Binance zk-SNARKs Proof of Solvency Independent Technical Assessment

Feb 14, 2023

Repositories:

https://github.com/binance/zkmerkle-proof-of-solvency

Commit:

c1884aae22cd17af023ac4424b4e6623eb0ea9dd

References:

- Announcement
- How to Verify Your Account Balance on Binance
- How zk-SNARKs Improve Binance's Proof of Reserves System
- Proof of solvency - technical specification
- Having a safe CEX: proof of solvency and beyond

Authors:

Luciano Ciattaglia (l.ciattaglia@hacken.io)
Bartosz Barwikowski (b.barwikowski@hacken.io)
Yaroslav Bratashchuk (y.bratashchuk@hacken.io)
Sofiane Akermoun (s.akermoun@hacken.io)

---

1 Introduction

On February 10th, Binance revealed a new and improved Proof of Reserves (PoR) verification system that leverages the power of both Merkle Trees and zk-SNARKs which marks a significant upgrade from the previous system that solely relied on plain Merkle Trees.

The integration of zk-proofs enhances the system's security by addressing previous vulnerabilities, such as the possibility of negative balances in fake accounts while preserving user privacy during the verification process.

Hacken conducted an independent technical assessment on the recently released verification system and discovered a critical vulnerability that could lead to the creation of fake debt. This issue has been reported and promptly fixed.

2 Project Summary

In the project we identified 1 critical issue which allows to fake the total debt amount in the zero knowledge proof circuit, 1 medium severity issue and 2 other low severity issues. The critical and medium severity issues have been already fixed. However, any proof generated before those issues were fixed cannot be verified to be valid, as the critical one allowed for the total debt amount to be tampered. Although the proofs may appear to be valid, it is not possible to ensure that they were not modified due to the vulnerability. The other low severity issues are very unlikely to be abused and do not need to be addressed immediately.

The project has 1157 dependencies, all of them with checksum verification. There were found 42 vulnerabilities within all dependencies, with 16 of them having public exploits available. 22 with high severity and 20 with medium. None of the vulnerable functions are currently being used in the project.

It uses a forked version of gnark made on Sep 2022 for the circuits and Poseidon with BN254 hash function to hash the user information and the Sparse Merkle Tree (SMT) data structure to store the hashes. The SMT is implemented using the BSMT library, and its maximum depth is set to be 28, which means that this Proof Of Solvency approach may be used for more than 250M users.

The code quality is clean and organized.

The README.md contains instructions on how to run tools one by one, and motivation behind the circuits is also detailed.

The Panic is used for main function error handling, so all the tools crash with a stack trace in case of an error.

The sample user data (balance sheets) is provided in order to test tools manually. There is a way to fetch (probably production) Postgres configuration from the AWS storage if the remote_password_config flag is provided to the tools that use Postgres.

There was a function to generate fake accounts in the witness service, which was commented out but still left in the code (probably for manual testing purposes). EmptyAccounts are generated in the witness, and they are used in case the last account's batch size is less than 864.

The git history log has been modified several times, and as a result, the git metadata is mixed in some places.

3 Vulnerabilities

3.1 [Critical][Fixed] TotalDebt manipulation vulnerability caused by overflow of BasePrice

The code contains a critical error that enables it to create false user debt, reducing the number of assets needed. This occurs because there is a method to circumvent the assertion that checks if the user's debt exceeds their equity.

There is a bug in the system that allows for bypassing because the BasePrice parameter can be set to an extremely high value. This vulnerability exists because the parameter is not checked for value range, making it easy to manipulate. Although the BasePrice is publicly accessible, it would be simple to identify if it has been changed. However, there is a method to modify the BasePrice in a way that would be undetectable by other users, making it possible to exploit the vulnerability without being detected.

As an optimization, the code splits all the users into batches, each with 864 users. The batches are linked with each other by sharing information about assets and the cryptographic hashes. Each exchange asset is shared using three variables: TotalEquity, TotalDebt and BasePrice. The hash of asset is calculated from one big integer, which is calculated using the following formula:

$$TotalEquity * 2^{128} + TotalDebt * 2^{64} + BasePrice$$

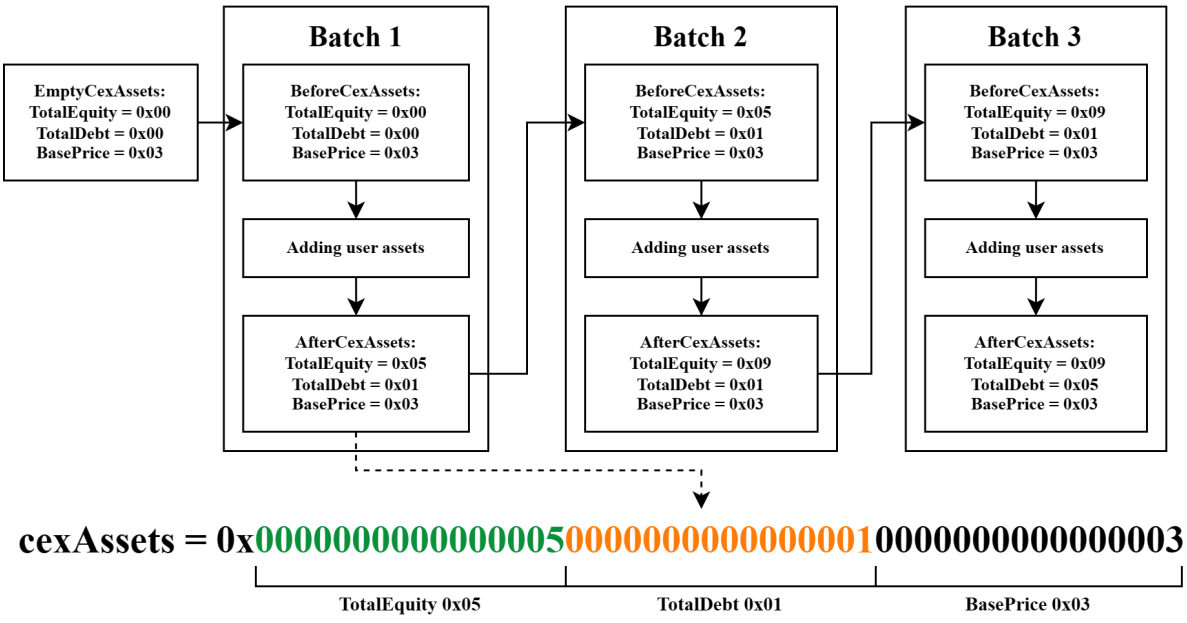
The problem is, that in the code responsible for doing these calculations, only *TotalEquity* and *TotalDebt* are checked if they are greater or equal to 0 and lower than 2^{64}. The value of BasePrice is not being checked, which allows to set it to value higher than $2^{64} - 1$ which makes it possible to modify the value of *TotalDebt* and *TotalEquity*. Because of that, it is possible to generate the same value for different parameters, for example both *TotalDebt* = 2, *BasePrice* = 3 and *TotalDebt* = 1, *BasePrice* = $2^{64} + 3$ will have value of $2 * 2^{64} + 3$. The source code responsible for this calculations:

```
// verify whether beforeCexAssetsCommitment is computed correctly
for i := 0; i < len(b.BeforeCexAssets); i++ {
        CheckValueInRange(api, b.BeforeCexAssets[i].TotalEquity)
        CheckValueInRange(api, b.BeforeCexAssets[i].TotalDebt)
        cexAssets[i] = api.Add(api.Mul(b.BeforeCexAssets[i].TotalEquity, utils.Uint64MaxValueFrSquare),
                api.Mul(b.BeforeCexAssets[i].TotalDebt, utils.Uint64MaxValueFr), b.BeforeCexAssets[i].BasePrice)
        afterCexAssets[i] = b.BeforeCexAssets[i]
}
actualCexAssetsCommitment := poseidon.Poseidon(api, cexAssets...)
api.AssertIsEqual(b.BeforeCEXAssetsCommitment, actualCexAssetsCommitment)
```

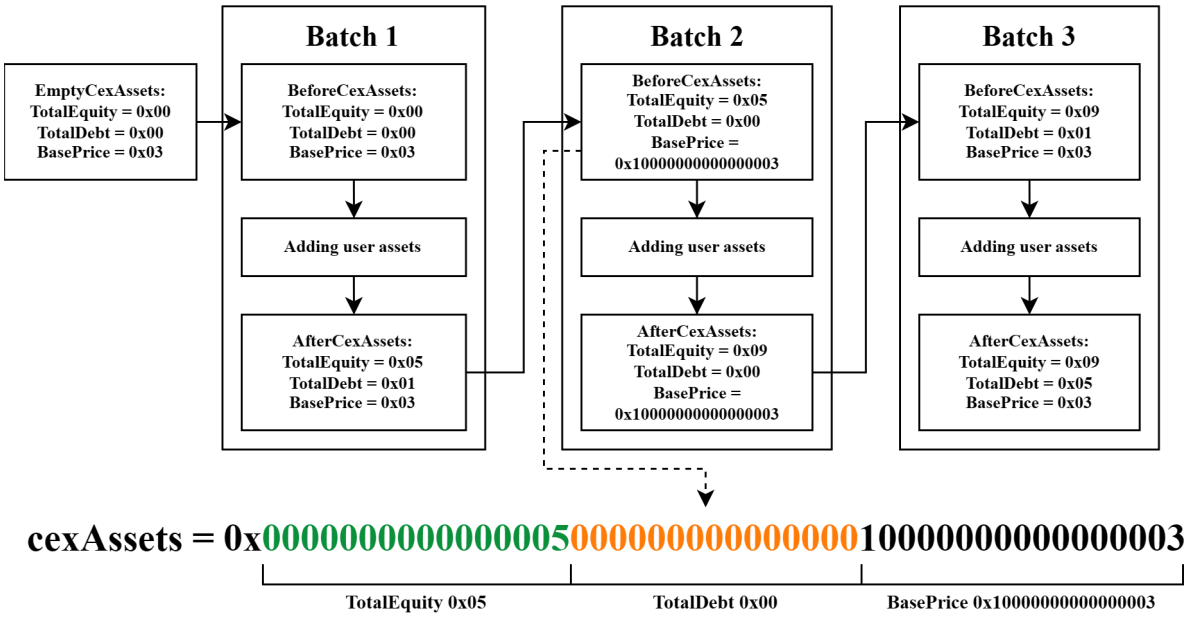
The lack of validation of *BasePrice* allows it to be modified between batches, by lowering the *TotalDebt* by 1, the *BasePrice* can be increased by 2^{64} and vice versa. Because of that, it is possible to generate almost unlimited debt. A user with 1 coin with *BaseValue* greater than 2^{64} (million of dollars) can have almost any debt, the assertion responsible for checking if users have lower debt than equity won't work correctly.

It is possible to generate the debt without anyone noticing it, it is possible by creating a batch of 864 fake users with huge debt but also with a single coin with modified *BaseValue*, which will cover the whole debt. The below diagrams demonstrate how the value of *BasePrice* can be modified in a single batch.

Correct batch calculation



Batch calculcation with BasePrice overflow



Hacken team created a dedicated repository to prove the existence of this issue: https://github.com/hknio/zkmerkle-proof-of-solvency-debt-bug.

The issue with the fix proposal was reported to Binance Team which confirmed the issue and merged fix proposed by Hacken team: https://github.com/binance/zkmerkle-proof-of-solvency/pull/5.

3.2 [Medium][Fixed] TotalDebt value underflow caused by integer overflow

When *TotalEquity* and *TotalDebt* is calculated from user assets, it is possible that it becomes bigger than 2^{64}, an example case is when two users have both 2^{63} debt and equity, then the sum of their debt and equity will be equal to 2^{64}. The code responsible for the calculations:

```
for j := 0; j < len(userAssets); j++ {
        CheckValueInRange(api, userAssets[j].Debt)
        CheckValueInRange(api, userAssets[j].Equity)
        totalUserEquity = api.Add(totalUserEquity, api.Mul(userAssets[j].Equity, b.BeforeCexAssets[j].BasePrice))
        totalUserDebt = api.Add(totalUserDebt, api.Mul(userAssets[j].Debt, b.BeforeCexAssets[j].BasePrice))

        afterCexAssets[j].TotalEquity = api.Add(afterCexAssets[j].TotalEquity, userAssets[j].Equity)
        afterCexAssets[j].TotalDebt = api.Add(afterCexAssets[j].TotalDebt, userAssets[j].Debt)
}
```

When the value of *TotalEquity* or *TotalDebt* will become higher than 2^{64}, then the next part of code, responsible for calculating integer used by hash function (*tempAfterCexAssets)* will work incorrectly because of overflows:

```
for j := 0; j < len(tempAfterCexAssets); j++ {
        tempAfterCexAssets[j] = api.Add(api.Mul(afterCexAssets[j].TotalEquity, utils.Uint64MaxValueFrSquare),
                api.Mul(afterCexAssets[j].TotalDebt, utils.Uint64MaxValueFr), afterCexAssets[j].BasePrice)
}
```

When *TotalEquity* exceeds 2^{64} then the proof in the next batch will be incorrect, however when *TotalDebt* exceeds 2^{64}, then it will overflow into *TotalEquity*. For example, *TotalDebt* equal to exactly 2^{64} would be equivalent to *TotalEquity* equal 1 and *TotalDebt* equal 0. This allows to lower the value of *TotalDebt* in the similar way as it was done in the case of the first issue with *BasePrice*, however it would not be beneficial in any way so this issue is not critical.

We recommend adding additional *CheckValueInRange* for *TotalEquity* and *TotalDebt* when calculating *tempAfterCexAssets*.

The issue was addressed and fixed by Binance Team: https://github.com/binance/zkmerkle-proof-of-solvency/pull/6

3.3 [Low] Potential omission of users

The current system of verification lacks a mechanism to confirm the completeness of the provider's inclusion of their users in the Merkle Tree. It is uncertain whether the provider may have excluded some users, who they presume will either not perform a verification of the proof or whose objections, in the event that they do not receive a proof, will not be given due consideration.

In the current implementation, the prover knows which users do the verification process as they need to download the configuration files from their website. Simplifying the process of choosing which users should be included and which ones can be omitted.

While unlikely this would happen in practice, to address this issue, it is necessary a trusted third party, as they become more readily available to support crypto exchanges, must verify that all users were included in the Merkle Tree without any exclusions.

3.4 [Low] Merkle Root hash integrity

When users download the Merkle tree and each user config from the frontend, the Merkle root hash is included in the user_config.json file, but there is no way to check the integrity of this hash across all Binance users in order to be sure that this root hash wasn't tampered depending on the client IP or other parameters of the users.


```
{
  "AccountIndex": 9,
  "AccountIdHash": "0000041cb7323211d0b356c2fe6e79fdaf0c27d74b3bb1a4635942f9ae92145b",
  "Root": "29591ef3a9ed02605edd6ab14f5dd49e7dbe0d03e72a27383f929ef3efb7514f",
  "Assets": [{"Index":7,"Equity":123456000,"Debt":0}],
  "Proof": ["DrPpFsm4/5HntRTf8M3dbgpdrxq3Q8lZkB2ngysW2js=","G1WgD/CvmGApQgmIX0rE0BlSifkw6IfNwY
  "TotalEquity": 123456000,
  "TotalDebt": 0
}
```


To counteract this, the Merkle root should be signed by a trusted third-party auditor or be published on the blockchain as a public bulletin board, so users can easily verify the transaction's inclusion and the validity of the Merkle root hash they got from their user_config.json. It should be done in a single transaction, which will be easy to detect. It's also possible to address this issues by publishing the hash root in a social media that the proved doesn't control.

3.5 [Informational] Total amount of users inference

The user downloads a proof.csv file in the verification config containing the total amount of batches and their commitments. The current number of batches at the moment of this assessment is 49.789. If we multiply this by the number of users per batch (currently 864), we can infer that the total number of users is around 43.015.104, as one leaf is equal to one user in the current implementation. Still, this amount can also include a number of empty leaves, so it's only an approximation.
Randomizing the number of empty leaves in bigger numbers can solve this issue.

4 [Informational] Vulnerabilities in dependencies

- Improper Signature Verification affecting golang.org/x/crypto/ssh package
- Denial of Service (DoS) affecting golang.org/x/net/html package
- NULL Pointer Dereference affecting golang.org/x/net/html package
- Authorization Bypass Through User-Controlled Key affecting github.com/emicklei/go-restful/v3 package
- Authorization Bypass affecting github.com/emicklei/go-restful/v3 package
- Improper Input Validation affecting github.com/ethereum/go-ethereum/core package
- Insecure Randomness affecting github.com/satori/go.uuid package

After checking every exploit of these vulnerabilities we found that none of the vulnerable functions are being currently used in the project, but it is suggested to update these dependencies in order to avoid them completely.