Audit Report

Persistence Bridge

v1.0

January 20, 2021

Table of Contents

Table of Contents 2

License 4

Disclaimer 4

Introduction 6

- Purpose of this Report 6
- Codebase Submitted for the Audit 6
- Methodology 7

How to read this Report 8

Project Risk Analysis 9

- Code Quality 9
- Project Inherent Risk 9

Summary of Findings 11

Detailed Findings 13

- TLS connection to CASP is prone to man-in-the-middle attacks 13
- Lack of Kafka message deduplication can lead to accidental transaction replay 13
- BadgerDB and Kafka operations do not happen atomically, potentially leading inconsistent state and missed bridged transactions 14
- Mnemonic/seed phrase as well as a bearer token exists in the codebase 14
- Retry logic can cause unlimited fees and potentially spend user funds 15
- Unbonding of all funds will fail 15
- Signing may fail 15
- CASP API token can be read by any user on the machine 16
- Transaction page calculation using floating point numbers could be off by one, causing transactions to be missed 16
- Wrong validation before setting Ethereum public key in configuration 17
- The current Kafka replication factor of 1 might lead to lost messages 17
- Usage of loops for retrying without timeouts may lead to program running out of memory or deadlock on shutdown 18
- Usage of background contexts that are not canceled might lead to slow shutdowns 18
- When no validator is in database the program does not panic 19
- Unchecked error during initialization and start of chain 19
- Usage of leaking time. Tick will lead the program to run out of memory 19
- Usage of magic numbers throughout the codebase 20
- Erroneous batch size counting logic will lead to bigger batch sizes than set 20
- CASP pub key prefix is sliced off instead of asserted to be 04, length is not asserted 21
- Hardcoded ethereum contract addresses might not work on different networks 21

Configuration can be updated even if it is sealed 22
Shutdown is sealed even if bridge stop signal is set to false which prevents setting the signal subsequently to true 22
Only the first public key from CASP is used, potentially leading to unexpected behaviour 22
Unrelated test code as part of the repo 23
Unused code 23

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of this Report

Oak Security has been engaged by Persistence to perform a continuous security audit of the pBridge cross-blockchain bridge implementation.

The objectives of the audit are as follows:

1. Determine the correct functioning of the bridge implementation, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following GitHub repository:

https://github.com/persistenceOne/persistenceBridge/releases/tag/v0.4.0

Commit hash: `d3960e444faad9b5dbecba58a94c986b17492a86`

Persistence uses Unbound's Crypto Asset Security Platform ("CASP") to handle threshold signatures. CASP as well as other dependencies (such as BadgerDB, Apache Kafka and libraries) used by the bridge have not been audited as part of this present security audit.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line by line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

How to read this Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending,** **Acknowledged,** or **Resolved**. Informational notes do not have a status, since we consider them optional recommendations.

Project Risk Analysis

Audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

This section is meant to provide an indicator of the remaining risk. Users of the system should exercise caution.

Code Quality

In order to subjectively quantify the remaining risk, we provide a measure of the following code quality indicators: **code complexity**, **code readability**, **level of documentation** and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium	-
Level of Documentation	Medium	-
Test Coverage	Low	There are no tests in the codebase

Project Inherent Risk

In addition to code quality indicators, risk can be characterized by the nature of the project or protocol.

Criteria	Risk Level	Comment
Modularity Is the project a self-contained unit, such as a single smart contract, or does it have many moving parts?	High	The project relies on multiple components: A go-ethereum node, a Cosmos SDK chain, Unbound's Crypto Asset Security Platform ("CASP"), a BadgerDB, and Apache Kafka.
Technology Complexity Does the project rely on different technologies interacting with each other?	High	Yes, see above.

Degree of Experimentation Is the project implementing a well-known concept or does it implement experimental concepts, such as new economic models?	**Medium**	Bridges are fairly well understood, however the current design involves certain features that may lead to unexpected results: There are different groups of message types that are potentially executed in a non-deterministic order. There is no guarantee that a send and unbond message from Ethereum is not executed as an unbond and then send message. Likewise, any failure (e. g. due to networking conditions) will lead to a change of execution order of transactions.
External Dependencies Does the project interact with external components, such as other protocols or oracles?	**High**	The Persistence bridge relies on multiple external dependencies. For instance, it does not verify the proof of work/validator signatures or that transactions are actually included in a block through a light client implementation, but rather trusts a running go-ethereum node to return valid data. Additionally, there is no check for connection to the right chain of the received blocks. Likewise, signing and key management is handled by Unbound's Crypto Asset Security Platform ("CASP"), whose configuration is not part of the codebase audited. Furthermore, the correct functioning of the bride relies on the configuration of Apache Kafka, which is also not part of the present codebase. For example, a configuration of more than one partition would break the ordering assumption of messages, potentially leading to a deadlock of signing operations and a halt of the bridge.

Due to the lack of tests and the medium to high risks involved with the bridge, we recommend the project to undergo ongoing security audits by multiple providers. We also recommend creating a treasury with an insurance fund for ongoing maintenance and compensation in case of any exploits. Alternatively, insurance coverage could be acquired.

Summary of Findings

No	Description	Severity	Status
1	TLS connection to CASP is prone to man-in-the-middle attacks	Critical	Resolved
2	Lack of Kafka message deduplication can lead to accidental transaction replay	Critical	Resolved
3	BadgerDB and Kafka operations do not happen atomically, potentially leading inconsistent state and missed bridged transactions	Critical	Resolved
4	Mnemonic/seed phrase as well as a bearer token exists in the codebase	Critical	Resolved
5	Retry logic can cause unlimited fees and potentially spend user funds	Critical	Acknowledged
6	Unbonding of all funds will fail	Critical	Resolved
7	Signing may fail	Major	Resolved
8	CASP API token can be read by any user on the machine	Major	Resolved
9	Transaction page calculation using floating point numbers could be off by one, causing transactions to be missed	Major	Resolved
10	Wrong validation before setting Ethereum public key in configuration	Minor	Resolved
11	The current Kafka replication factor of 1 might lead to lost messages	Minor	Acknowledged
12	Usage of loops for retrying without timeouts may lead to program running out of memory or deadlock on shutdown	Minor	Resolved
13	Usage of background contexts that are not canceled might lead to slow shutdowns	Minor	Acknowledged
14	When no validator is in database the program does not panic	Minor	Resolved
15	Unchecked error during initialization and start of chain	Minor	Resolved

16	Usage of leaking time. Tick will lead the program to run out of memory	Minor	Resolved
17	Usage of magic numbers throughout the codebase	Minor	Resolved
18	Erroneous batch size counting logic will lead to bigger batch sizes than set	Minor	Resolved
19	CASP pub key prefix is sliced off instead of asserted to be 04, length is not asserted	Informational	Resolved
20	Hardcoded ethereum contract addresses might not work on different networks	Informational	Resolved
21	Configuration can be updated even if it is sealed	Informational	Resolved
22	Shutdown is sealed even if bridge stop signal is set to false which prevents setting the signal subsequently to true	Informational	Resolved
23	Only the first public key from CASP is used, potentially leading to unexpected behaviour	Informational	Acknowledged
24	Unrelated test code as part of the repo	Informational	Resolved
25	Unused code	Informational	Resolved

Detailed Findings

1. TLS connection to CASP is prone to man-in-the-middle attacks

Severity: Critical

In `application/rest/casp/getSignOperation.go:19`, `application/rest/casp/getUncompressedPublicKeys.go:25` as well as in `application/rest/casp/postSignData.go:39`, `InsecureSkipVerify` is set to `true`. In this mode, certificates are not verified. That implies that an attacker can easily intercept the TLS connection with a man-in-the-middle attack, giving them access to trigger/participate in signing of arbitrary data through CASP.

Recommendation

We recommend using a tls config with the correct root certificates and unskip the certificate verification (see the tls package docs for an example).

Status: Resolved

The bridge and the CASP server run on a local network, so man-in-the-middle attacks are very unlikely. The Persistence team changed `InsecureSkipVerify` to be a config parameter though in v0.5.0 to allow TLS connections.

2. Lack of Kafka message deduplication can lead to accidental transaction replay

Severity: Critical

The current architecture receives messages from Kafka, processes them, and then marks the processed messages in Kafka. Any error during processing will lead to a restart of the consumer, which can leave a partially processed message not marked in Kafka. After the restart, that same message will be processed again. That can be highly problematic, for example in cases where a transaction was already sent to the blockchain. An unbond that's triggered twice will lead to users losing funds.

Recommendation

We recommend deduplicating any messages by checking on-chain whether a message was already sent. Alternatively, we recommend tracking in the database message progress to allow recovery from any partially executed messages.

Status: Resolved

3. BadgerDB and Kafka operations do not happen atomically, potentially leading inconsistent state and missed bridged transactions

Severity: Critical

The current architecture writes to BadgerDB and Kafka in different operations and does not ensure that changes are reverted if errors occur.

An example of this issue is the removal of the last validator by `RemoveCommand`. Through `application/commands/removeValidator.go:43` or `49`, the BadgerDB will be updated, but if the last validator was removed, an error will be returned in line `63` before the `Redelegate` message is sent to Kafka in line `85`. That leads to an inconsistent state between BadgerDB and Kafka, and the redelegate message will have to be triggered manually.

Another example is the Ethereum block handle logic, in which a loop is used to iterate over every transaction in `ethereum/block.go:21`. In every iteration, BadgerDB may be updated. If an error happens in a subsequent iteration in the `collectEthTx` function, that error will bubble up the stack without a reversion of BadgerDB changes. That will leave BadgerDB in an inconsistent state where some transactions have been written and others have not. Additionally, Kafka will not receive messages for any of the transactions in the block, not even the ones that have made it into BadgerDB.

The same issue exists all over the codebase. Such data inconsistencies are hard to detect and even harder to recover from.

Recommendation

We recommend changing the architecture such that any BadgerDB changes are running in an ACID transaction and will be reverted if any error occurs. Likewise, we recommend adopting Kafka transactions to ensure that all related messages are committed to Kafka atomically, e. g. either all Ethereum transactions for a block are written, or none. We recommend committing the BadgerDB transaction only after the Kafka transaction was successfully committed.

Status: Resolved

4. Mnemonic/seed phrase as well as a bearer token exists in the codebase

Severity: Critical

In `application/outgoingTx/tendermint.go:117`, a 24 word mnemonic/seed phrase is specified in a comment. Additionally, `application/casp/sign_test.go:14` contains a valid bearer token for authentication with CASP. Depending on the usage of those secrets, anyone with access to the codebase could potentially steal funds or at least sign malicious transactions.

Recommendation

We recommend removing any potential secrets/private keys from the codebase (and history in the repository) and instead pass sensitive information in via the environment.

Status: Resolved

5. Retry logic can cause unlimited fees and potentially spend user funds

Severity: Critical

When transactions on Tendermint fail, they are queued for re-execution in `kafka/handler/toTendermint.go:100` and in `tendermint/onNewBlock.go:57`. Since there is no limit on the number of retries, an unlimited amount of fees can be wasted. As the fees come out of the same account that holds locked user tokens, those user funds could be proportionally spent.

Recommendation

We recommend putting a limit on the number of retries and shut down the bridge if the limit is reached. That approach limits gas consumption and allows for manual recovery.

Status: Acknowledged

6. Unbonding of all funds will fail

Severity: Critical

Due to the “less than” condition in `kafka/handler/ethUnbond.go:70`, unbonding of all funds will fail. That implies that the last user trying to unbond their tokens will be unable to exit the protocol.

Recommendation

We recommend changing the condition to “less than or equal to”.

Status: Resolved

7. Signing may fail

Severity: Major

The `for` loop in the `GetCASPSigningOperationID` function at `application/casp/signing.go:15` only ever runs once. If the `SignData` query returns a true busy value, the function sleeps for some time and will then return without an

error in line 23. That will lead to a failure of the signing process whenever there is a delay between processing of the different parties participating in the signing process. Such a failure will return errors to the callers until a retry is tried through the Kafka queue.

Recommendation

If the intention here is to retry after the sleep, we recommend adding a continue statement after line 18. Otherwise, we recommend returning an error.

Status: Resolved

8. CASP API token can be read by any user on the machine

Severity: Major

In `application/commands/init.go:38`, the configuration is written to `config.toml` with a chmod of `0644`. That allows any user on the machine to read the configuration file, which includes the CASP API token.

Recommendation

We recommend removing the CASP API token from the configuration file and instead pass it as an environment variable. We also recommend changing the `config.toml` chmod to `0600`.

Status: Resolved

9. Transaction page calculation using floating point numbers could be off by one, causing transactions to be missed

Severity: Major

In `tendermint/listener.go:105`, the number of transaction pages is calculated using `float64` types. Go's float types are subject to the inaccuracies of integer representation of IEEE 754. That could lead to the page number being off by one, which in the worst case could mean that transactions on the last page are missed by the bridge.

Recommendation

We recommend using integer division instead and determining whether an extra page is needed by checking modulo being zero.

Status: Resolved

10. Wrong validation before setting Ethereum public key in configuration

Severity: Minor

In `application/configuration/configuration.go:115`, a validation check for non-emptiness of `caspTMPublicKey` is performed, but then the `caspEthPublicKey` configuration value is assigned.

Recommendation

We recommend performing the non-empty validation on `caspEthPublicKey`.

Status: Resolved

11. The current Kafka replication factor of 1 might lead to lost messages

Severity: Minor

In `application/constants/kafka.go:21`, `ReplicationFactor` is set to `1`, which implies that there will be only one copy of each partition. Without any replications, a data loss of messages is more likely.

We only consider this issue minor since the bridge can recover from data loss by processing the affected blocks again. A loss of some of the messages within a block might be hard to recover from though.

Additionally, it is important to keep `NumPartitions` set to `1`. Any higher value might lead to non-deterministic ordering of messages, which potentially could lead to deadlocks of the bridge nodes where they wait for signatures from each other on different transactions.

Recommendation

We recommend setting `ReplicationFactor` to `2`.

Status: Acknowledged

Persistence intends to set the `ReplicationFactor` to `2` in staging servers as this is configurable.

12. Usage of loops for retrying without timeouts may lead to program running out of memory or deadlock on shutdown

Severity: Minor

In multiple places in the codebase, loops are used to retry async operations, e. g. in `application/casp/signing.go:15` and `32`. Those loops might run indefinitely, with further invocations through the messaging queue creating an ever-increasing memory consumption. Additionally, those loops might block a shutdown of the program.

Recommendation

We recommend adding timeouts to those loops. To allow proper and timely shutdown, we recommend creating an outer context and then deriving a child context using `WithTimeout` for easy cancellation propagation.

Status: Resolved

Partially resolved for CASP signing through introduction of an attempt limit in `GetCASPSignature` function in `application/casp/signing.go:31`. Other loops still exhibit this issue, and the attempt limit introduced still does not allow cancellation of ongoing loops.

13. Usage of background contexts that are not canceled might lead to slow shutdowns

Severity: Minor

In multiple places in the codebase, `context.Background` is used, but not canceled on shutdown. That may lead to slow shutdown times since requests will continue to block until they finish before a function can return.

An example is the `consumeToEthMsgs` function, where cancellation of the context would cancel the pending Kafka consumer in `kafka/routines.go:62`. Without a context cancellation, the Kafka consumer may keep the application running for a long time.

Recommendation

We recommend using a tree of contexts throughout the application that can propagate cancellation on shutdown. We also recommend canceling the top-level context on application shutdown. Finally, usage of the errgroup package might further simplify the shutdown logic, making shutdown flags unnecessary.

Status: Acknowledged

Persistence intends to implement this recommendation when refactoring the code in the future.

14. When no validator is in database the program does not panic

Severity: Minor

In `application/commands/addValidator.go:63` and `application/commands/showValidator.go:45`, when no validators are in the database (validator count is zero), a log entry is emitted that states that the program will panic, but no panic is triggered.

Recommendation

We recommend using `log.Fatalln` instead of `log.Println` to actually panic.

Status: Resolved

15. Unchecked error during initialization and start of chain

Severity: Minor

The `err` returned from `fileInputAdd` in `tendermint/chain.go:16` is not checked for a `nil` value.

Recommendation

We recommend checking whether the `err` is nil and returning the `err` if not.

Status: Resolved

16. Usage of leaking time. Tick will lead the program to run out of memory

Severity: Minor

The `time.Tick` used in `kafka/handler/toEth.go:19`, `kafka/handler/toTendermint.go:21` and `kafka/routines.go:157` will lead to ever increasing memory consumption, since `time.Tick` leaks the underlying Ticker, which means it cannot be garbage collected.

Recommendation

We recommend using `time.NewTicker` (which can be garbage collected) instead of `time.Tick`.

Status: Resolved

17. Usage of magic numbers throughout the codebase

Severity: Minor

In multiple places of the codebase, magic numbers (hardcoded value) are used. That makes changes of parameters difficult. An example is the 12 block confirmation time used in `ethereum/listener.go:52` and `ethereum/onNewBlock.go:48`.

Recommendation

We recommend putting magic numbers into the configuration. To find all magic numbers, we recommend using the `gomnd` linter.

Status: Resolved

18. Erroneous batch size counting logic will lead to bigger batch sizes than set

Severity: Minor

In multiple places of the codebase, counting and verification of the number of messages in a batch is erroneous, which will cause the batch size enforcement to fail:

- In `kafka/handler/msgDelegate.go:89`, `checkCount` is not called. There should be a check for the number of messages.
- In `kafka/handler/msgSend.go:51`, `m.Count` is not increased. It should be increased by the number of messages.
- In `kafka/handler/msgSend.go:59`, `m.Count` is not increased. It should be increased by 1 for the `ToTendermint` message.
- In `kafka/handler/msgSend.go:60`, `loop` could be 0 already, since it could also be decreased through the call of `WithdrawRewards`. The check in line `61` should be `loop <= 0`.
- In `kafka/handler/msgUnbond.go:42`, `checkCount` is called, but it is checked after a message has been added in line `34`. It should be called before line `34` instead.
- In `kafka/handler/redelegate.go:88`, `checkCount` is not called. There should be a check for the number of messages.
- In `kafka/handler/retryTendermint.go:46`, `m.Count` is set to a value that overwrites the previous count. The assignment should instead be `m.Count -= loop`.
- In `kafka/handler/retryTendermint.go:57`, `checkCount` is called, but it is checked after messages have been added in line `41` and `49`. It should be called before line `41` instead.
- In `kafka/handler/retryTendermint.go:41`, no check is done whether the rewards of all validators can be withdrawn, which means that partial reward withdrawals are possible. There should be a check that skips reward withdrawals

whenever the remaining messages in the batch is less than the validator count, as is done in `kafka/handler/msgSend.go:27`.

Recommendation

We recommend fixing the issues as indicated in the list above.

Status: Resolved

19. CASP pub key prefix is sliced off instead of asserted to be 04, length is not asserted

Severity: Informational

In `application/casp/publicKey.go:41`, the first two runes of the `caspPubKey` string are sliced off. A comment states that those runes should be `04`. Additionally, in line `48`, the length of the `pubKeyBytes` slice is not validated.

Recommendation

We recommend asserting that those two bytes are `04` as opposed to slicing the string. We also recommend asserting the length of `pubKeyBytes`.

Status: Resolved

20. Hardcoded ethereum contract addresses might not work on different networks

Severity: Informational

In `application/constants/ethereum.go:5` and `10`, `LiquidStakingAddress` and `TokenWrapperAddress` hold hard coded values to the liquid staking and token wrapper contracts. As a best practice, different private keys/multisigs should be used for deploying on different networks (i. e. testnets and mainnet), which will result in different addresses for the contracts.

Recommendation

We recommend retrieving the contract addresses from the environment or passing them as flags.

Status: Resolved

21. Configuration can be updated even if it is sealed

Severity: Informational

In the function `GetAppConfig` at `application/configuration/configuration.go:23`, a pointer to the config is returned. That allows the caller to change configuration values, independent of the value of the `seal`.

Recommendation

We recommend returning the config struct instead of a pointer to it.

Status: Resolved

22. Shutdown is sealed even if bridge stop signal is set to false which prevents setting the signal subsequently to true

Severity: Informational

In the function `SetBridgeStopSignal` at `application/shutdown/shutdown.go:16`, `seal` is set to `true` even if the bridge stop signal is set to `false`. Due to the condition in line 14, that prevents the stop signal to be set in a subsequent call with a true value.

Recommendation

We recommend only setting `seal` to true if the stop signal is `true`.

Status: Resolved

23. Only the first public key from CASP is used, potentially leading to unexpected behaviour

Severity: Informational

In `application/casp/address.go:19` as well as `31`, the first available public key is used only. If more than one public key is returned by CASP, the others are ignored, which might lead to unexpected behaviour.

Recommendation

We recommend asserting that exactly one key was retrieved and exiting with an error otherwise.

Status: Acknowledged

While creating keys on CASP, the Persistence team ensures that only one key is generated per coin.

24. Unrelated test code as part of the repo

Severity: Informational

The tests in `application/casp/sign_test.go` are not testing any functionality implemented in the casp package.

Recommendation

We recommend moving that test file into a separate package.

Status: Resolved

25. Unused code

Severity: Informational

All functions in `kafka/utils/db.go` and in `kafka/utils/utils.go` are currently unused.

Recommendation

We recommend removing any unused code to improve readability and increase maintainability of the codebase.

Status: Resolved