Indigo Protocol

Audit Report - Revision 1

MLabs Audit Team

November 7, 2022 (Revised December 9, 2022)

Contents

Revision Notes

Revision 1 - 2022-12-09

Revision 1 of the issued *Audit Report* (issued 2022-11-07) brings the following changes to the original document:

1. A vulnerability that was found to be a duplicate was removed from the list of identified vulnerabilities. Any changes resulting from its removal have been applied throughout the report.
2. Technical and implementation details were omitted from the description of the vulnerabilities.
3. Some text was rephrased for readability purposes.
4. Some members of the MLabs Audit team conducted a post-audit review of proposed fixes for the identified vulnerabilities. The inclusion of a status tag for each vulnerability reflects the findings of this review.

Index

References

- Audit Meta Issue
- Indigo GitHub Repository
- CVSS-Scale
- Common Vulnerability Scoring System
- NVD Calculator

1. Disclaimer

This audit report is presented without warranty or guarantee of any type. Neither MLabs nor its auditors can assume any liability whatsoever for the use, deployment or operations of the audited code. This report lists the most salient concerns that have become apparent to MLabs' auditors after an inspection of the project's codebase and documentation, given the time available for the audit. Corrections may arise, including the revision of incorrectly reported issues. Therefore, MLabs advises against making any business or other decisions based on the contents of this report.

An audit does not guarantee security. Reasoning about security requires careful considerations about the capabilities of the assumed adversaries. These assumptions and the time bounds of the audit can impose realistic constraints on the exhaustiveness of the audit process. Furthermore, the audit process involves, amongst others, manual inspection and work which is subject to human error.

MLabs does not recommend for or against the use of any work or supplier mentioned in this report. This report focuses on the technical implementation provided by the project's contractors and subcontractors, based on the information they provided, and is not meant to assess the concept, mathematical validity, or business validity of their product. This report does not assess the implementation regarding financial viability nor suitability for any purpose. *MLabs does not accept responsibility for any loss or damage howsoever arising which may be suffered as result of using the report nor does it guarantee any particular outcome in respect of using the code on the smart contract.*

2. Background

2.1. Scope

During the audit, MLabs has inspected the code contained in the provided files and attempted to locate problems that can be found in the following categories:

1. Unclear or wrong specifications, that could lead to unwanted behaviour.
2. Wrongful implementation.
3. Vulnerabilities that can be leveraged by an attacker.
4. Code quality concerns and comments.

Where possible, MLabs have provided recommendations to address the relevant issues.

2.2. Parties Involved in Audit Process

In this document, we will refer to the MLabs Audit Team as *MLabs*. This clarification is made to avoid confusion with the MLabs Development Team. Although MLabs (the company) has been involved in both the development and audit of the protocol, the two teams are separate, and distinct - with no conflict of interest.

2.3. Methodology

2.3.1. Information

MLabs analysed the validator and minting scripts from the Indigo git remote IndigoProtocol/smart-contracts starting with commit `bcf9ed05`. The base commit was later updated to commit `82e69a21` at the request of Indigo.

During the audit process, vulnerabilities discovered were posted to the repository as "GitHub Issues", and tracked under one overarching ticket, namely *Audit Issue #439*. Some of the findings were accompanied by

additional tests or proofs of behaviour, which were committed to the repository as separate branches.

2.3.1.1. Audited File Checksums

The following checksums are those of files captured by commit `bcf9ed05`, and were generated using the following `sha256` binary:

```
$ sha256sum --version
sha256sum (GNU coreutils) 9.0
```

The *checksums* of the smart-contract implementation code are:

```
4927...d023  Indigo/Contracts/Liquidity/OnChain.hs
52a1...26b0  Indigo/Contracts/Liquidity/Common.hs
f7af...1cb5  Indigo/Contracts/Governance/Gov/OnChain.hs
c0fb...ee91  Indigo/Contracts/Governance/Gov/Common.hs
a396...14c9  Indigo/Contracts/Governance/Execute/OnChain.hs
f654...8258  Indigo/Contracts/Governance/Execute/Common.hs
b2af...58c5  Indigo/Contracts/Governance/VersionRegistry/OnChain.hs
3733...6fa1  Indigo/Contracts/Governance/VersionRegistry/Common.hs
b658...6e38  Indigo/Contracts/Governance/Poll/OnChain.hs
47b6...ed0e  Indigo/Contracts/Governance/Poll/Common.hs
a64a...ffc4  Indigo/Contracts/Oracle/OnChain.hs
7f9a...1d2d  Indigo/Contracts/Oracle/Common.hs
a067...cd42  Indigo/Contracts/Collector/OnChain.hs
525d...8b58  Indigo/Contracts/Collector/Common.hs
34a5...8846  Indigo/Contracts/Treasury/OnChain.hs
ef0e...5cd4  Indigo/Contracts/Treasury/Common.hs
73ba...444c  Indigo/Contracts/Staking/OnChain.hs
3a2c...9e9d  Indigo/Contracts/Staking/Common.hs
355c...1192  Indigo/Contracts/CDP/OnChain.hs
ab99...a4f8  Indigo/Contracts/CDP/Common.hs
0808...5b24  Indigo/Contracts/Helpers.hs
3d9c...1465  Indigo/Contracts/StabilityPool/OnChain.hs
b697...aed3  Indigo/Contracts/StabilityPool/Common.hs
fb9b...86b7  Indigo/Utils/Spooky/Helpers.hs
874d...82d9  Indigo/Utils/Spooky.hs
dcd1...59c2  Indigo/Utils/Helpers.hs
0eba...c5df  Indigo/Data/Decimal.hs
f91a...a896  Indigo/Data/Token.hs
```

The *checksum* for the provided documentation is:

```
e4a1...c450  Indigo-Yellow-Paper.pdf
```

2.3.2. Audit Timeline

During the first sprint, MLabs conducted an exploratory audit of the protocol, examining all of the provided modules and documentation to familiarize themselves with the codebase and the protocol. This was followed by a structured exploration phase in sprints two through four, which focused on individual smart contract vulnerability exploration and penetration testing.

In the final two sprints, the findings were organized into the audit report. This task was done concurrently with reviewing some of the proposed fixes and exploring additional vulnerabilities. The review of the proposed fixes was not originally part of the project scope, but was undertaken as a good faith effort. However, the MLabs Audit Team cannot guarantee that the proposed fixes do not introduce new vulnerabilities when considered in the context of the overall protocol implementation.

2.3.3. Metrics

2.3.3.1. CVSS

The audit used the Common Vulnerability Scoring System and the NVD Calculator to provide a standardised measure for the severity of the identified vulnerabilities. Although MLabs Audit Team recognises that some of the parameters of the tools may not be relevant for the audit of a Cardano protocol, the team believes that using a standard is still valuable in providing a more unbiased severity metric for the findings.

2.3.3.2. Severity Levels

The aforementioned CVSS calculations were then benchmarked using the CVSS-Scale metric, receiving a grade spanning from `Low` to `Critical`. This additional metric allows for an easier, human understandable grading, whilst leveraging the CVSS standardised format.

3. Audit

3.1. Executive Summary

The audit findings can be categorised as the following vulnerability types:

1. other-redeemer ×2
2. utxo-contention ×2
3. cheap-spam ×2
4. locked-ada ×1
5. missing-incentive ×1
6. bad-incentive ×1
7. multiple-satisfaction ×1
8. incorrect-logic ×4
9. insufficient-tests ×3
10. incorrect-documentation ×1
11. insufficient-documentation ×1
12. unbounded-protocol-datum ×1
13. foreign-utxo-tokens ×1

3.1.1. Code Coverage

The entirety of the codebase was audited making use of hypotheses formulated on the technical specification, together with speculative analysis based on vulnerability types.

3.2. Vulnerabilities

Critical Severity Vulnerabilities

3.2.1. Incorrect validation of *CDP* liquidation transactions allows burning arbitrary amount of *iAsset* from the Stability Pool

Status: *Fixed*

Severity	CVSS	Vulnerability type
Critical	9.5	incorrect-logic

3.2.1.1. Description

Incorrect validation of *CDP* liquidation transactions allows anyone to burn an arbitrary amount of *iAsset* from the Stability Pool.

Frozen *CDPs* are eligible for "liquidation" by a Stability Pool associated with the same iAsset. *CDP* liquidation transactions come in two forms:

1. Full liquidation transactions that liquidate all of the minted *iAssets* associated with a *CDP.*
2. Partial liquidation transactions that liquidate only a part of the minted *iAssets* associated with a *CDP.*

An incorrect validation of full liquidation transactions was identified in the function:

`Indigo.Contracts.CDP.OnChain.validateLiquidateCDP,`

allowing anyone to burn arbitrary amount of *iAssets* from the Stability Pool regardless of the amount of *iAssets* actually minted by the *CDP* being liquidated.

For example, If a frozen *CDP* has 1 minted *iAsset*, and the Stability Pool has 1,000,000 *iAsset* staked. Then a full liquidation transaction can be submitted that burns all 1,000,000 *iAssets* from the Stability Pool.

3.2.1.2. Recommendation

The validation of full liquidation transactions should include a check to ensure that the burned *iAsset* amount does not exceed the amount of *iAsset* minted by the *CDP* being liquidated.

3.2.1.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Audit - Liquidity Pool Liquidation Exploit #441
5. Liquidation exploit test
6. CDP validator
7. CDP validateLiquidateCDP

3.2.2. Incorrect validation of poll shard merging allows stealing of poll tokens

Status: *Fixed*

Severity	CVSS	Vulnerability type
Critical	9.5	incorrect-logic

3.2.2.1. Description

Incorrect validation of poll shard merging allows anyone to steal all poll tokens and *Ada* from a poll's shards, while invalidating the poll. This results in a total compromise of the protocol.

Multiple polls can be active at the same time, each with their own set of poll shards. Incorrect validation allows for the merge operation to be performed for one poll with a set of poll shards from an entirely separate poll. This results in no restrictions being placed on the output location of the poll shards' values, allowing an attacker to steal the poll tokens and *Ada* from the poll shards. Further, the poll to which the poll shards actually belonged is now invalidated and concludes through expiration, as its poll shards cannot be tallied.

With poll tokens under an attacker's control, they can create more "counterfeit" poll tokens, and fake poll shards and poll managers to make any changes they desire to the protocol.

3.2.2.2. Recommendation

A PR with the recommended fix was provided during the audit, and contains corresponding test cases.

3.2.2.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Audit Merge Shards Exploit #450
5. Fix for #450 - #471
6. Vulnerable validators:
    1. `validateMergeShards`
    2. `validateMergeShardsManager`
7. `validatePoll`
8. `validateCreateProposal`
9. `validatePollManager`
10. Poll token asset class
11. Exploit transaction example
12. Exploit test case

3.2.3. Incorrect validation of Proposal execution allows stealing of the Upgrade Token

Status: *Fixed*

Severity	CVSS	Vulnerability types
Critical	9.3	multiple-satisfaction

3.2.3.1. Description

A user can steal an upgrade token by executing two similar proposals in the same transaction.

A passed proposal results in an $eUTxO$ with an `Upgrade Token`. Spending this $eUTxO$ - that is executing a proposal - is governed by the `Execute` validator. There are pairs of `Upgrade Tokens` for which the validation logic allows for the pair to be executed simultaneously in a single transaction. For example two `Text Proposals` or two `Modify Protocol Params` proposals for the same parameters. This is due to insufficient validation logic in the `validateExecute` function checks. This allows for a transaction spending two `Upgrade Tokens`, burning one of them as required by the validator, but sending the remaining one to a wallet address of an attacker.

An actor in possession of an `Upgrade Token` could execute arbitrary proposals.

3.2.3.2. Recommendation

The insufficient check applies to the `Execute` validator. The validation of proposal execution should check for the unique update token in the inputs. The check should happen in the `validateExecute` function.

3.2.3.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Audit - Proposal Exploit #466
5. Proposal exploit test
6. Execute validator
7. Proposal types
8. Passed proposal upgrade token

Medium Severity Vulnerabilities

3.2.4. Missing check in the *CDP* validator to ensure that inputs from other scripts are spent with the correct redeemer is unsafe

Status: *Not Fixed*

Severity	CVSS	Vulnerability type
Low	5.7	other-redeemer

3.2.4.1. Description

The *Other Redeemer* vulnerability was identified in the *CDP* validator and specifically the:

`Indigo.Contracts.CDP.OnChain.validateLiquidateCDP`

function that validates spending from the *CDP* validator with the *CDP's* `Liquidate` redeemer. The function implicitly relies that the transaction spends an input from the Stability Pool validator with the Stability Pool's `LiquidateCDP` redeemer, but performs no explicit checks to ensure it, which is considered an unsafe practice.

In the future, if the Stability Pool validator is modified it could inadvertently introduce a vulnerability.

In the pre-audit engagement, a vulnerability of this type was indeed found and reported.

3.2.4.2. Recommendation

Add an explicit check in the `Indigo.Contracts.CDP.OnChain.validateLiquidateCDP` function using the `Indigo.Utils.Helpers.usesSpendRedeemer` function to assert that the transaction spends an input from the Stability Pool validator using the correct Stability Pool's `LiquidateCDP` redeemer.

3.2.4.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. CDP validator
5. CDP validateLiquidateCDP
6. Stability Pool validator
7. Stability Pool validateLiquidateCDP

3.2.5. Lack of restriction on collector value may halt distribution of protocol fees

Status: *Fixed*

Severity	CVSS	Vulnerability type
Medium	6.6	foreign-utxo-tokens

3.2.5.1. Description

`Collector` outputs allowed by the protocol may contain arbitrary value. This may allow an attacker to block the distribution of protocol fees, and is allowed under two circumstances.

3.2.5.1.1. Disclaimer

These exploits are hypothetical and there are no tests to demonstrate them.

3.2.5.2. Recommendation

1. In any automation involving collector outputs, filter out collectors with large numbers of tokens so they are not used to create transactions which may exceed the execution unit limit.
2. In the staking manager's `Distribute` validator, do not allow *non-Ada* tokens in the collector outputs.

3.2.5.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Audit - Collector value overflow #491
5. Vulnerable validators
    1. `validateCollectorScript`
    2. `validateDistribute`
6. Staking manager

3.2.6. Creation of duplicate *iAssets* results in invalid protocol state

Status: *Not Fixed*

Severity	CVSS	Vulnerability type
Medium	4.3	incorrect-logic

3.2.6.1. Description

The Indigo Protocol tracks a "whitelist" and a "blacklist" of currently valid, respectively invalid *iAssets*. A valid *iAsset* has a Stability Pool, can be minted through *CDPs*, etc., while an invalid *iAsset* cannot. The whitelist of *iAssets* is tracked through all *iAsset* outputs with *an Oracle*. The blacklist of *iAssets* is everything else, including the list of all *iAsset* outputs without an Oracle.

The Indigo Protocol does not prevent *iAssets* with the same `name` from being whitelisted through the governance system (see "Governance" in the Indigo Paper), but also expects *iAsset* names to be unique identifiers, for example to maintain that exactly one Stability Pool exists per *iAsset*. If two *iAssets* with the same name were to be created, this would result in two Stability Pools for the same *iAsset* name. **This would represent an invalid state for the protocol with undetermined behaviour**.

3.2.6.2. Recommendation

There are two options:

1. Disallow a proposal from being created which proposes an *iAsset* with the same name as an existing *iAsset*.
2. For any proposal meeting the above criteria, disallow it from being executed.

The first option is preferable, but performance constraints may require the second option.

In either case, this would require a list of the current *iAsset* name to be maintained. The list could be held in the datum of the Governance output. The list would need to be updated during the execution of `ProposeAsset` and `MigrateAsset` proposals, and the Governance output could be used as a reference input to acquire the current whitelist in order to check for a duplicate *iAsset*.

3.2.6.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Paper
4. Audit - Protocol allows but does not account for duplicate *iAssets* #481
5. Relevant validators:
    1. `validateExecute`
    2. `validateCreateProposal`

3.2.7. Missing check in the Poll validator to ensure that inputs from other scripts are spent with the correct redeemer is unsafe

Status: *Fixed*

Severity	CVSS	Vulnerability type
Medium	5.7	other-redeemer

3.2.7.1. Description

The *Other Redeemer* vulnerability was identified in the Poll validator and specifically the

`Indigo.Contracts.Governance.Poll.OnChain.validateVote` function that validates spending from the Poll validator with the Poll's `Vote` redeemer. The function implicitly relies that the transaction spends an input from the Staking validator with the Staking's `Lock` redeemer, but performs no explicit checks to ensure it, which is considered an unsafe practice.

In the future, if the Poll validator is modified it could inadvertently introduce a vulnerability.

In the pre-audit engagement, a vulnerability of this type was indeed found and reported.

3.2.7.2. Recommendation

Add an explicit check in the `Indigo.Contracts.Governance.Poll.OnChain.validateVote` function to assert that the transaction spends an input from the Staking validator using the correct redeemer.

3.2.7.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Poll validator
5. Poll validateVote
6. Staking validator
7. Staking validateLock

3.2.8. Contention and low-cost transactions in the Staking Manager may cause the protocol to stall

Status: *Fixed*

Severity	CVSS	Vulnerability type
Medium	6.9	utxo-contention cheap-spam

3.2.8.1. Description

The Indigo Protocol has a single output known as the Staking Manager which keeps track of the status of the Indy stakers. All staking must be done through the Staking Manager, which creates a single point of contention; this is acknowledged in the Indigo Yellowpaper.

The `Distribute` validator of the staking manager allows repeatable, low-cost transactions whenever there is *Ada* in a collector output. This enables malicious actors to spam the Staking Manager UTxO at low cost, effectively **stalling the protocol's ability to maintain its staking positions and distribute protocol fees**.

3.2.8.2. Recommendation

In the staking manager's distribute validator:

1. Require the collector inputs to transfer as much *Ada* as possible to the staking manager. This could also be done in the collector validator, but it is probably more efficient to do it in the staking manager.
2. Require a minimum amount to be transferred from the collectors to the staking manager, for example 2 *Ada*.

The first point stops attackers from using an existing collector with a large amount of protocol fees. The second point deters attackers from creating their own collectors with small amounts of *Ada*, just to use those collectors to pass a few *Lovelace* at a time to the staking manager.

3.2.8.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Audit - Staking Manager allows cheap spam #502
5. Vulnerable validators:
    1. `validateDistribute`
    2. `validateCollectorScript`

Low Severity Vulnerabilities

3.2.9. Potential Governance Power Misuse

Status: *Not Fixed*

Severity	CVSS	Vulnerability type
Low	4.4	bad-incentive

3.2.9.1. Description

The Governance system is an essential and powerful aspect of the Indigo protocol. Many aspects of Indigo can be changed through the use of Governance enabled powers, for example: the *MCR* of an *iAsset*, validators for the various contracts (like the Stability Pool), etc. However, Governance enabled powers could be misused.

3.2.9.2. Recommendation

While there is no clear solution to this problem, we can mitigate it by having more control over INDY tokens, which are used for voting on the Indigo protocol.

3.2.9.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Corresponding GH issue
4. Governance contracts

3.2.10. No lower bound for *CDP* positions can dis-incentivize *CDP* liquidation

Status: *Fixed*

Severity	CVSS	Vulnerability type
Low	4.6	missing-incentive

3.2.10.1. Description

Indigo protocol allows creation of arbitrarily small *CDPs* which could disincentivize freezing and liquidation of such *CDPs* and in turn compromise the minimal collateral ratio goal of the Indigo protocol.

This happens when the collateral surplus contained within a *CDP* that liquidators hope to acquire is smaller than the total transaction cost of freezing and liquidating the *CDP*. Effectively, the liquidation would incur a net loss on the Stability Provider that is trying to liquidate the *CDP*.

3.2.10.2. Recommendation

Before we discuss the solution for the problem, we think it is worth looking into what we are trying to achieve. Currently, in some circumstances the protocol misses to implement an incentive to liquidate a *CDP* holding some 'small' amounts. We want to account for such circumstances by implementing a proper incentive to liquidate

any under collateralized *CDP*. However, there's no need to incentivize *everyone* to liquidate since the *CDP* will eventually be liquidated by the users who are incentivized to do so.

The solution for this missing incentive problem requires careful balance between:

1. Users who own some substantial amount of *iAsset* present in the Stability Pool,
2. Typical percent of Stability Pool depletion when liquidating a *CDP*.
3. The price at which liquidation will occur.

Incentivizing every user of the Stability Pool is not particularly good, because some users may own *iAssets* in the Stability Pool that are close to zero (for example: 0.000001%), hence their reward when liquidating a *CDP* will also be close to zero (R * 0.000001) unless the *CDP* whose liquidation is performed contains massive amount of collateral. Therefore, if we want to incentivize such users to liquidate the *CDPs* then the lower bound for liquidating the *CDP* will be very high, which means very few people will be able to open the *CDP*.

Hence, we want to find a good balance between users that own at least some substantial amount of *iAssets* present in the Stability Pool (something like: 1%) and create a lower bound based on that. Such that every user who owns at least 1% of the Stability Pool is guaranteed to have a profit when liquidating a *CDP*.

One way to achieve this is by using the following formula:

```
R - (P + F) > 0
```

Where

- N = amount of *iAssets*,
- F = *Tx* Fee for liquidation of *CDP* + *Tx* Fee for Freezing the *CDP*.
- P = Price of *iAsset* at which liquidation will occur * Depletion percent of Stability Pool * N,
- R = *MCR* * Price of the *iAsset* * Min percent of Stability Pool a user should own * N.

Now, we solve for N, by substituting appropriate values in the equation.

3.2.10.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. CDP validator

3.2.11. Locked *Ada* values in multiple protocol UTxOs

Status: *Not Fixed*

Severity	CVSS	Vulnerability type
Low	4.3	locked-ada

3.2.11.1. Description

The *Locked-Ada* vulnerability was identified in multiple protocol UTxOs.

The associated validators, namely Version Registry, Poll Manager and Stability Pool, produce outputs that can become indefinitely locked once they become obsolete due to the expiry mechanisms implemented in the protocol.

This impacts financial sustainability of the protocol but also compromises the availability of *Ada* that ought to be circulating in Cardano.

3.2.11.2. Recommendation

Each protocol UTxO that is planned for obsolescence, should be redeemable preferably by the same user that bare the original *Ada* cost of producing the output.

General approach to doing so is to extend the UTxO with a datum that contains

1. Authentication information that the redeeming user must provide in the redeeming transaction (public key hash, authentication token, etc.)
2. Expiry time denoting when the output min utxo *Ada* value can be redeemed by the user

And then, extend the validator with a special redeemer to enable such redeeming transactions. The validator should validate when the UTxO is spent with the appropriate authentication information as indicated by the datum, after the expiry deadline had passed (also indicated in the datum).

3.2.11.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Poll Manager validator
5. Stability Pool validator
6. Version Registry validator
7. Value locked by protocol #477
8. Minimum *Ada* value requirement

3.2.12. Contention in the Stability Pool may cause the protocol to stall

Status: *Fixed*

Severity	CVSS	Vulnerability type
Low	4.2	utxo-contention cheap-spam

3.2.12.1. Description

Indigo protocol uses a single *UTxO* for every Stability Pool for which the participants compete to spend when submitting transactions conveying Stability Pool actions. This results in only one Stability Pool action being accepted per slot, while others get discarded. This issue is already acknowledged by Indigo as described in their yellow paper.

Additionally, a *Cheap Spam* vulnerability was identified in the Stability Pool validator that allows idempotent `AdjustAccount` actions to be performed in the Stability Pool for a very low cost.

Compounding both vulnerabilities, it enables malicious actors to cheaply spam the contended Stability Pool UTxO effectively stalling the protocol's ability to liquidate *CDPs* and other key actions.

3.2.12.2. Recommendation

To address the Cheap Spam vulnerability, the protocol can introduce an additional fee to such actions or one of the following changes:

1. Require initial deposit of corresponding *iAsset* when creating an Account in the Stability Pool,
2. Require delay when Adjusting Stability Pool Accounts to make such actions accepted less often.

To address the UTxO contention issue however would require a more significant design change. Here we propose a design that could alleviate the contention problem.

The contention exists because the Indigo design couples *CDP* liquidation with the Stability Pool, as each liquidation transaction requires a single input from the Stability Pool UTxO. If we agree that the overall goal is to maintain a global *MCR* (minimal collateral ratio) for a certain iAsset, it seems like we can achieve this without the Stability Pool as it stands now, but any protocol mechanism that can restore MCR.

The *CDP* 'takeover' protocol mechanic simply allows to change ownership to a *CDP* if that *CDP* is deemed 'unhealthy' (i.e. became under-collateralized).

1. Anyone can deposit *Ada* or burn the corresponding *iAssets* with such a 'unhealthy' *CDP* only IFF such an action makes the *CDP* 'healthy' again (i.e. over-collateralized),
2. In the same transaction, a new *CDP* owner can be set effectively enabling 'takeover',
3. Once the *CDP* is 'healthy' the *MCR* is re-established,
4. Full or partial liquidation would be reduced to closing a *CDP* or withdrawing *Ada_s by burning _iAssets*.

Few implications of such design:

1. Doesn't require a special 'Frozen' state for *CDP*.,
2. Completely decouples liquidation from the very notion of a Stability Pool,
3. All protocol participants are in direct competition to perform 'takeovers',
4. There's no global contention point as *N CDPs* can be taken over with *N* parallel transactions.
5. If nobody acts on an 'unhealthy' *CDP* it can naturally recover if the price turns favourable again.

The Stability Pool could be reimagined as a separate Cardano dApp that is managed and operated independently. It is possible that a third part could employ some version of the current Stability Pool to operate liquidation efforts, staking and reward distributions. Such Stability Pool operators could additionally add 'fee' mechanisms to account for running the automation infrastructure and transaction costs, and generally cover for the 'yield' service.

3.2.12.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Stability Pool validator
5. Stability Pool validateAdjustAccount

3.2.13. It is possible for the staking position datum to grow infinitely, resulting in the datum not fitting into the Tx

Status: *Not Fixed*

Severity	CVSS	Vulnerability type
Low	3.2	incorrect-logic

3.2.13.1. Description

The staking position datum contains a `Map Integer (Integer, POSIXTime)`. This `Map` stores all the proposals that a user has voted on, and is unbounded - i.e. no upper bound exists for the number of proposals a user can vote on. The unbounded property may result in the datum becoming too large to fit in a Cardano transaction.

To cause this behaviour, a user would need to vote on *many* simultaneous proposals. To avoid such behaviour, we recommend benchmarking, and potentially enforcing the number of proposals that a user can vote on simultaneously.

3.2.13.2. Recommendation

An upper limit should be set on the number of proposals a user can vote on simultaneously. It is possible to derive this upper bound from the benchmarks.

3.2.13.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Staking Position Datum
4. GH Issue mentioning this problem

Unclassified Severity Vulnerabilities

3.2.14. Insufficient tests for reward collection and distribution

Status: *Fixed*

Severity	CVSS	Vulnerability types
None	N/A	insufficient-tests

3.2.14.1. Description

There are no tests asserting the correctness of collecting and distributing protocol fees. A lack of tests means the behaviour of the code is not fully determined and cannot be relied upon. The logic behind reward calculation is never tested, and the following sequences of transactions are untested:

- Distribute from collector to staking manager;
- Distribute from collector to staking manager;
- Performing the above with multiple active staking positions;
- Adding protocol fees to an existing collector through a different validation path.

3.2.14.2. Recommendation

Add relevant tests to increase the confidence that the implementation indeed works as specified. Tests should cover the scenarios above, covering the positive case and also failing as expected for incorrect reward calculation or unauthorised spend attempts. Tests should attempt to steal the collected fees, spending `Collector` owned $eUTxO$s to a private address.

Two PRs were opened which attempt to address these issues, however they were not considered as a part of the audit:

1. Collector Test Suite #479
2. Protocol Fee Tests #480

3.2.14.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Audit - No Collector fee tests #446
5. Audit - No staking reward tests #469
6. Collector Test Suite #479
7. Protocol Fee Tests #480
8. Staking validator
9. Relevant validator parts:
    1. `validateDistribute`
    2. `validateUnstake`
    3. `validateAdjustStakedAmount`
    4. `validateCollectorScript`

3.2.15. Lack of tests for helper functions

Status: *Not Fixed*

Severity	CVSS	Vulnerability type
None	N/A	insufficient-tests

3.2.15.1. Description

There are currently no unit tests for the helper functions in `Indigo.Utils.Helpers`. A lack of tests means the behaviour of the code is not fully determined and cannot be relied upon. These functions are used extensively throughout the codebase so they should be well tested. They are tested indirectly through the transaction tests, but this method is more likely to miss edge cases.

3.2.15.2. Recommendation

Add unit tests for the helper functions in `Indigo.Utils.Helpers`.

3.2.15.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. `Indigo.Utils.Helpers`
5. Audit - No helper function unit tests #470

3.2.16. Documentation Enhancements

Status: *Fixed*

Severity	CVSS RATING	vulnerability types
None	0.0	insufficient-documentation incorrect-documentation

3.2.16.1. Description

Proper documentation is crucial to avoid introducing new bugs in the future.

Codebase generally contains few comments, but the consistent naming and module structure together with self-commenting validator checks make it a lesser problem.

Following are issues that could be improved still:

1. `RecordEpochToScaleToSum` endpoint is missing from the yellow-paper.
2. Minimal collateral ratio is passed in code as a percentage. This is undocumented both in the types and functions using it.
3. `sessSnapshot` field is confusingly named. Elsewhere "snapshot" refers to values with `StabilityPoolSnapshot` type.
4. The implementation of the tests is poorly documented.

3.2.16.2. Recommended Fix

We recommend expanding *section 3.2.4* of the Indigo Yellowpaper with point on `RecordEpochToScaleToSum` redeemer. The `overCollateralized` function should have its `ratio` argument documented. Also worth considering is a `newtype` for decimals understood as percentages. Consider renaming the `sessSnapshot` field for consistency. Finally improving the documentation of the test suite would go a long way in making sure the tests cover all cases.

3.2.16.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. #478 - Audit - Documentation enhancements
5. `RecordEpochToScaleToSum` endpoint
6. `overCollateralized` function
7. `sessSnapshot` field

3.2.17. Missing integration tests with the real Cardano environment risks unexpected issues during deployment

Status: *Not Fixed*

Severity	CVSS	Vulnerability type
None	N/A	insufficient-tests

3.2.17.1. Description

Currently, Indigo's test suite uses the Plutus Simple Model testing framework, which is similar to emulator traces and is used to estimate program resource usage and assert correctness conveniently and quickly.

However, testing and running Plutus programs in a real Cardano network is essential as various issues can arise when dealing with not deterministic parts of the blockchain like contention, slot timing, block validation, etc.

3.2.17.2. Recommendation

Test the Indigo Plutus program behaviour using any of the available integration test frameworks, specifically Plutip or using CTL integration with Plutip.

3.2.17.3. References

1. CVSS 3.1 Qualitative Severity Rating Scale
2. MLabs vulnerability classification
3. Indigo Yellow Paper
4. Plutip
5. Plutus Simple Model
6. cardano-transaction-library
7. CTL integration with Plutip

4. Conclusion

During the six week period MLabs inspected the on-chain code of the Indigo protocol revealing 17 vulnerabilities, 3 of them critical. These enable a malicious actor to burn assets of a Stability Pool or compromise the governance part of the protocol by creating fake votes or executing arbitrary proposals. We also identified vulnerabilities of medium severity including contention issues, missing incentives and ones that (while not directly exploitable) remain a risk factor with the potential to compound in the event of further protocol changes. Lastly we pinpointed holes in the test suite, documentation shortcomings and cases of value locked in the protocol due to minimum *Ada* requirement. The list is not exhaustive, containing only issues identified by the team in the limited timeframe of the audit. MLabs has summarised the findings in the given report, outlining potential dangers and offering recommendations. The report is provided without a guarantee of any kind, but we hope it proves useful. Already before the audit conclusion, the Indigo development team has responded with fixes. MLabs has provided feedback to some of these, but a thorough review needs to be carried out by the responsible Indigo developers.

5. Appendix

5.1. Vulnerability types

The following list of vulnerability types represents a list of commonly found vulnerabilities in Cardano smart contract protocol designs or implementations. The list of types is actively updated and added to as new vulnerabilities are found.

5.1.1. Other redeemer

ID: other-redeemer

Test: Transaction can avoid some checks when it can successfully spend a UTxO or mint a token with a redeemer that some script logic didn't expect to be used.

Property: A validator/policy should check explicitly whether the 'other' validator/policy is invoked with the expected redeemer.

Impacts:

- Bypassing checks

5.1.2. Other token name

ID: other-token-names

Test: Transaction can mint additional tokens with some 'other' token name of 'own' currency alongside the intended token name.

Property: A policy should check that the total value minted of their 'own' currency symbol doesn't include unintended token names.

Impacts:

- Stealing protocol tokens
- Unauthorised protocol actions

Example:

A common coding pattern that introduces such a vulnerability can be observed in the following excerpt:

```
vulnPolicy rmr ctx = do
  ...
  assetClassValueOf txInfoMint ownAssetClass == someQuantity
  ...
```

The recommended coding pattern to use in order to prevent such a vulnerability can be observed in the following excerpt:

```
safePolicy rmr ctx = do
  ...
  txInfoMint == (assetClassValue ownAssetClass someQuantity)
  ...
```

5.1.3. Unbounded Protocol datum

ID: unbounded-protocol-datum

Test: Transaction can create protocol UTxOs with increasingly bigger protocol datums.

Property: A protocol should ensure that all protocol datums are bounded within reasonable limits.

Impacts:

- Script XU and/or size overflow
- Unspendable outputs
- Protocol halting

Example:

A common design pattern that introduces such vulnerability can be observed in the following excerpt:

```
data MyDatum = Foo {
  users :: [String],
  userToPkh :: Map String PubKeyHash
}
```

If the protocol allows these datums to grow indefinitely, eventually XU and/or size limits imposed by the Plutus interpreter will be reached, rendering the output unspendable.

The recommended design patterns is either to limit the growth of such datums in validators/policies or to split the datum across different outputs.

5.1.4. Arbitrary UTxO datum

ID: arbitrary-utxo-datum

Test: Transaction can create protocol UTxOs with arbitrary datums.

Property: A protocol should ensure that all protocol UTxOs hold intended datums.

Impacts:

- Script XU overflow
- Unspendable outputs
- Protocol halting

5.1.5. Unbounded protocol value

ID: unbounded-protocol-value

Test: Transaction can create increasingly more protocol tokens in protocol UTxOs.

Property: A protocol should ensure that protocol values held in protocol UTxOs are bounded within reasonable limits.

Impacts:

- Script XU overflow
- Unspendable outputs
- Protocol halting

5.1.6. Foreign UTxO tokens

ID: foreign-utxo-tokens

Test: Transaction can create protocol UTxOs with foreign tokens attached alongside the protocol tokens.

Property: A protocol should ensure that protocol UTxOs only hold the tokens used by the protocol.

Impacts:

- Script XU overflow
- Unspendable outputs
- Protocol halting

5.1.7. Multiple satisfaction

ID: multiple-satisfaction

Test: Transaction can spend multiple UTxOs from a validator by satisfying burning and/or paying requirements for a single input while paying the rest of the unaccounted input value to a foreign address.

Property: A validator/policy should ensure that all burning and paying requirements consider all relevant inputs in aggregate.

Impacts:

- Stealing protocol tokens
- Unauthorised protocol actions
- Integrity

Example:

A common coding pattern that introduces such a vulnerability can be observed in the following excerpt:

```
vulnValidator _ _ ctx =
  ownInput ← findOwnInput ctx
  ownOutput ← findContinuingOutput ctx
  traceIfFalse "Must continue tokens" (valueIn ownInput == valueIn ownOutput)
```

Imagine two outputs at `vulnValidator` holding the same values

A. `TxOut ($FOO x 1 + $ADA x 2)` B. `TxOut ($FOO x 1 + $ADA x 2)`

A transaction that spends both of these outputs can steal value from one spent output by simply paying `$FOO x 1 + $ADA x 2` to the 'correct' address of the `vulnValidator`, and paying the rest `$FOO x 1 + $ADA x 2` to an arbitrary address.

5.1.8. Locked Ada

ID: locked-ada

Test: Protocol locks Ada value indefinitely in obsolete validator outputs.

Property: Protocol should include mechanisms to enable redeeming any Ada value stored at obsolete validator outputs.

Impacts:

- Financial sustainability

- Cardano halting

5.1.9. Locked non Ada values

ID: locked-nonada-values

Test: Protocol indefinitely locks some non-Ada values that ought to be circulating in the economy.

Property: Protocol should include mechanisms to enable redeeming any non-Ada value stored at obsolete validator outputs.

Impacts:

- Financial sustainability
- Protocol halting

5.1.10. Missing UTxO authentication

ID: missing-utxo-authentication

Test: Transaction can perform a protocol action by spending or referencing an illegitimate output of a protocol validator.

Property: All spending and referencing of protocol outputs should be authenticated.

Impacts:

- Unauthorised protocol actions

Example:

Checking only for validator address and not checking for an authentication token.,

5.1.11. Missing incentive

ID: missing-incentive

Test: There is no incentive for users to participate in the protocol to maintain the intended goals of the protocol.

Property: All users in the Protocol should have an incentive to maintain the intended goals of the protocol

Impacts:

- Protocol stalling
- Protocol halting

5.1.12. Bad incentive

ID: bad-incentive

Test: There is an incentive for users to participate in the protocol that compromises the intended goals of the protocol.

Property: No users of the protocol should have an incentive to compromise the intended goals of the protocol.

Impacts:

- Protocol stalling
- Protocol halting

5.1.13. UTxO contention

ID: utxo-contention

Test: The protocol requires that transactions spend a globally shared UTxO(s) thereby introducing a contention point.

Property: The protocol should enable parallel transactions and contention-less global state management if possible.

Impacts:

- Protocol stalling
- Protocol halting

5.1.14. Cheap spam

ID: cheap-spam

Test: A transaction can introduce an idempotent or useless action/effect in the protocol for a low cost that can compromise protocol operations.

Property: The protocol should ensure that the cost for introducing a salient action is sufficient to deter spamming.

Severity increases when compounded with the `utxo-contention` vulnerability.

Impacts:

- Protocol stalling
- Protocol halting

5.1.15. Insufficient tests

ID: insufficient-tests

Test: There is piece of validation logic that tests do not attempt to verify.

Property: Every piece of validator code gets meaningfully executed during tests.

Impacts:

- Correctness

5.1.16. Incorrect documentation

ID: incorrect-documentation

Test: There is a mistake or something confusing in existing documentation.

Property: Everything documented is clear and correct.

Impacts:

- Correctness
- Maintainability

5.1.17. Insufficient documentation

ID: insufficient-documentation

Test: There is a lack of important documentation.

Property: Everything of importance is documented.

Impacts:

- Comprehension
- Correctness

5.2. Terminology and Abbreviations

Throughout the report, the following abbreviations and terms have been used:

- *Ada*: Cardano digital currency.
- *CDP*: collateralized debt position.
- *eUTxO*: extended unspent transaction output. Used interchangeably with *utxo* in the context of the report.
- *iAsset*: Indigo synthetic Asset.
- *Indy*: Indigo utility token.
- *Lovelace*: Cardano digital currency subdivision. One million *Lovelace* is equal to one *Ada*.
- *MCR*: minimum collateral ratio.
- *Staker*: a user currently staking Indy tokens int the Indigo protocol.
- *Tx*: transaction.
- *User*: a user of the Indigo protocol.