low	1.0	Fixed
L5: Ambiguous error `LiquidityAlreadyDeployed`	Low	1.0	Fixed
W1: Incorrect value of `issuedSupply` in `_estimateTokenCost` function	Warning	1.0	Fixed
W2: Ignoring failure of calling `decimals` function of the base pair token	Warning	1.0	Fixed
W3: Approval for the ITS token manager is needed before teleporting	Warning	1.0	Acknowledged
W4: Documentation disrepency with code	Warning	1.0	Partially fixed
W5: No verification on which chain the user teleports tokens to	Warning	1.0	Acknowledged
W6: Code size is too large	Warning	1.0	Fixed
W7: Whitelisted addresses cannot be updated	Warning	1.0	Acknowledged

Finding title	Severity	Reported	Status
W8: Amount calculation uses `maxPrice` for calculation, resulting in operating with smaller amount of tokens	Warning	1.0	Reported
I1: No validation for zero `params.amount`	Info	1.0	Fixed
I2: No explanation for using specific constants for gas limit	Info	1.0	Fixed
I3: Misleading information in NatSpec	Info	1.0	Fixed
I4: Incorrect NatSpec title in `Printed.sol` contract	Info	1.0	Acknowledged
I5: Inconsistent contract names	Info	1.0	Acknowledged
I6: Unused errors	Info	1.0	Partially fixed
I7: Unused interfaces and libraries	Info	1.0	Partially fixed
I8: Unused events	Info	1.0	Partially fixed
I9: Unused functions	Info	1.0	Fixed
I10: Unused imports	Info	1.0	Partially fixed
C8: DOS when handling initial price fix due to incorrect liquidity parameter	**Critical**	2.0	Fixed
C9: CREATE2 deployment of LZChannel results in different addresses across chains due to varying lzEndpoint values	**Critical**	2.0	Fixed

Finding title	Severity	Reported	Status
H4: Overestimated base token amount required for ERC20 trades after graduation	**High**	2.0	Fixed
L6: Treasury accepts any NFT token instead of only LP position NFTs	Low	2.0	Acknowledged
W9: Merchantmoe Liquidity Module is not implemented correctly	Warning	2.0	Fixed
W10: Value overflow may limit realistic token configurations	Warning	2.0	Fixed
W11: Confusing parameter semantics in `_buy` function due to overloaded `params.amount` usage	Warning	2.0	Acknowledged
W12: HyperSwap behavior may differ from UniswapV3 due to modified source code	Warning	2.0	Acknowledged
I11: Missing permit invalidation mechanism	Info	2.0	Fixed
I12: Missing events for critical operations that distinguish behavior	Info	2.0	Acknowledged
C10: Removed approve consumption allows approve vulnerability	**Critical**	2.1	Fixed

Finding title	Severity	Reported	Status
M5: All Printed tokens are sent to the liquidity pool even if they are used as basePair tokens during graduation	Medium	2.1	Fixed
L7: Usage of transfer in Telecoin	Low	2.1	Fixed
L8: `priceAfter` is used as `effectivePrice`	Low	2.1	Fixed
L9: Inconsistent refund for ERC20 and native tokens	Low	2.1	Fixed
L10: permitWitnessCall forwards msg.value instead of signed call.nativeValue	Low	2.1	Acknowledged
W13: Attacker can create tokens with the same symbol on different chains to drain legitimate liquidity	Warning	2.1	Acknowledged
W14: Tokens are refunded and not spent on liquidity pool	Warning	2.1	Partially fixed
W15: Attacker can front-run the liquidity pool creation and make graduation fail	Warning	2.1	Acknowledged
I13: Withdraw forwards all gas despite 3000-gas comment	Info	2.1	Fixed
I14: `IntentProxy` constructor has `payable` parameter despite non-payable function	Info	2.1	Fixed
I15: `_isTeleporting` is not correctly described	Info	2.1	Fixed

Finding title	Severity	Reported	Status
I16: Token URI is not in base64 format as mentioned in the documentation	Info	2.1	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Dmytro Khimchenko	Lead Auditor
Naoki Yoshida	Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The Printr protocol is a platform for deploying ERC20 tokens, which are internally called telecoins. The protocol allows users to print telecoins on multiple chains with the same address. For that purpose, users need to call the `print` function with the same parameters on different chains. During the deployment process, the protocol creates a local bonding curve for the telecoin on the current chain. The bonding curve is a constant product curve used for buying and selling telecoins. Before deploying the telecoin, users must specify parameters for the telecoin such as initial price, maximum token supply, symbol, chains where the telecoin should be printed, base pair token, completion threshold after which the telecoin will be graduated, and other parameters. After deploying the telecoin, users can buy and sell telecoins on the current chain. Telecoins can also be teleported to other chains using the teleportation mechanism via Axelar network and LayerZero. After graduation of the telecoin, liquidity is deployed on Uniswap, PancakeSwap, Algebra, TraderJoe, or other DEX protocols. After deployment of liquidity, users can still use the Printr protocol to buy and sell telecoins.

Trust Model

Printr allows permissionless printing of telecoins, buying and selling of any telecoin, and teleportation of users' telecoins to other chains. Distribution of fees is a permissioned operation.

The protocol requires users to trust the Printr contract owner who maintains UUPS upgradeability privileges. Token creators must also be trusted, as they can:

- establish different base prices across chains, potentially enabling arbitrage;
- define custom base token pairs beyond standard options like WETH.

Issues related to access control are C6 and L1.

Fuzzing

Manually-guided differential fuzz tests were developed during the review to test the correctness and robustness of the system. The fuzz tests employ fork testing technique to test the system with external contracts exactly as they are deployed on the mainnet. This is crucial to detect any potential integration issues.

The differential fuzz tests keep their own Python state according to the system's specification. Assertions are used to verify the Python state against the on-chain state in contracts.

The list of all implemented execution flows and invariants is available in Appendix B.

The most severe findings that have been discovered by fuzzing are C1, C2, C4, C5 C7 by implementing in Wake testing framework.

Findings

The following section presents the list of findings discovered in this revision.

For the complete list of all findings, Go back to Findings Summary

C1: Native Token is not refunded after Uniswap trade

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PrintrTrading.sol	Type:	Logic error

Description

The Printr protocol uses Uniswap for routing the trades of the telecoin after its graduation. The Printr protocol allows partial fills for the trades on Uniswap. However, if the native token is used for the trade and the trade is partially filled, the native token stays in the Printr contract and is not refunded to the user.

This issue arises in the following functions:

- `PrintrTrading.buy`
- `PrintrTrading.spend`

Listing 1. Excerpt from PrintrTrading.buy

```
189 if (curve.completionThreshold == 0) {
190     if (maxPrice != 0) {
191         params.amount = amount * maxPrice / PRECISION;
192     } else {
193         if (msg.value > 0) {
194             params.amount = msg.value;
195         } else {
196             revert LiquidityAlreadyDeployed();
197         }
198     }
199
200     _spendInLiquidityPool(curve, params);
201     return params;
```

Listing 2. Excerpt from PrintrTrading.spend

```
238 if (curve.completionThreshold == 0) {
239     params.amount = _spendInLiquidityPool(curve, params);
240     return params;
241 }
```

As a result, the user loses the native token amount that has not been used for the trade.

Exploit scenario

1. Alice, a user, tries to buy a token that has graduated with the native token.
2. The trade is partially filled, and the native token stays in the Printr contract.
3. Alice loses the native token amount that has not been used for the trade.
4. Bob, a user who initiates a trade on a token that has not graduated, receives the native token amount that Alice lost during her trade.

Recommendation

Refund the native token amount that has not been used for the Uniswap trade.

Fix 1.1

The issue was fixed by the recommendation.

Go back to Findings Summary

C2: Incorrect fee calculation for bridging via ITS

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PrintrTeleport.sol	Type:	Logic error

Description

During calculation of the fees that the user needs to pay for the teleportation of a specific amount of telecoins via ITS, the fee is calculated incorrectly. The reason is the incorrect assignment of the values returned from the function.

Listing 3. Excerpt from PrintrTeleport

```
567 (uint256 convertedFee,,,,) = IPrintrTrading(address(
    this)).estimateTokenCost(tokenAddress, bipsAmount);
568
569 basePair = curve.basePair;
570
571 if (curve.basePair == wrappedNativeToken) {
572     // Base pair is native, add converted fee to native fee
573     nativeFee = flatFee + convertedFee;
574     basePairFee = 0;
575 } else {
576     // Base pair is ERC20
577     nativeFee = flatFee;
578     basePairFee = convertedFee;
```

The function above assigns `convertedFee` to the first return value from `PrintrTrading.estimateTokenCost`, which is `availableAmount` (meme tokens), not the cost (base pair currency) as intended.

The code treats `convertedFee`, which represents the amount of meme tokens, as if it is base pair currency, adding it directly to `nativeFee` in base pair currency or using it as `basePairFee`.

Since meme token amounts are typically large numbers (e.g., 1000 * 10^18), users get charged this raw number as base pair currency, resulting in fees similar to 0.001 ETH (flatFee) + 1000 ETH (1000 * 10^18 wei) vastly inflated total fees.

These function are used in:

- `PrintrTeleport.quoteTeleportFee`
- `Telecoin._processTeleportFee`

As a result, every ITS transaction that estimates its fee by using `PrintrTeleport.quoteTeleportFee` will vastly overpay the fees or fail if it does not provide enough funds because of the `transfer` function usage which is showed below:

Listing 4. Excerpt from PrintrTeleport

```
77  */
78 uint8 public constant TELEPORT_MESSAGE = 0x01;
79
80 /**
81  * @notice LayerZero endpoint ID for Solana
82  */
83 uint256 public constant LZ_SOLANA_EID = 30_168;
84
85 /**
86  * @notice Constructor to initialize the PrintrTeleport contract
87  * @param storageParams Storage parameters for PrintrStorage initialization
88  * @param params Teleport deployment parameters including LayerZero
   configuration
89  */
90 constructor(
91     DeploymentParams memory storageParams,
92     TeleportDeployParams memory params
93 ) PrintrStorage(storageParams) {
94     require(params.lzEndpoint != address(0), InvalidEndpoint());
95     require(params.solanaLzPeer != bytes32(0), InvalidSolanaProgram());
```

Exploit scenario

First scenario:

1. Bridging fee is configured to 0.1% of the total teleported amount of tokens. Base pair currency is ETH.
2. Alice, a user, tries to teleport her 100,000 tokens from Mainnet to Arbitrum using ITS.
3. Alice uses the `PrintrTeleport.quoteTeleportFee` function to estimate the fee. The calculation results in 100 ETH as a fee.
4. Alice sends the fee to the Printr contract with the `PrintrTeleport.teleport` function and massively overpays the fees to the Printr contract.

Second scenario:

1. Bridging fee is configured to 0.1% of the total teleported amount of tokens. Base pair currency is ETH.
2. Alice, a user, tries to teleport her 100,000 tokens from Mainnet to Arbitrum using ITS.
3. Alice correctly estimates fees manually and pays the correct amount of fees to the `PrintrTeleport.teleport` function to be successful.
4. The transaction reverts as the Printr contract tries to transfer a huge amount of ETH to the Printr contract.

Recommendation

Assign the correct return value from the `PrintrTrading.estimateTokenCost` function to the `convertedFee` variable.

Fix 1.1

The issue was fixed by assigning the correct returning value to the `convertedFee` variable.

Go back to Findings Summary

C3: `teleport` function always fails for the ITS transactions as all fees are transferred to the Printr

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PrintrTeleport.sol	Type:	Logic error

Description

The `teleport` function calculates the amount of fees that must be paid for the teleportation to the Printr team and for Interchain Token Service bridging. However, it always transfers all calculated fees to the Printr team and there are no remaining funds for the Interchain Token Service bridging.

Listing 5. Excerpt from PrintrTeleport

```
186 function quoteTeleportFee(
187     address token,
188     TeleportParams calldata params,
189     TeleportProtocol protocol
190 ) external returns (uint256 nativeFee, uint256 basePairFee, address
    basePair) {
191     if (protocol == TeleportProtocol.ITS) {
192         // Calculate the base teleport fees
193         (nativeFee, basePairFee, basePair) = _calculateTeleportFee(token,
    params.amount, itsFlatFee, itsBipsFee);
194 
195         // Estimate ITS/Axelar gas fee for the cross-chain transfer
196         uint256 itsGasFee = _quoteItsGasFee(token, params);
197 
198         // Add ITS gas fee to the native fee
199         nativeFee += itsGasFee;
200 
201         return (nativeFee, basePairFee, basePair);
202     } else if (protocol == TeleportProtocol.LZ) {
```

The above function calculates the amount of fees that must be paid for the

teleportation to the Printr team and for Interchain Token Service bridging.

Listing 6. Excerpt from Telecoin

```
77 function teleport(
78     IPrintrTeleport.TeleportParams calldata params,
79     IPrintrTeleport.TeleportProtocol protocol
80 ) public payable virtual {
81     if (protocol == IPrintrTeleport.TeleportProtocol.UNSPECIFIED) {
82         revert InvalidProtocol();
83     }
84     if (protocol == IPrintrTeleport.TeleportProtocol.ITS) {
85         // Get the teleport fee using the unified function
86         (uint256 nativeTeleportFee,,) =
   IPrintrTeleport(printr).quoteTeleportFee(address(this), params, protocol);
87
88         // Transfer the fee to printr
89         if (nativeTeleportFee > 0) {
90             payable(printr).transfer(nativeTeleportFee);
91         }
92
93         interchainTransfer(params.destChain, params.destAddress,
   params.amount, params.metadata);
94         return;
95     }
```

All calculated fees are transferred to the Printr contract without remaining funds for the Interchain Token Service bridging.

As a result, every ITS transaction that estimates its fee by using the `PrintrTeleport.quoteTeleportFee` function will fail because of overpaying the fees to the Printr contract.

Exploit scenario

1. Alice, a user, attempts to teleport a token from Mainnet to Arbitrum using the ITS;
2. Alice calculates the amount of fees that must be paid for the teleportation to the Printr team and for Interchain Token Service bridging

using the `PrintrTeleport.quoteTeleportFee` function;

3. Alice sends the calculated fees to the Printr contract with the `teleport` function; and
4. the ITS transaction fails because of overpaying the fees to the Printr contract and not enough funds for the Interchain Token Service bridging.

Recommendation

Update the `quoteTeleportFee` function to separate the amount of fees that must be paid for the teleportation to the Printr team and for Interchain Token Service bridging and send only the appropriate amount of funds to the Printr contract.

Fix 1.1

The issue was fixed by sending only the appropriate amount of funds to the Printr contract and leaving the remaining funds for the Interchain Token Service bridging.

Go back to Findings Summary

C4: Inverted condition for creation of `PrintrTeleportingTelecoin`

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PrintrPrinting.sol	Type:	Logic error

Description

The `PrintrPrinting.print` function contains a condition that determines whether a token of type `PrintrTeleportingTelecoin` should be created. However, the condition is incorrectly implemented. Instead of evaluating if a `PrintrTeleportingTelecoin` should be created, it evaluates if a `PrintrMainTelecoin` should be created.

Listing 7. Excerpt from PrintrPrinting

```
86 tokenAddress = _deployToken(
87     telecoinParams, tokenData.telecoinId, telecoinParams.chains.length == 1
   && tokenData.chainIndex == 0
88 );
```

Listing 8. Excerpt from PrintrPrinting

```
235 (bool success, bytes memory data) = telecoinFactory.delegatecall(
236     abi.encodeWithSelector(
237         ITelecoinFactory.deployToken.selector,
238         deployParams,
239         address(treasury),
240         10 ** unpacked.maxTokenSupplyE * PRECISION /
    telecoinParams.chains.length, // initialSupply
241         isTeleporting // isMainTelecoin if only one chain
242     )
```

Listing 9. Excerpt from TelecoinFactory

```
43 function deployToken(
```

```
44      ITelecoin.TelecoinDeployParams memory params,
45      address treasury,
46      uint256 initialSupply,
47      bool isTeleporting
48  ) external payable returns (address token) {
49      // Get token manager address from ITS
50      params.itsTokenManager =
51
    IInterchainTokenService(params.interchainTokenService).tokenManagerAddress(pa
    rams.interchainTokenId);
52
53      // Deploy token contract using CREATE3
54      token = _create3(
55          abi.encodePacked(
56              isTeleporting ? type(PrintrTeleportingTelecoin).creationCode :
    type(PrintrMainTelecoin).creationCode,
57              abi.encode(params, treasury, initialSupply)
58          ),
59          params.telecoinId
60      );
61  }
```

As a result, a token of type `PrintrMainTelecoin` is created when a `PrintrTeleportingTelecoin` should be created.

Exploit scenario

1. Alice, a user, attempts to print a new token on Mainnet, Optimism, and Arbitrum;
2. the token is printed successfully on all three chains;
3. the registration of the printed token is successful on Mainnet; and
4. Alice attempts to teleport the token to other chains using the `PrintrTeleport.teleport` function. However, the transaction reverts because the Axelar ITS operates with a main telecoin when it should operate with a teleporting telecoin.

As a result, all printed tokens that are supposed to be teleportable cannot be teleported.

Recommendation

Update the condition to correctly evaluate whether `PrintrTeleportingTelecoin` should be created.

Fix 1.1

The issue was fixed by updating the logic of processing the condition to correctly evaluate whether `PrintrTeleportingTelecoin` or `PrintrMainTelecoin` should be created.

Go back to Findings Summary

C5: Only sending to Solana peer address even for EVMs at LayerZero causes wrong receiver address

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PrintrTeleport.sol	Type:	Logic error

Description

The receiver of the transaction from LayerZero is set to the Solana peer address even for message transfers to EVM chains. This causes the transaction to succeed on the source chain but fail on the destination chain.

Listing 10. Excerpt from PrintrTeleport

```
489 receiver: solanaLzPeer,
```

Exploit scenario

1. Alice, a user, tries to transfer to another EVM.
2. The teleport transaction succeeds; however, the receiver address is set to the Solana peer address.
3. The transaction fails on the destination EVM chain, and Alice loses the tokens.

Recommendation

Set the correct receiver address for EVM chains.

Fix 1.1

The issue was fixed by adding self address and solana peer by condition.

Go back to Findings Summary

C6: Missing access control in `witnessTeleport`

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PrintrTeleport.sol	Type:	Access control

Description

The function `witnessTeleport` is missing access control. This results in anyone being able to consume anyone's tokens that are approved to the Printr contract.

Listing 11. Excerpt from PrintrTeleport

```
242 function witnessTeleport(
243     address signer,
244     address token,
245     TeleportParams calldata params,
246     TeleportProtocol protocol
247 ) external payable {
248     _teleport(signer, token, params, protocol);
249 }
```

Exploit scenario

1. Alice, a victim, wants to operate in Printr and therefore sends a transaction to approve.
2. Bob, an attacker, sees this transaction and sends a `witnessTeleport` transaction that sends tokens to Bob's entity.
3. Alice sends an operation transaction; however, the tokens have already been consumed by Bob.

Recommendation

Add access control for the witness operation.

Fix 1.1

The issue was fixed by implementing verification that the message sender is Telecoin, which is ERC20Witness. Only `permitWitnessCall` can call with this ABI, and the first parameter is an address that is verified in this function.

Go back to Findings Summary

C7: Recipient address is left-padded during encoding and right-padded during decoding for LayerZero

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PrintrTeleport.sol	Type:	Arithmetics

Description

The recipient address for LayerZero is encoded with padding by `bytes32(bytes20(address))` which does right padding. However, the Address library decodes with `address(uint160(uint256(bytes32)))` which assume left padding. This results in LayerZero sending the transaction to a different address.

Listing 12. Excerpt from PrintrTeleport

```
694 /**
695  * @notice Converts destination address bytes to bytes32 format
696  * @dev Handles both 20-byte addresses and 32-byte addresses
697  * @param destAddress The destination address bytes
698  * @return The destination address as bytes32
699  */
700 function _parseRecipientToBytes32(
701     bytes calldata destAddress
702 ) internal pure returns (bytes32) {
703     if (destAddress.length == 20) {
704         // EVM address - pad with zeros on the left
705         return bytes32(bytes20(destAddress));
706     } else if (destAddress.length == 32) {
707         // Already bytes32 (Solana address)
708         return bytes32(destAddress);
709     } else {
710         revert InvalidCalldata();
711     }
712 }
```

Exploit scenario

Alice, a user, tries to teleport to her address on a different chain. The tokens are transferred to the destination chain; however, they are sent to a different address from the one specified during the source chain transaction. Alice loses the tokens.

Recommendation

Use the correct calculation or library to specify the correct address.

Fix 1.1

The issue was fixed by using a library for the conversion.

Go back to Findings Summary

H1: Loss of precision during conversion of `uint256` to `uint80`

High severity issue

Impact:	Medium	Likelihood:	High
Target:	PrintrTeleport.sol	Type:	Data validation

Description

The Printr protocol has a `print` function for printing telecoins. For printing telecoins, the user must specify the parameters with which the telecoin will be printed. Depending on these parameters, the value of the virtual reserve of the pool is calculated. The virtual reserve value represents the equivalent amount of basePair tokens to the initial amount of telecoins in the pool. This value is used in different calculations for buying and selling telecoins.

Listing 13. Excerpt from PrintrPrinting._estimateTokenCost

```
350 uint256 curveConstant = curve.virtualReserve * initialTokenReserve;
351 uint256 tokenReserve = curveConstant / (curve.virtualReserve +
    curve.reserve);
```

Listing 14. Excerpt from PrintrPrinting._quoteTokenAmount

```
422 uint256 curveConstant = curve.virtualReserve * initialTokenReserve;
423 uint256 tokenReserve = curveConstant / (curve.virtualReserve +
    curve.reserve);
424
425 // curveBudget = 100 % / 101 %, excluding trading fee
426 uint256 curveBudget = (BIPS_SCALAR * baseSpend) / (BIPS_SCALAR +
    tradingFee);
427 tokenAmount = tokenReserve - curveConstant / (curve.virtualReserve +
    curve.reserve + curveBudget);
```

It is calculated by this formula:

Listing 15. Excerpt from PrintrPrinting.print

```
151 uint256 virtualReserve = initialTokenReserve * unpacked.initialPrice /
    PRECISION;
```

Initially, it is saved in `uint256` type, but later it is converted to `uint80` type. However, if the value of the virtual reserve is too large, the value is truncated, breaking the constant product formula invariant.

Listing 16. Excerpt from PrintrPrinting.print

```
158 curve = Curve({
159     basePair: basePair,
160     totalCurves: uint16(telecoinParams.chains.length),
161     maxTokenSupplyE: unpacked.maxTokenSupplyE,
162     virtualReserve: uint80(virtualReserve),
163     reserve: 0,
164     completionThreshold: uint64(unpacked.completionThreshold)
165 });
```

As a result, the curve is not valid.

Exploit scenario

1. Alice, a user, deploys a telecoin by specifying `chains.length` value to `1`, `max_token_supply_e` to `9`, and `initial_price` to `10^16`.
2. The `virtualReserve` variable evaluates to 10^25, which requires at least 84 bits for representation of this value, while only 80 bits are available.
3. Alice tries to buy telecoins from the printing mechanism and the constant product formula invariant is broken.

As a result, all calculations based on the virtual reserve value are incorrect.

Recommendation

Do not use `uint80` type for the virtual reserve value. Use `uint128` instead and

add an assertion that if the virtual reserve value is larger than 10^38, the telecoin cannot be printed due to restrictions. Otherwise, `uint256` type is recommended.

Partial solution 1.1

The issue was addressed by introducing an additional variable, virtualReserveE, to store the exponent of the virtual reserve value, while virtualReserve represents the mantissa. However, if the value does not contain trailing zeros (i.e., it is not of the form 10 * 10**18) and instead resembles a number such as 1_123_456_789_123_456_789, the applied compression algorithm — which repeatedly divides the value by 10 — can result in precision loss, as significant digits are truncated during the scaling process.

Go back to Findings Summary

H2: `SELF` immutable address pointing to implementation contract

High severity issue

Impact:	Medium	Likelihood:	High
Target:	PrintrTeleport.sol	Type:	Access control

Description

The contract is upgradeable; however, the `SELF` is set during implementation contract construction. The variable is used for verification of the cross-chain message sender at the `allowInitializePath` function.

Exploit scenario

1. Alice, a user, tries to transfer tokens via cross-chain.
2. The transfer transaction at the source chain succeeds.
3. However, the transaction at the destination chain fails for EVM networks, causing loss of tokens because of different address.

Recommendation

Make `SELF` always point to the proxy contract.

Fix 1.1

The issue was fixed by adding a new contract for handling LayerZero.

Go back to Findings Summary

H3: Lack of destination address verification

High severity issue

Impact:	High	Likelihood:	Medium
Target:	PrintrTeleport.sol	Type:	Data validation

Description

The LayerZero teleport can transfer to a chain where Printr is not deployed. LayerZero's official OApp handles this by setting the destination address for each chain.

LayerZero Source

Exploit scenario

1. Alice, a user, tries to teleport to another chain.
2. Alice specifies a chain that Telecoin does not support or is not yet deployed on.
3. The transaction at the destination chain fails, and Alice loses the tokens.

Recommendation

Add verification of the chain and destination address similarly to how LayerZero OApp does.

Acknowledgment 1.1

The issue was acknowledged by the team.

Go back to Findings Summary

M1: Unchecked call in `_refund` function

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Intent.sol	Type:	Data validation

Description

The `_refund` function in the `Intent` contract is used for refunding the leftover ETH and tokens to the sender after the intent is executed. However, the function does not check if the call is successful and also does not check if the ERC20 token transfers were successful.

This issue has been found by static analysis.

Listing 17. Excerpt from Intent

```
425 function _refund(address sender, address[] memory refundTokens) internal {
426     // Refund any leftover ETH
427     uint256 ethBalance = address(this).balance;
428     if (ethBalance > 0) {
429         // Intentionally allow transfer to fail for refunds
430         (bool success,) = sender.call{ value: ethBalance }("");
431         success;
432     }
433 
434     // Refund any specified tokens
435     for (uint256 i = 0; i < refundTokens.length; ++i) {
436         if (refundTokens[i] != address(0)) {
437             // Get token balance
438             uint256 balance = IERC20(refundTokens[i]).balanceOf(address(
    this));
439             if (balance > 0) {
440                 // Intentionally allow transfer to fail for refunds
441                 try IERC20(refundTokens[i]).transfer(sender, balance) { }
    catch { }
442             }
443         }
444     }
```

```
445 }
```

Exploit scenario

Alice, a user, calls the `execute` function to execute an intent. The intent is successfully executed, but the refunding of the leftover ETH and tokens to the sender fails. As a result, Alice loses the leftover ETH and tokens.

Recommendation

Add verification that the call is successful and the ERC20 token transfers were successful.

Fix 1.1

The issue was fixed by returning the refund in the wrapped native token if transaction of native token failed.

Go back to Findings Summary

M2: Variable `priceLimit` is ignored for specific cases

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	PrintrTrading.sol	Type:	Data validation

Description

The Printr protocol allows specifying the price limit for trades. However, the price limit is ignored when the price is above the specified price limit in the `PrintrTrading.buy` function and when it is below the specified price limit in the `PrintrTrading.sell` function.

This issue arises in the following functions:

Listing 18. Excerpt from PrintrTrading._estimateTokenCost

```
356 if (priceLimit != 0) {
357     // Calculate the maximum tokenReserve that would result in
    effectivePrice <= priceLimit
358     // priceLimit = PRECISION * curveConstant / (tokenReserve -
    tokenAmount)^2
359     // tokenAmount = tokenReserve - sqrt(PRECISION * curveConstant /
    priceLimit)
360
361     uint256 sqrtValue = Math.sqrt(PRECISION * curveConstant / priceLimit);
362     if (sqrtValue < tokenReserve) {
363         uint256 maxAmountFromPriceLimit = tokenReserve - sqrtValue;
364         // Take the minimum between requested amount and price limit amount
365         if (maxAmountFromPriceLimit < tokenAmount) {
366             tokenAmount = maxAmountFromPriceLimit;
367         }
368     }
369 }
```

Listing 19. Excerpt from PrintrTrading._estimateTokenRefund

```
238 if (curve.completionThreshold == 0) {
239     params.amount = _spendInLiquidityPool(curve, params);
240     return params;
241 }
```

As a result, the specified `priceLimit` is ignored and the trade is executed at a price that is higher than the specified `priceLimit` in the `PrintrTrading.buy` function and at a price that is lower than the specified `priceLimit` in the `PrintrTrading.sell` function.

Exploit scenario

1. Alice, a user, tries to buy a token and specifies a price limit.
2. The trade is executed at a price that is higher than the specified price limit.
3. Alice loses money because tokens are bought at a price higher than her specified limit.

Recommendation

Add an alternative branch of the `priceLimit` check to ensure that no tokens can be bought at a price higher than the specified `priceLimit` in the `PrintrTrading.buy` function and no tokens can be sold at a price lower than the specified `priceLimit` in the `PrintrTrading.sell` function.

Fix 1.1

The issue was fixed as specified in recommendation.

Go back to Findings Summary

M3: `teleportFrom` function does not work due to insufficient approval for `TokenManager` to burn tokens

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	Teleporting.sol	Type:	Logic error

Description

The `teleportFrom` function is used for teleporting tokens via ITS or LZ when approval is given to the spender by the approver. However, the approval to the spender is insufficient for the function to execute successfully.

Listing 20. Excerpt from InterchainStandard

```
66 function interchainTransferFrom(
67     address sender,
68     string calldata destinationChain,
69     bytes calldata recipient,
70     uint256 amount,
71     bytes calldata metadata
72 ) public payable {
73     _decreaseAllowance(sender, msg.sender, amount);
74 
75     _beforeInterchainTransfer(sender, destinationChain, recipient, amount,
   metadata);
76 
77 
   IInterchainTokenService(interchainTokenService).transmitInterchainTransfer{
   value: address(this).balance }(
78         interchainTokenId, sender, destinationChain, recipient, amount,
   metadata
79     );
80 }
```

In the function above, the allowance of the spender is consumed.

Listing 21. Excerpt from Teleporting

```
109 fallback() external {
110     if (msg.data.length < 4) {
111         revert InvalidFunctionSelector();
112     }
113
114     bytes4 selector = bytes4(msg.data[:4]);
115     (address account, uint256 amount) = abi.decode(msg.data[4:], (address,
    uint256));
116
117     // teleportIn selector: 0x40c10f19 (mint))
118     if (selector == 0x40c10f19) {
119         teleportIn(account, amount);
120         return;
121     }
122     // teleportOut selector: 0x9dc29fac (burn) or 0x79cc6790 (burnFrom)
123     if (selector == 0x9dc29fac || selector == 0x79cc6790) {
124         teleportOut(account, amount);
125         return;
126     }
127
128     revert InvalidFunctionSelector();
```

Later, when `TokenManager` tries to burn the tokens from the spender, it needs approval from the spender.

Listing 22. Excerpt from Teleporting

```
50 function teleportOut(address from, uint256 value) public {
51     if (msg.sender != itsTokenManager && msg.sender != printr) {
52         revert UnauthorizedAccount(msg.sender);
53     }
54     if (from == address(0)) {
55         revert ERC20InvalidSender(address(0));
56     }
57
58     if (msg.sender == itsTokenManager) {
59         IPrintrTeleport(printr).processInterchainTransfer(telecoinId, from,
    value);
60     }
61
62     emit TeleportOut(telecoinId, from, value);
```

```
63     _decreaseAllowance(from, msg.sender, value);
64     _update(from, address(this), value);
65 }
```

Besides, there is a similar issue with the telecoins, which are supposed to not be teleportable and use `LOCK_UNLOCK` token manager type. .Excerpt from [InterchainTokenService.transmitInterchainTransfer]

```
592     function transmitInterchainTransfer(
593         bytes32 tokenId,
594         address sourceAddress,
595         string calldata destinationChain,
596         bytes memory destinationAddress,
597         uint256 amount,
598         bytes calldata metadata
599     ) external payable whenNotPaused {
600         amount = _takeToken(tokenId, sourceAddress, amount, true);
601 
602         bytes memory data = _decodeMetadata(metadata);
603 
604         _transmitInterchainTransfer(tokenId, sourceAddress,
    destinationChain, destinationAddress, amount, data, msg.value);
605     }
```

Listing 23. Excerpt from [InterchainTokenService._takeToken]

```
1145     function _takeToken(bytes32 tokenId, address from, uint256 amount, bool
     tokenOnly) internal returns (uint256) {
1146         (bool success, bytes memory data) = tokenHandler.delegatecall(
1147             abi.encodeWithSelector(ITokenHandler.takeToken.selector,
     tokenId, tokenOnly, from, amount)
1148         );
1149         if (!success) revert TakeTokenFailed(data);
1150         amount = abi.decode(data, (uint256));
1151 
1152         return amount;
1153     }
```

Listing 24. Excerpt from [TokenHandler.takeToken]

```
67     function takeToken(bytes32 tokenId, bool tokenOnly, address from, uint256
```

```
    amount) external payable returns (uint256) {
68          address tokenManager = _create3Address(tokenId);
69          (uint256 tokenManagerType, address tokenAddress) =
    ITokenManagerProxy(tokenManager).getImplementationTypeAndTokenAddress();
70
71          if (tokenOnly && msg.sender != tokenAddress) revert
    NotToken(msg.sender, tokenAddress);
72
73          if (
74              tokenManagerType ==
    uint256(TokenManagerType.NATIVE_INTERCHAIN_TOKEN) || tokenManagerType ==
    uint256(TokenManagerType.MINT_BURN)
75          ) {
76              _burnToken(ITokenManager(tokenManager), tokenAddress, from,
    amount);
77          } else if (tokenManagerType ==
    uint256(TokenManagerType.MINT_BURN_FROM)) {
78              _burnTokenFrom(tokenAddress, from, amount);
79          } else if (tokenManagerType == uint256(TokenManagerType.LOCK_UNLOCK))
    {
80              _transferTokenFrom(tokenAddress, from, tokenManager, amount);
81          ...
```

When `InterchainTokenService` tries to transfer the tokens from the spender, it needs the approval from the spender, even though the approval the user gave to the Printr contract is not used in any way. As a result, users need to give approval specifically to the `InterchainTokenService` to transfer the tokens, not to the Printr contract.

As a result, a similar problem arises when the `teleportFrom` function is executed for the telecoins, which are supposed to not be teleportable and use `LOCK_UNLOCK` token manager type.

Exploit scenario

1. Bob, a token holder, grants approval to Alice, a user, for 100,000 telecoins.
2. Alice calls the `teleportFrom` function to teleport 100,000 telecoins from Bob to Alice for teleportation.

3. The transaction reverts because it requires additional approval from Alice to the `TokenManager` to burn the tokens.

The severity of the finding is decreased to Medium because there is a workaround for the issue: Alice first uses the `transferFrom` function to transfer the tokens to herself, approves the `TokenManager` to burn her tokens, and then calls the `teleport` function to teleport the tokens.

Recommendation

Refactor the function to allow `teleportFrom` to execute with only the approval from the spender.

Acknowledgment 1.1

The issue was acknowledged by the team with the comment:

> *Intended by design, user needs to approve ITS for burnFrom.*

Go back to Findings Summary

M4: Potential overflow on teleport amount conversion

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	PrintrTeleport.sol	Type:	Data validation

Description

The contract converts the teleporting number to `uint64` after dividing by 10^9. However, the number can potentially overflow, and users lose tokens. This will be a problem if someone tries to teleport more than ~18.4 billion tokens.

Listing 25. Excerpt from PrintrTeleport

```
478 ctx.convertedAmount = uint64(params.amount / 1e9);
479 // Truncate to 9 decimals for cross-chain compatibility
480 ctx.truncatedAmount = (params.amount / 1e9) * 1e9;
```

Exploit scenario

Alice, a user, calls teleport to transfer a large amount. However, because of overflow, Alice receives a much smaller number of tokens at the destination chain.

Recommendation

Add verification that overflow does not happen. Alternatively, use a larger bit number.

Fix 1.1

The issue was fixed by adding overflow verification.

Go back to Findings Summary

L1: Everyone can link telecoins to other chains

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	PrintrInterchain.sol	Type:	Access control

Description

The `linkEvmInterchainToken` function is used for linking telecoins with other chains. However, anyone can call this function to link telecoins to other chains, even if they have not been listed in the `PrintrPrinting.print` function. This can lead to telecoins being linked to chains that are not supposed to have the telecoins printed.

Listing 26. Excerpt from PrintrInterchain

```
87 function linkEvmInterchainToken(
88     TelecoinParams calldata tokenParams,
89     string calldata destinationChain
90 ) external payable whenNotPaused {
91     if (tokenParams.chains.length == 0) {
92         revert InvalidLength();
93     }
94     // Verify that the first chain in params matches the current chain
95     if (keccak256(bytes(tokenParams.chains[0].toTrimmedString())) !=
   currentChainHash) {
96         revert WrongChainName();
97     }
98
99     // Generate telecoin ID and get token address
100     bytes32 telecoinId = _getTelecoinId(tokenParams);
101     bytes32 itsTokenId = _getInterchainTokenId(telecoinId,
   tokenParams.chains[0].toTrimmedString());
102     address tokenAddress = IPrintrGetters(address(
   this)).getTokenAddress(telecoinId);
103
104     // Link token to destination chain using MINT_BURN type manager
105     bytes32 linkedTokenId = IInterchainTokenFactory(itsFactory).linkToken{
   value: msg.value }(
```

```
106         telecoinId,
107         destinationChain,
108         abi.encodePacked(tokenAddress),
109         IInterchainTokenFactory.TokenManagerType.MINT_BURN_FROM,
110         abi.encodePacked(owner()),
111         msg.value
112     );
113
114     if (linkedTokenId != itsTokenId) {
115         revert TokenIdMismatch();
116     }
117 }
```

This behavior applies to all tokens, not only `PrintrTeleportingTelecoin` but also other telecoin types.

Exploit scenario

1. Alice, a user, wants to link a telecoin to another chain.
2. Alice calls the `linkEvmInterchainToken` function to link the telecoin to another chain, and the transaction succeeds.
3. The telecoin is linked to the other chain in the Axelar ecosystem.
4. Alice tries to teleport the telecoin to the other chain, and the transaction succeeds.
5. However, the transaction fails on the destination chain because the telecoin does not exist on the destination chain.

The severity of the finding is increased to Low because of the chaining with W5.

Recommendation

Restrict the linking of telecoins to chains where telecoins are not printed.

Acknowledgment 1.1

The issue was acknowledged by the team with the comment:

> *Intended, token can be deploy on any chain, but without bonding curve*

Go back to Findings Summary

L2: Missing refund in `teleport` function

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	PrintrTeleport.sol	Type:	Logic error

Description

The `teleport` function is used for teleporting tokens via ITS. However, it does not refund the sender if the user sends more funds than needed.

Listing 27. Excerpt from Telecoin

```
77 function teleport(
78     IPrintrTeleport.TeleportParams calldata params,
79     IPrintrTeleport.TeleportProtocol protocol
80 ) public payable virtual {
81     if (protocol == IPrintrTeleport.TeleportProtocol.UNSPECIFIED) {
82         revert InvalidProtocol();
83     }
84     if (protocol == IPrintrTeleport.TeleportProtocol.ITS) {
85         // Get the teleport fee using the unified function
86         (uint256 nativeTeleportFee,,) =
   IPrintrTeleport(printr).quoteTeleportFee(address(this), params, protocol);
87
88         // Transfer the fee to printr
89         if (nativeTeleportFee > 0) {
90             payable(printr).transfer(nativeTeleportFee);
91         }
92
93         interchainTransfer(params.destChain, params.destAddress,
   params.amount, params.metadata);
94         return;
95     }
```

Exploit scenario

1. Alice, a user, wants to teleport tokens via ITS.

2. Alice sends a transaction to teleport tokens and sends twice the required funds.
3. The transaction is successful.
4. However, Alice is not refunded the extra funds she sent.

Recommendation

Refund the sender if the user sends more funds than needed.

Acknowledgment 1.1

The issue was acknowledged by the team with the comment:

> *Acknowledged, we want to send gas overhead to ITS.*

Go back to Findings Summary

L3: Limited gas for ETH transfers causes withdrawal failures for smart contract wallets

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Treasury.sol	Type:	Code quality

Description

The `Treasury.withdraw` function allows withdrawing the base currency of the chain to the recipient. However, the function uses a hardcoded gas value of 3000 for ETH transfers, which can cause withdrawal failures for smart contract wallets.

Listing 28. Excerpt from Treasury

```
90 if (token == address(0)) {
91     // Handle native currency withdrawal
92     (bool success,) = recipient.call{ value: amount, gas: 3000 }("");
93     if (!success) {
94         revert FailedWithdrawal();
95     }
96 } else {
```

Additionally, with new EIP-7702, more EOAs can have fallback functions for processing incoming ETH. See link to EIP-7702

Exploit scenario

1. Alice, the Treasury owner, has execution code configured on her EOA. The EOA has a fallback function for processing incoming ETH.
2. Alice calls the `Treasury.withdraw` function to withdraw the base currency of the chain to the recipient.
3. The transaction fails because the gas value is insufficient for the ETH

transfer.

Recommendation

Remove the hardcoded gas value for ETH transfers.

Fix 1.1

The issue was fixed by removing the hardcoded gas value for ETH transfers.

Go back to Findings Summary

L4: Possible overflow in `print` function

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	PrintrPrinting.sol	Type:	Overflow/Underflow

Description

The `print` function in `PrintrPrinting.sol` is susceptible to arithmetic overflow when `TelecoinParams.basePrices` is set to small values. The overflow occurs in the following calculation:

Listing 29. Excerpt from PrintrPrinting

```
154 tokenData.completionPrice =
155     PRECISION * (virtualReserve * initialTokenReserve) /
    completionTokenReserve / completionTokenReserve;
```

While this overflow is caught by Solidity's built-in checks and results in a transaction revert, it prevents the function from working with certain valid small base prices.

Exploit scenario

Alice, a user, attempts to create a new token with the following parameters:

```
packed_params = pack_params(
    maxTokenSupplyE=6,  # 6 = 1_000_000 tokens
    completionThreshold=5000,  # 50%
    initialPrice=int(1e18),  # 1 WETH per token
    initialBuySpending=0
)

token_params = PrintrTelecoin.TelecoinParams(
    salt=bytes32(0),
```

```
    creatorAddresses=abi.encode_packed(alice),
    name=make_bytes32_string("TestToken"),
    symbol=make_bytes32_string("TEST"),
    packedParams=packed_params,
    chains=[make_bytes32_string("mainnet")],
    basePairs=[address_to_bytes32(WETH_address)],
    basePrices=abi.encode_packed(uint128(basePrice))
)
```

With these specific parameters, when `basePrice` is set to any value below 8,636,169 wei, the transaction reverts due to arithmetic overflow, preventing Alice from creating tokens with valid small base prices.

Recommendation

Add input validation to ensure `basePrice` is above the minimum threshold that would cause overflow based on the other parameters provided to the function. Alternatively, restructure the calculation to handle small base prices without overflow.

Fix 1.1

The issue was fixed by modifying the calculation formula for `tokenData.completionPrice` to prevent arithmetic overflow when `TelecoinParams.basePrices` contains small values. The new implementation restructures the mathematical operations to avoid the overflow condition that previously caused transaction reverts with valid small base prices.

Go back to Findings Summary

L5: Ambiguous error `LiquidityAlreadyDeployed`

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	IPrintrStorage.sol	Type:	Code quality

Description

The error names do not match their actual meanings.

The error is used in `PrintrTrading.sol`. In the `buy` function, it is thrown when liquidity is deployed and neither `msg.value` nor `maxPrice` is provided. The actual meaning is "Cannot determine payment amount for liquidity pool purchase."

Another place of error usage is `PrintrTrading.sol`, in `_quoteLiquidityExactOutput`. It is thrown when the liquidity module's `quoteExactOutput` delegatecall fails; however, the actual meaning is "Failed to get quote from liquidity pool."

Exploit scenario

Alice, a user, tries to operate with the contract. If the contract reverts the transaction, Alice cannot find the reason unless she debugs the transaction in detail.

Recommendation

Use errors that correspond to their meanings, preferably adding parameters that describe the situation for easier debugging.

Fix 1.1

The custom error InvalidQuoteResult was added for quote failures.

Go back to Findings Summary

W1: Incorrect value of `issuedSupply` in `_estimateTokenCost` function

Impact:	Warning	Likelihood:	N/A
Target:	PrintrPrinting.sol	Type:	Data validation

Description

The Printr protocol uses the `_estimateTokenCost` function to calculate the cost of buying a specific amount of tokens. The value of `issuedSupply` is calculated incorrectly due to precision loss. As a result, the emitted event has an incorrect value of `issuedSupply`.

Listing 30. Excerpt from PrintrTrading._estimateTokenCost

```
390 issuedSupply = initialTokenReserve - curveConstant / (curve.virtualReserve +
    curve.reserve + curveCost);
```

This calculation suffers from precision loss due to the division operation. A better way to calculate the value of `issuedSupply` is the following:

```
uint256 issuedSupply2 = initialTokenReserve - (tokenReserve - availableAmount);
```

Recommendation

Refactor the calculation of `issuedSupply` to use the formula without precision loss.

Fix 1.1

The issue was fixed by using the formula without precision loss.

Go back to Findings Summary

W2: Ignoring failure of calling `decimals` function of the base pair token

Impact:	Warning	Likelihood:	N/A
Target:	PrintrPrinting.sol	Type:	Data validation

Description

The `PrintrPrinting` contract ignores the failure of calling the `decimals` function on the base pair token and continues execution with 18 decimals as the default. This can result in incorrect price calculation and incorrect amount of tokens being bought.

Listing 31. Excerpt from PrintrPrinting

```
122 {
123     // Calculate required initial spending adjusted for chain-specific base
    price
124     uint256 convertedInitialSpending =
125         PRECISION * unpacked.initialBuySpending / basePrice /
    telecoinParams.chains.length;
126
127     try IERC20Metadata(basePair).decimals() returns (uint8 decimals) {
128         if (decimals < 18) {
129             convertedInitialSpending = convertedInitialSpending / 10 ** (18
    - decimals);
130             unpacked.initialPrice = unpacked.initialPrice / 10 ** (18 -
    decimals);
131         }
132         if (decimals > 18) {
```

Recommendation

Add verification that the `decimals` function is successful, otherwise revert the transaction.

Fix 1.1

The issue was fixed by adding verification that the `decimals` call was successful, otherwise revert the transaction.

Go back to Findings Summary

W3: Approval for the ITS token manager is needed before teleporting

Impact:	Warning	Likelihood:	N/A
Target:	Teleporting.sol	Type:	Code quality

Description

The Printr protocol provides the ability to teleport tokens from one chain to another via Interchain Token Service. However, before teleporting, a user needs to approve the ITS token manager to burn tokens, which is not good practice to give third-party protocols permissions for the tokens. If there is any vulnerability in the integrated contract to which the user gave approval, the user's tokens can be stolen.

The flow of teleporting via ITS is the following:

Listing 32. Excerpt from cross-chain

```
        bytes32 _interchainTokenId,
        address _tokenManager
    ) ERC20(name, symbol) {
        interchainTokenId = _interchainTokenId;
        tokenManager = _tokenManager;
        _setupRole(MINTER_ROLE, _tokenManager);
    }

    function burn(address account, uint256 amount) external {
        require(msg.sender == tokenManager, "Only token manager");
        _burn(account, amount);
    }

    function mint(address account, uint256 amount) external {
        require(msg.sender == tokenManager, "Only token manager");
        _mint(account, amount);
    }
}
```

In the code snippet above, the Printr protocol calls the `InterchainTokenService.transmitInterchainTransfer` which will initiate the teleporting process.

At a specific moment of the call execution, the `InterchainTokenService` contract will execute the following function:

Listing 33. Excerpt from TokenHandler.takeToken

```
67     function takeToken(bytes32 tokenId, bool tokenOnly, address from, uint256
   amount) external payable returns (uint256) {
68         address tokenManager = _create3Address(tokenId);
69         (uint256 tokenManagerType, address tokenAddress) =
   ITokenManagerProxy(tokenManager).getImplementationTypeAndTokenAddress();
70 
71         if (tokenOnly && msg.sender != tokenAddress) revert
   NotToken(msg.sender, tokenAddress);
72 
73         if (
74             tokenManagerType ==
   uint256(TokenManagerType.NATIVE_INTERCHAIN_TOKEN) || tokenManagerType ==
   uint256(TokenManagerType.MINT_BURN)
75         ) {
76             _burnToken(ITokenManager(tokenManager), tokenAddress, from,
   amount);
77         } else if (tokenManagerType ==
   uint256(TokenManagerType.MINT_BURN_FROM)) {
78             _burnTokenFrom(tokenAddress, from, amount);
79         ...
```

Let's assume that the token manager type is configured correctly to the `MINT_BURN` type. In this case, the `_burnToken` function will be called to burn the tokens.

Listing 34. Excerpt from TokenHandler._burnToken

```
167     function _burnToken(ITokenManager tokenManager, address tokenAddress,
    address from, uint256 amount) internal {
168         tokenManager.burnToken(tokenAddress, from, amount);
169     }
```

Exploit scenario

1. Alice, a token holder, grants approval to the ITS token manager to burn her tokens.
2. A vulnerability exists in the `InterchainTokenService` contracts, which makes it possible to steal the tokens. Similar to C6.
3. Bob, a malicious actor, sees the approval and exploits the vulnerability in the `InterchainTokenService` contracts to steal the tokens from Alice.

Recommendation

Refactor the teleporting flow to make possible for user to approve their tokens to the contracts, which the Printr protocol has control over.

Acknowledgment 1.1

The issue was acknowledged by the team.

Go back to Findings Summary

W4: Documentation disrepency with code

Impact:	Warning	Likelihood:	N/A
Target:	PrintrTeleport.sol	Type:	Code quality

Description

There is folder named `docs` in the `evm` folder. There is supposed to be documentation for the codebase. However, the information provided in the documentation is misleading and does not match the codebase.

One of the example from the documentation that says misleading information is:

Listing 35. Excerpt from cross-chain

```
62 ## Axelar ITS Integration
63
64 ### Interchain Token Standard
65
66 The protocol implements Axelar's Interchain Token Standard for cross-chain
   token functionality:
67
68 ```solidity
69 contract InterchainToken is ERC20, InterchainStandard {
70     bytes32 public immutable interchainTokenId;
71     address public immutable tokenManager;
72
73     constructor(
74         string memory name,
75         string memory symbol,
76         uint8 decimals,
77         bytes32 _interchainTokenId,
78         address _tokenManager
79     ) ERC20(name, symbol) {
80         interchainTokenId = _interchainTokenId;
81         tokenManager = _tokenManager;
82         _setupRole(MINTER_ROLE, _tokenManager);
83     }
84
85     function burn(address account, uint256 amount) external {
```

```
        require(msg.sender == tokenManager, "Only token manager");
        _burn(account, amount);
    }

    function mint(address account, uint256 amount) external {
        require(msg.sender == tokenManager, "Only token manager");
        _mint(account, amount);
    }
}
```

However, the Printr protocol implements different handling of the `TokenManager` calls:

Listing 36. Excerpt from Teleporting

```
fallback() external {
    if (msg.data.length < 4) {
        revert InvalidFunctionSelector();
    }

    bytes4 selector = bytes4(msg.data[:4]);
    (address account, uint256 amount) = abi.decode(msg.data[4:], (address, uint256));

    // teleportIn selector: 0x40c10f19 (mint))
    if (selector == 0x40c10f19) {
        teleportIn(account, amount);
        return;
    }
    // teleportOut selector: 0x9dc29fac (burn) or 0x79cc6790 (burnFrom)
    if (selector == 0x9dc29fac || selector == 0x79cc6790) {
        teleportOut(account, amount);
        return;
    }

    revert InvalidFunctionSelector();
}
```

Recommendation

Rewrite the documentation to match the codebase.

Partial solution 1.1

The issue was fixed by rewriting documentation. However, multiple documentation files are still outdated:

- `evm/docs/architecture/storage-pattern.md`

Listing 37. Excerpt from storage-pattern

```
 9 ### Storage Namespace
10
11 The protocol uses a single, well-defined storage namespace:
12
13 ```solidity
14 library PrintrStorageLib {
15     // keccak256(abi.encode(uint256(keccak256("printr.storage.main")) - 1)) &
   ~bytes32(uint256(0xff))
16     bytes32 internal constant PRINTR_STORAGE_LOCATION =
17         0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcd00;
18 }
19 ```
```

No such values are present in the codebase.

Listing 38. Excerpt from PrintrStorage

```
54 /// @dev ERC7201 storage location for Storage
55 /// @dev
   keccak256(abi.encode(uint256(keccak256("printr.storage.PrintrStorage")) - 1))
   &
56 /// ~bytes32(uint256(0xff))
57 bytes32 private constant PRINTR_STORAGE_LOCATION =
58     0x4115e7e53fb5d2198d5cadf1cb530b9c70f06c4e1b4565d7cc7245d4040dcb00;
```

The document mentioned above invalidly describes the storage layout `Storage` and mentions the storage access pattern, which is implemented in the codebase with a function under a different name `_storage` instead of `_getStorage`. The mentioned storage version is not present in the codebase. Check this file and rewrite it to match the codebase.

- evm/docs/concepts/advanced-features.md

Listing 39. Excerpt from advanced-features

```
171 ### Solution
172
173 The protocol automatically adjusts calculations:
174
175 ```solidity
176 function _adjustForDecimals(
177     uint256 amount,
178     address token
179 ) internal view returns (uint256) {
180     uint8 decimals = IERC20Metadata(token).decimals();
181
182     if (decimals < 18) {
183         // Scale up for tokens with fewer decimals
184         return amount * 10 ** (18 - decimals);
185     } else if (decimals > 18) {
186         // Scale down for tokens with more decimals
187         return amount / 10 ** (decimals - 18);
188     }
189
190     return amount;
191 }
192 ```
```

No such function is present in the codebase.

Listing 40. Excerpt from advanced-features

```
201 ## Stack Optimization Patterns
202
203 ### Challenge
204
205 Solidity's stack depth limitation (16 variables) requires careful management
    in complex functions.
206
207 ### Solution: Memory Structs
208
209 ```solidity
210 // Instead of many local variables
211 function complexOperation() {
```

```
    uint256 var1 = ...;
    uint256 var2 = ...;
    address var3 = ...;
    // Stack too deep error!
}

// Use memory struct
struct OperationContext {
    uint256 var1;
    uint256 var2;
    address var3;
    // ... more variables
}

function complexOperation() {
    OperationContext memory ctx;
    ctx.var1 = ...;
    ctx.var2 = ...;
    ctx.var3 = ...;
    // No stack issues!
}
```

No such pattern is implemented in the codebase.

Listing 41. Excerpt from advanced-features

```
### Reentrancy Protection

All value-transferring functions use OpenZeppelin's ReentrancyGuard:

```solidity
function buy(...) external payable nonReentrant {
    // Protected from reentrancy attacks
}
```
```

No such security mechanism is implemented in the codebase.

- `evm/docs/concepts/cross-chain.md`

There is no mention of the usage of `LayerZero` for the EVM chains.

Listing 42. Excerpt from cross-chain

```
88 ```solidity
89 contract Token is IToken, ERC20, ERC20Permit, InterchainStandard {
90     bytes32 public immutable tokenId;           // Universal identifier
   (deploySalt)
91     uint256 public immutable maxSupply;
92     address public immutable printr;
93     address public immutable tokenManager;
94     CompletionStatus public completionStatus;
95
96     constructor(
97         string memory _name,
98         string memory _symbol,
99         uint256 _maxSupply,
100         address _its,
101         address _tokenManager,
102         bytes32 _tokenId,
103         bytes32 _interchainTokenId
104     ) ERC20(_name, _symbol) ERC20Permit(_name) InterchainStandard(_its,
   _interchainTokenId) {
105         tokenId = _tokenId;                     // Set universal ID
106         // interchainTokenId is handled by InterchainStandard parent
107         maxSupply = _maxSupply;
108         printr = msg.sender;
109         tokenManager = _tokenManager;
110     }
111 }
112 ```
```

The described structure of the Token is not the same as the one in the codebase.

This list of discrepancies is not complete and we recommend reviewing all other files in the `docs` folder and rewriting them to match the codebase.

Partial solution 2.0

The following disrepency in the documentation have been additionally found:

Listing 43. Excerpt from pancake

```
177 ### Factory Interface
178
179 ```solidity
180 interface IPancakeV3Factory {
181     function createPool(address tokenA, address tokenB, uint24 fee)
182         external returns (address pool);
183     function getPool(address tokenA, address tokenB, uint24 fee)
184         external view returns (address);
185 }
186 ```
```

The following described function does not exist in the codebase.

Go back to Findings Summary

W5: No verification on which chain the user teleports tokens to

Impact:	Warning	Likelihood:	N/A
Target:	Telecoin.sol	Type:	Logic error

Description

The `teleport` function does not verify which chain the user teleports tokens to. However, during the `print` function, users specify the chains on which they want the printed token to exist.

Listing 44. Excerpt from Telecoin

```
77 function teleport(
78     IPrintrTeleport.TeleportParams calldata params,
79     IPrintrTeleport.TeleportProtocol protocol
80 ) public payable virtual {
81     if (protocol == IPrintrTeleport.TeleportProtocol.UNSPECIFIED) {
82         revert InvalidProtocol();
83     }
84     if (protocol == IPrintrTeleport.TeleportProtocol.ITS) {
85         // Get the teleport fee using the unified function
86         (uint256 nativeTeleportFee,,) =
   IPrintrTeleport(printr).quoteTeleportFee(address(this), params, protocol);
87
88         // Transfer the fee to printr
89         if (nativeTeleportFee > 0) {
90             payable(printr).transfer(nativeTeleportFee);
91         }
92
93         interchainTransfer(params.destChain, params.destAddress,
   params.amount, params.metadata);
94         return;
95     }
96     if (protocol == IPrintrTeleport.TeleportProtocol.LZ) {
97         // Use LayerZero for teleportation via PrintrTeleport
98         IPrintrTeleport(printr).transmitLzSend{ value: msg.value
   }(telecoinId, msg.sender, params);
99     }
```

```
100 }
```

The function above contains no verification on which chain the user teleports tokens to. This can lead to users teleporting tokens to chains on which the token is not supposed to be printed or exist.

This behavior applies to all tokens, not only `PrintrTeleportingTelecoin` but also other telecoin types.

Recommendation

Update the `teleport` function to verify which chain the user teleports tokens to if it is important for the token to not exist on chains other than those specified during the `print` function.

Acknowledgment 1.1

The issue was acknowledged by the team.

Go back to Findings Summary

W6: Code size is too large

Impact:	Warning	Likelihood:	N/A
Target:	TelecoinFactory.sol	Type:	Configuration

Description

The following contracts have runtime bytecode sizes that are too large:

- TelecoinFactory

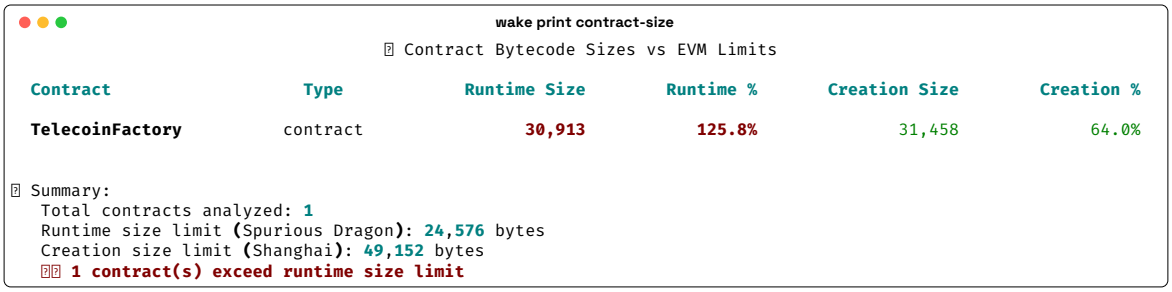


Figure 1. Contract size

Recommendation

Reduce the code size of the mentioned contracts for the project to be deployable to the Mainnet.

Fix 1.1

The issue was fixed by dividing the contract into smaller contracts `TeleportingTelecoinFactory` and `TeleportingMainFactory`.

Go back to Findings Summary

W7: Whitelisted addresses cannot be updated

Impact:	Warning	Likelihood:	N/A
Target:	Intent.sol	Type:	Access control

Description

The `Intent` contract inherits from `Whitelist` contract but lacks functionality to update whitelisted addresses after deployment. This immutability poses a security risk as there is no way to remove compromised addresses or add new legitimate ones, potentially leading to:

- inability to revoke access from compromised addresses;
- no way to add new legitimate addresses; and
- permanent lockout if all whitelisted addresses become inaccessible.

Recommendation

Add administrative functions to the `Whitelist` contract that allow privileged roles to:

- add new addresses to the whitelist;
- remove addresses from the whitelist; and
- update existing whitelisted addresses.

Ensure these functions are protected by appropriate access control mechanisms.

Acknowledgment 1.1

The issue was acknowledged by the team.

Go back to Findings Summary

W8: Amount calculation uses `maxPrice` for calculation, resulting in operating with smaller amount of tokens

Impact:	Warning	Likelihood:	N/A
Target:	PrintrTeleport.sol	Type:	Arithmetics

Description

The amount calculation uses `maxPrice` for calculation, resulting in operating with a smaller amount of tokens. Users are still able to use the `spend` function to buy a specific amount of tokens.

Listing 45. Excerpt from PrintrTrading

```
190 if (maxPrice != 0) {
191     params.amount = amount * maxPrice / PRECISION;
192 } else {
193     if (msg.value > 0) {
```

Recommendation

Use the current price for amount calculation.

Go back to Findings Summary

I1: No validation for zero `params.amount`

Impact:	Info	Likelihood:	N/A
Target:	PrintrTrading.sol	Type:	Code quality

Description

The Printr protocol provides functionality for buying and selling tokens. However, there is no validation for zero `params.amount`, which can lead to incorrect calculations and protocol behavior.

Listing 46. Excerpt from PrintrTrading

```
387 }
388
389 // Calculate projected issuedSupply based on new curve reserve
390 issuedSupply = initialTokenReserve - curveConstant / (curve.virtualReserve +
    curve.reserve + curveCost);
391
392 // Calculate and round up minimum fee (1 wei) if percentage would round to 0
```

Exploit scenario

Recommendation

Validate that `params.amount` is not zero when `priceLimit` is not specified and revert the transaction if it is zero.

Fix 1.1

The issue was fixed by validating that `params.amount` is not zero and reverting the transaction if it is zero.

Go back to Findings Summary

I2: No explanation for using specific constants for gas limit

Impact:	Info	Likelihood:	N/A
Target:	PrintrTeleport.sol	Type:	Code quality

Description

There is no explanation why specific constant which is responsible for gas limit is used and will be enough for the operation with the metadata.

Listing 47. Excerpt from PrintrTeleport

```
651     uint256 executionGasLimit = params.metadata.length == 0 ? 200_000 :
    500_000;
652
653     // Estimate the gas fee for the hub message
654     gasFee = gasService.estimateGasFee(
655         params.destChain,
656         interchainTokenService.toHexString(), // Convert address to string
    for Axelar
657         payload,
658         executionGasLimit,
659         "" // Empty params for standard estimation
660     );
661
662     return gasFee;
663 }
```

It is possible that the constant is not enough for the operation with the metadata.

Exploit scenario

Recommendation

Add an explanation why the constant is used and will be enough for the operation with the metadata.

Fix 1.1

The issue was fixed by adding an explanation why the constant is used.

Go back to Findings Summary

I3: Misleading information in NatSpec

Impact:	Info	Likelihood:	N/A
Target:	Teleporting.sol	Type:	Code quality

Description

The NatSpec of the `Teleporting.teleportOut` function contains misleading information.

Listing 48. Excerpt from Teleporting

```
32 function teleportIn(address to, uint256 value) public {
33     if (msg.sender != itsTokenManager && msg.sender != printr) {
34         revert UnauthorizedAccount(msg.sender);
35     }
36     if (to == address(0)) {
37         revert ERC20InvalidReceiver(address(0));
38     }
39
40     emit TeleportIn(telecoinId, to, value);
41     _update(address(this), to, value);
42 }
```

In the function above, the NatSpec states that the function can be called by the token handler. However, the function can be called only by the ITS token manager or Printr.

Exploit scenario

Recommendation

Update the NatSpec to specify that the function can be called only by the ITS token manager or Printr.

Fix 1.1

The issue was fixed by updating condition which entity can call the function

and now it can be called by the token manager. Besides, the NatSpec was updated to specify which entity can call the function.

Go back to Findings Summary

I4: Incorrect NatSpec title in `Printed.sol` contract

Impact:	Info	Likelihood:	N/A
Target:	Printed.sol	Type:	Code quality

Description

The contract in `Printed.sol` contains an incorrect NatSpec documentation where the `@title` tag specifies `PrintrTelecoin` instead of the actual contract name. This inconsistency in documentation makes the code harder to maintain and may confuse developers about the contract's identity.

Recommendation

Update the NatSpec documentation in `Printed.sol` to use the correct contract name in the `@title` tag:

```
/// @title Printed
```

Acknowledgment 1.1

The issue was acknowledged by the team.

Go back to Findings Summary

I5: Inconsistent contract names

Impact:	Info	Likelihood:	N/A
Target:	PrintrTeleportingTelecoin.sol, PrintrMainTelecoin.sol, Teleporting.sol	Type:	Code quality

Description

Multiple contract files in the codebase violate the Solidity style guide's naming convention, which explicitly states that "Contract and library names should match their filenames." The following inconsistencies were identified:

- files `PrintrTeleportingTelecoin.sol` and `PrintrMainTelecoin.sol` both declare their contract name as `PrintrTelecoin`; and
- file `Teleporting.sol` declares its contract name as `TeleportingTelecoin`.

This naming inconsistency makes the codebase harder to navigate and maintain.

Recommendation

Rename the contracts to match their respective filenames:

- rename contract in `PrintrTeleportingTelecoin.sol` to `PrintrTeleportingTelecoin`;
- rename contract in `PrintrMainTelecoin.sol` to `PrintrMainTelecoin`; and
- rename file `Teleporting.sol` to `TeleportingTelecoin.sol` to match its contract name.

Acknowledgment 1.1

The issue was acknowledged by the team.

Go back to Findings Summary

I6: Unused errors

Impact:	Info	Likelihood:	N/A
Target:	src/evm/*	Type:	Code quality

Description

There are unused errors in the codebase:

Listing 49. Excerpt from IIntent

```
15 error Unauthorized();
```

Listing 50. Excerpt from IIntent

```
21 error InvalidAuthorization();
```

Listing 51. Excerpt from IIntent

```
39 error RefundFailed();
```

Listing 52. Excerpt from IInterchainTokenFactory

```
13 error ZeroAddress();
14 error InvalidChainName();
15 error InvalidMinter(address minter);
16 error NotMinter(address minter);
17 error NotSupported();
18 error RemoteDeploymentNotApproved();
19 error InvalidTokenId(bytes32 tokenId, bytes32 expectedTokenId);
20 error ZeroSupplyToken();
21 error NotToken(address tokenAddress);
```

Listing 53. Excerpt from IInterchainTokenService

```
41 error InvalidChainName();
42 error NotRemoteService();
43 error TokenManagerDoesNotExist(bytes32 tokenId);
```

```
44 error ExecuteWithInterchainTokenFailed(address contractAddress);
45 error ExpressExecuteWithInterchainTokenFailed(address contractAddress);
46 error TokenManagerDeploymentFailed(bytes error);
47 error InterchainTokenDeploymentFailed(bytes error);
48 error InvalidMessageType(uint256 messageType);
49 error InvalidMetadataVersion(uint32 version);
50 error InvalidExpressMessageType(uint256 messageType);
51 error TakeTokenFailed(bytes data);
52 error GiveTokenFailed(bytes data);
53 error TokenHandlerFailed(bytes data);
54 error EmptyData();
55 error PostDeployFailed(bytes data);
56 error ZeroAmount();
57 error CannotDeploy(TokenManagerType);
58 error CannotDeployRemotelyToSelf();
59 error InvalidPayload();
60 error GatewayCallFailed(bytes data);
61 error EmptyTokenName();
62 error EmptyTokenSymbol();
63 error EmptyParams();
64 error EmptyDestinationAddress();
65 error EmptyTokenAddress();
66 error NotSupported();
67 error NotInterchainTokenFactory(address sender);
```

Listing 54. Excerpt from ILiquidityModule

```
55 error SqrtPriceOutOfBounds();
```

Listing 55. Excerpt from IPrintrStorage

```
28 error ExceedsIssuedSupply();
```

Listing 56. Excerpt from IPrintrTeleport

```
27 error TeleportInFailed();
```

Listing 57. Excerpt from IPrintrTeleport

```
67 error MetadataMissingTeleportFee();
```

Exploit scenario

Recommendation

Remove the unused errors.

Partial solution 1.1

The issue was fixed by removing the unused errors. However, there is still one unused error in the codebase:

Listing 58. Excerpt from IPrintrTeleport

```
37 error UnauthorizedSender();
```

Go back to Findings Summary

I7: Unused interfaces and libraries

Impact:	Info	Likelihood:	N/A
Target:	src/evm/*	Type:	Code quality

Description

There are unused interfaces in the codebase:

Listing 59. Excerpt from IMerchantmoeLB

```
19 interface ILBFactory {
```

Listing 60. Excerpt from IMerchantmoeLB

```
26 interface ILBRouter {
```

There is unused library in the codebase:

Listing 61. Excerpt from AddressBytes

```
10 library AddressBytes {
```

Exploit scenario

Recommendation

Remove the unused library and interfaces.

Partial solution 1.1

The issue was fixed by removing the unused library and interfaces. However, there is still one unused interface in the codebase:

Listing 62. Excerpt from IMerchantmoeLB

```
11 interface ILBPair {
```

Go back to Findings Summary

I8: Unused events

Impact:	Info	Likelihood:	N/A
Target:	src/evm/*	Type:	Code quality

Description

There are unused events in the codebase:

Listing 63. Excerpt from IInterchainTokenFactory

```
25 event DeployRemoteInterchainTokenApproval(
```

Listing 64. Excerpt from IInterchainTokenFactory

```
35 event RevokedDeployRemoteInterchainTokenApproval(
```

Listing 65. Excerpt from IInterchainTokenService

```
82 event InterchainTransfer(
```

Listing 66. Excerpt from IInterchainTokenService

```
90 event InterchainTransferReceived(
```

Listing 67. Excerpt from IInterchainTokenService

```
99 event TokenMetadataRegistered(address indexed tokenAddress, uint8 decimals);
100 event LinkTokenStarted(
```

Listing 68. Excerpt from IInterchainTokenService

```
108 event InterchainTokenDeploymentStarted(
```

Listing 69. Excerpt from IInterchainTokenService

```
116 event TokenManagerDeployed(
```

Listing 70. Excerpt from IInterchainTokenService

```
119 event InterchainTokenDeployed(
```

Listing 71. Excerpt from IInterchainTokenService

```
127 event InterchainTokenIdClaimed(bytes32 indexed tokenId, address indexed
    deployer, bytes32 indexed salt);
```

Listing 72. Excerpt from IAlgebra

```
18 event PoolCreated(address indexed token0, address indexed token1, address
   pool);
```

Listing 73. Excerpt from ILynex

```
20 event PoolCreated(
```

Listing 74. Excerpt from ILynex

```
85 event IncreaseLiquidity(uint256 indexed tokenId, uint128 liquidity, uint256
   amount0, uint256 amount1);
```

Listing 75. Excerpt from ILynex

```
92 event DecreaseLiquidity(uint256 indexed tokenId, uint128 liquidity, uint256
   amount0, uint256 amount1);
```

Listing 76. Excerpt from ILynex

```
99 event Collect(uint256 indexed tokenId, address recipient, uint256 amount0,
   uint256 amount1);
```

Listing 77. Excerpt from IMerchantmoe

```
20 event PoolCreated(
```

Listing 78. Excerpt from IMerchantmoe

```
85 event IncreaseLiquidity(uint256 indexed tokenId, uint128 liquidity, uint256
   amount0, uint256 amount1);
```

Listing 79. Excerpt from IMerchantmoe

```
92 event DecreaseLiquidity(uint256 indexed tokenId, uint128 liquidity, uint256
   amount0, uint256 amount1);
```

Listing 80. Excerpt from IMerchantmoe

```
99 event Collect(uint256 indexed tokenId, address recipient, uint256 amount0,
   uint256 amount1);
```

Listing 81. Excerpt from ITraderJoe

```
19 event LBPairCreated(
```

Listing 82. Excerpt from IUniswap

```
12 event OwnerChanged(address indexed oldOwner, address indexed newOwner);
```

Listing 83. Excerpt from IUniswap

```
20 event PoolCreated(
```

Listing 84. Excerpt from IUniswap

```
28 event FeeAmountEnabled(uint24 indexed fee, int24 indexed tickSpacing);
```

Listing 85. Excerpt from IUniswap

```
121 }
```

Listing 86. Excerpt from IUniswap

```
146 /// @param recipient The address of the account that received the collected
    tokens
```

Listing 87. Excerpt from IUniswap

```
152 /// @dev Throws if the token ID is not valid.
```

Listing 88. Excerpt from IUniswap

```
160 /// @return tickUpper The higher end of the tick range for the position
```

Listing 89. Excerpt from IPrintrFeeDistribution

```
34 event FeeDistributionFailed(address indexed recipient, uint256 amount, string
   reason);
```

Exploit scenario

Recommendation

Remove the unused events.

Partial solution 1.1

The issue was addressed by removing the previously identified unused events. However, several unused events still remain in the codebase:

Listing 90. Excerpt from IInterchainTokenFactory

```
25 event RevokedDeployRemoteInterchainTokenApproval(
```

Listing 91. Excerpt from IInterchainTokenService

```
54 event InterchainTransfer(
```

Listing 92. Excerpt from IInterchainTokenService

```
71 event TokenMetadataRegistered(address indexed tokenAddress, uint8 decimals);
```

Listing 93. Excerpt from IInterchainTokenService

```
72 event LinkTokenStarted(
```

Listing 94. Excerpt from IInterchainTokenService

```
80 event InterchainTokenDeploymentStarted(
```

Listing 95. Excerpt from IInterchainTokenService

```
91 event InterchainTokenDeployed(
```

Listing 96. Excerpt from IInterchainTokenService

```
99 event InterchainTokenIdClaimed(bytes32 indexed tokenId, address indexed
   deployer, bytes32 indexed salt);
```

Listing 97. Excerpt from IAlgebra

```
18 event PoolCreated(address indexed token0, address indexed token1, address
   pool);
```

Listing 98. Excerpt from IAlgebra

```
20 /// @notice Returns the pool address for a given pair of tokens, or address 0
   if it does not exist
```

Listing 99. Excerpt from ILynex

```
92 event DecreaseLiquidity(uint256 indexed tokenId, uint128 liquidity, uint256
   amount0, uint256 amount1);
```

Listing 100. Excerpt from ILynex

```
99 event Collect(uint256 indexed tokenId, address recipient, uint256 amount0,
   uint256 amount1);
```

Listing 101. Excerpt from IMerchantmoe

```
20 event PoolCreated(
```

Listing 102. Excerpt from IMerchantmoe

```
85 event IncreaseLiquidity(uint256 indexed tokenId, uint128 liquidity, uint256
   amount0, uint256 amount1);
```

Listing 103. Excerpt from IMerchantmoe

```
92 event DecreaseLiquidity(uint256 indexed tokenId, uint128 liquidity, uint256
   amount0, uint256 amount1);
```

Listing 104. Excerpt from IMerchantmoe

```
99 event Collect(uint256 indexed tokenId, address recipient, uint256 amount0,
   uint256 amount1);
```

Listing 105. Excerpt from ITraderJoe

```
19 event LBPairCreated(
```

Listing 106. Excerpt from IUniswap

```
12 event OwnerChanged(address indexed oldOwner, address indexed newOwner);
```

Listing 107. Excerpt from IUniswap

```
28 event FeeAmountEnabled(uint24 indexed fee, int24 indexed tickSpacing);
```

Listing 108. Excerpt from IUniswap

```
135 event IncreaseLiquidity(uint256 indexed tokenId, uint128 liquidity, uint256
    amount0, uint256 amount1);
```

Listing 109. Excerpt from IUniswap

```
141 event DecreaseLiquidity(uint256 indexed tokenId, uint128 liquidity, uint256
    amount0, uint256 amount1);
```

Listing 110. Excerpt from IUniswap

```
149 event Collect(uint256 indexed tokenId, address recipient, uint256 amount0,
    uint256 amount1);
```

Go back to Findings Summary

I9: Unused functions

Impact:	Info	Likelihood:	N/A
Target:	src/evm/*	Type:	Code quality

Description

There are unused functions in the codebase:

Listing 111. Excerpt from AddressBytes

```
25 function toAddress(
```

Listing 112. Excerpt from AddressBytes

```
44 function toBytes(
```

Exploit scenario

Recommendation

Remove the unused functions.

Fix 1.1

The issue was fixed by removing the unused functions.

Go back to Findings Summary

I10: Unused imports

Impact:	Info	Likelihood:	N/A
Target:	src/evm/*	Type:	Code quality

Description

There are unused functions in the codebase:

Listing 113. Excerpt from ITeleporting

```
4 import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
5 import "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";
```

Listing 114. Excerpt from AlgebraLiquidity

```
6 import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
```

Listing 115. Excerpt from LynexLiquidity

```
7 import "@openzeppelin/contracts/utils/math/Math.sol";
```

Listing 116. Excerpt from MerchantmoeLiquidity

```
8 import "@openzeppelin/contracts/utils/math/Math.sol";
```

Listing 117. Excerpt from WAGMILiquidity

```
6 import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
7 import "@openzeppelin/contracts/utils/math/Math.sol";
```

Exploit scenario

Recommendation

Remove the unused imports.

Partial solution 1.1

The issue was fixed by removing the unused imports. However, there is still one unused import in the codebase:

Listing 118. Excerpt from MerchantmoeLiquidity

```
6 import "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";
```

Go back to Findings Summary

Report Revision 1.1

Revision Team

Member's Name	Position
Dmytro Khimchenko	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

Overview

Since there were no comprehensive changes in this revision, the complete overview is listed in the Executive Summary section Revision 1.1.

Report Revision 2.0

Revision Team

Member's Name	Position
Dmytro Khimchenko	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The system has not been changed since the previous revision.

Trust Model

The trust model has not been changed since the previous revision.

Fuzzing

The fuzzing approach is the same as in the previous revision. The fuzzing tests have been updated to reflect the fixes made.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

C8: DOS when handling initial price fix due to incorrect liquidity parameter

Critical severity issue

Impact:	High	Likelihood:	High
Target:	liquidity/*	Type:	Logic error

Description

The Printr protocol deploys liquidity to external DEXes when a token reaches its completion threshold. Before deployment, if the current pool price differs from the expected graduation price, the protocol fixes the price by creating a temporary position, executing a swap, and then removing the temporary position.

The issue arises in the `_decreaseLiquidity` function, which is called to remove liquidity from the temporary position. All liquidity module implementations use `type(uint128).max` as the liquidity parameter:

Listing 119. Excerpt from LynexLiquidity._decreaseLiquidity

```
236 function _decreaseLiquidity(
237     uint256 positionId
238 ) internal override {
239     lynexPositionManager.decreaseLiquidity(
240         ILynexPositionManager.DecreaseLiquidityParams({
241             tokenId: positionId,
242             liquidity: type(uint128).max,
243             amount0Min: 0,
244             amount1Min: 0,
245             deadline: block.timestamp
246         })
247     );
248 }
```

Listing 120. Excerpt from AlgebraLiquidity._decreaseLiquidity

```
233 function _decreaseLiquidity(
234     uint256 positionId
235 ) internal override {
236     algebraPositionManager.decreaseLiquidity(
237         IAlgebraPositionManager.DecreaseLiquidityParams({
238             tokenId: positionId,
239             liquidity: type(uint128).max,
240             amount0Min: 0,
241             amount1Min: 0,
242             deadline: block.timestamp
243         })
244     );
245 }
```

Listing 121. Excerpt from UniswapV3Liquidity._decreaseLiquidity

```
335 function _decreaseLiquidity(
336     uint256 positionId
337 ) internal virtual override {
338     uniPositionManager.decreaseLiquidity(
339         INonfungiblePositionManager.DecreaseLiquidityParams({
340             tokenId: positionId,
341             liquidity: type(uint128).max,
342             amount0Min: 0,
343             amount1Min: 0,
344             deadline: block.timestamp
345         })
346     );
347 }
```

However, Uniswap V3 and its forks (including Lynex and Algebra) require that the position's liquidity is greater than or equal to the requested liquidity amount. Since no position can have `type(uint128).max` liquidity, the transaction will always revert:

```
function decreaseLiquidity(DecreaseLiquidityParams calldata params)
    external
    payable
    returns (uint256 amount0, uint256 amount1)
```

```
{
    require(params.liquidity > 0);
    Position storage position = _positions[params.tokenId];

    uint128 positionLiquidity = position.liquidity;
    require(positionLiquidity >= params.liquidity); // This will always fail
    ...
}
```

As a result, liquidity deployment fails whenever initial price fixing is required, preventing tokens from graduating to external DEXes.

Exploit scenario

1. Alice, a user, purchases enough tokens to reach the completion threshold.
2. The pool's current price differs from the expected graduation price, triggering the price fix mechanism.
3. The protocol calls `BaseLiquidityModule._fixInitialPrice`, which creates a temporary position and executes a swap.
4. The protocol attempts to remove the temporary position by calling `_decreaseLiquidity` with `type(uint128).max` as the liquidity parameter.
5. The external DEX's position manager reverts because the position's liquidity is less than `type(uint128).max`.
6. The entire liquidity deployment transaction fails, and the token cannot graduate to the external DEX.

As a result, any transaction that would cause graduation of the token fails, and the token cannot graduate to the external DEX.

Recommendation

Retrieve the position's actual liquidity amount and use it as the parameter for the `decreaseLiquidity` call instead of using `type(uint128).max`.

Fix 2.1

The issue was fixed by retrieving the position's actual liquidity amount and using it as the parameter for the `decreaseLiquidity` call instead of using `type(uint128).max`.

Go back to Findings Summary

C9: CREATE2 deployment of LZChannel results in different addresses across chains due to varying lzEndpoint values

Critical severity issue

Impact:	High	Likelihood:	High
Target:	PrintrTeleport.sol	Type:	Logic error

Description

The LZChannel contract is deployed using CREATE2 (via `new LZChannel{salt: salt}()` syntax), which calculates the deployment address based on the deployer address, salt, and contract bytecode. However, the contract bytecode includes the constructor parameters, including the `lzEndpoint` address.

Since CREATE2 includes constructor parameters in the address calculation, the same salt will produce different LZChannel addresses on chains with different LZEndpoint values. This breaks cross-chain compatibility, preventing token transfers to these chains or causing token loss due to unexpected destination addresses.

Listing 122. Excerpt from PrintrTeleport

```
894 function _getOrDeployLzChannel(
895     TeleportProtocol protocol
896 ) internal returns (address channel) {
897     // Validate that protocol is a LayerZero protocol
898     if (
899         protocol != TeleportProtocol.LZ_FAST && protocol !=
    TeleportProtocol.LZ_SECURE
900             && protocol != TeleportProtocol.LZ_SLOW
901     ) {
902         revert InvalidProtocol();
903     }
904
```

```
905     channel = _calculateLzChannelAddress(protocol);
906
907     if (channel.code.length == 0) {
908         bytes32 solanaPeer = _getSolanaPeerForProtocol(protocol);
909         bytes32 salt = keccak256(bytes("LZChannel"));
910         address deployed = address(new LZChannel{ salt: salt }(lzEndpoint,
    protocol, solanaPeer));
911         if (deployed != channel) {
912             revert Create2AddressMismatch();
913         }
914     }
915 }
```

Exploit scenario

1. Alice, a user, initiates a cross-chain token transfer from Ethereum mainnet to HyperEVM using the protocol.
2. The protocol assumes the destination LZChannel address on HyperEVM is the same as Ethereum that uses CREATE2 with the same salt.
3. Bob, the protocol operator, deploys LZChannel on HyperEVM, but the different `lzEndpoint` address results in a different contract address than expected.
4. Alice's LayerZero messages are sent to the calculated address on HyperEVM, which does not contain the actual LZChannel contract.
5. Alice's tokens are permanently lost as they are sent to an address without the proper contract logic.

Recommendation

Use CREATE3 deployment or avoid passing the endpoint address as a constructor parameter.

Fix 2.1

The issued fixed by using custom create3 deployer.

Go back to Findings Summary

H4: Overestimated base token amount required for ERC20 trades after graduation

High severity issue

Impact:	Medium	Likelihood:	High
Target:	PrintrTrading.sol	Type:	Logic error

Description

When users purchase tokens with ERC20 base tokens after token graduation, the protocol overestimates the amount of base tokens required from the user. The issue arises in the `buy` function when liquidity has already been deployed and a price limit is specified.

The function calculates the required base token amount by multiplying the desired token amount by the maximum price limit:

Listing 123. Excerpt from PrintrTrading.buy

```
190 // Route to liquidity pool if already deployed (threshold = 0)
191 if (curve.completionThreshold == 0) {
192     if (maxPrice != 0) {
193         params.amount = amount * maxPrice / PRECISION;
194     } else {
195         if (msg.value > 0) {
196             params.amount = msg.value;
197         } else {
198             revert LiquidityAlreadyDeployed();
199         }
200     }
201
202     _spendInLiquidityPool(curve, params);
```

This formula `amount * maxPrice / PRECISION` is fundamentally incorrect because it assumes all tokens can be purchased at a constant price. However, Automated Market Makers (AMMs) use dynamic pricing where the price

changes continuously during the swap:

- the first token costs less;
- each subsequent token costs progressively more;
- the price increases along the bonding curve as liquidity is consumed; and
- the actual cost is the integral of the price function over the swap range, not a simple multiplication.

Additionally, `maxPrice` represents a slippage protection limit (the maximum price the user is willing to accept), not the actual execution price. The `maxPrice` is later converted to `sqrtPriceLimitX96` for AMM price limit enforcement. The actual average execution price will be lower than this maximum, especially for smaller trades or trades with conservative slippage settings.

The correct cost calculation requires querying the AMM to simulate the swap, which accounts for liquidity distribution, fee tiers, tick ranges, and price impact. The protocol already has `_quoteLiquidityExactOutput` function for this purpose, but it is not used here.

The overestimated amount is then used to transfer tokens from the user:

Listing 124. Excerpt from PrintrTrading._spendInLiquidityPool

```
765     IERC20(curve.basePair).safeTransferFrom(params.account, address(this),
    params.amount);
766 }
```

As a result, the protocol attempts to transfer more base tokens from the user than actually required for the swap. While the excess tokens are refunded for ERC20 tokens (line 787), the initial transfer may fail if the user has not approved sufficient tokens based on the overestimated amount, even though they have enough tokens for the actual swap cost.

Exploit scenario

1. Alice, a user, wants to buy 10,000 tokens of a newly graduated meme token using USDC as the base token.
2. The token has just graduated and has a small liquidity pool with shallow liquidity, which is typical for newly graduated tokens.
3. The current pool price is 0.10 USDC per token. Due to the low liquidity, Alice sets `maxPrice` to 0.50 USDC per token as a slippage protection limit to account for potential price impact.
4. The protocol incorrectly calculates the required base amount as 10,000 * 0.50 = 5,000 USDC, treating `maxPrice` as the execution price.
5. The actual swap cost through the AMM would be approximately 1,800 USDC. Due to low liquidity, the price moves from 0.10 USDC to approximately 0.30 USDC during the swap, resulting in an average execution price of 0.18 USDC per token.
6. Alice has approved exactly 2,500 USDC, which provides a 38% buffer over the actual swap cost and should be more than sufficient.
7. The protocol attempts to transfer 5,000 USDC from Alice, but the transfer fails due to insufficient allowance.
8. Alice's transaction reverts, even though she has approved twice the amount needed for the actual swap cost.
9. This issue is particularly severe for newly graduated tokens, where low liquidity causes significant price movements, leading users to set higher `maxPrice` values for protection, which in turn causes even larger overestimation by the protocol.

Recommendation

Use the `_quoteLiquidityExactOutput` function to accurately estimate the required base token amount before attempting to transfer tokens from the

user. This ensures that only the necessary amount is requested, preventing transaction failures due to overestimation.

Fix 2.1

The issue was fixed by using the `_quoteLiquidityExactOutput` function to accurately estimate the required base token amount before attempting to transfer tokens from the user.

Go back to Findings Summary

L6: Treasury accepts any NFT token instead of only LP position NFTs

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	Treasury.sol	Type:	Access control

Description

The contract is supposed to only receive LP position NFTs, but it accepts any NFT. Users may send NFTs by mistake, causing them to be permanently locked.

Exploit scenario

1. Alice, a user, mistakenly sends a valuable NFT to the Treasury contract instead of the intended LP position NFT.
2. The contract accepts the NFT without validation.
3. Alice's NFT becomes permanently locked in the contract with no recovery mechanism.

Recommendation

Validate that `msg.sender` is the expected LP position NFT contract before accepting NFT transfers.

Acknowledgment 2.1

The issue was acknowledged by the team.

Go back to Findings Summary

W9: Merchantmoe Liquidity Module is not implemented correctly

Impact:	Warning	Likelihood:	N/A
Target:	MerchantmoeLiquidity.sol	Type:	Configuration

Description

The protocol uses different liquidity modules for deploying liquidity after the token graduation. One of the modules that can be used for the Mantle chain is the Merchantmoe Liquidity Module. However, the interaction with the Merchantmoe protocol is implemented incorrectly, causing every transaction that interacts with the Merchantmoe protocol to revert.

The following function calls the `MerchantmoeFactory.createPool` function, which does not exist in the Merchantmoe protocol.

Listing 125. Excerpt from MerchantmoeLiquidity.createPool

```
143     );
144
145     IERC20(params.tokenIn).forceApprove(address(merchantmoeRouter), 0);
146 }
147
148 /**
149  * @notice Sets up token approvals for Merchantmoe position manager
150  * @param token Token address
151  * @param basePair Base pair address
152  */
153 function _setupApprovals(address token, address basePair) internal override
    {
154     IERC20(basePair).forceApprove(address(merchantmoePositionManager),
    type(uint256).max);
155     IERC20(token).forceApprove(address(merchantmoePositionManager),
    type(uint256).max);
156 }
157
158 /**
159  * @notice Cleans up token approvals after deployment
```

```
160  * @param token Token address
161  * @param basePair Base pair address
162  */
163 function _cleanupApprovals(address token, address basePair) internal
    override {
164     IERC20(basePair).forceApprove(address(merchantmoePositionManager), 0);
165     IERC20(token).forceApprove(address(merchantmoePositionManager), 0);
166 }
167
168 /**
169  * @notice Gets the current pool price from Merchantmoe
170  * @param token Token address
171  * @param basePair Base pair address
172  * @return Current sqrt price in X96 format
173  */
174 function _getCurrentPoolPrice(address token, address basePair) internal view
    override returns (uint160) {
```

As a result, no liquidity can be deployed to the Merchantmoe protocol.

Recommendation

Integrate with the Merchantmoe protocol correctly or remove the Merchantmoe Liquidity Module from the protocol.

Fix 2.1

The issue was fixed by deleting integration with Merchantmoe protocol.

Go back to Findings Summary

W10: Value overflow may limit realistic token configurations

Impact:	Warning	Likelihood:	N/A
Target:	Printr	Type:	Logic error

Description

The `curveConstant` calculation and intermediate variables in the `effectivePrice` function within `_estimateTokenCost` are susceptible to uint256 overflow (maximum value ≈ 10^77).

The overflow occurs through the following calculations:

1. Initial Token Reserve almost equals the total supply of the token being created. Common values reach 10^32 (e.g., PEPE, SHIB) or 10^26 (e.g., Ethereum).
2. `virtualReserve` is calculated as `(relative token price) × InitialTokenReserve`. The relative token price ranges from 0.00001 to 1000 in ETH or USD as base token.
3. Assume the relative telecoin price is 1000 of base token.
4. `curveConstant` is calculated as `InitialTokenReserve × virtualReserve`. With typical values: 10^32 × 10^32 × 1000 ≈ 10^67.
5. The overflow occurs when the internal calculation in `effectivePrice` computes `curveConstant × PRECISION`. With PRECISION = 10^18, this yields: 10^67 × 10^18 ≈ 10^85, which exceeds the uint256 maximum.

Recommendation

Ensure the `curveConstant` does not exceed 10^59 to avoid transaction revert. Ensure users set values that prevent overflow from occurring.

Fix 2.1

The issue is fixed by division ealier to preventing overflow. The divide by multiply effect is limited, because it used for price limit and effectivePrice that is used for event emission or return value of view function.

Go back to Findings Summary

W11: Confusing parameter semantics in `_buy` function due to overloaded `params.amount` usage

Impact:	Warning	Likelihood:	N/A
Target:	PritrTrading.sol	Type:	Code quality

Description

The `params.amount` field in the `_buy` function changes meaning during execution: initially representing tokens to buy, then base currency cost, and finally tokens purchased. This overloading reduces code clarity and increases maintenance risk.

Recommendation

Use return values or different variables to maintain consistent variable semantics.

Acknowledgment 2.1

The issue was acknowledged by the team.

Go back to Findings Summary

W12: HyperSwap behavior may differ from UniswapV3 due to modified source code

Impact:	Warning	Likelihood:	N/A
Target:	PrintrTrading.sol	Type:	Denial of service

Description

HyperSwap uses UniswapV3 with modified source code, causing behavior to differ from the official UniswapV3 implementation.

The `NonfungiblePositionManager` should be compatible with UniswapV3, but token graduation fails during edge cases. Since HyperSwap's `NonfungiblePositionManager` is not open source, the specific cause of these failures cannot be determined.

Recommendation

Verify HyperSwap's behavior through testing or use alternative liquidity pools with confirmed compatibility.

Acknowledgment 2.1

The issue was acknowledged by the team.

Go back to Findings Summary

I11: Missing permit invalidation mechanism

Impact:	Info	Likelihood:	N/A
Target:	ERC20.sol	Type:	Code quality

Description

The ERC20 implementation includes EIP-2612 permit functionality, which allows users to approve token spending via off-chain signatures. However, the implementation lacks a mechanism for users to invalidate outstanding permit signatures without executing them first.

Listing 126. Excerpt from ERC20.permit

```
143 function permit(
144     address owner,
145     address spender,
146     uint256 value,
147     uint256 deadline,
148     uint8 v,
149     bytes32 r,
150     bytes32 s
151 ) public virtual {
152     if (deadline < block.timestamp) {
153         revert ERC2612ExpiredSignature(deadline);
154     }
155
156     // Unchecked because the only math done is incrementing
157     // the owner's nonce which cannot realistically overflow.
158     unchecked {
159         bytes32 structHash =
160             keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
    nonces[owner]++, deadline));
161
162         bytes32 hash = keccak256(abi.encodePacked("\x19\x01",
    DOMAIN_SEPARATOR(), structHash));
163
164         address recoveredAddress = ecrecover(hash, v, r, s);
165
166         if (recoveredAddress == address(0) || recoveredAddress != owner) {
167             revert ERC2612InvalidSigner(recoveredAddress, owner);
```

```
168         }
169
170         allowance[recoveredAddress][spender] = value;
171     }
172
173     emit Approval(owner, spender, value);
174 }
```

Currently, if a user signs a permit off-chain and later wants to cancel it (for example, if they suspect their private key is compromised, signed a phishing transaction, or simply changed their mind), they have no direct way to invalidate the permit before it is used. Their only options are:

- execute the permit themselves and then revoke the approval (requiring two transactions and gas costs); or
- wait for the permit deadline to expire.

This limitation is particularly relevant in the context of Printr's cross-chain token system (Telecoin with LayerZero integration), where:

- users interact with multiple chains, increasing the attack surface for phishing attempts;
- permit signatures may be shared across different interfaces and chains; and
- the complexity of cross-chain interactions makes it more likely that users might want to cancel pending permits.

While this is not a vulnerability (permits are secured by signatures and deadlines), adding a permit invalidation function would provide users with better control over their approvals and serve as a security response mechanism in case of key compromise or phishing attacks.

Exploit scenario

Recommendation

Add a function that allows users to manually increment their nonce, thereby invalidating all outstanding permit signatures:

```
function invalidateNonce() external {
    nonces[msg.sender]++;
}
```

Fix 2.1

Additional functionality was added.

Go back to Findings Summary

I12: Missing events for critical operations that distinguish behavior

Impact:	Info	Likelihood:	N/A
Target:	LZChannel, Printr	Type:	Code quality

Description

The protocol would benefit from additional events for critical operations. For example, the `lzReceive` function lacks events that indicate LayerZero message reception. The existing `TeleportIn` event provides insufficient context for tracking operations.

Recommendation

Add events for critical execution branches, especially where off-chain components need to track operations.

Acknowledgment 2.1

The issue was acknowledged by the team.

Go back to Findings Summary

Report Revision 2.1

Revision Team

Member's Name	Position
Dmytro Khimchenko	Lead Auditor
Naoki Yoshida	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The revision introduces changes to the liquidity fee collection mechanism. The protocol now collects liquidity fees via liquidity modules, which required modifications in the `PrintrFeeDistribution` contract and the liquidity modules. The `PrintrTrading._buy` function was updated to change the calculation logic. Additionally, minor changes were made to the interaction with Uniswap. The `PrintrTeleport` contract received minor updates. New functionality was added to the `ERC20Witness` contract. A new contract `OftStandard` was introduced to the protocol. Finally, minor changes were made to the NFT logic.

Trust Model

The trust model has not been changed since the previous revision.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

C10: Removed approve consumption allows approve vulnerability

Critical severity issue

Impact:	High	Likelihood:	High
Target:	Telecoin.sol	Type:	Logic error

Description

The refactoring in the Telecoin contract removed the `_decreaseAllowance` function call. This allows any user to claim Telecoin tokens from any other user who has approved Printr as a spender.

Listing 127. Excerpt from Telecoin

```
101 function teleportFrom(
102     address from,
103     IPrintrTeleport.TeleportParams calldata params
104 ) public payable virtual {
105     if (params.protocol == IPrintrTeleport.TeleportProtocol.UNSPECIFIED) {
106         revert InvalidProtocol();
107     }
108
109     if (params.protocol == IPrintrTeleport.TeleportProtocol.ITS) {
110         _itsTeleport(from, params);
111         return;
112     }
113
114     _lzTeleport(from, params);
115 }
```

Exploit scenario

1. Alice, a user, wants to interact with Printr and approves Printr to spend her Telecoin tokens.
2. Bob, an attacker, observes the approval transaction.

3. Bob calls `teleportFrom` to teleport Alice's tokens to his address.
4. Alice sends a transaction to interact with Printr; however, the tokens were already consumed by Bob's transaction.
5. Bob profits from the stolen tokens.

Recommendation

Add the `_decreaseAllowance` call in the `teleportFrom` function or remove the `teleportFrom` function, since sequentially calling `transferFrom` and `teleport` performs the same operation.

Fix 2.2

The issue was fixed by recommendation.

Go back to Findings Summary

M5: All Printed tokens are sent to the liquidity pool even if they are used as basePair tokens during graduation

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	BaseLiquidityModule.sol	Type:	Logic error

Description

The Printr protocol makes it possible to use already printed tokens as `basePair` for printing new tokens. All tokens are stored in the `Treasury` contract and are sent to the liquidity pool during graduation. However, during the calculation of how many tokens should be sent to the liquidity pool, the protocol uses the `balanceOf` function, effectively sending all the tokens that the protocol has in the `Treasury`.

Listing 128. Excerpt from BaseLiquidityModule

```
104 (ctx.tokenAmount, ctx.baseAmount) = _calculateTokenAmounts(params);
```

If the token is multi-chain, it grabs the liquidity from all pools where this token is used.

Listing 129. Excerpt from BaseLiquidityModule

```
197 } else {
198     // but for Teleporting telecoin, take all tokens from treasury to avoid
    rounding dust
199     tokenAmount = IERC20(params.token).balanceOf(address(params.treasury));
200 }
```

As a result, bonding curves for tokens where a printed token is used as `basePair` will be invalid.

Exploit scenario

1. Alice, a user, prints a token `TOKEN1`.
2. Bob and Charlie, users, print tokens `TOKEN2` and `TOKEN3` using `TOKEN1` as `basePair`.
3. Users trade the tokens `TOKEN1`, `TOKEN2` and `TOKEN3` in the local bonding curves.
4. Graduation of `TOKEN1` occurs and the protocol takes all `TOKEN1` tokens from the `Treasury` and sends them to the liquidity pool.
5. Local bonding curves for `TOKEN2` and `TOKEN3` are invalid because they do not have enough liquidity.

As a result, `TOKEN2` and `TOKEN3` cannot be traded on the local bonding curves because there are no `TOKEN1` tokens.

Recommendation

Calculate the token amount that should be sent to the liquidity pool without relying on the `balanceOf` function.

Fix 2.2

The issue was fixed by calculating the token amount that should be sent to the liquidity pool without relying on the `balanceOf` function.

Go back to Findings Summary

L7: Usage of transfer in Telecoin

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	Telecoin.sol	Type:	Logic error

Description

The `transfer` function is used for sending native token, and this is not recommended because of the limited gas. Additional logic in Printr may always revert here.

Listing 130. Excerpt from Telecoin

```
205 function _itsTeleport(
206     address from,
207     IPrintrTeleport.TeleportParams calldata params
208 ) internal {
209     // Get the teleport fee using the unified function
210     (uint256 totalNativeFee,,, uint256 bridgeFee) =
    IPrintrTeleport(printr).quoteTeleportFee(address(this), params);
211
212     // Transfer only the protocol fee to printr (excluding bridge gas fee)
213     uint256 protocolFee = totalNativeFee - bridgeFee;
214     if (protocolFee > 0) {
215         payable(printr).transfer(protocolFee);
216     }
```

Exploit scenario

1. The Printr contract is updated so that the receive function performs some operation.
2. Alice, a user, tries to use this function but fails because of the gas limit in the transfer function.

Recommendation

Use the call function to transfer native tokens to the Treasury.

Fix 2.2

The issue was fixed by using the `.call` function to transfer native tokens to the Treasury.

Go back to Findings Summary

L8: `priceAfter` is used as `effectivePrice`

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	LiquidityPool.sol	Type:	Logic error

Description

`priceAfter` is used as `effectivePrice` in the `estimateTokenCost` function.

However, the comment states "@return effectivePrice Final price per token after the trade execution", which describes the `finalPrice`.

The `priceAfter` value is an approximation for some liquidity modules. The value is used only as the return value of a view function.

Additionally, `effectivePrice` in the liquidity module can potentially underflow in the calculation `priceAfter = 2 * averagePrice - currentPrice`.

Exploit scenario

1. Alice, an integrator, builds a trading bot that calls the `estimateTokenCost` function to determine trade profitability.
2. The function returns `priceAfter` as `effectivePrice`, which is an approximation.
3. Alice's bot executes a trade based on this estimate, but the actual effective price differs, resulting in an unexpected loss.
4. Bob, another integrator, calls the `estimateTokenCost` function when `currentPrice > 2 * averagePrice`.
5. The calculation `priceAfter = 2 * averagePrice - currentPrice` underflows, causing the function to revert or return incorrect values.

Recommendation

Document the behavior and its limitations. Alternatively, calculate the effective trade price in the liquidity module and return it as the trade price.

Fix 2.2

The issue was fixed by renaming `effectivePrice` to `priceAfter` in every place where it is used.

Go back to Findings Summary

L9: Inconsistent refund for ERC20 and native tokens

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	PrintrTrading.sol	Type:	Logic error

Description

The functions `PrintrTrading.buy` and `PrintrTrading.spend` refund native tokens to `recipient` but refund ERC20 tokens to `msg.sender`.

Listing 131. Excerpt from PrintrTrading

```
202 _buy(curve, params);
203 // Refund excess ETH payment if any
204 _refundNativeValue(recipient);
```

Listing 132. Excerpt from PrintrTrading

```
780 }
781
782 /**
783  * @notice Executes a buy order through the liquidity pool
784  * @dev Handles token swaps via the liquidity module
785  *      Uses sqrtPriceLimitX96 for partial fills instead of amountOutMinimum
786  * @param curve Curve configuration containing pool details
787  * @param params Trading parameters and limits
788  * @param gracefulFailure If true, return 0 on failure instead of reverting
    (for graduation flow)
789  */
```

Exploit scenario

1. Alice, a user, wants to buy a token using ERC20 tokens and specifies herself as the recipient.

2. Bob, a relayer, calls the function on Alice's behalf, becoming the `msg.sender`.
3. The ERC20 token refund goes to Bob instead of Alice.

Recommendation

Ensure the same behavior for both ERC20 and native tokens.

Fix 2.2

The issue was fixed by refunding native tokens to the recipient and not the account which performs a trade.

Go back to Findings Summary

L10: permitWitnessCall forwards msg.value instead of signed call.nativeValue

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	ERC20Witness.sol	Type:	Logic error

Description

The `msg.value` is used for executing a signed call:

Listing 133. Excerpt from ERC20Witness

```
190 // Validate msg.value is sufficient for the native value transfer
191 if (msg.value < call.nativeValue) {
192     revert ERC20InvalidCallData();
193 }
```

Listing 134. Excerpt from ERC20Witness

```
211 // Execute the external call
212 (bool success, bytes memory returnData) = call.target.call{ value: msg.value
    }(callData);
213 if (!success) {
```

This allows running an unexpected call by specifying a different native token amount than the user expected. Although there is a condition `msg.value < call.nativeValue`, it is still possible to create an unexpected transaction, for example, through the `msg.value` branch.

Exploit scenario

1. Alice, a user, wants to use `permitWitnessCall` to run a specific function.
2. Alice simulates and signs the transaction.

3. Bob, the attacker, sees the call and runs this operation with a different `msg.value`.
4. Bob's transaction executes a different execution branch than Alice expected.

Recommendation

Use `call.nativeValue` for the call and refund native tokens if necessary.

Acknowledgment 2.2

The issue was acknowledged by the team with the comment:

> *Acknowledged, allowed by design.*

Go back to Findings Summary

W13: Attacker can create tokens with the same symbol on different chains to drain legitimate liquidity

Impact:	Warning	Likelihood:	N/A
Target:	PrintrPrinting.sol	Type:	Trust model

Description

The Printr protocol allows users to create tokens with the same symbol and token address on different chains using different base tokens. This enables a malicious actor to create a legitimate token on one chain and a controlled token on another chain, then exploit cross-chain transfers to drain legitimate liquidity.

An attacker can create `USDC/TOKEN1` pool on Chain A and `ATTACKER_TOKEN/TOKEN1` pool on Chain B. By minting `ATTACKER_TOKEN` on Chain B, the attacker purchases `TOKEN1`, transfers it to Chain A, and sells it for the legitimate `USDC`.

As a result, users who provide liquidity to the legitimate pool can have their funds fully liquidated by the malicious actor who prints the legitimate token on first sight.

Exploit scenario

1. Alice, a malicious actor, creates token `TOKEN1` on Chain A with `USDC` as the base token, establishing a legitimate-looking liquidity pool and creates `TOKEN1` on Chain B with `ATTACKER_TOKEN` as the base token.
2. Bob and other users, believing `TOKEN1` is legitimate, provide liquidity to the `USDC/TOKEN1` pool on Chain A.
3. Alice mints unlimited `ATTACKER_TOKEN` on Chain B to purchase `TOKEN1`.
4. Alice teleports `TOKEN1` from Chain B to Chain A.

5. Alice sells `TOKEN1` for `USDC` on Chain A, draining the legitimate liquidity.

As a result, Bob and other users lose their provided liquidity as Alice repeatedly executes this cycle to extract value from the legitimate pool.

Recommendation

Implement mechanism to flag potentially malicious tokens.

Acknowledgment 2.2

The issue was acknowledged by the team with the comment:

> *Bogus tokens will not show up in the Printr UI.*

Go back to Findings Summary

W14: Tokens are refunded and not spent on liquidity pool

Impact:	Warning	Likelihood:	N/A
Target:	PrintrTrading.sol	Type:	Logic error

Description

During graduation, there is a recalculation of how many base tokens should be used for swapping in the liquidity pool. However, the calculation of the remaining tokens is incorrect. The code always takes base tokens from the user that are calculated for spending on the local bonding curve and does not calculate how many tokens the user wants to spend on the liquidity pool due to the reassignment of `params.amount`.

Listing 135. Excerpt from PrintrTrading

```
623 params.amount = ctx.buyAmount;
```

As a result, the value of `remainingTokens` will always be 0, meaning that no user tokens are effectively swapped.

Listing 136. Excerpt from PrintrTrading

```
671 uint256 remainingTokens = params.amount - ctx.buyAmount;
```

The severity of the issue is lower because no tokens are taken from the user - the user simply does not spend all the tokens they intended to.

Recommendation

Refactor the code to correctly calculate the remaining tokens and swap them in the liquidity pool.

Partial solution 2.2

The issue was partially fixed by not updating the `params.amount` variable and updating it only when transaction gracefully fails. However, the logic of the calculation is partially incorrect, as for calculating the remaining meme tokens amount which can be bought, the protocol uses `_estimateTokenCost` function, which is supposed to be primarily used for local curve calculations. The problem occurs when `tokenReserve` equals to `tokenAmount` which leads to a division by zero error.

Listing 137. Excerpt from PrintrTrading

```
395 // Round up the cost if there's any remainder to prevent precision attacks
396 if ((curveConstant / (tokenReserve - tokenAmount)) * (tokenReserve -
    tokenAmount) < curveConstant) {
397     curveCost += 1;
```

As a result,

Go back to Findings Summary

W15: Attacker can front-run the liquidity pool creation and make graduation fail

Impact:	Warning	Likelihood:	N/A
Target:	PrintrPrinting.sol	Type:	Configuration

Description

Anyone can create a liquidity pool in Uniswap, initialize the price, and provide liquidity with only the base token making it one-sided liquidity. This can cause the graduation to fail and the transaction will buy tokens only from the local curve.

Exploit scenario

1. Alice, a malicious actor, front-runs the token printing to manipulate the price and make Telecoin expensive on the pool.
2. Alice provides one-sided liquidity by providing only the base token to Liquidity Pool.
3. During the graduation swap, Printr attempts to sell Telecoin to decrease its price.
4. The swap cannot correct the price because of Alice's liquidity.
5. Providing liquidity by Printr fails due to the price difference.

The severity of the issue is decreased because the attacker needs to provide one-sided liquidity and will not take any profit from the transaction.

Recommendation

Monitor the protocol for front-running of the pool creation.

Acknowledgment 2.2

The issue was acknowledged by the team with the comment:

> *Acknowledged, this is a known issue and we are working on a solution.*

Go back to Findings Summary

I13: Withdraw forwards all gas despite 3000-gas comment

Impact:	Info	Likelihood:	N/A
Target:	Treasury.sol	Type:	Code quality

Description

The comment states "Uses 3000 gas limit for ETH transfers to prevent griefing", but the code uses the `call()` function, which forwards all available gas, instead of the `transfer()` function.

Listing 138. Excerpt from Treasury

```
78 /**
79  * @notice Withdraws tokens or native currency from the treasury to a
   recipient
80  * @dev Only callable by Printr contract. Uses 3000 gas limit for ETH
   transfers to prevent griefing
81  * @param token Address of token to withdraw (address(0) for native ETH)
82  * @param recipient Address that will receive the withdrawn funds
83  * @param amount Amount of tokens or wei to withdraw from treasury
84  * @custom:throws WrongAccess if caller is not the authorized Printr contract
85  * @custom:throws ZeroAddress if recipient is the zero address
86  * @custom:throws FailedWithdrawal if native currency transfer fails
87  */
88 function withdraw(
```

Recommendation

Update the comment to reflect that the `call()` function forwards all available gas.

Fix 2.2

The issue was fixed by updating the comment.

Go back to Findings Summary

I14: `IntentProxy` constructor has `payable` parameter despite non-payable function

Impact:	Info	Likelihood:	N/A
Target:	IntentProxy.sol	Type:	Code quality

Description

The `sender` parameter in the delegatecall is marked as `payable`. However, the constructor function is not payable.

Listing 139. Excerpt from IntentProxy

```
17 constructor(
18     bytes memory delegateData
19 ) {
20     (bool success, bytes memory data) = payable(
   msg.sender).delegatecall(delegateData);
```

Recommendation

Remove the `payable` modifier from the `sender` parameter and send native tokens to the proxy before execution. Alternatively, mark the `executeIntentAddress` and `signedExecuteIntentAddress` functions as `payable` and forward `msg.value` during `IntentProxy` creation.

Fix 2.2

The issue was fixed by removing the `payable` modifier from the `sender` parameter.

Go back to Findings Summary

I15: `_isTeleporting` is not correctly described

Impact:	Info	Likelihood:	N/A
Target:	Printed.sol	Type:	Code quality

Description

The comment states that the `_isTeleporting` parameter specifies whether this is a main telecoin (true) or a wrapped telecoin (false). However, the main telecoin always sets `_isTeleporting` to false.

Listing 140. Excerpt from Printed

```
49 /**
50  * @notice Initializes the token with its basic parameters
51  * @param _printr Address of the Printr contract that created this token
52  * @param _treasury Address to receive the initial token supply
53  * @param _initialSupply The initial supply to mint to treasury
54  * @param _isTeleporting Whether this is a main telecoin (true) or wrapped
   telecoin (false)
55  */
56 constructor(
57     address _printr,
58     address _treasury,
59     uint256 _initialSupply,
60     bool _isTeleporting
61 ) {
```

Recommendation

Update the comment to reflect the actual implementation.

Fix 2.2

The issue was fixed by introducing `TeleportType` enum for describing the type of telecoin.

Go back to Findings Summary

I16: Token URI is not in base64 format as mentioned in the documentation

Impact:	Info	Likelihood:	N/A
Target:	Telecoin.sol	Type:	Code quality

Description

The code below declares a base64-encoded data URL (data:application/json;base64,) but then just appends raw, unencoded JSON text. This creates an invalid data URL that browsers and NFT marketplaces will not be able to parse correctly.

Listing 141. Excerpt from PrintrDev

```
73 function tokenURI(
74     uint256 id
75 ) public view virtual override(ERC721, IERC721Metadata) returns (string
   memory) {
76     _requireOwned(id);
77
78     // Token ID is the token address
79     address token = idToToken(id);
80
81     // Basic JSON metadata - can be enhanced later
82     return string(
83         abi.encodePacked(
84             "data:application/json;base64,",
85             bytes(
86                 string(
87                     abi.encodePacked(
88                         '{"name":"PRINTR Dev Position #',
89                         id.toString(),
90                         '","description":"Developer rights for PRINTR
   telecoin at ',
91                         token.toHexString(),
92                         '","image":"https://cdn.printr.money/nft/',
93                         id.toHexString(32),
94                         '.svg"}'
95                     )
96                 )
```

```
97             )
98         )
99     );
100 }
```

Recommendation

Use the `Base64` library to encode the JSON text correctly.

Fix 2.2

The issue was fixed by using the `Base64` library to encode the JSON text correctly.

Go back to Findings Summary

Report Revision 2.2

Revision Team

Member's Name	Position
Dmytro Khimchenko	Lead Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

Overview

Since there were no comprehensive changes in this revision, the complete overview is listed in the Executive Summary section Revision 2.2.

Appendix A: How to cite

Please cite this document as:

> Ackee Blockchain Security, Audit Report | Printr: Protocol, 8.12.2025.

Appendix B: Wake Findings

This section lists the outputs from the Wake framework used for testing and static analysis during the audit.

B.1. Fuzzing

The following table lists all implemented execution flows in the Wake fuzzing framework.

ID	Flow	Added
F1	Print a new Telecoin token	1.0
F2	Buy Telecoin from Printr	1.0
F3	Sell Telecoin from Printr	1.0
F4	Buy Telecoin with spend	1.0
F5	Register Telecoin to ITS	1.0
F6	Link Telecoin to ITS	1.0
F7	Teleport Telecoin via ITS	1.0
F8	Teleport Telecoin via ITS with teleportFrom	1.0
F9	Teleport Telecoin via LZ	1.0
F10	Teleport Telecoin via LZ with teleportFrom	1.0

Table 4. Wake fuzzing flows

The following table lists the invariants checked after each flow.

ID	Invariant	Added	Status
IV1	Transactions do not revert except where explicitly expected	1.0	**Fail** (C2, C3, C4)

ID	Invariant	Added	Status
IV2	Predicted contract addresses match actual deployed addresses during deployment	1.0	Success
IV3	Token printing creates correct initial state with zero amounts and reserves	1.0	Success
IV4	TelecoinPrinted event contains the expected telecoin ID	1.0	Success
IV5	CurveCreated event contains the correct creator address	1.0	Success
IV6	Initial TokenTrade event has expected amount, cost, initial price, issued supply, and expected reserve	1.0	Success
IV7	Initial TokenTrade event is marked as a buy transaction with the correct token and trader addresses	1.0	Success
IV8	Token trading events contain accurate transaction data	1.0	Success
IV9	Buy transaction TokenTrade events have correct token address, trader address, and buy flag	1.0	Success
IV10	Buy transaction TokenTrade events have correct amount, cost, effective price, issued supply, and reserve calculations	1.0	**Fail** (C1, W1)
IV11	Sell transaction TokenTrade events have correct token address, trader address, and sell flag (not isBuy)	1.0	Success

ID	Invariant	Added	Status
IV12	Sell transaction TokenTrade events have correct amount, refund cost, effective price, issued supply, and reserve calculations	1.0	Success
IV13	Spend transaction TokenTrade events have correct token address, trader address, and buy flag	1.0	Success
IV14	Spend transaction TokenTrade events have correct amount, cost, and reserve calculations	1.0	Success
IV15	Token balances are correctly maintained and updated	1.0	Success
IV16	User token balances match the expected amounts after buy transactions	1.0	Success
IV17	Python state token balances always match actual on-chain ERC20 token balances	1.0	Success
IV18	Python state ETH balances always match actual on-chain native token balances	1.0	**Fail** (C1)
IV19	Completion threshold and graduation mechanics work correctly	1.0	Success
IV20	Completion threshold flag is set when token reaches graduation threshold	1.0	Success
IV21	TokenGraduated event contains correct token address and total supply at graduation	1.0	Success
IV22	Graduation events trigger when completion threshold is reached during buy operations	1.0	Success
IV23	Graduation events trigger when completion threshold is reached during spend operations	1.0	Success

ID	Invariant	Added	Status
IV24	Swap events occur after graduation when expected	1.0	Success
IV25	Price limit mechanisms function correctly	1.0	**Fail** (M2)
IV26	Price calculations are consistent with bonding curve mathematics	1.0	Success
IV27	Bonding curve from getCurve parameters are maintained correctly	1.0	Success
IV28	Curve constant calculations match expected values with minimal precision error (≤1)	1.0	Success
IV29	Base token reserves match the curve reserve values stored on-chain	1.0	Success
IV30	Completion thresholds match the curve completion threshold values	1.0	Success
IV31	Maximum token supplies match the curve maximum token supply values	1.0	Success
IV32	Virtual reserves match the curve virtual reserve values	1.0	Success
IV33	Base pair addresses match the curve base pair values	1.0	Success
IV34	Total curves count matches the curve total curves values	1.0	Success
IV35	Meme token reserves match actual treasury token balances with minimal precision error (≤1)	1.0	Success
IV36	Cross-chain and interchain functionality works correctly	1.0	Success

ID	Invariant	Added	Status
IV37	Telecoin parameters match between registration and linking operations	1.0	Success
IV38	Interchain token IDs match between teleport events and expected values	1.0	Success
IV39	Teleport amounts match between events and intended transfer amounts	1.0	Success
IV40	TeleportFrom operations correctly transfer tokens from owner to destination via spender	1.0	Success
IV41	Fee calculations and distributions are accurate	1.0	Success
IV42	Trading fees are correctly calculated and deducted from transactions	1.0	Success
IV43	Reserve calculations properly account for fees (reserve = cost - fee)	1.0	Success
IV44	Fee amounts are correctly distributed to treasury and fee recipients	1.0	Success
IV45	Uniswap integration functions correctly after graduation	1.0	Success
IV46	Swap events contain correct token amounts and directions after graduation	1.0	Success
IV47	Base token amounts spent in Uniswap swaps match expected calculations	1.0	Success
IV48	Telecoin amounts received in Uniswap swaps match expected calculations	1.0	Success
IV49	Refunds are correctly calculated and distributed when base token amount exceeds swap cost	1.0	Success

ID	Invariant	Added	Status
IV50	Error handling and revert conditions work as expected	1.0	Success
IV51	SwapFailed errors are properly handled and do not cause unexpected state changes	1.0	Success
IV52	Invalid registration attempts with incorrect parameters are properly rejected	1.0	Success
IV53	Invalid linking attempts with incorrect parameters are properly rejected	1.0	Success
IV54	Teleport operations with invalid states are properly rejected	1.0	Success
IV55	Price calculations maintain precision through multiple operations	1.0	Success
IV56	State transitions are atomic and consistent	1.0	Success
IV57	Token state progresses correctly from PRINTED to REGISTERED to LINKED	1.0	Success
IV58	All state updates within a transaction are consistent with each other	1.0	Success
IV59	Failed transactions do not cause partial state updates	1.0	Success

Table 5. Wake fuzzing invariants

Appendix C: Wake AI Findings

This section lists vulnerabilities identified by Wake AI, an LLM-powered audit tool used for AI-assisted vulnerability discovery during the audit. Wake AI leverages large language models to understand code context and reason about complex contract behavior, complementing manual review.

C.1. Discovered Findings

The following table contains true-positive findings identified by Wake AI. These findings are included regardless of whether they were also discovered independently by auditors during manual review.

Finding title	Severity	Reported	Discoverer
C1: Native Token is not refunded after Uniswap trade	**Critical**	1.0	Wake AI, Auditor
C3: `teleport` function always fails for the ITS transactions as all fees are transferred to the Printr	**Critical**	1.0	Wake AI, Auditor
C4: Inverted condition for creation of `PrintrTeleportingTelecoin`	**Critical**	1.0	Wake AI, Auditor
C6: Missing access control in `witnessTeleport`	**Critical**	1.0	Wake AI, Auditor
C7: Recipient address is left-padded during encoding and right-padded during decoding for LayerZero	**Critical**	1.0	Wake AI, Auditor
M1: Unchecked call in `_refund` function	Medium	1.0	Wake AI, Auditor
L2: Missing refund in `teleport` function	Low	1.0	Wake AI, Auditor

Finding title	Severity	Reported	Discoverer
L4: Possible overflow in `print` function	Low	1.0	Wake AI, Auditor
W3: Approval for the ITS token manager is needed before teleporting	Warning	1.0	Wake AI, Auditor
W4: Documentation disrepency with code	Warning	1.0	Wake AI, Auditor
I1: No validation for zero `params.amount`	Info	1.0	Wake AI, Auditor
W9: Merchantmoe Liquidity Module is not implemented correctly	Warning	2.0	Wake AI, Auditor
I11: Missing permit invalidation mechanism	Info	2.0	Wake AI, Auditor
I12: Missing events for critical operations that distinguish behavior	Info	2.0	Wake AI, Auditor
L7: Usage of transfer in Telecoin	Low	2.1	Wake AI, Auditor
L10: permitWitnessCall forwards msg.value instead of signed call.nativeValue	Low	2.1	Wake AI
W15: Attacker can front-run the liquidity pool creation and make graduation fail	Warning	2.1	Wake AI, Auditor
I13: Withdraw forwards all gas despite 3000-gas comment	Info	2.1	Wake AI
I14: `IntentProxy` constructor has `payable` parameter despite non-payable function	Info	2.1	Wake AI

Finding title	Severity	Reported	Discoverer
I15: `_isTeleporting` is not correctly described	Info	2.1	Wake AI
I16: Token URI is not in base64 format as mentioned in the documentation	Info	2.1	Wake AI

Table 6. Table of findings identified by Wake AI

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz