Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

panther.

Panther Protocol

Veridise Inc.

April 25, 2025

► **Prepared For:**

Panther
https://www.pantherprotocol.io/

► **Prepared By:**

Evgeniy Shishkin
Mark Anthony
Alp Bassa
Kostas Ferles

► **Contact Us:**

contact@veridise.com

► **Version History:**

April 25, 2025	V8
March 7, 2025	V7
January 30, 2025	V6
January 24, 2025	V5
January 20, 2025	V4
December 27, 2024	V3
December 24, 2024	V2
December 20, 2024	V1
November 25, 2024	Initial Draft

Contents

1 Executive Summary

From Sep. 26, 2024 to Dec. 6, 2024 and from Feb. 21, 2025 to Feb. 28, 2025, Panther engaged Veridise to conduct a security assessment of their Panther Protocol. The security assessment covered the core smart contracts and zero-knowledge circuits of the Panther Protocol. Compared to the previous version, which Veridise has audited, the new version has undergone substantial refactoring (especially on the circuits side) and also includes several new features. Veridise conducted the assessment over 171 person-days, with 3 security analysts reviewing the project over 57 days on commit `a16a43e`. The review strategy involved a tool-assisted analysis of the program source code performed by Veridise security analysts as well as thorough code review.

Project Summary. Panther Protocol is a privacy-preserving, compliance-oriented multi-chain digital asset management system that supports ERC-20, ERC-721, and ERC-1155 tokens. The protocol relies heavily on advanced cryptographic techniques and Zero-Knowledge Proof technology to implement its core features.

Users of the protocol are able to perform the following functions in a privacy preserving manner:

- Deposit tokens into the protocol
- Withdraw tokens from the protocol
- Transfer tokens from one internal account to another
- Swap deposited tokens on selected decentralized exchanges*
- Provide evidence of asset movements to a designated third party upon request
- Stake their assets†
- Earn rewards in the form of Panther Reward Points (PRP) points through participation in protocol-wide activities
- Convert PRP points into ZKP tokens (Panther Protocol's native token)
- Seamlessly transfer funds between different supported networks ‡

To provide an additional layer of privacy, Panther Protocol implements the ERC-4337 Account Abstraction proposal. This allows external Bundler nodes§ to execute user transactions on behalf of users, while hiding the true origin of the transaction. Additionally, this feature allows users to pay fees for transactions in protocol-native ZKP tokens, rather than Ether.

Users can choose to send their transactions directly to the protocol, covering all incurred costs themselves, or they can rely on another mechanism called the *Bus Queue*. When using the latter, they submit their transactions to a special queue rather than directly into the protocol. When the queue is full, a special node called *ZMiner* finalizes it by sealing it into protocol storage for a reward.

* Currently, it is UniswapV3 and Quickswap
† Out of scope of this security assessment
‡ Not yet fully implemented
§ Also known as Relayers

The advantage of this approach is that, instead of each user doing the finalization individually for each transaction, the ZMiner only does it once for a batch of transactions, greatly reducing the overhead costs for users. Compared to a direct method of processing transactions, the bus queue aims to reduce the costs but requires more time for funds to reach the protocol.

Panther Protocol is strongly committed to privacy and compliance. In terms of compliance, the protocol implements the following measures:

- Users of the protocol are assigned to *Zones*. Each zone has its own policies regarding transaction limits, potential destinations, KYC/KYT requirements, and more.
- Almost all transactions require a KYC/KYT certificate tied to the specific transaction being processed and issued by an external trusted service provider[¶].

Code Assessment. The Panther Protocol developers provided the source code of the Panther Protocol contracts and circuits for the code review. The source code appears to be mostly original code written by the Panther Protocol developers. It contains some documentation in the form of READMEs and documentation comments on functions, storage variables, and circuit signals. To facilitate the Veridise security analysts' understanding of the code, the Panther Protocol developers shared some online documentation, describing a high-level structure of the protocol as well as some design documents that explained several design decisions in more detail.

The source code contains a test suite, which the Veridise security analysts studied to understand the way users are expected to interact with the protocol and document protocol invariants.

Summary of Issues Detected. The security assessment uncovered 72 issues, 17 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, the audit uncovered, among others, issues where fees were not properly accounted (V-PAN-VUL-006, V-PAN-VUL-061), issues where users could disable or bypass security-critical checks (V-PAN-VUL-015, V-PAN-VUL-016), business logic errors that could lead to lost funds (V-PAN-VUL-007, V-PAN-VUL-011), as well as under-constrained circuits (V-PAN-VUL-001).

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the implementation and security practices of the Panther Protocol. The recommendations listed below are solely Veridise's opinion for improving the long-term security and stability of the protocol, and may not align with the Panther Protocol developers' viewpoint. Furthermore, these recommendations are based on the initial version of the code reviewed by Veridise and may not necessarily apply to the current version of the protocol. For any recommendations the developers incorporated during the fix review of this security assessment, we provide a post-audit update below.

Improve testing. Several of the issues uncovered by this review could have been caught by some sort of testing. After carefully reviewing the existing suite, the Veridise analysts noticed that several important components of the system are under-tested. The protocol can significantly improve its security and robustness by:

1. Introducing unit and regression tests wherever they are lacking.

[¶] Currently, PureFi

2. Introducing integration tests that include both the circuits and the smart contracts. This will help developers to ensure that the contracts and circuits are in sync.
3. Introducing negative tests. These tests must check when things are expected to fail. These tests will ensure that the protocol has some safeguards against malicious inputs.

Post fix review update. The Panther Protocol developers have increased test coverage significantly after the fix review phase of the protocol.

Standardize data validation. In the current version of the protocol, data validation is scattered across the smart contracts. It is recommended to standardize the way data validation is performed across the code base. Otherwise, the protocol might become susceptible to missing data validation vulnerabilities in the future, since developers might assume that another part of the code base is performing data validation.

Use well-established libraries. Several components of the protocol are well-established concepts already implemented in audited and well-tested libraries. Instead of using these mature libraries, the Panther team has decided to re-implement several components themselves. The Veridise team uncovered a few issues related to these components, all of which have been fixed by the Panther team. Using a well-established library would simultaneously simplify the development of the protocol and improve its security. *Developer response:* The Panther Protocol developers chose to re-implement some concepts mainly to optimize gas usage.

Implement storage layout suitable for the Diamond Pattern. Related to the recommendation above, the Panther team decided to make use of the Diamond Pattern to make its contracts upgradable. However, instead of implementing one of the example layouts from the ERC-2535, they decided to implement a layout based on storage gaps. Furthermore, the Panther developers did not provide a script for upgrading the facets of the diamond. Without such a script and due to the complex nature of the protocol, future upgrades should be subject to detailed scrutiny. It is encouraged to either implement one of the storage layouts described in EIP-2535 or add rigorous testing for the upgrade logic while maintaining the original layout.

Introduce guidelines for approving tokens. The Panther Protocol is configured to work with a set of approved tokens (ERC-20, ERC-721, and ERC-1155). Even though the tokens must be approved by the protocol's governance, there are several assumptions around these tokens that are currently implicit. For instance, approving tokens with user hooks (e.g., ERC-777) might make the protocol prone to reentrancy attacks. Therefore, appropriate guidelines for approving tokens must be shared with the protocol's governance members.

Disclaimer. Given the complexity of the Panther Protocol, the size of the code base, and the scope of the proposed changes, the Veridise team cannot make guarantees about the absence of high/critical issues in the protocol.

We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Panther Protocol	`a16a43e`	Solidity, Circom	EVM

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Sep. 26–Dec. 6, 2024	Manual & Tools	3	156 person-days
Feb.21–Feb.27, 2024	Manual	3	15 person-days

Table 2.3: Vulnerability Summary.[a]

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	7	7	7
High-Severity Issues	10	10	10
Medium-Severity Issues	11	11	11
Low-Severity Issues	14	14	14
Warning-Severity Issues	28	28	25
Informational-Severity Issues	2	2	2
TOTAL	72	72	69

[a] The first column counts all the issues reported by the Veridise analysts. Not all of those issues were acknowledged by the Panther team.

Table 2.4: Category Breakdown.

Name	Number
Logic Error	31
Data Validation	24
Maintainability	8
Cryptographic Vulnerability	4
Access Control	2
Reentrancy	1
Usability Issue	1
Authorization	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Panther Protocol's smart contracts and ZK circuits. During the assessment, the security analysts aimed to answer questions such as:

Does the code contain any of Solidity-specific vulnerabilities, such as:

- Reentrancies
- Arithmetic overflows leading to denial of service
- Silent integer overflows or underflows
- Out-of-gas attacks
- Insufficient input parameters validation
- Inability to receive funds when needed or potentially locked funds

Do the circuits contain any of Circom-specific vulnerabilities, such as:

- Under-constrained signals
- Arithmetic overflows or underflows in the underlying field of Circom
- Errors related to the incorrect usage of the Circom's standard library

General business logic-related questions:

- Do the token funds correctly track in all usage scenarios, including depositing, withdrawing, transferring, and swaps?
- Are rewards correctly nominated and accounted for when they are issued?
- Does the protocol's fee calculation correctly account for all necessary fees?
- Does the protocol integrate correctly with supported decentralized exchanges?
- Has the protocol implemented all necessary access control checks?

Cryptography-related and privacy-related questions:

- Does the protocol use solid field-tested cryptographic primitives?
- Are all cryptographic mechanisms properly implemented and configured?
- Is there a possibility of attacks on the protocol specific cryptographic schemes?
- Has the hashing been applied correctly and is it immune to brute force attacks?
- Could there be any privacy leaks during normal protocol operation?

Protocol-wide attack vectors:

- Is there a possibility of impersonating funds of other users?
- Is it possible to intentionally or accidentally cause the protocol to stall when it can no longer process user requests?
- Can funds become inaccessible to legitimate users?
- Are there risks of double spend attacks?
- Could there be griefing attacks, where the goal is to cause damage to the protocol for malicious purposes?

- Might there be ways to deceive legitimate users?

Compliance-related attack vectors:

- Is it possible to circumvent KYC/KYT verification or reuse previously issued certificates?
- Can certificates be impersonated?
- Are there any ways to bypass restrictions imposed by Zones, such as transfer limits, destination restrictions, blacklists, etc.?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology. To address the questions above, the security assessment involved a combination of human experts and automated program analysis & testing tools. In particular, the security assessment was conducted with the aid of the following techniques:

- *Static analysis.* To identify potential common vulnerabilities, security analysts leveraged Veridise's custom smart contract and ZK circuit analysis tool Vanguard. This tool is designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables, and ZK vulnerabilities, such as under-constrained or unconstrained signals.
- *Formal Verification.* Security analysts leveraged Veridise's custom verification tool called Picus to determine if certain security critical sub-circuits (e.g., the tree updater and balance checker) might be under-constrained. Specifically, Picus checks that every output signal of a circuit is uniquely determined by its inputs. If this is not the case, Picus returns two concrete witnesses that demonstrate an output signal can take different values given the same input signals.

Scope. The scope of this security assessment is limited to the following files provided by the Panther Protocol developers:

- `contracts/protocol/v1/Account.sol`
- `contracts/protocol/v1/FeeMaster.sol`
- `contracts/protocol/v1/PantherPoolV1.sol`
- `contracts/protocol/v1/core/facets/AppConfiguration.sol`
- `contracts/protocol/v1/core/facets/FeeMasterTotalDebtController.sol`
- `contracts/protocol/v1/core/facets/PrpConversion.sol`
- `contracts/protocol/v1/core/facets/PrpVoucherController.sol`
- `contracts/protocol/v1/core/facets/ZAccountsRegistration.sol`
- `contracts/protocol/v1/core/facets/ZSwap.sol`
- `contracts/protocol/v1/core/facets/ZTransaction.sol`
- `contracts/protocol/v1/PantherTrees.sol`
- `contracts/protocol/v1/trees/facets/BlacklistedZAccountsIdsRegistry.sol`
- `contracts/protocol/v1/trees/facets/ForestTree.sol`
- `contracts/protocol/v1/trees/facets/ProvidersKeysRegistry.sol`
- `contracts/protocol/v1/trees/facets/StaticTree.sol`
- `contracts/protocol/v1/trees/facets/ZAssetsRegistryV1.sol`
- `contracts/protocol/v1/trees/facets/ZNetworksRegistry.sol`
- `contracts/protocol/v1/trees/facets/ZZonesRegistry.sol`

- contracts/protocol/v1/trees/errMsgs/PantherTreesErrMsgs.sol
- contracts/protocol/v1/trees/errMsgs/ProvidersKeysErrMsgs.sol
- contracts/protocol/v1/trees/errMsgs/ZAccountsRegistryErrMsgs.sol
- contracts/protocol/v1/trees/facets/forestTrees/busTree/BusQueues.sol
- contracts/protocol/v1/trees/facets/forestTrees/busTree/MiningRewardsSignatureVerifier.sol
- contracts/protocol/v1/trees/facets/forestTrees/busTree/MiningRewards.sol
- contracts/protocol/v1/trees/facets/forestTrees/BusTree.sol
- contracts/protocol/v1/trees/facets/staticTrees/ProvidersKeysRegistry/ProvidersKeysSignatureVerif.sol
- contracts/protocol/v1/trees/facets/staticTrees/StaticRootUpdater.sol
- contracts/protocol/v1/trees/interfaces/IMinersNetRewardReserves.sol
- contracts/protocol/v1/trees/interfaces/IStaticSubtreesRootsGetter.sol
- contracts/protocol/v1/trees/interfaces/IStaticTreeRootUpdater.sol
- contracts/protocol/v1/trees/libraries/ZAssetEncodingUtils.sol
- contracts/protocol/v1/trees/storage/AppStorage.sol
- contracts/protocol/v1/trees/storage/BlacklistedZAccountsIdsRegistryStorageGap.sol

- contracts/protocol/v1/trees/storage/Constants.sol
- contracts/protocol/v1/trees/storage/ProvidersKeysRegistryStorageGap.sol
- contracts/protocol/v1/trees/storage/StaticTreeStorageGap.sol
- contracts/protocol/v1/trees/storage/ZAssetsRegistryStorageGap.sol
- contracts/protocol/v1/trees/storage/ZNetworksRegistryStorageGap.sol
- contracts/protocol/v1/trees/storage/ZZonesRegistryStorageGap.sol
- contracts/protocol/v1/trees/utils/Constants.sol
- contracts/protocol/v1/trees/utils/merkleTrees/BinaryUpdatableTree.sol
- contracts/protocol/v1/trees/utils/merkleTrees/DegenerateIncrementalBinaryTree.sol

- contracts/protocol/v1/trees/utils/PantherPoolAuth.sol
- contracts/protocol/v1/trees/utils/zeroTrees/Constants.sol
- contracts/protocol/v1/vault/BalanceViewer.sol
- contracts/protocol/v1/vault/EthEscrow.sol
- contracts/protocol/v1/vault/StealthEthPull.sol
- contracts/protocol/v1/vault/StealthExec.sol
- contracts/protocol/v1/verifier/Verifier.sol
- contracts/protocol/v1/verifier/VerifyingKeyProvider.sol
- circuits/circuits/ammV1.circom
- circuits/circuits/ammV1Top.circom
- circuits/circuits/mainAmmV1.circom
- circuits/circuits/mainTreeBatchUpdaterAndRootChecker.circom
- circuits/circuits/mainZAccountRegistrationV1.circom
- circuits/circuits/mainZAccountRenewalV1.circom
- circuits/circuits/mainZSwapV1.circom
- circuits/circuits/mainZTransactionV1.circom
- circuits/circuits/zAccountRegistrationV1.circom
- circuits/circuits/zAccountRegistrationV1Top.circom
- circuits/circuits/zAccountRenewalV1.circom

- circuits/circuits/zAccountRenewalV1Top.circom
- circuits/circuits/zSwapV1.circom
- circuits/circuits/zSwapV1Top.circom
- circuits/circuits/zTransactionV1.circom
- circuits/circuits/templates/balanceChecker.circom
- circuits/circuits/templates/dataEscrowElGamalEncryption.circom
- circuits/circuits/templates/merkleInclusionProof.circom
- circuits/circuits/templates/merkleTreeBuilder.circom
- circuits/circuits/templates/merkleTreeInclusionProof.circom
- circuits/circuits/templates/merkleTreeUpdater.circom
- circuits/circuits/templates/networkIdInclusionProver.circom
- circuits/circuits/templates/nullifierHasher.circom
- circuits/circuits/templates/partiallyFilledChainBuilder.circom
- circuits/circuits/templates/pubKeyDeriver.circom
- circuits/circuits/templates/rewardsExtended.circom
- circuits/circuits/templates/selectable3TreeInclusionProof.circom
- circuits/circuits/templates/selectable3TreeInclusionProofChecker.circom
- circuits/circuits/templates/selector3.circom
- circuits/circuits/templates/treeBatchUpdaterAndRootChecker.circom
- circuits/circuits/templates/trustProvidersKyt.circom
- circuits/circuits/templates/trustProvidersMerkleTreeLeafIDAndRuleInclusionProver.circom
- circuits/circuits/templates/trustProvidersNoteInclusionProver.circom
- circuits/circuits/templates/utils.circom
- circuits/circuits/templates/utxoNoteHasher.circom
- circuits/circuits/templates/utxoNoteInclusionProver.circom
- circuits/circuits/templates/zAccountBlackListLeafInclusionProver.circom
- circuits/circuits/templates/zAccountNoteHasher.circom
- circuits/circuits/templates/zAccountNoteInclusionProver.circom
- circuits/circuits/templates/zAccountNullifierHasher.circom
- circuits/circuits/templates/zAssetChecker.circom
- circuits/circuits/templates/zAssetNoteInclusionProver.circom
- circuits/circuits/templates/zeroPaddedInputChecker.circom
- circuits/circuits/templates/zNetworkNoteInclusionProver.circom
- circuits/circuits/templates/zoneIdInclusionProver.circom
- circuits/circuits/templates/zZoneNoteHasher.circom
- circuits/circuits/templates/zZoneNoteInclusionProver.circom
- circuits/circuits/templates/zZoneZAccountBlackListExclusionProver.circom

Methodology. Veridise security analysts reviewed the reports of previous audits for Panther Protocol, inspected the provided tests, and read the Panther Protocol documentation. They then began a review of the code assisted by both static analyzers and formal verifiers.

During the security assessment, the Veridise security analysts regularly met with the Panther Protocol developers to ask questions about the code.

Limitations. Due to the scope of the assessment, the recommendations given in this report are limited to the functional specification provided by the Panther Protocol developers. The overall security of the system can be compromised if any component outside the scope of the security

assessment is vulnerable. For the Panther Protocol, such components include, but are not limited to, the following:

1. **Circuit deployment:** If the circuits are not deployed according to industry standards, i.e., following a secure trusted setup ceremony, the whole protocol can be at risk in case the common reference string (CRS) is leaked.
2. **Cryptographic hash functions:** The Panther Protocol circuits make substantial use of Poseidon hashes, therefore, the security of the protocol is tied to the security guarantees of the underlying cryptographic hashes. Even though the Poseidon hash has been tested in contracts deployed on Ethereum mainnet, we would encourage the developers of Panther Protocol to evaluate the trade-offs between Poseidon hashes and other alternatives like Pedersen hashes. Some initial research on this topic can be found here.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-PAN-VUL-001	Babyjubjub suborder constraints not . . .	Critical	Fixed
V-PAN-VUL-002	zAccountRenewalV1 can validate multiple . . .	Critical	Fixed
V-PAN-VUL-003	Incorrect processing of Deposit+Withdraw . . .	Critical	Fixed
V-PAN-VUL-004	Nullifier verification can be disabled	Critical	Fixed
V-PAN-VUL-005	Bundler fee amount is not checked in . . .	Critical	Fixed
V-PAN-VUL-006	getQuoteAmount function misuse leads to . . .	Critical	Fixed
V-PAN-VUL-007	ZSwap is missing logic for depositing tokens	Critical	Fixed
V-PAN-VUL-008	Possibility of zAccountId overflow	High	Fixed
V-PAN-VUL-009	Blacklist states cannot be represented . . .	High	Fixed
V-PAN-VUL-010	Unsafe use of Num2Bits(254) on blacklist leaf	High	Fixed
V-PAN-VUL-011	Incorrect handling of WEth tokens during . . .	High	Fixed
V-PAN-VUL-012	Incorrect ZKP balance accounting in . . .	High	Fixed
V-PAN-VUL-013	PantherPool does not update Vault . . .	High	Fixed
V-PAN-VUL-014	ZKP tokens get accrued incorrectly during . . .	High	Fixed
V-PAN-VUL-015	ZoneIdInclusionProver check can be bypassed	High	Fixed
V-PAN-VUL-016	Zone related limits can be bypassed	High	Fixed
V-PAN-VUL-017	zAccountRenewalV1 circuit does not . . .	High	Fixed
V-PAN-VUL-018	Incorrect token identifier is used for swap	Medium	Fixed
V-PAN-VUL-019	KYT signature verification process fails for . . .	Medium	Fixed
V-PAN-VUL-020	forTxReward gets abstracted away in the . . .	Medium	Fixed
V-PAN-VUL-021	Data escrow encrypted message . . .	Medium	Fixed
V-PAN-VUL-022	PrpConversion does not account ZKP . . .	Medium	Fixed
V-PAN-VUL-023	Incorrect perUtxoReward value can halt . . .	Medium	Fixed
V-PAN-VUL-024	PureFi session identifiers may be insecure	Medium	Fixed
V-PAN-VUL-025	Pontential reentrancy via safeTransferETH	Medium	Fixed
V-PAN-VUL-026	Custom encryption scheme lacks security . . .	Medium	Fixed
V-PAN-VUL-027	Potential bypass of extended KYT for . . .	Medium	Fixed
V-PAN-VUL-028	Improper HMAC implementation	Medium	Fixed
V-PAN-VUL-029	Insufficient source address check in . . .	Low	Fixed
V-PAN-VUL-030	Zones registry accepts root as an argument . . .	Low	Fixed
V-PAN-VUL-031	kytSignedMessageChargedAmountZkp is . . .	Low	Fixed
V-PAN-VUL-032	Unconditional reward during the debt . . .	Low	Fixed
V-PAN-VUL-033	Malleable ECDSA implementation	Low	Fixed
V-PAN-VUL-034	Sub-contracts of BinaryUpdatableTree do . . .	Low	Fixed
V-PAN-VUL-035	Storage variable forestRoot is not updated . . .	Low	Fixed
V-PAN-VUL-036	Incorrect loop upper bound in . . .	Low	Fixed

4.1 Detailed Description of Issues

4.1.1 V-PAN-VUL-001: Babyjubjub suborder constraints not applied correctly

Severity	Critical	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`utils.circom`		
Location(s)	Template BabyJubJubSubOrderTag()		
Confirmed Fix At	8fdab18		

The template `BabyJubJubSubOrderTag` is used to ensure that the input signal is less than the BabyJubJub suborder. Internally, it makes use of the `LessThan` template from circomlib to perform this range check. This constraint is important to ensure that operations performed on points within the subgroup remain deterministic. See snippet below for context.

As it is currently implemented, the range check is not effective because the output of the `LessThan` template is not enforced to be `1`. So, the circuit does not actually enforce that the input signal is less than the suborder.

```
template BabyJubJubSubOrderTag(isActive) {
  signal input in;
  signal output {sub_order_bj_sf} out;
  var suborder =
    2736030358979909402780800718157159386076813972158567259200215660948447373041;
  component n2b;
  if ( isActive ) {
      n2b = LessThan(251);
      n2b.in[0] <== in;
      n2b.in[1] <== suborder;
  }
  out <== in;
}
```

Snippet 4.1: Snippet from template `BabyJubJubSubOrderTag()`

This tag is applied to several signals in the circuits. Because the range check is not performed correctly, the operations that these signals are involved in can be rendered non-deterministic.

One of the signals that `BabyJubJubSubOrderTag` is used to constrain throughout the circuits is the `zAccountUtxoInNullifierPrivKey`. This key is used to verify the `zAccountUtxoInNullifierPubKey` using the circomlib template `BabyPbk`. It is also involved in the nullifier hash along with the UTXO commitment. See snippet below for details.

```
component zAccountNullifierPubKeyChecker = BabyPbk();
zAccountNullifierPubKeyChecker.in <== zAccountUtxoInNullifierPrivKey;
zAccountNullifierPubKeyChecker.Ax === zAccountUtxoInNullifierPubKey[0];
zAccountNullifierPubKeyChecker.Ay === zAccountUtxoInNullifierPubKey[1];

component zAccountUtxoInNullifierHasher = ZAccountNullifierHasher();
zAccountUtxoInNullifierHasher.privKey <== zAccountUtxoInNullifierPrivKey;
zAccountUtxoInNullifierHasher.commitment <== zAccountUtxoInNoteHasher.out;
```

Because the `zAccountUtxoInNullifierPrivKey` is not correctly enforced to be less than the BabyJubJub suborder, we can have multiple values of a private key which correspond to the same public key which renders the circuit non-deterministic. A PoC can be found below which illustrates how the public key generated using `BabyPbk` for the private keys `1` and `suborder + 1` is the same.

```
include "circomlib/poseidon.circom";
include "circomlib/babyjub.circom";

template Example () {
    signal input zAccountUtxoInNullifierPrivKey;

    var suborder =
    2736030358979909402780800718157159386076813972158567259200215660948447373041;

    component pubKeyFromInput = BabyPbk();
    pubKeyFromInput.in <== zAccountUtxoInNullifierPrivKey;

    component pubKeyFromSuborder = BabyPbk();
    pubKeyFromSuborder.in <== suborder + 1;

    pubKeyFromInput.Ax === pubKeyFromSuborder.Ax;
    pubKeyFromInput.Ay === pubKeyFromSuborder.Ay;

}

component main { public [ zAccountUtxoInNullifierPrivKey ] } = Example();

/* INPUT = {
    "zAccountUtxoInNullifierPrivKey": "1"
} */
```

As the `zAccountUtxoInNullifierPrivKey` is involved in the nullifier hash, this can be used to create multiple nullifiers for the same UTXO commitment.

Additionally, the `LessThan` template assumes that its input signal fits within `251` bits i.e the argument the template is instantiated with. There is no other constraint applied to the input signal which ensures that it is constrained to `251` bits. If this is not followed it can render the circuit non-deterministic, as the `LessThan` template can be made to overflow internally.

Impact An attacker can create and consume multiple nullifiers using the same UTXO commitment.

Recommendation Inside the template `BabyJubJubSubOrderTag`, constrain the output of the `LessThan` template to be `1`.

The input signal should also be constrained to fit within `251` bits.

Developer Response The developers fixed the issue at commit 8fdab18.

4.1.2 V-PAN-VUL-002: zAccountRenewalV1 can validate multiple nullifiers for the same UTXO commitment

Severity	Critical	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`zAccountRenewalV1.circom`		
Location(s)	Template ZAccountRenewalV1		
Confirmed Fix At	903edd3		

The pre-image to the `zAccount` note commitments in the circuits contain the zAccount nullifier public key which is a point on the BabyJubJub curve. The private key of the private-public key pair is used to generate nullifiers for the UTXO commitments by hashing the private key with the commitment. The purpose of this nullifier is to create a binding identity with the UTXO commitment, so that it cannot be spent more than once. See snippet below for context.

Snippet from `ZAccountRenewalV1()`

```
component zAccountUtxoInNullifierHasher = ZAccountNullifierHasher();
zAccountUtxoInNullifierHasher.privKey <== zAccountUtxoInNullifierPrivKey;
    // r * RootPrivKey
zAccountUtxoInNullifierHasher.commitment <== zAccountUtxoInNoteHasher.out;

component zAccountUtxoInNullifierHasherProver = ForceEqualIfEnabled();
zAccountUtxoInNullifierHasherProver.in[0] <== zAccountUtxoInNullifier;
zAccountUtxoInNullifierHasherProver.in[1] <== zAccountUtxoInNullifierHasher
    .out;
zAccountUtxoInNullifierHasherProver.enabled <== zAccountUtxoInNullifier;
```

The renewal process is important to comply with regulations, ensuring that each user is KYC verified periodically. The KYC requirements may also change in the future, so this ensures that each user is in compliance with changing regulations.

In the `zAccountRenewalV1` circuit, there is no constraint which enforces that the `zAccountUtxoInNullifierPubKey[2]` is derived from the `zAccountUtxoInNullifierPrivKey`. Therefore, any private key can fulfill the constraints for such a UTXO commitment. This allows generating multiple `zAccountUtxoInNullifiers` by using different values for `zAccountUtxoInNullifierPrivKey`.

Impact Because `zAccountRenewalV1` allows creating multiple nullifiers for the same UTXO note commitment, an attacker can repeat a renewal process many times for the same zAccount UTXO. This allows them to bypass the periodic KYC compliance requirements, as well as changing KYC regulations.

The renewal process also allows depositing and withdrawing ZKP tokens as part of its workflow. An attacker can spend the same UTXO multiple times to withdraw ZKP tokens and drain the protocol.

Recommendation Add constraints which verify that signals `zAccountUtxoInNullifierPubKey[2]` are generated from signal `zAccountUtxoInNullifierPrivKey`.

Developer Response The developers fixed the issue at commit 903edd3.

4.1.3 V-PAN-VUL-003: Incorrect processing of Deposit+Withdraw transactions

Severity	Critical	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`DepositAndWithdrawHandler.sol`		
Location(s)	_processDepositAndWithdraw		
Confirmed Fix At	f891ff0		

The function `_processDepositAndWithdraw` is responsible for handling deposit and/or withdrawal transactions in the Panther protocol. Transactions in the protocol can be either deposits or withdrawals, but the current implementation assumes that the function will only receive transactions of one type at a time.

```
function _processDepositAndWithdraw(
   uint256[] calldata inputs,
   uint16 transactionType,
   uint96 protocolFee
) internal {
   uint96 depositAmount = inputs[MAIN_DEPOSIT_AMOUNT_IND].safe96();
   uint96 withdrawAmount = inputs[MAIN_WITHDRAW_AMOUNT_IND].safe96();
   (uint8 tokenType, address tokenAddress) = inputs[MAIN_TOKEN_IND]
            .getTokenTypeAndAddress();
   if (transactionType.isDeposit()) {
      // process deposit
      // (code removed)
   }
   if (transactionType.isWithdrawal()) {
      // process withdrawal
      // (code removed)
   }
```

Snippet 4.2: Snippet from `_processDepositAndWithdraw()`

Impact For all transactions where both a deposit and withdrawal occur at the same time, the proper token transfer will not take place, while the UTXO will be successfully created. This could be used to deplete the protocol by generating fake UTXOs for funds that are not actually being transferred to the Vault, and then withdrawing those UTXOs later.

Recommendation In case the input transaction is of a mixed type, both the `isDeposit()` and `isWithdrawal()` methods should return true.

Developer Response The developers fixed the issue at commit f891ff0.

4.1.4 V-PAN-VUL-004: Nullifier verification can be disabled

Severity	Critical	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`zSwapV1.circom`		
Location(s)	Template ZSwapV1		
Confirmed Fix At	69db60e		

In the template `ZSwapV1`, the nullifier `zAccountUtxoInNullifier` is used to ensure that the zAccount input UTXO can only be spent once.

The component `ZAccountNullifierHasher` creates the nullifier hash from the `zAccountUtxoInNullifierPrivKey` and the zAccount input UTXO commitment. This hash is verified against the input signal `zAccountUtxoInNullifier`. See snippet below for the implementation.

```
1  // Hashes priv key and account note to get nullifier hash
2  component zAccountUtxoInNullifierHasher = ZAccountNullifierHasher();
3  zAccountUtxoInNullifierHasher.privKey <== zAccountUtxoInNullifierPrivKey;
4  zAccountUtxoInNullifierHasher.commitment <== zAccountUtxoInHasher.out;
5
6  // Verifies the nullifier hash is same as the input nullifier
7  component zAccountUtxoInNullifierHasherProver = ForceEqualIfEnabled();
8  zAccountUtxoInNullifierHasherProver.in[0] <== zAccountUtxoInNullifier;
9  zAccountUtxoInNullifierHasherProver.in[1] <== zAccountUtxoInNullifierHasher.out;
10 zAccountUtxoInNullifierHasherProver.enabled <== zAccountUtxoInSpendPrivKey;
```

Snippet 4.3: Snippet from template `ZSwapV1()`

The template `ForceEqualIfEnabled` which performs the nullifier verification, can be disabled if `enabled` is set to `0`. As see in the code snippet, `enabled` is assigned the value of `zAccountUtxoInSpendPrivKey`.

By assigning a value of `0` to this private key, the nullifier verification can be disabled, which will allow any nullifier to be associated with this particular zAccount UTXO commitment. This is possible because there are no other constraints applied to `zAccountUtxoInNullifier` and `0` is a valid `zAccountUtxoInSpendPrivKey`(as explained below).

The `zAccountUtxoInSpendPrivKey` is derived from the root spending private key and used to generate the derived spending public key. A value of `0` for the `zAccountUtxoInSpendPrivKey` can be derived by multiplying the root spend private key with `0`. The spending public key generated from such a private key will be the point at infinity `(0, 1)`. The above attack is possible because this private/public key-pair still satisfies all the other constraints in the circuit and is a valid key-pair.

Impact An attacker can disable the nullifier verification for `zAccountUtxoInNullifier` and spend the same zAccount input UTXO infinitely many times. There are no other constraints on `zAccountUtxoInNullifier`, and it can be replaced with any value if the nullifier check is disabled.

This particular nullifier verifies the commitment to the users zAccount input UTXO, ensuring that it can only be spent once. This bug will allow a user to spend this UTXO as many times as they want.

Recommendation The nullifier verification should always be enabled. Instead of using `zAccountUtxoInSpendPrivKey` to enable it conditionally, use `1` to enable it perpetually.

Developer Response The developers fixed the issue at commit 69db60.

4.1.5 V-PAN-VUL-005: Bundler fee amount is not checked in PayMaster

Severity	Critical	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`PayMaster.sol`		
Location(s)	validatePaymasterUserOp , postOp		
Confirmed Fix At	dfe6b30		

Panther Protocol implements ERC-4773 to enable users to submit requests (so called user operations) using third-party service nodes called bundlers (or relayers). Among other things, this standard allows users to pay for submitted transactions using tokens other than ETH. In case of Panther, ZKP token is to be used for that.

The ERC4773 standard is implemented through three contracts: EntryPoint, PayMaster, and Account. The main logic for processing is implemented in the pre-existing contract EntryPoint. Panther implements the PayMaster and Account contracts only.

The PayMaster contract is responsible for covering the expenses associated with user operation processing. This includes two callback functions: `validatePaymasterUserOp` and `postOp`. These functions are called by EntryPoint before and after a user's operation is executed, respectively.

EntryPoint uses these functions to determine whether a given user operation can be paid for by the funds held by PayMaster. If it can, EntryPoint will charge the specified amount from PayMaster's deposit and transfer it to the recipient after the operation has been completed.

The current implementation of PayMaster ensures that the specified number of ZKPs associated with the user's transaction through the paymasterCompensation parameter are sufficient to cover the requiredPreFund amount requested by the Bundler. This amount encompasses both the actual gas costs and the service fee that must be paid to the Bundler for their services.

The issue here is that neither the `validatePaymasterUserOp` nor the `postOp` callbacks of the Paymaster limit the funds requested by the Bundler to cover expenses. This logic would work well if user operations always succeeded, as the associated ZKPs would cover the requested costs. However, if the submitted user operation fails for any reason, the requested amount will still be charged.

Impact The current implementation is vulnerable to the following attack vector:

1. After the Paymaster has accumulated a considerable amount of fees, the attacker calls the`Paymaster.claimEthAndRefundEntryPoint` function to swap all the ZKPs for ETH and send them to the EntryPoint's deposit.
2. The attacker submits a dummy user operation through the `EntryPoint.handleOps` , setting themselves as the beneficiary and setting a large enough `requiredPreFund` that would require all Paymaster's funds to cover it.
3. In the submitted user operation, the attacker specifies a high enough `paymasterCompensation` value measured in ZKPs. The current ZKP balance of their user's account should not contain this amount of ZKPs.
4. The processing of this user operation inside Panther will revert, since the user account doesn't have the necessary amount of ZKPs, but the requested funds will still be sent to

the attacker beneficiary address, because the PayMaster approved it in both `validatePaymasterUserOp` and `postOp` callback functions.

After this is completed, the whole PayMaster deposit gets drained.

Recommendation PayMaster callbacks shall check that the requested amount is within a reasonable range.

Developer Response The developers fixed the issue at commit dfe6b30.

4.1.6 V-PAN-VUL-006: getQuoteAmount function misuse leads to incorrect fees accounting

Severity	Critical	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`FeeMaster.sol`		
Location(s)	getNativeRateInZkp , getZkpRateInNative		
Confirmed Fix At	002129c		

The `UniswapV3PriceFeed.getQuoteAmount()` function is used to calculate the amount of output tokens the caller would receive in exchange for a given amount of input tokens, based on the current exchange rate.

However, there is a catch - if the address of the input token is greater than the address of the output token, the result will be reverted: the input token will be treated as the output token in this case.

Functions `FeeMaster.getNativeRateInZkp()` and `FeeMaster.getZkpRateInNative()` do not handle this specific situation.

Impact If `WETH` address happens to be greater than `ZKP_TOKEN` address, results of the functions will be incorrect, rendering the `accountFees()` function logic incorrect.

Recommendation The described corner case has to be properly handled in the functions.

Developer Response The developers fixed the issue at commit 002129c.

4.1.7 V-PAN-VUL-007: ZSwap is missing logic for depositing tokens

Severity	Critical	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`ZSwap.sol`		
Location(s)	swapZAsset		
Confirmed Fix At	4a4dc21		

The Panther protocol's swap functionality allows users to exchange their tokens for other types of tokens on an external exchange privately. The tokens that users want to swap can be either existing UTXOs that belong to the user and are stored in the protocol or external tokens that have not been deposited in the protocol yet. The deposit process is expected to occur during the same transaction as the swap, making it more convenient compared to depositing as a separate step.

The issue is that even if a user has entered a non-zero value for a deposit, indicating their willingness to deposit external tokens, this action does not actually take place in the `ZSwap` contract. Instead, the transaction is processed in the circuit as if the tokens had been successfully deposited, despite no actual transfer to the Vault taking place.

```
function swapZAsset(
        uint256[] calldata inputs,
        SnarkProof calldata proof,
        uint32 transactionOptions,
        uint96 paymasterCompensation,
        bytes memory swapData,
        bytes calldata privateMessages
    ) external returns (uint256 zAccountUtxoBusQueuePos) {
       // ... skipped ...
       // @audit: no deposit processing here
       uint160 circuitId = circuitIds[TT_ZSWAP];
       verifyOrRevert(circuitId, inputs, proof);
       (
          bytes32[2] memory zAssetUtxos,
          uint256 zAssetAmountScaled
       ) = _getKnownPluginOrRevert(swapData).processSwap(swapData, inputs);
       (
          zAccountUtxoQueueId,
          zAccountUtxoIndexInQueue,
          zAccountUtxoBusQueuePos
       ) = PANTHER_TREES.insertZSwapUtxos(
                inputs,
                zAssetUtxos,
                transactionOptions,
                miningReward
       );
       // ... skipped ...
    }
```

Impact This issue allows an attacker to steal tokens directly from the Vault by creating new UTXOs with tokens that they claim to have deposited during the swap process.

Recommendation The ZSwap contract must either correctly process non-zero deposits or disallow them entirely.

Developer Response The developers fixed the issue at commit 4a4dc21.

4.1.8 V-PAN-VUL-008: Possibility of zAccountId overflow

Severity	High	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	ZAccountsRegistration.sol		
Location(s)	registerZAccount()		
Confirmed Fix At	3d40447		

For each new zAccount being registered in the Panther, the function `registerZAccount()` has to be called. The ZAccount identifier is a `uint256` value generated by a monotonically increasing counter inside the `getNextZAccountId()` function. However, when assigned to a variable, this value gets downcasted to `uint24` type. Unlike arithmetic operations, down-casts do not lead to runtime errors in Solidity, and hence the overflow will stay unnoticed.

```
1  function registerZAccount(
2          G1Point memory _pubRootSpendingKey,
3          G1Point memory _pubReadingKey,
4          uint8 v,
5          bytes32 r,
6          bytes32 s
7      ) external {
8      // ... skipped
9      uint24 zAccountId = uint24(_getNextZAccountId());
10     // ... skipped
11     masterEOAs[zAccountId] = masterEoa;
12     zAccounts[masterEoa] = _zAccount;
13     emit ZAccountRegistered(masterEoa, _zAccount);
14 }
```

Snippet 4.4: Snippet from `registerZAccount()`

Impact The `uint24` type can only hold 16,777,216 unique values. If the `registerZAccount` function is called that many times (with a different public key each time, to pass the check on line 162), the zAccountId will wrap around due to the down-cast, leading to the overwriting of previously registered zAccounts.

If an attacker knows that a specific EOA or zAccountId is going to receive a payment soon, they may try to overwrite the information about this zAccountId and replace the original ZAccount with their own structure containing their public keys. Since the unspent transaction output (UTXO) will be linked to the attacker's keys, they will then be able to spend this UTXO.

Also, to overflow the counter, it will take roughly 40 minutes on Polygon network at worst.

Recommendation If the `uint24` type is dictated by a lower-level protocol implementation and can't be increased at this time, it is recommended to at least add a check that will prevent the rewriting of zAccounts that are already linked to some masterEOA. This will stop the attack from being potentially profitable.

Developer Response The developers fixed the issue at commit 3d40447.

4.1.9 V-PAN-VUL-009: Blacklist states cannot be represented within the circom field

Severity	High	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	zAccountBlackListLeafInclusionProver.circom		
Location(s)	Template ZAccountBlackListLeafInclusionProver		
Confirmed Fix At	1763ca4, 966f691		

The template `ZAccountBlackListLeafInclusionProver` checks if a particular `ZAccount` is blacklisted or not. It takes as input the blacklist `leaf` and uses the `zAccountId` to derive the path to the leaf and build the index that the `zAccountId` corresponds to within the leaf.

The `zAccountId` has `24` bits. Of those, the `16` MSB serve as the path index to the leaf inside the `zAccountBlacklist` merkle tree. A leaf within the tree has `254` bits, and each bit denotes the activation of blacklisting for a particular `zAccountId`. And the `8` LSB of the `zAccountId` are used to determine which bit of the leaf to check for its blacklist status. If the `8` LSB of the `zAccountId` denote the value `200`, then the leaf at index `200` is checked for the activation flag. See snippet below for the implementation.

```
1
2  // Build the index inside leaf
3  component b2n_zAccountIdInsideLeaf = Bits2Num(8);
4  for (var j = 0; j < 8; j++) {
5      b2n_zAccountIdInsideLeaf.in[j] <== n2b_zAccountId.out[j];
6  }
7
8  assert(b2n_zAccountIdInsideLeaf.out < 254); // regular scalar field size
9
10 // switch-on single bit
11 component n2b_leaf = Num2Bits(254);
12 n2b_leaf.in <== leaf;
13
14 component is_zero[254];
15
16 for(var i = 0; i < 254; i++) {
17     // is_zero[i].out == 1 only when i == b2n_zAccountIdInsideLeaf.out
18     is_zero[i] = IsZero();
19     is_zero[i].in <== i - b2n_zAccountIdInsideLeaf.out;
20     // make sure that for our zAccountId LSB inside leaf, the bit is zero,
21     // for example: zAccountId LSB = 200, for i = 200, is_zero[i].out == 1 --> if
       n2b_leaf.out[i] == 1, then the assertion will fail
22     // which means that our zAccountId is blacklisted !
23     is_zero[i].out * n2b_leaf.out[i] === 0;
24 }
```

Snippet 4.5: Snippet from template `ZAccountBlackListLeafInclusionProver()`

There is an assumption that all possible states represented by the `ZAccountBlackList` leaf are representable within the circom field. This is incorrect because the leaf has `254` bits and therefore it can represent a certain range of values which are larger than the circom prime. This has a subtle implication that given a particular blacklist leaf, if activating the blacklist status for a particular `zAccountId` pushes the value of the leaf to be larger than the circom prime, then a

merkle inclusion proof for that leaf cannot be created.

Let's understand this with an example. We consider the leaf which denotes the blacklist status of `zAccountIds` in the range `0-253`. Lets suppose that currently the `zAccountIds` `252` and `251` are banned so the corresponding bits in the leaf are activated. Now, we want to ban the `zAccountId` 253 and we activate the corresponding bit. This particular `ZAccountBlackList` state is denoted by the leaf value `2^253 + 2^252 + 2^251`. But, this value is larger than the circom prime. See snippet below for context.

```
sage: a = 2^253 + 2^252 + 2^251
// Below is the circom prime
sage: b =
    21888242871839275222246405745257275088548364400416034343698204186575808495617
sage: a > b
True
```

Snippet 4.6: Snippet of POC in `sagemath`

Therefore for this particular leaf value, a blacklist inclusion proof cannot be created. This implies that because certain states of the blacklist leaf are not representable, depending on the current state of a `ZAccountBlackList` leaf, it may not be possible to ban certain `zAccountIds`. For illustration, in the above example the zAccount with id `253` could not be banned.

Impact If the value of a leaf within the `ZAccountBlackListMerkleTree` is sufficiently large, then it may not be possible to further blacklist `zAccountIds` represented within the leaf. A malicious entity can orchestrate such a leaf state, to protect their other `zAccountIds` from being banned.

Recommendation Increase the height of the `ZAccountBlackListMerkleTree` to `17` and decrease the leaf bit space to `128`. This will allow a blacklist leaf to safely represent all possible states.

Alternatively, update `_getNextZAccountId` in the smart contracts, to also skip value `253` during the account generation process. This function currently skips the values `254` and `255` for the 8 `LSBs` of a `zAccountId`, to ensure that a blacklist leaf can represent the state within `254` bits. This exclusion reduces the leaf bit space to `253`, which will allow the leaf to safely represent all possible states.

Developer Response The developers fixed the issue at commits 1763ca4 and 966f691.

4.1.10 V-PAN-VUL-010: Unsafe use of Num2Bits(254) on blacklist leaf

Severity	High	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`zAccountBlackListLeafInclusionProver.circom`		
Location(s)	Template ZAccountBlackListLeafInclusionProver		
Confirmed Fix At	6dfcc56		

The template `ZAccountBlackListLeafInclusionProver` uses the circomlib template `Num2Bits(254)` to represent a leaf within the blacklist merkle tree. See this V-PAN-VUL-009 for a more detailed description on how the blacklist inclusion mechanism works.

As described here, this can have detrimental effects on the codebase because `Num2Bits` is not deterministic when the template parameter is greater or equal to 254.

```
template Num2Bits(n) {
    signal input in;
    signal output out[n];
    var lc1=0;

    var e2=1;
    for (var i = 0; i<n; i++) {
        out[i] <-- (in >> i) & 1;
        out[i] * (out[i] -1 ) === 0;
        lc1 += out[i] * e2;
        e2 = e2+e2;
    }

    lc1 === in;
}
```

Snippet 4.7: `Num2Bits` implementation

Because the `zAccountBlacklistMerkleTree` makes use of the leaf bit representation to signify blacklist status of a particular `zAccountId`, by using an alternate `out` value for the leaf bit representation an adversary can prove that their account id is excluded from the blacklist merkle tree when it may in fact be banned. See snippet below for context.

```
assert(b2n_zAccountIdInsideLeaf.out < 254); // regular scalar field size

// switch-on single bit
component n2b_leaf = Num2Bits(254);
n2b_leaf.in <== leaf;

component is_zero[254];

for(var i = 0; i < 254; i++) {
    // is_zero[i].out == 1 only when i == b2n_zAccountIdInsideLeaf.out
    is_zero[i] = IsZero();
    is_zero[i].in <== i - b2n_zAccountIdInsideLeaf.out;
    // make sure that for our zAccountId LSB inside leaf, the bit is zero,
    // for example: zAccountId LSB = 200, for i = 200, is_zero[i].out == 1 --> if
    n2b_leaf.out[i] == 1, then the assertion will fail
```

```
    // which means that our zAccountId is blacklisted !
    is_zero[i].out * n2b_leaf.out[i] === 0;
}
```

Snippet 4.8: Snippet from template `ZAccountBlackListLeafInclusionProver()`

Impact The use of `Num2Bits(254)` in the above scenario renders the circuit non-deterministic. Therefore an attacker can create a bogus proof that their `zAccountId` is excluded from the blacklist Merkle Tree when it is actually banned.

Recommendation It is recommended to use `Num2Bits_strict`, which ensures that the bit representation is smaller than the field's prime.

Developer Response The developers fixed the issue at commit 01995d1 and 6dfcc56.

4.1.11 V-PAN-VUL-011: Incorrect handling of WEth tokens during swap

Severity	High	Commit	a16a43e
Type	Logic Error	Status	Fixed
File(s)	`UniswapV3RouterPlugin.sol` , `QuickswapRouterPlugin.sol`		
Location(s)	execute()		
Confirmed Fix At	b483953		

The described issue is related to both UniswapV3 swap plugin and Quickswap plugin contracts.

The `execute()` function is called during the token swap transaction. If the swap destination token happens to be WETH, this function will wrongly unwrap the tokens and send it to the native balance of the Vault contract, instead of WETH balance.

```
function execute(
   PluginData calldata pluginData
) external payable returns (uint256 amountOut)
{
  (address tokenIn, address tokenOut) =
     pluginData.getTokenInAndTokenOut(WETH);

  address recipient = _getOutputRecipient(tokenOut);

  amountOut = _execute(
            tokenIn,
            tokenOut,
            amountIn,
            nativeInputAmount,
            recipient,
            data
        );
   if (tokenOut == WETH)
        withdrawWethAndTransferToVault(amountOut);
}
```

Snippet 4.9: Snippet from `execute()`

Impact Since the output UTXO will be tied to WETH, and not native Ether, the user will not be able to spend their swapped tokens.

Recommendation It is recommended to convert WETH tokens to native Ether only if the target token is set to be native Ether. Currently, the check for this is done after the token substitution in the `getTokenInAndOut` function, which is not correct.

Developer Response The developers fixed the issue at commit b483953.

4.1.12 V-PAN-VUL-012: Incorrect ZKP balance accounting in ProcessConversion

Severity	High	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`ConversionHandler.sol`		
Location(s)	_processConversion()		
Confirmed Fix At	867f30c		

The function `ConversionHandler._processConversion()` is responsible for swapping PRP tokens for ZKP tokens during a PRP conversion transaction. This function keeps track of the actual PRP and ZKP reserve balances. After the swap has been made, this function attempts to update the reserve balances with the actual values. However, the ZKP reserve balance is updated with an outdated value.

```
uint256 zkpBalance = zkpToken.safeBalanceOf(address(this));
require(prpVirtualBalance * zkpBalance >= _prpReserve * _zkpReserve,
        ERR_LOW_CONSTANT_PRODUCT
);
_update(prpVirtualBalance, zkpBalance);
_lockZkp(zkpToken, zkpAmountOutScaled);
```

Snippet 4.10: Snippet from `_processConversion()`

Here, the ZKP balance is stored in the `zkpBalance` variable, which is later used in the `_update()` function. However, the actual ZKP balance may change after this update, due to the `_lockZKP` call, which transfers ZKP tokens from the PantherPool to the Vault.

Impact Improperly accounted ZKP balance will break the PRP/ZKP swap logic, leading to incorrect swap rates and funds loss for users.

Recommendation The ZKP accounting logic should be executed after the `_lockZkp` call, and not before.

Developer Response The developers fixed the issue at commit cf263e3.

4.1.13 V-PAN-VUL-013: PantherPool does not update Vault allowance for ZKP tokens

Severity	High	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`PrpConversion.sol`		
Location(s)	see the description		
Confirmed Fix At	d951ac9		

During a PRP to ZKP conversion transaction, the PrpConversion component (a part of PantherPool) requests the Vault to transfer ZKP tokens from PantherPool to Vault, using the `VaultV1.lockAsset()` function. To do this, PantherPool needs to provide sufficient allowance to Vault. Currently, allowance is only set once in the `PrpConversion.initPool()` function and never updated afterwards.

```
function initPool(uint256 prpVirtualAmount, uint256 zkpAmount) external onlyOwner {
    require(!initialized, ERR_ALREADY_INITIALIZED);
    uint256 zkpBalance = ZKP_TOKEN.safeBalanceOf(address(this));
    require(zkpBalance >= zkpAmount, ERR_LOW_INIT_ZKP_BALANCE);
    initialized = true;
    TransferHelper.safeIncreaseAllowance(ZKP_TOKEN, VAULT, zkpAmount);
    _update(prpVirtualAmount, zkpAmount);
    emit Initialized(prpVirtualAmount, zkpAmount);
}
```

Snippet 4.11: Snippet from `PrpConversion.initPool()`

Impact The initial ZKP vault allowance will be spent at some point, making all future transfers unable to function. This will break the PRP's ZKP conversion feature.

Recommendation It is necessary to regularly update the ZKP Vault allowance on behalf of PantherPool.

Developer Response The developers fixed the issue at commit d951ac9.

4.1.14 V-PAN-VUL-014: ZKP tokens get accrued incorrectly during PRP conversion

Severity	High	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`ConversionHandler.sol`		
Location(s)	_processConversion()		
Confirmed Fix At	62af617		

The protocol rewards users with a certain amount of PRP points for completing required actions within the protocol. These PRP points can then be converted into ZKP tokens, which have real market value.

Once the protocol has calculated the number of ZKPs corresponding to a given amount of PRP, it transfers this amount of ZKP tokens from the PantherPool to the Vault. However, there is an issue with this process, as the amount is divided by the `scale` factor before being transferred to the Vault using the `lockZkp` function, which is not accurate.

```
function _processConversion(
    address zkpToken,
    uint96 zkpAmountOutMin,
    uint256[] calldata inputs
) internal returns (uint256 zkpAmountOutScaled)
{
  // ... [VERIDISE] skipped
  zkpAmountOut = getAmountOut(prpWithdrawAmount, _prpReserve, _zkpReserve);
  // ... [VERIDISE] skipped
  zkpAmountOutScaled = zkpAmountOutRounded / scale;
  // ... [VERIDISE] skipped
  _lockZkp(zkpToken, zkpAmountOutScaled);
}
```

Snippet 4.12: Snippet from `ConversionHandler._processConversion()`

Impact An incorrect number of ZKP tokens will be sent to the Vault. As this number is significantly lower than expected, this error could make the protocol insolvent as the Vault will not have enough ZKP tokens available to pay with.

However, there is a softening factor in the form of the `rescueErc20` function, which can help recover the unsent ZKP tokens. This function requires the involvement of protocol owners.

Recommendation It seems that developers wanted to send `zkpAmountOutRounded` instead of the `zkpAmountOutScaled` , and it is recommended to do exactly that.

Developer Response The developers fixed the issue at commit 62af617.

4.1.15 V-PAN-VUL-015: ZoneIdInclusionProver check can be bypassed

Severity	High	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`zoneIdInclusionProver.circom`		
Location(s)	ZoneIdInclusionProver		
Confirmed Fix At	0621e84		

The circuit `ZoneIdInclusionProver` is responsible for verifying if the transfer destination zone is permitted by the current user's zone. To do this, the user must provide their zone ID, a list of allowed target zone IDs, and the position of their zone ID in the list called `offset`.

The logic of the circuit assumes that the offset value should be less than 15, but does not enforce this constraint in the code. Therefore, the main checking loop will silently skip the check if the offset value is set to 15.

```
for(var i = 0; i < 15; i++) {
        is_equal[i] = IsEqual();
        is_equal[i].in[0] <== i;
        is_equal[i].in[1] <== offset;

        forceIsEqual[i] = ForceEqualIfEnabled();
        forceIsEqual[i].in[0] <== zoneId;
        forceIsEqual[i].in[1] <== b2n_zoneIds[i].out;
        // i == offset this is the exact portion of bits to check
        forceIsEqual[i].enabled <== enabled * is_equal[i].out;
    }
```

Snippet 4.13: Snippet from template `ZoneIdInclusionProver`

Impact An attacker is able to bypass target zone checks and send funds to a disallowed zone.

Recommendation It is recommended to constraint the `offset` signal to be less than 15.

Developer Response The developers fixed the issue at commit 0621e84.

4.1.16 V-PAN-VUL-016: Zone related limits can be bypassed

Severity	High	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`zSwapV1.circom`		
Location(s)	ZSwapV1		
Confirmed Fix At	54d4d0a		

Panther protocol implements a concept of *zones* as a measure to put compliance policies for different kinds of users. One such policy is the limited amount of funds that a user belonging to a specific zone is allowed to transfer within a certain time period. For example, in a 24-hour period, a user account of zone 0 may only be allowed to transfer a maximum of $1000 worth of tokens.

The `ZSwapV1` circuit logic implementing this functionality has an error. Consider the following snippet.

```
component isDeltaTimeLessEqThen = LessEqThan(32); // 1 if deltaTime <=
    zZoneTimePeriodPerMaximumAmount
isDeltaTimeLessEqThen.in[0] <== deltaTime;
isDeltaTimeLessEqThen.in[1] <== zZoneTimePeriodPerMaximumAmount;
signal zAccountUtxoOutTotalAmountPerTimePeriod <== Uint96Tag(ACTIVE)(
    isDeltaTimeLessEqThen.out * (totalBalanceChecker.totalWeighted +
    zAccountUtxoInTotalAmountPerTimePeriod ));

assert(zAccountUtxoOutTotalAmountPerTimePeriod <= zZoneMaximumAmountPerTimePeriod);
component
    isLessThanEq_zAccountUtxoOutTotalAmountPerTimePeriod_zZoneMaximumAmountPerTimePeriod
     = ForceLessEqThan(96);
isLessThanEq_zAccountUtxoOutTotalAmountPerTimePeriod_zZoneMaximumAmountPerTimePeriod.
    in[0] <== zAccountUtxoOutTotalAmountPerTimePeriod;
isLessThanEq_zAccountUtxoOutTotalAmountPerTimePeriod_zZoneMaximumAmountPerTimePeriod.
    in[1] <== zZoneMaximumAmountPerTimePeriod;
```

Snippet 4.14: Snippet from `ZSwapV1()`

The `zAccountUtxoOutTotalAmountPerTimePeriod` signal is only assigned a correct total amount if the `deltaTime` (difference between current time and the account UTXO creation time) is less than `zZoneTimePeriodPerMaximumAmount`. If it is larger, the value of the signal becomes 0.

The value of the `zAccountUtxoOutTotalAmountPerTimePeriod` signal is compared to the maximum allowed limit for the zone. If the signal value exceeds this limit, the circuit should deny any operations. However, because the value is 0, any `totalBalanceChecker.totalWeighted` amount is allowed, even if it exceeds the `zZoneTimePeriodPerMaximumAmount` limit.

Impact This issue allows for easy bypassing of one of the important zone policy, which could lead to serious compliance problems for the protocol.

Recommendation The current circuit logic does not implement the expected behavior correctly and should be fixed.

Developer Response The developers fixed the issue at commit 54d4d0a.

4.1.17 V-PAN-VUL-017: zAccountRenewalV1 circuit does not validate KYC certificates for expiry

Severity	High	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`zAccountRenewalV1.circom`		
Location(s)	ZAccountRenewalV1		
Confirmed Fix At	3375335		

The ZAccountRenewalV1 circuit does not validate the `kycSignedMessageTimestamp` signal value in any way.

Impact Since KYC certificates are not tracked on the smart-contract level, there is a possibility of reusing the same KYC certificate for all further account renewals, even if certificates become expired.

Recommendation The circuit must ensure that the KYC certificate provided is not expired. Alternatively, the protocol can track used KYC certificates at the smart contract level and not allow to reuse the same certificate twice. In this case, the KYC provider would handle all necessary checks.

Developer Response The developers fixed the issue at commit 3375335.

4.1.18 V-PAN-VUL-018: Incorrect token identifier is used for swap

Severity	Medium	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`SwapHandler.sol`		
Location(s)	_executeSwapAndVerifyOutput()		
Confirmed Fix At	cc5e8cd		

The `_executeSwapAndVerifyOutput()` function inside the SwapHandler contract is responsible for performing a swap operation across two tokens named existing token and incoming token. The current function implementation takes the incoming token identifier from the wrong parameter - the existing token.

```
function _executeSwapAndVerifyOutput(
   address plugin,
   address vault,
   uint256[] memory inputs,
   bytes memory swapData
) private returns (uint96 _outputAmount) {
   (uint8 existingTokenType, address existingTokenAddress) = inputs[
      ZSWAP_EXISTING_TOKEN_IND
   ].getTokenTypeAndAddress();

   (uint8 incomingTokenType, address incomingTokenAddress) = inputs[
      ZSWAP_EXISTING_TOKEN_IND
   ].getTokenTypeAndAddress();
        ...
}
```

Snippet 4.15: Snippet from `_executeSwapAndVerifyOutput()`

Impact The token swap functionality does not work.

Recommendation Use the correct parameter for the incoming token, which is `inputs[ZSWAP_INCOMING_TOKEN_IND]`.

Developer Response The developers fixed the issue at commit cc5e8cd.

4.1.19 V-PAN-VUL-019: KYT signature verification process fails for any non-zero signed message hash

Severity	Medium	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	Template TrustProvidersKyt		
Location(s)	trustProvidersKyt.circom		
Confirmed Fix At	ccb2a8b		

The circuit `TrustProvidersKyt` allows disabling the KYT verification check in certain scenarios. The verification is disabled if the amount related to the deposit/withdraw/internal transaction is `0`. Additionally, in the swap case, verification can be disabled if the smart contracts agree to a zero-hash for the signed message hash. This is done to account for cases where the swap is made for already KYT accepted funds.

The snippet below highlights the particular section of the circuit, which deals with the KYT verification activation for a deposit transaction. The output of the conditional is passed to the `BinaryTag`, which constrains the output to be binary and is then assigned to `isKytDepositCheckEnabled`. But, the result of this conditional can actually be non-binary.

```
1 signal isKytDepositCheckEnabled <== BinaryTag(ACTIVE)(
2     isSwap ? kytDepositSignedMessageHash * (1-isZeroDeposit.out)
3     :(1 - isZeroDeposit.out));
```

Snippet 4.16: Snippet from template `TrustProvidersKyt()`

If we are dealing with a swap which has non-zero deposit amounts, the output of the conditional will be equal to `kytDepositSignedMessageHash`. And the `kytDepositSignedMessageHash` is only `0` when the smart contracts agree on the zero-hash, otherwise it contains the hash of the signed message which is not binary. For such a case, the `BinaryTag` constraint will fail and it will disallow users to generate a proof.

This issue is repeated in the same circuit for signals `isKytWithdrawCheckEnabled` and `isKytInternalCheckEnabled`.

Impact The KYT verification process will fail for swaps where the involved amount and the signed message hash are both non-zero. This will disallow users to generate a proof in these scenarios.

Recommendation Replace `kytDepositSignedMessageHash` in the conditional with the output of the template `IsNotZero`. It should take `kytDepositSignedMessageHash` as its input.

Also apply the same recommendation for `kytWithdrawSignedMessageHash` and `kytSignedMessageHash`.

Developer Response The developers fixed the issue at commit ccb2a8b.

4.1.20 V-PAN-VUL-020: forTxReward gets abstracted away in the reward calculation

Severity	Medium	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`rewardsExtended.circom`		
Location(s)	Template RewardsExtended		
Confirmed Fix At	1055906		

In the circuit `rewardsExtended`, the total rewards a user receives is denoted by the intermediate signal `R` which is computed as the expression `S1 + S5`. The result of the expression `R` is then divided by the `prpScaleFactor` to represent the amount in terms of `PRP`. See snippet below for the implementation.

However, the signal `S1` which is assigned the value of `forTxReward`, is already represented in terms of `PRP`. Because the amount is added to `S5`, and then divided by `prpScaleFactor` again, the contribution that `forTxReward` has to the final reward amount will be extremely minute or negligible. It will effectively be abstracted away.

```
S1 <== forTxReward; // 2^40
S2 <== forDepositReward * depositScaledAmount;

///-------------------------///

S3 <== sum[nUtxoIn-1];
S4 <== forUtxoReward * S3; // 2^40 x 2^100 = 2^140
S5 <== (S4 + S2) * assetWeight; // 2^104 x 2^48 = 2^152
R <== S1 + S5; // 2^40 + 2^152 = 2^153 (at most)

component n2b = Num2Bits(253);
n2b.in <== R;

component b2n = Bits2Num(253-prpScaleFactor);
for (var i = prpScaleFactor; i < 253; i++) {
  b2n.in[i-prpScaleFactor] <== n2b.out[i];
}
```

Snippet 4.17: Snippet from template `RewardsExtended()`

Impact The reward amount for doing a transaction will get abstracted away when the total reward is scaled down by `prpScaleFactor`. This implies that the transaction reward will never actually be properly accounted for in the reward that the user receives.

Recommendation In the total rewards computation, add the `forTxReward` to the amount `S5` **after** it has been divided by `prpScaleFactor`.

Developer Response The developers fixed the issue at commit 1055906.

4.1.21 V-PAN-VUL-021: Data escrow encrypted message constructed from incorrect input

Severity	Medium	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`dataEscrowElGamalEncryption.circom`		
Location(s)	Template DataEscrowElGamalEncryption()		
Confirmed Fix At	7ff5ba7		

The template `DataEscrowElGamalEncryption` constructs encrypted messages which are published on-chain through events. The encrypted message contains the padding points, the scalar message and then the derived UTXO spending public key. An encrypted message at a particular index is derived using the expression `ephemeralRandom * pubKey + M + HidingPoint`, where `M` is the scalar message being encrypted and `ephemeralRandom` is the randomness used to generate the ephemeral public keys. See snippet below for context.

```
// ephemeralRandom * pubKey + M + HidingPoint
drv_mGrY[j] = BabyAdd();
drv_mGrY[j].x1 <== paddingPoint[j].Ax;
drv_mGrY[j].y1 <== paddingPoint[j].Ay;
drv_mGrY[j].x2 <== ephemeralPubKeyBuilder.sharedPubKey[j][0];
drv_mGrY[j].y2 <== ephemeralPubKeyBuilder.sharedPubKey[j][1];

drv_mGrY_final[j] = BabyAdd();
drv_mGrY_final[j].x1 <== drv_mGrY[j].xout;
drv_mGrY_final[j].y1 <== drv_mGrY[j].yout;
drv_mGrY_final[j].x2 <== ephemeralPubKeyBuilder.hidingPoint[0];
drv_mGrY_final[j].y2 <== ephemeralPubKeyBuilder.hidingPoint[1];

// encrypted data
encryptedMessage[j][0] <== drv_mGrY[j].xout;
encryptedMessage[j][1] <== drv_mGrY[j].yout;
```

Snippet 4.18: Snippet from template `DataEscrowElGamalEncryption()`

For the segment of the encrypted message which constructs the padding point, it uses an incorrect input to construct the encrypted data. As per the expression, the output of the component `drv_mGrY_final` should be used to construct the encrypted data, whereas currently the output of the component `drv_mGrY` is being used.

The current implementation constructs the encrypted data without including the hiding points. The hiding points are added to ensure that UTXO's from the same source but meant for different receivers cannot be deciphered by other receivers.

Impact The padding points data is encrypted without using the hiding points.

Recommendation Replace `drv_mGrY` with `drv_mGrY_final` in the location highlighted above.

Developer Response The developers fixed the issue at commit 7ff5ba7.

4.1.22 V-PAN-VUL-022: PrpConversion does not account ZKP reserve changes

Severity	Medium	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`PrpConversion.sol`		
Location(s)	rescueErc20		
Confirmed Fix At	88acc96		

The `PrpConversion` contract includes a function called `rescueErc20` that allows the protocol owner to transfer ERC20 tokens out of the Panther Pool. This function does not update the balance of reserve values, such as the ZKP reserve balance, after tokens transfer.

```
1 /// @dev May be only called by the {OWNER}
2 function rescueErc20(address token, address to, uint256 amount) external onlyOwner {
3    _claimErc20(token, to, amount);
4 }
```

Snippet 4.19: Snippet from `PrpConversion.rescueErc20()`

Impact If for some reason, the protocol owner decides to partially transfer ZKP tokens from the Panther Pool, it will cause the accounting of the ZKP reserves to be incorrect, leading to an inaccurate PRP for the ZKP swap rate. Additionally, the function `increaseZkpReserve()` may not work in certain cases due to a check on line 83.

Recommendation It is recommended to properly reflect a potential ZKP balance change in the `_zkpReserve` variable during the rescuing operation.

Developer Response The `rescueErc20` was removed from `PrpConversion.sol` in commit 88acc96.

4.1.23 V-PAN-VUL-023: Incorrect perUtxoReward value can halt the protocol

Severity	Medium	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`FeeAccountant.sol`		
Location(s)	_updateFeeParams		
Confirmed Fix At	df8f37f		

The `FeeAccountant` module uses the `scPerUtxoReward` storage parameter to calculate the miner's reward each time a transaction occurs. The `_accountDebtForBusTree()` function requires this parameter to be greater than zero.

The `_updateFeeParams()` function is used to set the value of `scPerUtxoReward`. It checks that the supplied value is greater than zero, but before setting `scPerUtxoReward`, the original value is divided by `1e12` and assigned to a storage variable. This step is not accounted for in the logic, so the final value may be zero.

Because `scPerUtxoReward` is measured in ZKP tokens with 18 decimal places, it's perfectly possible for the supplied value to be less than `1e12` .

```
function _updateFeeParams(
        uint96 perUtxoReward,
        uint96 perKytFee,
        uint96 kycFee,
        uint16 protocolFeePercentage
    ) internal returns (FeeParams memory _feeParams) {
        require(perUtxoReward > 0, "Zero per utxo reward");
        // ... skipped ...
        _feeParams = FeeParams({
            scPerUtxoReward: perUtxoReward.scaleDownBy1e12().safe32(),
            scPerKytFee: perKytFee.scaleDownBy1e12().safe32(),
            scKycFee: kycFee.scaleDownBy1e12().safe32(),
            protocolFeePercentage: protocolFeePercentage
        });

        feeParams = _feeParams;
    }
```

Snippet 4.20: Snippet from `example()`

Impact The reward accrual logic will revert, hence new transactions can not be processed.

Recommendation It is recommended to perform the greater than zero check after the parameter has been scaled down.

Developer Response The developers fixed the issue at commit df8f37f.

4.1.24 V-PAN-VUL-024: PureFi session identifiers may be insecure

Severity	Medium	**Commit**	a16a43e
Type	Cryptographic Vulnera	**Status**	Fixed
File(s)	see description		
Location(s)	See description		
Confirmed Fix At	234c61c, 8680a1b		

To make a deposit or a withdrawal, the Panther protocol requires a KYT check to be conducted by a trusted service provider. This check is performed off-chain, and after a successful response is received, the on-chain component will publish a hash of the KYT certificate.

The KYT certificate contains the following fields (we consider the deposit case here):

1. Package type - constant value
2. Timestamp - a timestamp of the KYT request, it has to be close to the time when the UTXO was created.
3. Sender - the funds sender address
4. Receiver - constant value - it is a Panther Vault contract address
5. Token Identifier - public value
6. Session ID - random 31 byte array
7. Rule ID - constant value
8. Amount - amount of tokens being deposited
9. Signer - the true beneficiary of the operation, it coincides with the ZAccountEOA that will be able to spend the UTXO.
   The Signer field is the most sensitive as it reflects the true beneficiary of the transaction, and should therefore never be disclosed. As most of the other certificate values are relatively easy to recover, the brute force resistance of this hash value relies on a single parameter - the Session ID - which is generated by PureFI after successful attestation. However, while it is claimed to be a random number, the real logic behind its generation is unknown, since the service is closed source, and therefore cannot be trusted.

Impact The privacy promise of the protocol relies on a value provided by a third-party service. If, for some reason, this value can be predicted or recovered, the privacy aspects of Panther will be greatly affected, which is not acceptable.

Recommendation It is recommended to find a way to add extra randomness coming from the client side into the KYT request.

Developer Response This is an expected and intentional limitation in the current version. The KYC/T provider is trusted to generate a random session ID for the attestation process. In this version, the designated provider, PureFi, is relied upon to generate these session IDs securely and randomly.

For future versions of the protocol, we plan to modify the KYC/T attestation format to include a random value generated by the user (via their browser). However, these updates cannot be introduced in the current version because they require changes to PureFi's API. PureFi has

committed to implementing these changes within a reasonable timeframe. Once these updates are made, the protocol code will be updated accordingly.

Update: The developers have provided a fix for this issue.

4.1.25 V-PAN-VUL-025: Pontential reentrancy via safeTransferETH

Severity	Medium	**Commit**	a16a43e
Type	Reentrancy	**Status**	Fixed
File(s)	Several files		
Location(s)	Function safeTransferETH		
Confirmed Fix At	781399d		

There is a potential reentrancy originating from `TransferHelper.safeTransferETH`, which may significantly impact the protocol. The reentrancy originates from the following low-level call inside `safeTransferETH`.

```
function safeTransferETH(address to, uint256 value) internal {
    // slither-disable-next-line low-level-calls
    (bool success, ) = to.call{ value: value }(new bytes(0));
    require(success, "TransferHelper: ETH transfer failed");
}
```

Snippet 4.21: Snippet from `TransferHelper.safeTransferETH()`

The main concern here is that this function is invoked in several locations throughout the protocol, including multiple security critical operations like withdrawals. Even though in most cases it is easy to prove that address `to` is that of a trusted contract, there are several cases that this is not the case. Notably, function `VaultV1.unlockAsset` which invokes `safeTransferETH` in the case of a native token transfer with an address (`lData.extAccount`) that may be controlled by external users (see snippet below). To make matters worse, function `unlockAsset` is also called by several other contracts of the protocol which significantly increases the attack surface of the protocol.

```
function unlockAsset(LockData calldata lData) external override onlyOwner {
            _checkLockData(lData);

            if (lData.tokenType == NATIVE_TOKEN_TYPE) {
                payable(lData.extAccount).safeTransferETH(lData.extAmount);
            }
            // ...
```

Snippet 4.22: Snippet from `VaultV1.unlockAsset`

Impact Given the complexity of the protocol, this reentrancy has a non-negligible probability of being exploitable.

Recommendation If possible, swap the low-level call with a call to `transfer`. Alternatively, consider introducing proper reentrancy guards in all places where address `to` is not trusted.

Developer Response The developers have provided a partial fix for this issue at commit 190889. Not all the protocol user-facing functions have been guarded with a reentrancy guard. Developers made this decision based on their desire to reduce gas usage.

Veridise Response The developers fixed the issue at the commit 781399.

4.1.26 V-PAN-VUL-026: Custom encryption scheme lacks security proofs

Severity	Medium	**Commit**	a16a43e
Type	Cryptographic Vulnera	**Status**	Fixed
File(s)	`circuits/templates/dataEscrowElGamalEncryption.circom`		
Location(s)			
Confirmed Fix At	fdabb7f		

The current version of the encryption scheme used by the protocol is non-standard. Specifically, given a single ephemeral key (given as ephemeral randomness), the protocol derives several new ephemeral keys by hashing the previous shared secret using a Poseidon hash. This is a very non-standard way of deriving ephemeral keys, which has not been analyzed from a cryptographic perspective.

Impact This exposes a significant attack surface to the core of the protocol, which revolves around privacy.

Recommendation A more standard way would be to establish a common secret using DH key exchange, then derive the common secret from there using some well studied key derivation methods, and use a symmetric encryption scheme. Alternatively, share the secret to be used by a symmetric key using ElGamal encryption.

Developer Response The developers have provided a fix for this issue.

Veridise Response The proposed fix emphasizes efficiency by using Poseidon as a drop-in replacement for traditional hash functions, optimizing performance in resource-constrained environments. While this approach streamlines execution by bypassing certain encoding and domain separation details, it reflects a practical trade-off given the computational limitations of the environment.

This approach bears some risks involved in designing and implementing custom algorithms and protocols, hence we suggest adopting standards and best practices in the future instead. Using Poseidon in duplex sponge mode for both encryption and authentication aligns better with its intended design.

4.1.27 V-PAN-VUL-027: Potential bypass of extended KYT for internal transactions

Severity	Medium	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`ZSwap.sol`		
Location(s)	_checkNonZeroPublicInputs()		
Confirmed Fix At	8047415		

The Zone security settings include the Zone Sealing flag. If this flag is enabled, all transactions within the zone must undergo an extended KYT check. This means that even internal transactions, such as swaps, must receive approval from the KYT provider.

The problem is that this additional KYT check can be circumvented if the user enters a zero-value for the `ZSWAP_KYT_INTERNAL_SIGNED_MESSAGE_HASH` field, due to the way the current check is implemented in the `trustProvidersKyt` module.

```
signal isKytInternalCheckEnabled <==
    BinaryTag(ACTIVE)(isSwap ? isKytSignedMessageHashIsZero.out * (1 - isZeroInternal
    .out) : (1 - isZeroInternal.out));
```

Snippet 4.23: Snippet from `trustProvidersKyt`

Please note that in the current implementation, the`isKytSignedMessageHashIsZero.out` signal equals 1 if the KYT signed message hash *is not zero*. This is due to a misspelling in the name of the signal.

Impact Bypassing internal KYT checks violates the security and compliance guarantees of the protocol.

Recommendation The ZSwap smart contract must ensure that the value of the `ZSWAP_KYT_INTERNAL_SIGNED_MESSAGE_HASH` signal is not zero within the `_checkNonZeroPublicInputs()` function or at some other suitable location.

Developer Response The developers have implemented a fix.

4.1.28 V-PAN-VUL-028: Improper HMAC implementation

Severity	Medium	**Commit**	a9a2e71
Type	Cryptographic Vulnera	**Status**	Fixed
File(s)	`circuits/templates/dataEscrowElGamalEncryption.circom`		
Location(s)	DataEscrowElGamalEncryption		
Confirmed Fix At	fdabb7f		

Standard HMAC as described for instance in RFC 2104 constructs from a derived key two new keys (inner and outer key) by (possibly) hashing, padding and taking a bitwise XOR with two fixed bitstrings `ipad` and `opad` for the inner and outer key respectively. The implementation follows this approach, while replacing standard SHA hash functions with Poseidon. There are however several issues:

- As the inner and outer keys are related, stronger assumptions are needed to ensure security. In the case of the standard HMAC, security is ensured by the fact that the hash function in that case is based on a Davies-Meyer compression function. This does not hold once the hash function is replaced by a Poseidon hash function.
- The standard HMAC is bitstring based and hence bitwise XOR operations are meaningful. When the hash function is replaced by a Poseidon hash, operations will be performed on field elements. Bitwise operations are more cumbersome and not always meaningful. In particular, during the XOR operation with `opad` to obtain the outer key can lead to results exceeding the field size.
- The `XOR()` template from the Circomlib library is used to obtain the XOR of field elements with the masking constants `ipad` and `opad` , all representing bitstrings in the binary basis. The circomlib `XOR()` template makes the assumption that the input signals are bits, rather than bitstrings represented by field elements. Hence this is an inadequate use of the circomlib library.

```
component innerXor = XOR();
innerXor.a <== kMac.out;
innerXor.b <== ipad;
```

Snippet 4.24: Snippet from `DataEscrowElGamalEncryption`

- The deviation from the standard (replacing SHA by Poseidon, having no padding) gives a scheme whose security has not been analyzed, which poses a potential risk.

Impact HMAC was devised to be used with a classical hash function operating on bitstrings, whereas Poseidon was intended to be used within a (duplex) sponge construction and operates on finite field elements. Operations like XOR are not necessarily meaningful. Deviations from the standard forms a potential security risk. Moreover, improper use of templates from the circomlib library can lead to unexpected behavior and lead to a vulnerability.

Recommendation Using a more basic approach like an Envelope MAC will be better than using HMAC with deviations.

Developer Response The developers switched to Envelope MAC scheme.

4.1.29 V-PAN-VUL-029: Insufficient source address check in UniswapV3Handler

Severity	Low	**Commit**	a16a43e
Type	Access Control	**Status**	Fixed
File(s)	`UniswapV3Handler.sol`		
Location(s)	uniswapV3SwapCallback()		
Confirmed Fix At	8c8c50d		

The `uniswapV3Callback` function is intended to allow the UniswapV3 Pool call the user contract before finalizing the swap. This function should only be called on behalf of the UniswapV3 pool that is performing the swap operation, since the callback is responsible for sending the proper amount of tokens into the pool. However, there is a missing access control check, which means anyone can call this function.

```
function uniswapV3SwapCallback(
   int256 amount0Delta,
   int256 amount1Delta,
   bytes calldata data) external
{
   (address pool, address token0, address token1) =
         abi.decode(data, (address, address, address));
   require(msg.sender == pool, "Invalid sender");
   if (amount0Delta > 0)
      token0.safeTransfer(pool, uint256(amount0Delta));
   if (amount1Delta > 0)
      token1.safeTransfer(pool, uint256(amount1Delta));
 }
```

Snippet 4.25: Snippet from `uniswapV3Callback`

Impact Since the `uniswapV3Callback` function transfers an arbitrary amount of tokens to a specified address, it can be exploited by an attacker to drain the FeeMaster contract's balance.

The issue has been reevaluated to be of Low severity after it has been clarified that FeeMaster is not expected to store any ERC20 tokens.

Recommendation It is recommended to implement an access control check for the caller address that will ensure it is an authentic UniswapV3 pool.

Developer Response The developers fixed the issue at commit 8c8c50d.

4.1.30 V-PAN-VUL-030: Zones registry accepts root as an argument and does not verify against the root in storage

Severity	Low	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	ZZonesRegistry, ZNetworksRegistry		
Location(s)	addZone() and addNetwork()		
Confirmed Fix At	547a4d5		

The function `addZone` is used to add a new zone to the zZones Merkle tree. Upon addition, it is first verified that the current leaf is a part of the merkle tree and the path to that leaf index is correct. Afterwards, the new leaf with the updated zone is inserted at that leaf index and the new merkle root is computed and emitted.

The `addZone` function takes the `curRoot` as an argument as opposed to reading the root from the `_currentRoot` storage variable. This `curRoot` variable is also not validated to be the actual stored `_currentRoot`. See snippet below for context.

```
function addZone(
  bytes32 curRoot,
  bytes32 curLeaf,
  bytes32 newLeaf,
  uint256 leafIndex,
  bytes32[] calldata proofSiblings
) external onlyOwner {
  bytes32 zZonesTreeRoot = update(
      curRoot,
      curLeaf,
      newLeaf,
      leafIndex,
      proofSiblings
  );

  _updateStaticRoot(zZonesTreeRoot, ZZONE_STATIC_LEAF_INDEX);
  _currentRoot = zZonesTreeRoot;
  emit ZZonesTreeUpdated(zZonesTreeRoot);
}
```

Snippet 4.26: Snippet from `addZone()`

Taking the current root of the tree as an argument allows the caller to update the entire tree instead of a leaf in the tree. Although this action is a privileged action, the tree could be replaced by mistake, which could result in the registered zones being removed.

The same issue is present in the `addNetwork` function as well.

Impact The owner can set the root of the concerned Merkle trees to any value. This may be done intentionally or unintentionally.

Recommendation Use the storage variable `_currentRoot` to get the current Merkle tree root instead of passing `curRoot` as an argument.

Developer Response The developers fixed the issue at commit 547a4d5.

4.1.31 V-PAN-VUL-031: kytSignedMessageChargedAmountZkp is conditionally constrained

Severity	Low	Commit	a16a43e
Type	Data Validation	Status	Fixed
File(s)	`trustProvidersKyt.circom`		
Location(s)	Template TrustProvidersDepositWithdrawKyt()		
Confirmed Fix At	5f2ff4c		

In the template `TrustProvidersDepositWithdrawKyt`, the `kytSignedMessageHashInternal` hash is constructed with the involved fields. The signal `kytSignedMessageChargedAmountZkp` is conditionally included in the `kytSignedMessageHashInternal` construction if `PureFI` has added the `KYT` signed message hash to their `KYT` process. Until then it is not included or anchored within the `KYT` signed message hash. See snippet below for the implementation.

```
component kytSignedMessageHashInternal;
var ENABLE_WHEN_IMPLEMENTED = 0;
if ( ENABLE_WHEN_IMPLEMENTED ) {
  kytSignedMessageHashInternal = Poseidon(10);

  kytSignedMessageHashInternal.inputs[0] <== kytSignedMessagePackageType;
  kytSignedMessageHashInternal.inputs[1] <== kytSignedMessageTimestamp;
  kytSignedMessageHashInternal.inputs[2] <== kytSignedMessageSender;
  kytSignedMessageHashInternal.inputs[3] <== kytSignedMessageReceiver;
  kytSignedMessageHashInternal.inputs[4] <== kytSignedMessageToken;
  kytSignedMessageHashInternal.inputs[5] <== kytSignedMessageSessionId;
  kytSignedMessageHashInternal.inputs[6] <== kytSignedMessageRuleId;
  kytSignedMessageHashInternal.inputs[7] <== kytSignedMessageAmount;
  kytSignedMessageHashInternal.inputs[8] <== kytSignedMessageSigner;
  kytSignedMessageHashInternal.inputs[9] <== kytSignedMessageChargedAmountZkp;
} else {
  kytSignedMessageHashInternal = Poseidon(9);

  kytSignedMessageHashInternal.inputs[0] <== kytSignedMessagePackageType;
  kytSignedMessageHashInternal.inputs[1] <== kytSignedMessageTimestamp;
  kytSignedMessageHashInternal.inputs[2] <== kytSignedMessageSender;
  kytSignedMessageHashInternal.inputs[3] <== kytSignedMessageReceiver;
  kytSignedMessageHashInternal.inputs[4] <== kytSignedMessageToken;
  kytSignedMessageHashInternal.inputs[5] <== kytSignedMessageSessionId;
  kytSignedMessageHashInternal.inputs[6] <== kytSignedMessageRuleId;
  kytSignedMessageHashInternal.inputs[7] <== kytSignedMessageAmount;
  kytSignedMessageHashInternal.inputs[8] <== kytSignedMessageSigner;

}
```

Snippet 4.27: Snippet from `TrustProvidersDepositWithdrawKyt()`

This can be problematic because the amount `kytSignedMessageChargedAmountZkp` is involved in the `BalanceChecker` which operates under the assumption that this amount is included within the hash commitment `kytSignedMessageHashInternal` and cannot be manipulated. But that is not always the case as mentioned above.

One of the constraints the balance checker circuit applies is to ensure that the inflow and

outflow of ZKP tokens across the input and output UTXO's remains the same. Of the signals involved in this equation zAccountUtxoInZkpAmount and zAccountUtxoOutZkpAmount are included in hash commitments and cannot be tampered. But, addedScaledAmountZk and chargedScaledAmountZkp are not included in any other constraints except this one and can be manipulated as long as the main balance constraint is satisfied.

The kytChargedScaledAmountZkp corresponds to the fees charged by the KYT provider and is included in a hash commitment conditionally. It can be manipulated until Purefi adds the field into the KYT signed message commitment. This allows the main balance equation to be manipulated to show that a very large amount of ZKP tokens was added through addedScaledAmountZkp. kytChargedScaledAmountZkp can then be increased on the R.H.S to match this increase on the L.H.S. The above still adheres to the applied constraints and allows a KYT provider to collude with a user and drain the protocol of ZKP tokens.

```
signal kytChargedScaledAmountZkp <== UintTag(ACTIVE,99)(
    kytDepositScaledChargedAmountZkp + kytWithdrawScaledChargedAmountZkp +
    kytInternalScaledChargedAmountZkp); // 96 + 3

// depositScaledAmount is RCed, together with zAccountUtxoInZkpAmount &
    addedScaledAmountZkp, 64 + 64 + 96
signal totalBalanceIn <== depositScaledAmount + totalUtxoInAmount + isZkpToken * (
    zAccountUtxoInZkpAmount + addedScaledAmountZkp );

signal totalBalanceOut <== withdrawScaledAmount + totalUtxoOutAmount + isZkpToken * (
     zAccountUtxoOutZkpAmount + chargedScaledAmountZkp + kytChargedScaledAmountZkp );

totalBalanceIn === totalBalanceOut;

////////////////////////////////////////
// [5] - Verify zAccountUtxoOutZkpAmount //
////////////////////////////////////////
component zAccountUtxoOutZkpAmountChecker = ForceEqualIfEnabled();
// disabled if zZKP token since if zZKP the balance is checked via totalBalance IN/
    OUT
zAccountUtxoOutZkpAmountChecker.enabled <== 1 - isZkpToken;
zAccountUtxoOutZkpAmountChecker.in[0] <== zAccountUtxoInZkpAmount +
    addedScaledAmountZkp;
zAccountUtxoOutZkpAmountChecker.in[1] <== zAccountUtxoOutZkpAmount +
    chargedScaledAmountZkp + kytChargedScaledAmountZkp;
```

Snippet 4.28: Snippet from template BalanceChecker()

Impact If the KYT signed message hash does not include the kytSignedMessageChargedAmountZkp, then a KYT provider can drain the protocol of ZKP tokens.

Recommendation Until the KYT charged amount is not included in the KYT signed message hash, it should not be used in other constraints as it can be manipulated.

Developer Response The developers fixed the issue at commit 5f2ff4c.

4.1.32 V-PAN-VUL-032: Unconditional reward during the debt rebalance operation

Severity	Low	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`FeeMaster.sol`		
Location(s)	rebalanceDebt		
Confirmed Fix At	137e8f3		

Panther Protocol rewards users for performing various protocol-wide activities. One such activity is debt rebalancing, which aims to replenish the native ETH reserves by converting all the accumulated fees collected in various tokens into ETH through a decentralized exchange. If the operation is successful, the user will receive a portion of PRP tokens, which can later be exchanged for ZKP tokens.

```
function rebalanceDebt(bytes32 secretHash, address sellToken) external {
   // ... skipped
   _grantPrpRewardsToUser(secretHash, GT_FEE_EXCHANGE);
}
```

Snippet 4.29: Snippet from `FeeMaster.rebalanceDebt()`

The issue here is that the reward is given without any conditions.

Impact Depending on the PRP reward amount, this may open up a possibility of unreasonable profiting by calling this function on each tiny increase in `sellToken` fee debt: in this case, the value of the reward may be greater than the value of the operation.

Recommendation It is recommended to make the reward contingent on certain conditions, perhaps by setting a minimum reserve increase threshold that a user must reach in order to be eligible for the PRP reward.

Developer Response The developers fixed the issue at commit 137e8f3.

4.1.33 V-PAN-VUL-033: Malleable ECDSA implementation

Severity	Low	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`EIP712SignatureVerifier.sol`		
Location(s)	function EIP712SignatureVerifier.recover		
Confirmed Fix At	cb5928f		

The Panther protocol implements its own verifier for ECDSA signatures (see attached snippet). It is well known that ECDSA signatures are malleable and special care must be taken when implementing the verification logic.

```
function recover(
    bytes32 hash,
    uint8 v,
    bytes32 r,
    bytes32 s
) internal pure returns (address signer) {
    signer = ecrecover(hash, v, r, s);
    require(signer != address(0), "ECDSA invalid signature");
}
```

Snippet 4.30: Snippet from `EIP712SignatureVerifier.recover()`

Impact Currently, the malleability of signatures does not pose any risk to the protocol since all operations benefit the signer. However, this might change in the future.

Recommendation Consider using a standard and well-tested implementation that performs all appropriate checks (e.g., the one from OpenZeppelin).

Developer Response The developers fixed the issue at commit cb5928f.

4.1.34 V-PAN-VUL-034: Sub-contracts of BinaryUpdatableTree do not validate proof lengths

Severity	Low	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	Several files		
Location(s)			
Confirmed Fix At	c301871		

Several sub-contracts of `BinaryUpdatableTree` call `BinaryUpdatableTree.update` without validating the length of input proof equals the height of the tree. This is problematic because `BinaryUpdatableTree.update` can succeed by providing an empty proof with an arbitrary new root. The functions that miss this check are the following:

1. `ZNetworksRegistry.addNetwork`
2. `ZZonesRegistry.addZone`
3. `BlacklistedZAccountsIdsRegistry.addZAccountIdToBlacklist`
4. `BlacklistedZAccountsIdsRegistry.removeZAccountIdFromBlacklist`

Impact All the above are privileged actions, so they would require a malicious owner to exploit the lack of data validation. Effectively, malicious owners can call any of the above functions which are meant to update a single leaf of a tree and update all leaves simultaneously.

Recommendation Validate that the length of the proof matches the height of the tree in all functions mentioned above.

Developer Response The developers fixed the issue at commit c301871.

4.1.35 V-PAN-VUL-035: Storage variable forestRoot is not updated due to shadowing

Severity	Low	**Commit**	a7b76bc
Type	Logic Error	**Status**	Fixed
File(s)	`ForestTree.sol`		
Location(s)	addUtxosToBusQueueAndTaxiTree		
Confirmed Fix At	49ec81d		

Within the ForestTree contract, if the BusTree or TaxiTree roots are updated, the ForestTree root should also be updated. External systems can check the current state of the ForestTree root by using the `ForestTree.getRoots()` function.

At one of the update locations, the update operation will not be carried out because the left side of the assignment statement is shadowed by a function parameter variable that has the same name as the variable being updated.

```
function addUtxosToBusQueueAndTaxiTree(
        bytes32[] memory utxos,
        uint8 numTaxiUtxos,
        uint256 cachedForestRootIndex,
        bytes32 **forestRoot**,
        bytes32 staticRoot,
        uint96 reward
    )
{
    // ...skipped...
    **forestRoot** = _cacheNewForestRoot(
            taxiTreeNewRoot,
            TAXI_TREE_FOREST_LEAF_INDEX
    );
}
```

Snippet 4.31: Snippet from `example()`

Impact The `forestRoot` storage variable value may become outdated, which could negatively affect systems that depend on it.

Recommendation It is recommended to rename the input parameter `forestRoot` to rule out the variable name shadowing defect.

Developer Response The developers fixed the issue at commit 49ec81d.

4.1.36 V-PAN-VUL-036: Incorrect loop upper bound in zAccountRenewalV1 circuit

Severity	Low	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`zAccountRenewalV1.circom, ammV1.circom and zAccountRegistrationV1.circom`		
Location(s)	ZAccountRenewalV1		
Confirmed Fix At	f9d6bf0		

One of the steps in the `ZAccountRenewalV1` process during the account renewal is to ensure that the account being renewed is not considered as blacklisted. To do this, the circuit performs the following checks:

```
// [8] - Verify zAccoutId exclusion proof
component zAccountBlackListInlcusionProver = ZAccountBlackListLeafInclusionProver(
    ZAccountBlackListMerkleTreeDepth);
zAccountBlackListInlcusionProver.zAccountId <== zAccountUtxoInId;
zAccountBlackListInlcusionProver.leaf <== zAccountBlackListLeaf;
zAccountBlackListInlcusionProver.merkleRoot <== zAccountBlackListMerkleRoot;
for (var j = 0; j < **ZZoneMerkleTreeDepth**; j++) {
  zAccountBlackListInlcusionProver.pathElements[j] <== zAccountBlackListPathElements[
    j];
}
```

Snippet 4.32: Snippet from `ZAccountRenewalV1()`

The issue here is that the loop has an incorrect upper bound: instead of having it as `ZAccountBlackListMerkleTreeDepth` another value `ZZoneMerkleTreeDepth` is specified.

Impact This error does not currently pose a direct security risk, as both template parameters happen to have the same concrete value. However, given that both are template parameters and can change, the circuit could become underconstrained, potentially leading to security issues such as bypassing blacklist checks.

Recommendation The loop upper bound should be constrained with the value `ZAccountBlackListMerkleTreeDepth`.

Developer Response The developers fixed the issue at commit f9d6bf0.

4.1.37 V-PAN-VUL-037: Potential silent overflow in ZAssetEncodingUtils

Severity	Low	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`ZAssetEncodingUtils.sol`		
Location(s)	encodeTokenIdRangeSizeWithScale		
Confirmed Fix At	ca3385d		

The library function `ZAssetEncodingUtils.encodeTokenIdRangeSizeWithScale()` does not validate the `scaleFactor` value.

```
function encodeTokenIdRangeSizeWithScale(
  uint32 tokenIdRangeSize,
  uint8 scaleFactor
) internal pure returns (uint96) {
  uint64 scale = uint64((10 ** scaleFactor));
  return (uint96(tokenIdRangeSize) << 64) | scale;
}
```

Snippet 4.33: Snippet from `ZAssetEncodingUtils.sol`

Impact Since this function is used in the `ZAssetRegistryV1` contract during the ZAsset inclusion process, if the caller specifies a scale factor that is greater than 19, the expression of `scale` will silently overflow, and the corresponding ZAsset leaf will contain an incorrect scale value. This will cause the protocol to function incorrectly with this asset type.

Recommendation Limit the `scaleFactor` value to be less or equal to 19.

Developer Response The developers fixed the issue at commit ca3385d.

4.1.38 V-PAN-VUL-038: Total released ZKP tokens may be accounted incorrectly

Severity	Low	Commit	a16a43e
Type	Logic Error	Status	Fixed
File(s)	ZkpReserveController.sol		
Location(s)	releaseZkps()		
Confirmed Fix At	1f70aeb		

The `ZkpReserveController` contract is designed to refill the `PRP/ZKP` pool with `ZKP` tokens. The contract releases some `ZKP` tokens linearly to the pool based on the number of blocks elapsed.

If the contract runs out of balance then it can be funded with `ZKP` tokens to continue the release process. If more `ZKP` tokens are sent to the contract to increase its balance, then the new balance will continue to be released linearly. See snippet below for the implementation.

```
function releaseZkps(bytes32 saltHash) external {
  uint64 contractBalance = uint64(ZKP_TOKEN.safeBalanceOf(address(this)));
  require(contractBalance > 0, "no zkp is available");

  uint64 _scReleasable = _scReleasableAmount();
  uint256 _releasable = _scReleasable.scaleUpBy1e12();

  if (_releasable > contractBalance) {
      _releasable = contractBalance;
  }

  ZKP_TOKEN.safeTransfer(PANTHER_POOL, _releasable);
  IPrpConversion(PANTHER_POOL).increaseZkpReserve();

  scTotalReleased += _scReleasable;
```

Snippet 4.34: Snippet from `releaseZkps()`

In the above implementation, `scTotalReleased` is updated by the `_scReleasable` amount, but it does not account for the case where the contract does not have enough balance to release the specified amount of tokens. When the amount of releasable tokens is larger than the contract balance, then `_releasable` is set to the contracts balance. However, in the given scenario `scTotalReleased` is still updated by `_scReleasable`, whereas it should be updated by the contracts balance.

If a new cycle of token release begins with an incorrect `scTotalReleased` that is greater than the actual tokens released, then it will release fewer tokens moving forward.

Instead, to do proper accounting, the `scTotalReleased` should be updated as follows: `scTotalReleased += _releasable.scaleDownBy1e12()`.

Impact If the contract balance is emptied and the contract's balance is refilled with `ZKP` tokens for release, then the `scTotalReleased` will be more than what has actually been released from the contract. The future release of tokens from that point will be accounted incorrectly.

Recommendation When the contract balance is less than the releasable amount of tokens, update `scTotalReleased` with the scaled down value of `_releasable`.

Developer Response The developers fixed the issue at commit 1f70aeb.

4.1.39 V-PAN-VUL-039: Potential unexpected Taxi Root value reset

Severity	Low	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	TaxiTree.sol		
Location(s)	_addUtxos()		
Confirmed Fix At	372252a		

The function `TaxiTree._addUtxos()` does not check for the passed `utxos` array length.

```
function _addUtxos(
        bytes32[] memory utxos
    ) internal returns (bytes32 newRoot) {
   newRoot;
   for (uint256 i = 0; i < utxos.length; ) {
      newRoot = _addUtxoToTaxiTree(utxos[i]);
      unchecked {
        ++i;
      }
   }
   _updateTaxiTreeRoot(newRoot, utxos.length);
}
```

Snippet 4.35: Snippet from `_addUtxos()`

Impact If the passed `utxos` array is empty, the function will overwrite the current Taxi Tree root with a zero value, which would break the protocol.

Recommendation It is recommended to implement the array length check before processing.

Developer Response The developers fixed the issue at commit 372252a.

4.1.40 V-PAN-VUL-040: Incorrect assertions in zAccountRegistrationV1 circuit

Severity	Low	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`zAccountRegistrationV1.circom`		
Location(s)	template ZAccountRegistrationV1		
Confirmed Fix At	9a96e30		

The circuit `ZAccountRegistrationV1` implements incorrect assertions on two signals:

- Signal `zAccountPrpAmount` is tagged as the `uint196` , but it is asserted to be less than 2**64
- Signal `zAccountZkpAmount` is tagged as the `uint64` , but it is asserted to be less than 2**252

```
1 // [3] - verify ZKP & PRP balance
2 // zkp amount, range-checked since zkpAmount is public signal, and zAssetScale
      controlled by smart-contracts
3 assert(0 <= zAccountZkpAmount < 2**252);
4 // prp amount decided by the protocol on smart contract level - range is checked
      since it is public signal
5 assert(0 <= zAccountPrpAmount < 2**64);
```

Snippet 4.36: Snippet from `ZAccountRegistrationV1()`

Impact This error allows the creation of proofs that should not be created in the first place, or prevents the creation of valid proofs on the client side.

Recommendation The bounds in both assertions must correctly match the expected range of values.

Developer Response The developers fixed the issue at commit 9a96e30.

4.1.41 V-PAN-VUL-041: _accountDebtForPaymaster() always returns zero

Severity	Low	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	FeeAccountant		
Location(s)	_accountDebtForPaymaster()		
Confirmed Fix At	7a8a820		

The function `_accountDebtForPaymaster` calculates and accounts the debt for the paymaster in terms of `ZKP` tokens and native tokens. It returns `paymasterFeeInNative` which is the value of the paymaster fee in terms of the native token. See snippet below for the implementation.

The `paymasterFeeInNative` is initialized by default to `0` and only used in the first conditional when the paymaster compensation in `ZKP` is `0`. Therefore it will always be `0`. It should be updated to the value of `paymasterDebtInNative` if the paymaster debt in native tokens is greater than `0`.

```
function _accountDebtForPaymaster(
  uint256 paymasterCompensationInZkp
) internal returns (uint256 paymasterFeeInNative) {
  if (paymasterCompensationInZkp == 0) return paymasterFeeInNative;

  (
      uint256 paymasterDebtInZkp,
      uint256 paymasterDebtInNative
  ) = _tryInternalZkpToNativeConversion(paymasterCompensationInZkp);

  if (paymasterDebtInZkp > 0) {
      _updateDebts(PAYMASTER, ZKP_TOKEN, int256(paymasterDebtInZkp));
  }

  if (paymasterDebtInNative > 0) {
      _updateDebts(
          PAYMASTER,
          NATIVE_TOKEN,
          int256(paymasterDebtInNative)
      );
  }
}
```

Snippet 4.37: Snippet from `_accountDebtForPaymaster()`

Impact The return value of `_accountDebtForPaymaster()` will always be `0`. This value is not used anywhere else in the contracts but is emitted through events. External actors monitoring the emitted events will not be able to correctly track the paymaster fee compensation.

Recommendation If `paymasterDebtInNative` is greater than `0`, then update `paymasterFeeInNative` to `paymasterDebtInNative`.

Developer Response The developers fixed the issue at commit 7a8a820.

4.1.42 V-PAN-VUL-042: Signal nInputs inside ZeroPaddedInputChecker is under-constrained

Severity	Low	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`zeroPaddedInputChecker.circom`		
Location(s)	Template ZeroPaddedInputChecker		
Confirmed Fix At	ce15587		

The template `zeroPaddedInputChecker` enforces that all `inputs[i]` after a certain index are zero values. But, if `nInputs` is larger than `max_inputs`, then the for loop iterating over `max_inputs` will always pass.

The current constraints attempt to enforce that `nInputs` is less than or equal to `max_nInputs` but `nInputs` is not validated to fit within `252` bits and the current constraint can be circumvented by using a value of `nInputs` which is larger than that.

```
1 assert(nInputs<=max_nInputs);
2 component isNInputsLessOrEqualToMax_nInputs;
3 isNInputsLessOrEqualToMax_nInputs = LessEqThan(252);
4 isNInputsLessOrEqualToMax_nInputs.in[0] <== nInputs;
5 isNInputsLessOrEqualToMax_nInputs.in[1] <== max_nInputs;
6 isNInputsLessOrEqualToMax_nInputs.out === 1;
```

Snippet 4.38: Snippet from template `ZeroPaddedInputChecker()`

Impact Because `nInputs` is not correctly validated to be less than `max_nInputs`, the less than comparison can be circumvented allowing an array of inputs which are not enforced to be zero padded as intended. This also violates a necessary assumption when creating the degenerate binary Merkle tree from the inputs later.

Recommendation Use `Num2Bits` to ensure that `nInputs` is constrained to be within `252` bits.

Developer Response The developers fixed the issue at commit ce15587.

4.1.43 V-PAN-VUL-043: Last element of pathIndices is unconstrained

Severity	Warning	Commit	a16a43e
Type	Data Validation	Status	Fixed
File(s)	merkleTreeInclusionProof.circom		
Location(s)	Template MerkleTreeInclusionProof		
Confirmed Fix At	4625fdc		

The template `MerkleTreeInclusionProof` computes the Merkle inclusion proof for a leaf in a modified Merkle binary tree with 3 child nodes at the leaf level. The template takes as argument `n_levels` which is the depth of the Merkle tree and takes as input the `leaf` along with the `pathIndices` and `pathElements` which are then used to reconstruct the Merkle tree root. Because of the modification, both `pathIndices` and `pathElements` contain `n_levels+1` elements in total. The template definition can be seen below.

Before reconstruction of the tree root, the template applies constraints on the elements of `pathIndices` to ensure they are binary signals. But while looping over the signals, only `n_levels` signals are constrained. As established above, `pathIndices` has `n_levels + 1` elements so the last index of the path remains unconstrained. See snippet below for context.

```
template MerkleTreeInclusionProof(n_levels) {
  signal input          leaf;
  signal input {binary} pathIndices[n_levels+1];
  signal input          pathElements[n_levels+1]; // extra slot for third leave

  // first, hash 3 leaves ...
  hashers[0] = Poseidon(3);

  // enforcing that 2 bits of the leaf level index can't be 11
  0 === pathIndices[0]*pathIndices[1];

  hashers[0].inputs[0] <== leaf + (pathIndices[0]+pathIndices[1])*(pathElements[0] -
    leaf);
  temp <== pathElements[0] + pathIndices[0]*(leaf - pathElements[0]);
  hashers[0].inputs[1] <== temp + pathIndices[1]*(pathElements[1] - pathElements[0]);
  hashers[0].inputs[2] <== pathElements[1] + pathIndices[1]*(leaf -pathElements[1]);

  for (var i = 0; i < n_levels; i++) {
      // enforce binary index
      pathIndices[i] - pathIndices[i] * pathIndices[i] === 0;
  }

  // ... then iterate through levels above leaves

  // 00000
  for (var i = 1; i < n_levels; i++) {
      // (outL,outR) = sel==0 ? (L,R) : (R,L)
      switchers[i-1] = Switcher();
      switchers[i-1].L <== hashers[i-1].out;
      switchers[i-1].R <== pathElements[i+1];
      switchers[i-1].sel <== pathIndices[i+1];
      hashers[i] = Poseidon(2);
```

```
32        hashers[i].inputs[0] <== switchers[i-1].outL;
33        hashers[i].inputs[1] <== switchers[i-1].outR;
34    }
35
36    root <== hashers[n_levels-1].out;
37  }
```

Snippet 4.39: Snippet from template `MerkleTreeInclusionProof()`

It is important to ensure that the `pathIndices` are binary because they are provided to the circomlib template `Switcher` which assumes that the input signal `sel` is binary. In this case, because the last path index is not constrained to be binary it can allow manipulation of the switcher output, which can be used by an attacker to forge a merkle inclusion proof. See snippet below for the `Switcher` implementation.

```
1  /*
2      Assume sel is binary.
3
4      If sel == 0 then outL = L and outR=R
5      If sel == 1 then outL = R and outR=L
6
7   */
8
9  template Switcher() {
10     signal input sel;
11     signal input L;
12     signal input R;
13     signal output outL;
14     signal output outR;
15
16     signal aux;
17
18     aux <== (R-L)*sel;    // We create aux in order to have only one multiplication
19     outL <==  aux + L;
20     outR <== -aux + R;
21 }
```

Snippet 4.40: Snippet from template `Switcher()`

At the moment the template `MerkleTreeInclusionProof` is only used inside `UtxoNoteInclusionProver` which is not currently used in any of the top level circuits. Therefore, the current severity of this issue is marked as a warning. However, if the template is used in its current state, it may introduce a critical vulnerability into the protocol.

Impact If the template `MerkleTreeInclusionProof` is used, it may allow an attacker to forge a Merkle inclusion proof for an element not contained within the Merkle tree.

Recommendation Loop over `n_levels + 1` instead of `n_levels` , when enforcing that the `pathIndices` are binary.

Developer Response The developers fixed the issue at commit 4625fdc.

4.1.44 V-PAN-VUL-044: The constraint on offset is not verified

Severity	Warning	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`trustProvidersMerkleTreeLeafIDAndRuleInclusionProver.circom`		
Location(s)	Template TrustProvidersMerkleTreeLeafIDAndRuleInclusionProver		
Confirmed Fix At	cd39575		

The template `TrustProvidersMerkleTreeLeafIDAndRuleInclusionProver` takes as input a `leafId`, `offset`, `rule` and `leafIDsAndRulesList` and enforces that the `leafId` and `rule` obtained at that `offset` in `leafIDsAndRulesList` are indeed the same as the inputs. The input signal `leafIDsAndRulesList` holds 10 elements of 24 bits each. The first 8 bits of an element are the rule and the next 16 bits are the `leafId`.

In this template, the input signal `offset` is assumed to be less than `10`. The circomlib `LessThan` template is used to apply this constraint as can be seen in the snippet below.

```
assert(offset < 10);

component offset_lessThan_10 = LessThan(4);
offset_lessThan_10.in[0] <== offset;
offset_lessThan_10.in[1] <== 10;

component n2b_leafIDsAndRulesList = Num2Bits(10 * 24);
n2b_leafIDsAndRulesList.in <== leafIDsAndRulesList;

component selector[10];
component b2n_leafIdAndRulesList[10];
component multiSum_leafIDAndRule = MultiSum(10);

for(var i = 0; i < 10; i++) {
    selector[i] = IsEqual();
    selector[i].in[0] <== i;
    selector[i].in[1] <== offset;

    b2n_leafIdAndRulesList[i] = Bits2Num(24);

    for(var j = 0; j < 24; j++) {
        b2n_leafIdAndRulesList[i].in[j] <== n2b_leafIDsAndRulesList.out[24 * i + j];
    }
    // selector is one only for specific place (0..9)
    multiSum_leafIDAndRule.in[i] <== selector[i].out * b2n_leafIdAndRulesList[i].out;
    }
// since only one of 0..9 is not zero -> the rolling sum works as mutiplexer, often
    abbreviated as "MUX,"
component n2b = Num2Bits(24);
n2b.in <== multiSum_leafIDAndRule.out;

component b2nRule = Bits2Num(8);
for (var i = 0; i < 8; i++) {
  b2nRule.in[i] <== n2b.out[i];
}

```

```
component isEqualRule = ForceEqualIfEnabled();
isEqualRule.in[0] <== rule;
isEqualRule.in[1] <== b2nRule.out;
isEqualRule.enabled <== enabled;

component b2nLeafId = Bits2Num(16);
for (var i = 8; i < 24; i++) {
  b2nLeafId.in[i-8] <== n2b.out[i];
}

component isEqualLeafId = ForceEqualIfEnabled();
isEqualLeafId.in[0] <== leafId;
isEqualLeafId.in[1] <== b2nLeafId.out;
isEqualLeafId.enabled <== enabled;
```

Snippet 4.41: Snippet from template `TrustProvidersMerkleTreeLeafIDAndRuleInclusionProver()`

But the output of `offset_lessThan_10` is not constrained to be equal to 1. This means that `offset` can take values in range `[0,15]` instead of `[0,10]` which is the intended range. At the moment this does not lead into an issue because of reasons explained below.

If the `offset` has a value larger than `10`, then the loop iterates over the entire `leafIDsAndRulesList` without matching any offset index and the value of `multiSum_leafIDAndRule` will be `0`. The `b2nRule` and `b2nLeafId` which are extracted from it also end up as `0`. Hence the equality comparison with the input signals `leafId` and `rule` fails, as there should be no valid `leafId` or `rule` with that value.

But, if in the future the `offset` is used in a different manner while still being constrained incorrectly, then it can introduce issues into the protocol.

Impact The constraint on `offset` is not applied correctly. It does not lead to an issue at the moment. But if any changes are made to the circuits and `offset` is used under the assumption that it is constrained correctly, it can introduce bugs into the protocol.

Recommendation Constrain the output of `offset_lessThan_10` to be equal to 1.

Developer Response The developers fixed the issue at commit cd39575.

4.1.45 V-PAN-VUL-045: Several Inconsistencies within zoneIdInclusionProver

Severity	Warning	**Commit**	a16a43e
Type	Usability Issue	**Status**	Fixed
File(s)	`zoneIdInclusionProver.circom`		
Location(s)	Template ZoneIdInclusionProver		
Confirmed Fix At	6c69604		

The template `zoneIdInclusionProver` has several inconsistencies between its implementation and specifications. They are listed below. The template code can be seen in the snippet below

- The input signal `zoneIds` is mentioned to fit in `256` bits. The comment should instead mention that it fits in `240` bits. This is evident from the implementation, where `15` zone ids each of `16` bits are derived from it.
- Component `n2b_zoneIds` is represented with `Num2Bits(254)`. The use of `Num2Bits(254)` to describe a component is dangerous as described in this issue. Moreover as per the requirement in this template `n2b_zoneIds` should be represented using `Num2Bits(240)`.
- The circuit defines `16` elements for the component array `b2n_zoneIds`, but only `15` of them are iterated over in the following loop. The last element is unconstrained and unused. As per the requirements of this template only `15` elements need to be defined.

```
template ZoneIdInclusionProver(){
  signal input enabled;
  signal input {uint16} zoneId;  // 16 bit
  signal input zoneIds;          // 256 bit
  signal input {uint4} offset;   // 4 bit

  assert(offset < 16);
  component n2b_zoneIds = Num2Bits(254);
  n2b_zoneIds.in <== zoneIds;

  component b2n_zoneIds[16];

  for(var i = 0, ii = 0; i < 15*16; i += 16) {
      b2n_zoneIds[ii] = Bits2Num(16);
      for ( var j = 0; j < 16; j++) {
          b2n_zoneIds[ii].in[j] <== n2b_zoneIds.out[i + j];
      }
      ii++;
  }

  component forceIsEqual[15];
  component is_equal[15];
  for(var i = 0; i < 15; i++) {
      is_equal[i] = IsEqual();
      is_equal[i].in[0] <== i;
      is_equal[i].in[1] <== offset;

      forceIsEqual[i] = ForceEqualIfEnabled();
      forceIsEqual[i].in[0] <== zoneId;
      forceIsEqual[i].in[1] <== b2n_zoneIds[i].out;
      // i == offset this is the exact portion of bits to check
```

```
32        forceIsEqual[i].enabled <== enabled * is_equal[i].out;
33    }
34 }
```

Snippet 4.42: Snippet from template `ZoneIdInclusionProver`

The inconsistencies mentioned above do not lead to issues, but they adversely affect readability and maintainability. They can also cause confusion for future developers which can lead to issues if the template is used incorrectly.

Impact The inconsistencies mentioned above negatively affect comprehension and readability of the code. This can cause issues in maintainability and usability of this template.

Recommendation Resolve the mentioned inconsistencies so that they match with the intended specifications.

Developer Response The developers fixed the issue at commit 6c69604.

4.1.46 V-PAN-VUL-046: NonZeroUintTag implemented incorrectly

Severity	Warning	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`utils.circom`		
Location(s)	Template NonZeroUintTag		
Confirmed Fix At	ff18cf5		

The template `NonZeroUintTag` is used to ensure that the input signal is greater than `0` if the template is active. It makes use of circomlib's `GreaterThan` template to perform this check. See snippet below for the implementation.

However the order of the inputs passed to `GreaterThan` is incorrect. As it is currently implemented, the template enforces that `in < 0`. Whereas the intention is to enforce that `in > 0`.

```
template NonZeroUintTag(isActive, nBits) {
    signal input in;
    signal output {non_zero_uint} out;

    assert(nBits < 252);

    component n2b;
    if ( isActive ) {
        n2b = GreaterThan(nBits);
        n2b.in[0] <== 0;
        n2b.in[1] <== in;
        n2b.out === 1;
    }
    out <== in;
}
```

Snippet 4.43: Snippet from template `NonZeroUintTag()`

This template is applied to some signals but it is not activated. Therefore at the moment it does not lead to a major issue. But it may be used in its current form in the future which can cause serious issues.

Additionally the `LessThan` template assumes that its input signal fits within `nBits` i.e the argument the template is instantiated with. If this is not followed it can render the circuit non-deterministic as the template can be made to overflow internally. Care should be taken to ensure this assumption is adhered to, if this template is to be used in the future.

Impact The template `NonZeroUintTag` performs the opposite function to what it is intended for. It is currently not activated, but if used in its current state it can cause severe issues.

Recommendation Reverse the order in which the inputs are passed to `GreaterThan`. The input signal which is to be verified should be the first signal and `0` should be the second.

Also add constraints to ensure that the input signal fits within `nBits`.

Developer Response The developers fixed the issue at commit ff18cf5.

4.1.47 V-PAN-VUL-047: Malicious pool can shadow valid pool

Severity	Warning	**Commit**	a16a43e
Type	Authorization	**Status**	Fixed
File(s)	`UniswapPoolsList.sol`		
Location(s)	_addPool		
Confirmed Fix At	fce61c7		

The Panther protocol owner adds approved exchange pools using the `Freemaster.addPool()` call. These pools are stored in the `pools` collection of the UniswapPoolsList contract, which maps 4-byte identifiers of the pool to the pool descriptor structure.

When a new pool is created, the key for that pool is calculated by taking the first 4 bytes from a hash of the two tokens addresses - the token pair - that the pool will serve. This key is not collision-resistant and could be exploited.

```
function _addPool(
        address _pool,
        address _tokenA,
        address _tokenB
    ) internal {
        bytes4 key = PoolKey.getKey(_tokenA, _tokenB);
        pools[key] = Pool({ _address: _pool, _enabled: true });
    }
```

Snippet 4.44: Snippet from `UniswapPoolsList._addPool()`

Impact A malicious actor can create their own token, and, by using the CREATE2, pick a contract address, such that if paired with a legitimate token address, it will occupy an existing record in the pool of pairs. In this scenario, all swap operations for the legitimate token pair would be sent to the malicious pool controlled by the attacker. For example, instead of tokens being sent to the ZKP/USDT pool, they may be sent to a malicious ZKP/MEM pool controlled by the attacker.

To execute this attack, the attacker would need to convince the Panther owners to add their token pair to their pool. While this may not be an easy task, it may be possible.

Recommendation It is recommended to increase the pool key up to 256 bits.

Developer Response The developers fixed the issue at commit fce61c7.

4.1.48 V-PAN-VUL-048: rangeCheck uses GreaterThan incorrectly

Severity	Warning	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	See description		
Location(s)	rangeCheck.circom		
Confirmed Fix At	f898606		

The templates `RangeCheckSingleSignal` and `RangeCheckGroupOfSignals` within `rangeCheck.circom` are used to perform range checks on a single input signal or a group of input signals respectively. Internally these templates make use of the `LessThan` and `GreaterThan` templates from circomlib to perform these checks. See snippet below for the implementation.

The template takes in `LessThanValue` and `GreaterThanValue` which are the upper and lower values of the range that the input signal should belong in.

```
template RangeCheckSingleSignal(maxBits, LessThanValue, GreaterThanValue) {
  signal input in;
  component less = LessThan(maxBits);
  less.in[0] <== in;
  less.in[1] <== LessThanValue;
  less.out === 1;

  component greater = GreaterThan(maxBits);
  greater.in[0] <== GreaterThanValue;
  greater.in[1] <== in;
  greater.out === 1;
}
```

Snippet 4.45: Snippet from `RangeCheckSingleSignal()`

Currently the order of the input signals to the `GreaterThan` template is reversed. It enforces that `GreaterThanValue > in && in < LessThanValue` whereas it should enforce that `GreaterThanValue < in[i] < LessThanValue`. It does not lead to a severe issue because this template is currently unused in the current scope of the circuits. If it is used in its current form in the future, it can lead to severe issues.

Additionally the `GreaterThan` and `LessThan` circuits assume that their input signals fit within `maxBits` i.e the argument the templates are instantiated with. Care should be taken to ensure this assumption is adhered to if these templates are used in the future.

Impact The intended use of templates `RangeCheckSingleSignal` and `RangeCheckGroupOfSignals` is to verify that the input signal or signals lie within the defined range of values. However, this range check is not implemented correctly and if used in its current form can cause severe issues.

Recommendation Reverse the order in which the inputs are passed to `GreaterThan`. The input signal which is to be verified should be the first signal and the number it is compared to should be the second.

The input signals should also be constrained to fit within `maxBits`.

Developer Response The developers fixed the issue at commit f898606.

4.1.49 V-PAN-VUL-049: The Vault does not provide receive function

Severity	Warning	**Commit**	a16a43e
Type	Maintainability	**Status**	Fixed
File(s)	`VaultV1.sol`		
Location(s)			
Confirmed Fix At	4e7e870		

The VaultV1 smart contract is responsible for accumulating funds, including native ETH. However, the contract does not implement the payable `receive()` function, and hence is not able to receive any native ETH out of the box.

Impact The vanilla VaultV1 contract will not be able to receive native ETH, so the functionality of the protocol may break. Currently, the approach seems to rely on a proxy contract that implements the receive function. However, this is not mentioned anywhere in the code, and if it is deployed in any other way, the entire protocol could break.

Recommendation It is recommended to add the receive function explicitly into the VaultV1 contract.

Developer Response Since the ensures the `VaultV1` contract contract is meant to be used behind a proxy

- direct ETH transfers to the implementation contract should be prevented to avoid locking funds
- ETH transfers should go through the proxy contract instead (implemented in `EIP173ProxyWithReceive`).

The commit `75c2f460` (hash to be updated) ensures the `VaultV1` contract won't accept direct ETH transfers.

4.1.50 V-PAN-VUL-050: Unauthorized events emission on behalf of Panther

Severity	Warning	Commit	a16a43e
Type	Access Control	Status	Fixed
File(s)	`FeeMaster.sol , PayMaster.sol`		
Location(s)			
Confirmed Fix At	dfe6b30		

There are a few places in the code where an arbitrary user can emit protocol events on behalf of the Panther.

- In the `FeeMaster.payOff` function, the `amount` isn't checked to be greater than 0. This allows to emit `PayOff` event for anyone.
- In the `PayMaster.postOp` function, the accompanying comment tells that this function should be called by the EntryPoint contract, but there is no check being done to enforce the origin of a call, hence anyone can emit the `UserOperationSponsored` event.
- The function `PayMaster.validatePaymasterUserOp` has to be called by the EntryPoint contract, but there is no check being done to enforce the origin of a call, hence anyone can emit the `ValidatePaymasterUserOpRequested` event.

Impact Depending on how these events are used by third-party systems or the client-side components of the protocol, there is a potential risk of unauthorized events causing damage.

Recommendation It is recommended to provide proper checks enforcing origin and/or correct function arguments.

Developer Response The developers fixed the issue at commit dfe6b30.

4.1.51 V-PAN-VUL-051: Several unnecessary magic constraints

Severity	Warning	**Commit**	a16a43e
Type	Maintainability	**Status**	Acknowledged
File(s)	Several files		
Location(s)	Several templates		
Confirmed Fix At	N/A		

Several templates introduce unnecessary "magic" constraints, as described here. Note that the link suggests doing that for any signals that do not participate in any constraints. If such a case does not exist, then there is no need for adding such constraints. The templates that introduce such constraints are the following: `AmmV1`, `ZAccountRegistrationV1`, `ZAccountRenewalV1`, `ZSwapV1`, `ZTransactionV1`, `TreeBatchUpdaterAndRootChecker` and their respective top level circuits `AmmV1Top`, `ZAccountRegistrationV1Top`, `ZAccountRenewalV1Top` and `ZSwapV1Top`.

Impact This adds several unnecessary constraints and can also lead to maintainability overheads in the feature.

Recommendation We recommend removing all such constraints and only introducing new ones for public signals that do not participate in any constraint.

Developer Response Developers have acknowledged the issue, but decided not to introduce changes at this point.

4.1.52 V-PAN-VUL-052: Scalar message encrypted using the incorrect shared public key

Severity	Warning	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`dataEscrowElGamalEncryption.circom`		
Location(s)	Template DataEscrowElGamalEncryption()		
Confirmed Fix At	14fb7f6		

The template `DataEscrowElGamalEncryption` constructs encrypted messages which are published on-chain through events. The encrypted message contains the padding points, the scalar message and then the derived UTXO spending public key. An encrypted message at a particular index is derived using the expression `ephemeralRandom * pubKey + M + HidingPoint`, where `M` is the scalar message being encrypted and `ephemeralRandom` is the randomness used to generate the ephemeral public keys.

As seen in the snippet, at the moment the shared public key at index `j` is used to encrypt the scalar message at index `j`. This is incorrect because it doesn't take into consideration the `offset` which needs to be applied to fetch the intended shared public key. The padding points are encrypted before the scalar message and therefore the public keys should be used from an offset of `PaddingPointsSize`.

```
for (var j = 0; j < ScalarsSize; j++) {
  // M = m * B8
  drv_mG[j] = BabyPbk();
  drv_mG[j].in <== scalarMessage[j];
  // require 'm < 2^64' - otherwise brute-force will be near to impossible
  assert(scalarMessage[j] < 2**64);

  // ephemeralRandom * pubKey + M + hidingPoint
  drv_mGrY[offset+j] = BabyAdd();
  drv_mGrY[offset+j].x1 <== drv_mG[j].Ax;
  drv_mGrY[offset+j].y1 <== drv_mG[j].Ay;
  drv_mGrY[offset+j].x2 <== ephemeralPubKeyBuilder.sharedPubKey[j][0];
  drv_mGrY[offset+j].y2 <== ephemeralPubKeyBuilder.sharedPubKey[j][1];

  drv_mGrY_final[offset+j] = BabyAdd();
  drv_mGrY_final[offset+j].x1 <== drv_mGrY[offset+j].xout;
  drv_mGrY_final[offset+j].y1 <== drv_mGrY[offset+j].yout;
  drv_mGrY_final[offset+j].x2 <== ephemeralPubKeyBuilder.hidingPoint[0];
  drv_mGrY_final[offset+j].y2 <== ephemeralPubKeyBuilder.hidingPoint[1];

  // encrypted data
  encryptedMessage[offset+j][0] <== drv_mGrY_final[offset+j].xout;
  encryptedMessage[offset+j][1] <== drv_mGrY_final[offset+j].yout;
```

Snippet 4.46: Snippet from `DataEscrowElGamalEncryption()`

Because the scalar message is encrypted using the incorrect shared public key, it will not be possible to successfully retrieve the scalar message which contains the data of the UTXO to be spent.

Impact As the scalar message ends up being encrypted with a shared public key meant for a different message, it will not be possible to decrypt the scalar message from the cipher text using its corresponding shared public key.

Recommendation Make use of the shared public key with index `offset + j` instead of `j` to encrypt the scalar message.

Developer Response The developers fixed the issue at commit 14fb7f6.

4.1.53 V-PAN-VUL-053: Ephemeral public key space can have collisions

Severity	Warning	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`dataEscrowElGamalEncryption.circom`		
Location(s)	Template EphemeralPubKeysBuilder		
Confirmed Fix At	9b37c9a		

In `EphemeralPubKeysBuilder`, the ephemeral keys are built starting from one ephemeral key which is passed in as ephemeral randomness. The new ephemeral keys are generated by hashing the previous shared secret using a Poseidon hash and passing the truncated result as an input to `BabyPbk`. See snippet below for the implementation.

The signal passed as input to `BabyPbk` should be less than the babyjubjub suborder for the circuit to be deterministic, and therefore the output of the Poseidon hash is truncated to `252` bits to ensure this requirement is met. But, this truncation only ensures that the signal is `252` bits in length, and it does not ensure that the signal is less than the suborder.

```
// next ephemeral pub-key - random * B8
ephemeralPubKey_eRandMultG[i] = BabyPbk();
ephemeralPubKey_eRandMultG[i].in <== ephemeralRandoms[i];
ephemeralPubKey[i][0] <== ephemeralPubKey_eRandMultG[i].Ax;
ephemeralPubKey[i][1] <== ephemeralPubKey_eRandMultG[i].Ay;

// make next e-rand: Poseidon(sharedKey.x, sharedKey.y) - 0..251 bit of it
if( i < nPubKeys - 1 ) {
    hash[i] = Poseidon(2);
    hash[i].inputs[0] <== sharedKey_eRandMultPubKey[i].out[0];
    hash[i].inputs[1] <== sharedKey_eRandMultPubKey[i].out[1];

    n2b_hash[i] = Num2Bits(254);
    n2b_hash[i].in <== hash[i].out;
    b2n_hash[i] = Bits2Num(252);

    for(var j = 0; j < 252; j++) {
        b2n_hash[i].in[j] <== n2b_hash[i].out[j];
    }
    ephemeralRandoms[i+1] <== b2n_hash[i].out;
}
```

Snippet 4.47: Snippet from template `EphemeralPubKeysBuilder()`

The output of the hash function is distributed uniformly and it will be larger than the suborder a significant number of times resulting in collisions in the ephemeral public key space.

Impact The ephemeral public key generation process can overflow causing collisions in the ephemeral public key space.

Recommendation The output of the Poseidon hash should be truncated to `251` bits instead of `252`. This ensures that the output is always less than the babyjubjub suborder.

Developer Response The developers fixed the issue at commit 9b37c9a.

4.1.54 V-PAN-VUL-054: trustProvidersKyt enabled flag is not universal

Severity	Warning	**Commit**	a16a43e
Type	Maintainability	**Status**	Fixed
File(s)	`trustProvidersKyt.circom`		
Location(s)	TrustProvidersInternalKyt , TrustProvidersDepositWithdrawKyt		
Confirmed Fix At	eb6fe3a		

The circuits `TrustProvidersInternalKyt` , `TrustProvidersDepositWithdrawKyt` have a special signal called `enabled` that is used to turn on or turn off all the checks in the circuit. The issue here is that this signal does not affect the constraint of the `kytSignedMessagePackageType` signal value, even if the `enabled` signal is set to 0.

```
template TrustProvidersDepositWithdrawKyt() {
  // ... skipped ...
  component isLessThanEq_createTime_DW_Timestamp = LessEqThanWhenEnabled(252);
  isLessThanEq_createTime_DW_Timestamp.enabled <== enabled;
  isLessThanEq_createTime_DW_Timestamp.in[0] <== createTime;
  isLessThanEq_createTime_DW_Timestamp.in[1] <== kytSignedMessageTimestamp +
    zZoneKytExpiryTime;

  // package type
  kytSignedMessagePackageType === 2;
  // ... skipped ...
}
```

Snippet 4.48: Snippet from `TrustProvidersDepositWithdrawKyt()`

Impact The `enabled` signal is not universal and does not guarantee that all circuit checks will be turned off. However, it would be reasonable to expect this from it. This may cause maintainability issues in the future.

Recommendation It is recommended to tie the `kytSignedMessagePackageType` signal constraints to the `enabled` signal as well in both circuits.

Developer Response The developers fixed the issue at commit eb6fe3a.

4.1.55 V-PAN-VUL-055: extraInputsHash should be used as the magical constraint

Severity	Warning	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	Several files		
Location(s)	Several templates		
Confirmed Fix At	527c358		

Several templates implement "magic" constraints, as described here. These magical constraints are only needed for signals that do not participate in any constraints as explained in V-PAN-VUL-051 issue. Along with the magical constraints, the templates also implement constraints to anchor the signal `extraInputsHash` as shown in the snippet below.

The extra magical constraints are not needed and instead the `extraInputsHash` should be used for that purpose. Note that the current constraint applied on `extraInputsHash` is different from the solution suggested in the link. Instead, the constraint should be changed to the recommended solution.

```
1 //
      ////////////////////////////////////////////////////////////////////////////////////////////////
2 // [0] - Extra inputs hash anchoring
3 //
      ////////////////////////////////////////////////////////////////////////////////////////////////
4 extraInputsHash === 1 * extraInputsHash;
```

Snippet 4.49: Snippet from template `TreeBatchUpdaterAndRootChecker`

Impact The current constraint applied on `extraInputsHash` deviates from the suggested solution for Groth16 ****Malleability.

Recommendation Update the constraint on `extraInputsHash` in the mentioned templates to `0 === 0 * extraInputsHash`.

Developer Response The developers fixed the issue at commit 527c358.

4.1.56 V-PAN-VUL-056: Range check on utxoInSpendPrivKey is disabled

Severity	Warning	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`zSwapV1Top.circom`		
Location(s)	Template ZSwapV1Top()		
Confirmed Fix At	0bc95b6		

The top level circuit `ZSwapV1Top` applies range checks on the input signals used in the circuit `ZSwapV1`. In this circuit, the range check on input signal `utxoInSpendPrivKey` is disabled because the tag is ignored as can be seen in the snippet below.

This is problematic because there are no other constraints applied on `utxoInSpendPrivKey`. This private key corresponds to the derived spending public key for the UTXO. If the private key is not verified to be less than the suborder, then multiple private keys can correspond to the same public key which can cause issues. This is explored in more detail in this V-PAN-VUL-001.

```
1 signal rc_utxoInSpendPrivKey[nUtxoIn] <== BabyJubJubSubOrderTagArray(IGNORE_ANCHORED,
      nUtxoIn)(utxoInSpendPrivKey);
2 signal rc_utxoInSpendKeyRandom[nUtxoIn] <== BabyJubJubSubOrderTagArray(
      IGNORE_ANCHORED,nUtxoIn)(utxoInSpendKeyRandom);
3 signal rc_utxoInAmount[nUtxoIn] <== Uint64TagArray(IGNORE_ANCHORED,nUtxoIn)(
      utxoInAmount);
4 signal rc_utxoInOriginZoneId[nUtxoIn] <== Uint16TagArray(IGNORE_ANCHORED,nUtxoIn)(
      utxoInOriginZoneId);
```

Snippet 4.50: Snippet from `ZSwapV1Top()`

At the moment, because `utxoInSpendPrivKey` is not involved in other constraints this should not lead to an issue. But it can added to other constraints in the future which will then introduce issues into the circuits because it is under-constrained.

Impact If `utxoInSpendPrivKey` is used in any other constraints in the future without validating its range, then it can lead to issues.

Recommendation Constrain `utxoInSpendPrivKey` to be less than the suborder, so the witness generation remains deterministic.

Developer Response The developers fixed the issue at commit 0bc95b6.

4.1.57 V-PAN-VUL-057: PartiallyFilledChainBuilder might behave in an unexpected way

Severity	Warning	**Commit**	a16a43e
Type	Maintainability	**Status**	Fixed
File(s)	`partiallyFilledChainBuilder.circom`		
Location(s)	PartiallyFilledChainBuilder		
Confirmed Fix At	c4c50ee		

The circuit template `PartiallyFilledChainBuilder` calculates the root hash of a degenerate binary Merkle tree, which is essentially a chain of elements. This template requires two inputs: a list of tree elements called `inputs` and the length of the list, called `nInputs`. There is also a compile-time parameter in this template called `max_nInputs`, which indicates the maximum length of element lists that the instantiated circuit can process.

If the value of the `nInputs` input signal exceeds `max_nInputs`, the template will not calculate any hashes and will return a value of 0.

Impact Since Panther's code often passes hash values into `enabled` signals to other circuits to turn security checks on or off, an attacker could potentially use this behavior to disable checks in certain contexts.

Recommendation It is recommended to enforce the `nInputs` signal value to be less than the `max_nInputs` value.

Developer Response The developers fixed the issue at commit c4c50ee.

4.1.58 V-PAN-VUL-058: ForestTree can rewrite TAXI root with zero

Severity	Warning	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`ForestTree.sol`		
Location(s)	addUtxosToBusQueueAndTaxiTree()		
Confirmed Fix At	372252a		

The function `ForestTree.addUtxosToBusQueueAndTaxiTree()` takes a parameter called `numTaxiUtxos` which represents the number of UTXOs to be inserted into the TAXI tree.

In case this parameter is zero, the current implementation of the function will rewrite the root of the Taxi Tree with a zero value, due to the way fuction `_addUtxos` behaves when called with an empty input array. The length of the `utxos` array gets overwritten to 0 in the assembly code snippet before the function call.

```
function addUtxosToBusQueueAndTaxiTree(
        bytes32[] memory utxos,
        uint8 numTaxiUtxos,
        uint256 cachedForestRootIndex,
        bytes32 forestRoot,
        bytes32 staticRoot,
        uint96 reward
    )
{
  if (numTaxiUtxos == 1) {
     taxiTreeNewRoot = _addUtxo(utxos[0]);
  } else {
     // solhint-disable-next-line no-inline-assembly
     assembly {
       // Load the length of the ‘arr‘ array
       let arrLength := mload(utxos)
       // Check if we need to modify the length
       if gt(arrLength, numTaxiUtxos) {
         // Set the new length of the array
         mstore(utxos, numTaxiUtxos)
        }
      }
      taxiTreeNewRoot = _addUtxos(utxos);
   }

  _cacheNewForestRoot(taxiTreeNewRoot, TAXI_TREE_FOREST_LEAF_INDEX);
}
```

Snippet 4.51: Snippet from `ForestTree.addUtxosToBusQueueAndTaxiTree()`

Impact If the function is called with a zero-valued `numTaxiUtxos`, the Taxi Tree will be nullified, preventing the protocol from operating any further. Currently, this function is never called with `numTaxiUtxos` set to zero. However, this might be violated in future versions of the protocol.

Recommendation It is recommended to modify the `TaxiTree._addUtxos` function to ensure that it returns the current taxi tree root value in the case where the passed `utxos` array is empty, instead of returning a zero value.

Developer Response The developers fixed the issue at commit 1611c25.

4.1.59 V-PAN-VUL-059: zAccount input commitment verification can be disabled

Severity	Warning	**Commit**	a16a43e
Type	Maintainability	**Status**	Fixed
File(s)	`ammV1.circom`		
Location(s)	Template AmmV1		
Confirmed Fix At	7537208		

In `ammV1.circom` the input signal `zAccountUtxoInCommitment` is verified to be the same as the hash computed from the `zAccount` note. This check is conditionally enabled using `ForceEqualIfEnabled` if `zAccountUtxoInCommitment` is non-zero. See snippet below for the implementation.

```
1 // [5] - Verify zAccountUtxoInUtxo commitment
2 component zAccountUtxoInHasherProver = ForceEqualIfEnabled();
3 zAccountUtxoInHasherProver.in[0] <== zAccountUtxoInCommitment;
4 zAccountUtxoInHasherProver.in[1] <== zAccountUtxoInNoteHasher.out;
5 zAccountUtxoInHasherProver.enabled <== zAccountUtxoInCommitment;
```

Snippet 4.52: Snippet from template `AmmV1()`

The use of `zAccountUtxoInCommitment` to enable the check is problematic because the verification can be disabled if a zero value is passed as input. This does not currently lead to an issue because i) The `zAccountNoteHash` is checked for inclusion in one of the UTXO merkle trees and ii) `zAccountUtxoInCommitment` is not used in any other constraints. But, that may change in the future and it is easy to overlook that there is no non-zero constraint or validation on `zAccountUtxoInCommitment`.

Furthermore, because `zAccountUtxoInCommitment` is not involved in any other constraints and it is not a public signal, the verification of the `zAccountUtxoInUtxo` commitment is unnecessary.

Impact The `zAccountUtxoInCommitment` may be used unknowingly in another constraint when it is not properly validated as its verification can be disabled.

Recommendation If `zAccountUtxoInCommitment` is intended to be used, then add a non-zero constraint to ensure that the verification step cannot be disabled.

Consider removing `zAccountUtxoInCommitment` along with its verification steps if it has no future use, as it is currently unnecessary.

Developer Response The developers fixed the issue at commit 7537208.

4.1.60 V-PAN-VUL-060: Extensive use of ForceEqualIfEnabled

Severity	Warning	**Commit**	a16a43e
Type	Maintainability	**Status**	Acknowledged
File(s)	Several files		
Location(s)	Several templates		
Confirmed Fix At	N/A		

The circuits of the project extensively use the `ForceEqualIfEnabled` template from Circomlib to perform critical checks. This template is generally used to verify if a user has accurately calculated a commitment. For instance, given a commitment `c`, the protocol uses `c` as the `enabled` flag and one of the two inputs of `ForceEqualIfEnabled`. The other input is typically constructed by the circuit from other user-provided signals (see snippet below). This design often allows skipping the equality check, so another protocol component must ensure that the check was not omitted (i.e., checking that `c != 0`). This could be the smart contracts or another template of the circuit.

This pattern can be found in both top-level circuits (i.e., main) and auxiliary templates. We'll explain why this design is risky in both cases in the following sections:

- ***Main Circuits***: There are two scenarios involving `c`. Firstly, if `c` is a private signal, the main circuit must ensure that `c` influences a public signal that is subsequently checked by a smart contract. Secondly, if `c` is a public signal, a smart contract must verify that the check wasn't disabled. For both scenarios, the check for `c` is deferred to smart contracts, which could potentially be omitted by mistake.
- ***Auxiliary Template***: This case presents a risk to maintainability in future protocol iterations, as developers might assume that the auxiliary template consistently enforces the equality.

```
...
// [5] - Verify zAccountUtxoInUtxo commitment
component zAccountUtxoInHasherProver = ForceEqualIfEnabled();
zAccountUtxoInHasherProver.in[0] <== zAccountUtxoInCommitment;
zAccountUtxoInHasherProver.in[1] <== zAccountUtxoInNoteHasher.out;
zAccountUtxoInHasherProver.enabled <== zAccountUtxoInCommitment;
```

Snippet 4.53: Snippet from `ZAccountRenewalV1`

Impact The existing design complicates safety considerations. Moreover, it may lead to subtle bugs in the future.

Recommendation Consider replacing all instances of `ForceEqualIfEnabled` with `IsEqual` unless it is necessary, and ensure that its output is one.

Developer Response Developers have acknowledged the issue but decided not to introduce changes at this point.

4.1.61 V-PAN-VUL-061: Imprecise fee value extraction in PluginDataDecoderLib

Severity	Warning	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`PluginDataDecoderLib.sol`		
Location(s)	decodeUniswapV3RouterExactInputSingleData()		
Confirmed Fix At	3a6f22f		

The library function `PluginDataDecoderLib.decodeUniswapV3RouterExactInputSingleData()` extracts important swap parameters from the binary string passed from the user.

The extraction is performed manually, in assembly, probably to make the execution cost of the operation cheaper.

The code related to this issue is shown in the following snippet:

```
function decodeUniswapV3RouterExactInputSingleData(
  bytes memory data
) internal pure returns (
    uint32 deadline,
    uint96 amountOutMinimum,
    uint24 fee,
    uint160 sqrtPriceLimitX96
) {
// ... skipped ...
assembly {
  let location := data
  // skip the 160 bits for plugin address
  let pluginData_1 := mload(add(location, add(0x20, 0x14)))
  // ...skipped...
  fee := and(shr(104, pluginData_1), 0xffffffff)
  // ... skipped ...
 }
}
```

Snippet 4.54: Snippet from `example()`

The `fee` value is computed here by shifting right the 256 bits word read from the pluginData and masking it with the value `0xffffffff` , which is 32 bits. Note, however, that the mask value should be `0xffffff` since the fee occupies only 24 bits.

This code still works correctly due to the fact that the length of the output variable `fee` is `uint24`, and the higher bits are simply discarded during the conversion from `uint32` to `uint24`.

Impact The described implicit conversion from `uint32` to `uint24` may lead to maintenance issues if the type of the fee variable needs to be changed.

Recommendation It is recommended to either provide the correct masking value, or, if it was done intentionally for any reason, provide a comment explaining the reason behind this decision and warning for developers.

Developer Response The developers fixed the issue at commit 3a6f22f.

4.1.62 V-PAN-VUL-062: Potential underflow in ZkpReserveController

Severity	Warning	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`ZkpReserveController.sol`		
Location(s)	_scReleasableAmount()		
Confirmed Fix At	4bf0f7e		

The function `ZkpReserveController._scReleasableAmount()` may cause an arithmetic underflow if protocol admins set the `scReleasablePerBlock` value without considering the current `scTotalReleased` amount.

```
function _scReleasableAmount() private view returns (uint64) {
 uint64 blockOffset = block.number.safe64() - startBlock;
 return (scReleasablePerBlock * blockOffset) - scTotalReleased;
}
```

Snippet 4.55: Snippet from `_scReleasableAmount()()`

Impact If the underflow occurs, the function `releaseZkps()` will stop working, preventing the AMM from being refilled with ZKP tokens and distributed among eligible users.

Recommendation It is recommended to check the `releasablePerBlock` parameter in the setter function `updateParams()`, or check for underflow in `_scReleasableAmount()` and revert with a specific reason if necessary.

Developer Response The developers fixed the issue at commit 4bf0f7e.

4.1.63 V-PAN-VUL-063: PrpVoucherHandler logic allows to set unreasonable voucher terms

Severity	Warning	Commit	a16a43e
Type	Logic Error	Status	Fixed
File(s)	PrpVoucherHandler.sol		
Location(s)	_updateVoucherTerms()		
Confirmed Fix At	9921c68		

The function `PrpVoucherController.updateVoucherTerms()` is called by protocol owners to set reward parameters tied to a specific voucher.

The function `PrpVoucherHandler._updateVoucherTerms()`, which is directly called from `PrpVoucherController.updateVoucherTerms()`, allows setting values of `_limit` and `_amount` such that the new rewards will not be issued due to an incorrect check in the function.

Consider the following snippet:

```
function _updateVoucherTerms(
   address _allowedContract,
   bytes4 _voucherType,
   uint64 _limit,
   uint64 _amount,
   bool _enabled
) internal {
   uint64 rewardsGenerated =
       voucherTerms[_allowedContract][_voucherType].rewardsGranted;
   require(_limit + _amount >= rewardsGenerated,
        "PrpVoucherController: Limit cannot be less than rewards generated");
   // ... update the voucher terms for the given voucher type
}
```

Snippet 4.56: Snippet from `PrpVoucherHandler._updateVoucherTerms()`

The check in the `require` statement allows you to pass a pair of `_limit` and `_amount` values, such that the new limit will be less than `rewardsGenerated`. This seems unreasonable, as the new limit value should be at least as large as `rewardsGenerated` for a given voucher. Otherwise, it contradicts the meaning of being a limit.

Impact In the event that such unreasonable parameters are passed and set, subsequent calls to `_generateRewards()` will not award rewards to users, as the limit has been reached.

Recommendation A more reasonable check would be `require(_limit >= rewardsGenerated + _amount);`. This check would ensure that the limit is large enough to cover at least one more reward of the specified amount.

Developer Response The developers fixed the issue at commit 9921c68.

4.1.64 V-PAN-VUL-064: ZkpReserveController configuration validation should be performed on scaled inputs

Severity	Warning	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`ZkpReserveController.sol`		
Location(s)	updateParams()		
Confirmed Fix At	c220722		

In the `ZkpReserveController`, the function `updateParams` is used to configure the amount of `ZKP` tokens releasable per block and the min rewardable amount. It also performs input validation to ensure that the input values are non-zero. See snippet below for the implementation.

```
function updateParams(
  uint256 releasablePerBlock,
  uint256 minRewardedAmount
) external onlyOwner {
  require(
      releasablePerBlock > 0 && minRewardedAmount > 0,
      ERR_INVALID_PARAMS
  );

  scReleasablePerBlock = releasablePerBlock.scaleDownBy1e12().safe64();
  scMinRewardableAmount = minRewardedAmount.scaleDownBy1e12().safe64();

  emit RewardParamsUpdated(releasablePerBlock, minRewardedAmount);
}
```

Snippet 4.57: Snippet from `updateParams()`

After the inputs are validated, they values are scaled down by `1e12`. It is possible that they may be mistakenly set to a value which when scaled down becomes `0`. The non-zero validations should be performed on the scaled down values of the inputs instead.

Impact Currently, the performed validations do not ensure that the `scReleasablePerBlock` and the

`scMinRewardableAmount` will be non-zero.

Recommendation Perform the non-zero validations on the scaled down version of the inputs.

Developer Response The developers fixed the issue at commit c220722.

4.1.65 V-PAN-VUL-065: Unchecked return in safeContractBalance

Severity	Warning	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`TransferHelper.sol`		
Location(s)	Function safeContractBalance		
Confirmed Fix At	ce20ce2		

Function `TransferHelper.safeContractBalance` ignores the return value of the call to `isDeployedContract` (see snippet below). This essentially renders the call to `isDeployedContract` as dead code.

```
/// @dev Get the Native balance of `_contract`
function safeContractBalance(
    address _contract
) internal view returns (uint256) {
    isDeployedContract(_contract);
    return _contract.balance;
}
```

Snippet 4.58: Snippet from `example()`

Clearly the intention here is to disallow addresses that are not deployed contracts. However, this function will return the balance of any address, including EOAs.

Impact Currently, this function is only called with addresses that are guaranteed to be deployed contracts. So, there are no security concerns surrounding this issue.

Recommendation Replace `isDeployedContract(_contract)` with `require(isDeployedContract(_contract))`.

Developer Response The developers fixed the issue at commit ce20ce2.

4.1.66 V-PAN-VUL-066: Imprecise isTaxiApplicable() value computation

Severity	Warning	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`TransactionOptions.sol`		
Location(s)	isTaxiApplicable()		
Confirmed Fix At	595ccff		

The function `isTaxiApplicable()` is used to check if the Taxi Tree is applicable to the current transaction by checking the corresponding bit in the `transactionOptions` parameter. However, this function is not implemented accurately. Consider the following snippet.

```
function isTaxiApplicable(
        uint32 transactionOptions
) internal pure returns (bool) {
  // The 1 MSB contains the TaxiEnabler
  return (transactionOptions >> 16) == 1;
}
```

Snippet 4.59: Snippet from `isTaxiApplicable()`

The implementation assumes that all bits after the 17th bit in the transaction options are zero, but in fact, these bits are stated to be reserved and can be any value, even if they are ignored.

Impact If the most significant bits of the `transactionOptions` after the 17th bit are not all zero, the check will fail, even if the Taxi Tree enabling bit is set to 1.

Recommendation The check should be implemented in a more precise manner, for example as

```
(transactionOptions >> 16) & 1 == 1
```

Developer Response The developers fixed the issue at commit 595ccff.

4.1.67 V-PAN-VUL-067: Insufficient input validation in several locations

Severity	Warning	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	see description		
Location(s)	See description		
Confirmed Fix At	21c1d90		

The following places in the code were identified as having insufficient checks for the provided inputs.

- Function `RingBufferTree._insertLeaf()` does not validate the `leafIndex` to be less than or equal to the `MAX_LEAF_INDEX`
- Function `CachedRoots._cacheNewForestRoot()` can be called before the initialization has happened
- Function `CachedRoots._initCacheForestRoot()` can be called several times
- Function `BusQueues._updateBusQueueRewardParams()` allows the `reservationRate` parameter to be equal to the `HUNDRED_PERCENT`, but the comment in the `_estimateRewarding()` function says the reservation parameter has to be less than that
- In the function `Bytecode.read()`, if the `offset` parameter happens to be equal to `size`, the function will read 0 bytes, and return an empty data, however the memory pointer will still be updated with a new value due to unnecessary memory allocation.
- Function `FeeMaster.updateProtocolZkpFeeDistributionParams()` does not provide any sanity checks for the protocol-wide parameters.
- Function `AppConfiguration.updateMaxBlockTimeOffset()` does not provide any reasonable bounds for the `_maxBlockTimeOffset` parameter.
- Function `AppConfiguration.updateCircuitId()` does not prevent the passed `circuitId` value to be equal to `address(0)`

Impact The stated observations may cause maintainability issues under certain circumstances.

Recommendation It is recommended to address the stated observations by implementing necessary checks.

Developer Response The developers fixed the issue at commit 21c1d90.

4.1.68 V-PAN-VUL-068: Users may receive no rewards in some cases

Severity	Warning	**Commit**	a16a43e
Type	Logic Error	**Status**	Fixed
File(s)	`PrpVoucherHandler.sol`		
Location(s)	_generateRewards()		
Confirmed Fix At	db98fab		

The function `_generateRewards` is used internally to generate rewards for the user on completion of certain actions. See snippet below for the implementation.

But, if the `rewardsGranted + prpToGrant` is greater than the `voucherTerm.limit`, then the user receives no rewards. This is not a good way to account for this edge case. Users with large reward amounts but close to the limit will not get rewards at all, instead of getting reduced rewards. Changing the amount params may also cause this.

Instead, for the above case, setting `prpToGrant` to `limit - rewardsGranted - 1` will ensure that the entire voucher limit is used.

```
function _generateRewards(
  bytes32 _secretHash,
  uint64 _amount,
  bytes4 _voucherType
) internal returns (uint256) {
  VoucherTerms memory voucherTerm = voucherTerms[msg.sender][
      _voucherType
  ];

  uint64 prpToGrant = _amount > 0 ? _amount : voucherTerm.amount;

  if (voucherTerm.rewardsGranted + prpToGrant > voucherTerm.limit)
      return 0;

  // we are setting the balance to non-zero to save gas
  if (balance[_secretHash] > ZERO_VALUE) {
      balance[_secretHash] += prpToGrant;
  } else {
      balance[_secretHash] = ZERO_VALUE + prpToGrant;
  }

  voucherTerms[msg.sender][_voucherType].rewardsGranted += prpToGrant;

  return prpToGrant;
}
```

Snippet 4.60: Snippet from `_generateRewards()`

Impact If the `rewardsGranted + prpToGrant` is larger than the `voucherTerm.limit`, then the user receives no rewards. This can happen repeatedly if the leftover amount in the voucher is extremely small.

It will also make it harder to gauge when to reset the voucher limit, because to an external observer the voucher limit is not used up.

Recommendation if the `rewardsGranted + prpToGrant` is greater than the `voucherTerm.limit`, then generate `limit - rewardsGranted - 1` as the `PRP` to grant.

Developer Response The developers fixed the issue at commit db98fab.

4.1.69 V-PAN-VUL-069: Instantiations of Num2Bits(254) can overflow

Severity	Warning	**Commit**	a16a43e
Type	Data Validation	**Status**	Fixed
File(s)	`dataEscrowElGamalEncryption.circom`		
Location(s)	Template DataEscrowElGamalEncryption		
Confirmed Fix At	956416e		

There are several instantiations of template `Num2Bits` where it argument is set to `254`. At the current commit/scope, there are the following locations:

- `dataEscrowElGamalEncryption.circom` (lines 123, 158, and 410) 3 instances
- `zAccountBlackListLeafInclusionProver.circom` (line 54) 1 instance
- `zoneIdInclusionProver.circom` (line 16) 1 instance

As described here, this can have detrimental effects on the codebase because this template is not deterministic when the template parameter is greater or equal to `254`.

Impact The application is currently using the output of several of these instantiations in a non-trivial manner. This is risky because attackers may be able exploit this fact to create proofs about bogus facts.

Recommendation It is recommended to use `Num2Bits_strict`, which ensures that the bit representation is smaller than the field's prime.

Developer Response The developers fixed the issue at commit 956416e.

4.1.70 V-PAN-VUL-070: Multiposeidon is prone to hash collisions

Severity	Warning	**Commit**	a16a43e
Type	Cryptographic Vulnera	**Status**	Acknowledged
File(s)	`circuits/templates/dataEscrowElGamalEncryption.circom`		
Location(s)	template MultiPoseidon		
Confirmed Fix At	N/A		

Template `MultiPoseidon` is designed to hash an arbitrary number of input signals. To achieve this, `MultiPoseidon` recursively builds a tree of `MultiPoseidon` hashes. The recursion terminates when the number of inputs are one supported by the native circom `Poseidon` implementation (see snippet below).

```
if ( n <= 15 ) {
    hash = Poseidon(n);
    for(var i = 0; i < n; i++) {
        hash.inputs[i] <== in[i];
    }
    out <== hash.out;
} else {
    var n1 = n\2;
    var n2 = n-n\2;
    m_poseidon[0] = MultiPoseidon(n1);
    m_poseidon[1] = MultiPoseidon(n2);
```

Snippet 4.61: Snippet from `MultiPoseidon()`

`MultiPoseidon` is prone to collision by construction. Consider a `MultiPoseidon` instantiation with `n` levels of recursion, by construction all hashes at level `n` and all hashes at level `n-1` will result to the same `MultiPoseidon` hash.

Impact Hash constructions prone to collisions can have significant impact on the overall protocol.

Recommendation Consider implementing a sponge construction, as recommended by the poseidon paper.

Developer Response The code base does not use this function as a universal hash. The usage is quite narrow and the likelihood of a vulnerability is quite low.

Veridise Response The Veridise team agrees with the Panther protocol developers. The circuits already guarantee that the correct proof height is provided. The only risk remaining is for entities that consume these hashes in the unlikely case where inner nodes can be interpreted as meaningful data for the protocol.

4.1.71 V-PAN-VUL-071: Duplicate code across files

Severity	Info	**Commit**	a16a43e
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	6429ea9		

There are multiple code elements which are duplicated across files.

- Template `Selector3` exists in `selector3.circom` and `merkleTreeInclusionProof.circom`.
- The template `Selectable3TreeInclusionProof` which resides in `selectable3TreeInclusionProof.circom` is functionally the same as `MerkleTreeInclusionProofDoubleLeavesSelectable` inside `merkleTreeInclusionProof.circom`.

Impact These duplicated templates will be harder to maintain and more prone to bugs, because a single change needs to be updated in multiple locations and it can easily be overlooked.

Recommendation Eliminate redundant copies mentioned above.

Developer Response The developers fixed the issue at commit 6429ea9.

4.1.72 V-PAN-VUL-072: Unused code

Severity	Info	**Commit**	a16a43e
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	d50eb32		

Description The following templates are not used in any of the main circuits:

- Template `UtxoNoteInclusionProver` in file `utxoNoteInclusionProver.circom`
- Template `DataEscrowElGamalEncryptionScalar` in file `dataEscrowElGamalEncryption.circom`
- Template `DaoDataEscrowSerializer` in file `dataEscrowElGamalEncryption.circom`
- Template `RangeCheckGroupOfSignals` in file `rangeCheck.circom`
- Template `RangeCheckSingleSignal` in file `rangeCheck.circom`

Impact These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

Recommendation Remove the unused constructs. If it is intended for future use, add some documentation stating where and how they are intended to be used.

Developer Response The developers fixed the issue at commit d50eb32.

Glossary

ERC-1155 The Ethereum fungible multi-token standard. See `https://eips.ethereum.org/EIPS/eip-1155` to learn more. 1, 3

ERC-20 The famous Ethereum fungible token standard. See `https://eips.ethereum.org/EIPS/eip-20` to learn more. 1, 3

ERC-2535 This proposal standardizes diamonds, which are modular smart contract systems that can be upgraded/extended after deployment, and have virtually no size limit. More technically, a diamond is a contract with external functions that are supplied by contracts called facets. Facets are separate, independent contracts that can share internal functions, libraries, and state variables. . 3

ERC-4337 An account abstraction proposal which completely avoids the need for consensus-layer protocol changes. Instead of adding new protocol features and changing the bottom-layer transaction type, this proposal introduces a higher-layer pseudo-transaction object called a UserOperation. Users send UserOperation objects into a new separate mempool. Bundlers package up a set of these objects into a single transaction by making a call to a special contract, and that transaction then gets included in a block. . 1

ERC-721 The Ethereum non-fungible token standard. See `https://eips.ethereum.org/EIPS/eip-721` to learn more. 1, 3

ERC-777 ERC777 is a standard for fungible tokens, and is focused around allowing more complex interactions when trading tokens. More generally, it brings tokens and Ether closer together by providing the equivalent of a msg.value field, but for tokens. . 3

zero-knowledge circuit A specific tool or technique used to encode computer programs as Zero-Knowledge proofs. It defines the rules and logic to verify that a program was run correctly.. 1