Flare

C-Chain Indexer Security Review

coinspect

C-Chain Indexer

Source Code Security Review

Version: v240409 | Prepared for: Flare | April 2024

Security Assessment

1. Executive Summary

    2.2 Findings where caution is advised

    2.3 Solved issues & recommendations

3. Scope

4 Assessment

    4.4 Additional Fixes

5. Detailed Findings

    FCCI-001 - Attackers can abuse fallback methods to hide transactions

© Coinspect 2024

FCCI-002 - Malicious contracts can fake function signatures

FCCI-003 - Secrets are printed to stdout

FCCI-004 - Secrets are stored in configuration files

6. Disclaimer

© Coinspect 2024

1. Executive Summary

In **January 2024**, **Flare** engaged Coinspect to perform a source code review of its **C Chain Indexer**. The objective of the project was to evaluate the security of the application.

The **C Chain Indexer** is in charge of interfacing with a C-Chain node and storing certain transaction data in a database for later consumption by other Flare systems.

	Solved	Caution Advised	Resolution Pending
High	0	0	0
Medium	1	2	0
Low	1	0	0
No Risk	0	0	0
Total	2	2	0

© Coinspect 2024

2. Summary of Findings

2.2 Findings where caution is advised

These issues have been addressed, but their risk has not been fully mitigated. Any future changes to the codebase should be carefully evaluated to avoid exacerbating these issues or increasing their probability.

Findings with a risk of None pose no threat, but document an implicit assumption which must be taken into account. Once acknowledged, these are considered solved.

Id	Title	Risk
FCCI-001	Attackers can abuse fallback methods to hide transactions	Medium
FCCI-002	Malicious contracts can fake function signatures	Medium

2.3 Solved issues & recommendations

These issues have been fully fixed or represent recommendations that could improve the long-term security posture of the project.

Id	Title	Risk
FCCI-003	Secrets are printed to stdout	Medium
FCCI-004	Secrets are stored in configuration files	Low

© Coinspect 2024

3. Scope

The scope was set to be the repository at C-Chain Indexer at commit `7968e6bb0d7147c9823c8422fd32dda65bc0ab0b`.

© Coinspect 2024

4 Assessment

The **C Chain Indexer** works as an indexer for certain blockchain transactions specified by its operator in a `.toml` configuration file. The indexer needs to have access to a C-Chain node which is able to answer its queries.

The indexer has only a few features:

- It must index past transactions if booted for the first time or if it is not in sync
- It must keep indexing transactions in real time as long as it runs
- It must periodically clean the DB from transactions that are too old to be relevant

It is important to consider that there is no direct way to interact with the indexer, as it has no HTTP API. The indexer is started and it only interacts with a node that provides it with answers and with its own DB. The main point of contact with untrusted entities comes in the transactions that it parses and their data.

4.1 Security assumptions

There are two key security assumptions:

1. The node the indexer uses must be fully trusted.
2. The node the indexer uses must return only finalized blocks via its RPC, the default for Avalanche nodes.

A soft assumption is that the transactions indexed must have as recipients a set of certain Flare-controlled smart contracts. In particular, the indexer assumes that the contracts implement a Solidity-style function dispatcher with 4-byte function signatures.

This is a *soft* assumption because the indexer can be configured to index transaction to arbitrary contracts. When consulting with Flare about this possibility, Flare answered that while the intended usage is the one described, it would be beneficial to point out issues arising from trying to use the indexer with arbitrary contracts. These issues have been reported with a LOW likelihood.

4.3 Testing

© Coinspect 2024

There is only one test in the system. The test only checks basic equalities related to the indexer's pointers to the oldest and newest block it has received and that no errors have been thrown during execution.

Coinspect recommends that more tests relating to transaction data being stored by the indexer are added.

4.4 Additional Fixes

In April 8, 2024 Coinspect reviewed additional changes and features as of commit `8c92f3c23a5a3c39f3fba3e184650b41983cfdd8`.

Most relevant changes are related to bug fixes when retrieving latest blocks, modifications in data types and structs, improvements in the logging and backoff/retry systems and validation of contract addresses and function signatures integrity.

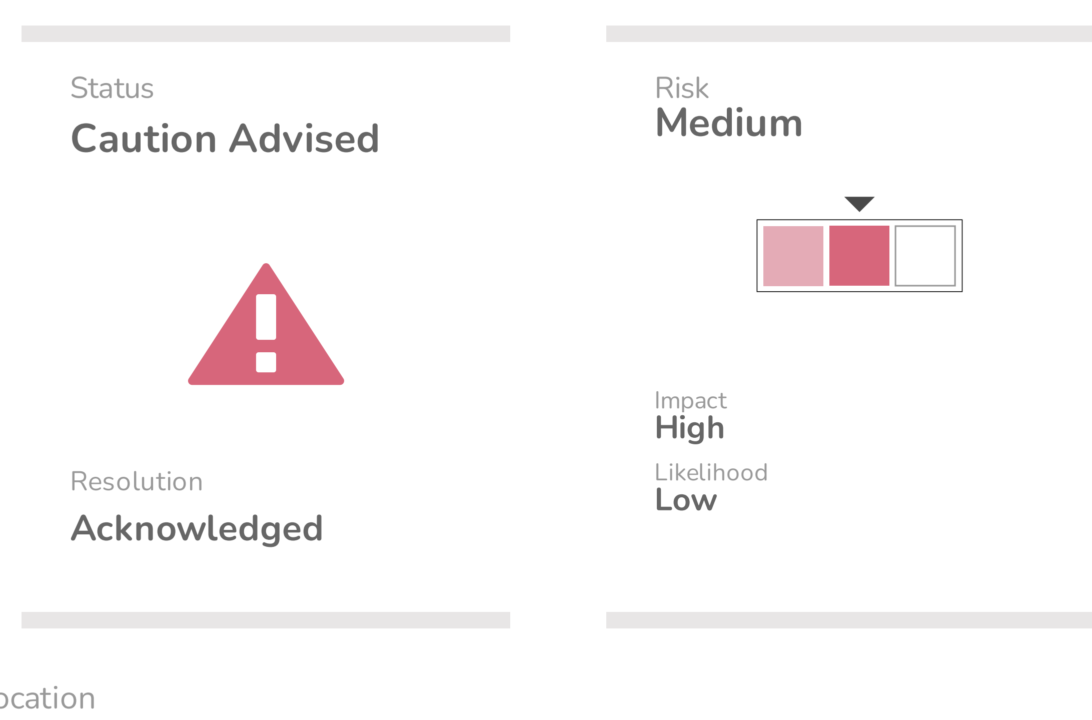
Coinspect did not find any issues as of this commit.

© Coinspect 2024

5. Detailed Findings

FCCI-001

Attackers can abuse fallback methods to hide transactions



Location

`indexer/blocks.go`

Description

Any call that executes logic via `fallback` or `receive` is not indexed, as a result, relevant transactions will not be included in the database.

The indexer skips transactions with a data size smaller than 4 bytes:

```
txData := hex.EncodeToString(tx.Data())
if len(txData) < 8 {
    continue
}
```

In other words, the indexer skips calls executing logic with less than 4 bytes of data that trigger a `fallback` or `receive` instruction.

A case in the Flare Protocol would be its `WNat` contract. Deposit transactions process incoming native transfers automatically by minting wrapped tokens:

```
receive() external payable {
    deposit();
}
```

Additionally, any relevant action performed via the constructor of a contract would never be indexed as transactions sent to the `address(0)` are ignored:

```
if tx.To() == nil {
    continue
}
```

If a transaction triggered from the constructor of a contract becomes relevant in the future, this indexer's implementation will omit it and will not be available for consumers.

Recommendation

Consider removing the minimum size of the data in transactions that can be indexed and adding a special string like "empty" to the `config.toml` file. This string would indicate that the empty transaction data is relevant for this contract and would get indexed.

Status

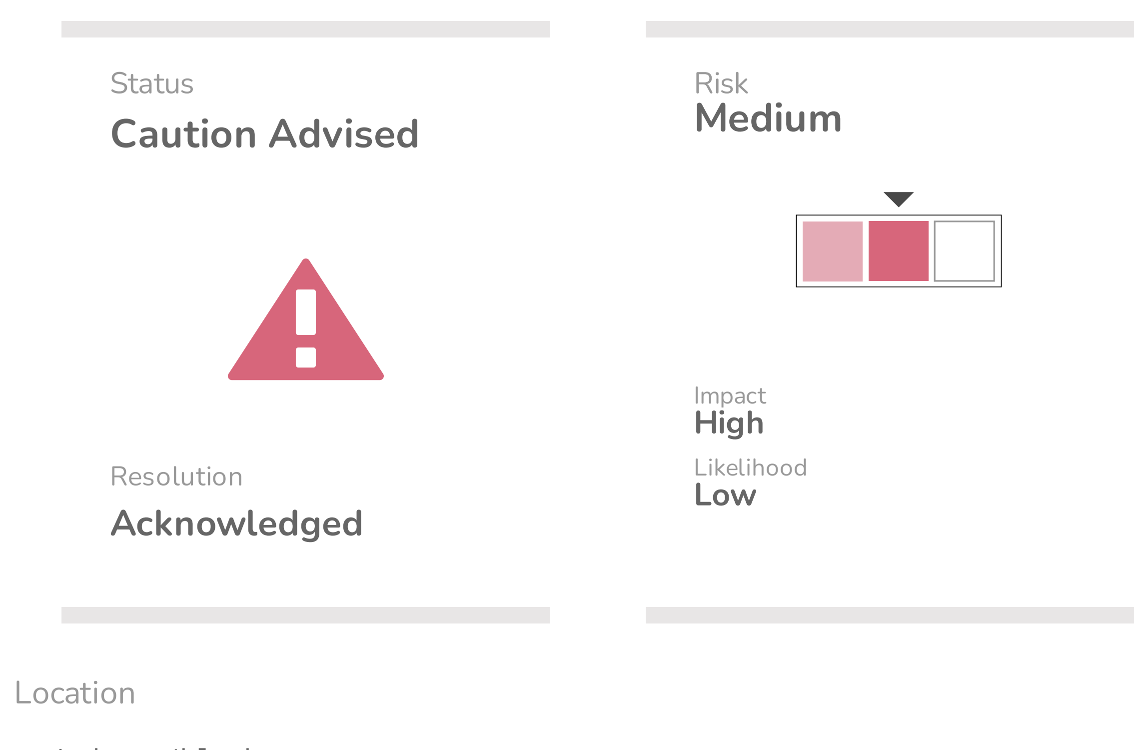
Acknowledged.

The Flare Team acknowledged the risk and stated that considers this as not needed in the context of the Flare Systems Protocol.

© Coinspect 2024

FCCI-002

Malicious contracts can fake function signatures



Location

`indexer/blocks.go`

Description

If transactions a malicious contract is configured to be indexed, the indexer can be tricked into believing certain method calls have happened when they have not.

The root cause of the issue is that the indexer assumes that contracts will follow the conventional Solidity-dispatcher interface. This interface mandates that the first four bytes of the `calldata` match a function selector.

```
txData := hex.EncodeToString(tx.Data())
if len(txData) < 8 {
        continue
}

funcSig := txData[:8]
```

```
                contractAddress := strings.ToLower(tx.To().Hex()[2:])
                check := false
                policy := transactionsPolicy{status: false,
collectEvents: false}

                for _, address := range []string{contractAddress,
undefined} {
                        if val, ok := ci.transactions[address]; ok {
                                for _, sig := range []string{funcSig,
undefined} {
                                        if pol, ok := val[sig]; ok {
                                                check = true
                                                policy.status =
policy.status || pol.status
                                                policy.collectEvents =
policy.collectEvents || pol.collectEvents
                                        }
                                }
                        }
                }
```

Nevertheless, this interface is not a requirement for the EVM, it is merely a common implementation detail of compilers.

If a malicious address is set as a valid `contractAddress`, then this contract can appear to the indexer to have called a function with a certain signature. In reality, the contract could take any arbitrary action, including simply being a `NOOP`.

The inverse is also true: a malicious contract is able to execute functions with a `txData` size of less than 4 bytes. An attacker can thus hide their functions calls.

Recommendation

Do not use the indexer with arbitrary contracts.

If the indexer needs to support a new contract, make sure that:

1. The contract is not upgradable or is upgradable only by a trusted party
2. The contract is in Solidity or languages that follow the 4-byte function selector convention.

Consider adding a warning to the `.toml` config file that warns operators of the dangers of adding arbitrary contracts to the indexer.

Status

Acknowledged.

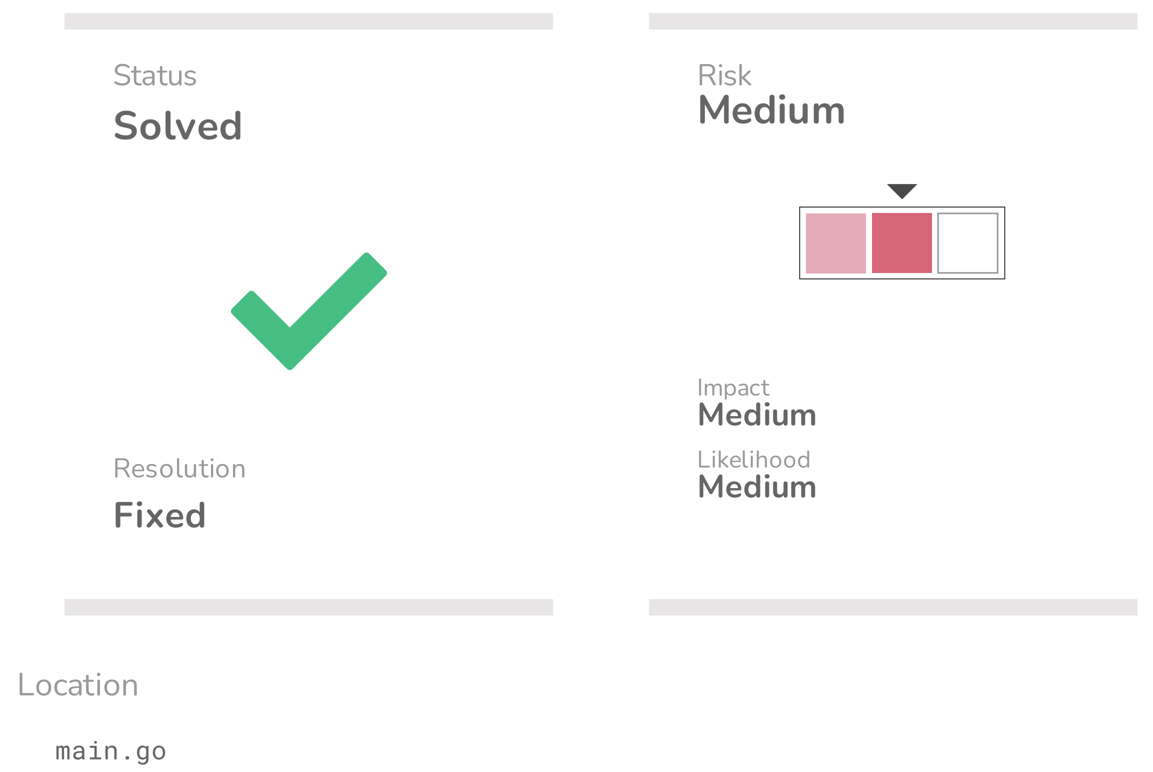
© Coinspect 2024

The Flare Team stated that the indexer will be used only for known contracts.

© Coinspect 2024

FCCI-003

Secrets are printed to stdout



Location

`main.go`

Description

Secrets are leaked to `stdout` and potentially to log-collecting services. This makes it possible for attackers that compromise any log collecting service to capture the database credentials and the API Keys for the node. Consider how this compounds with the risk described in **FCCI-04**.

The `CChain` struct's `String` method prints both the database credentials and the node's API key:

```
func (cc ChainConfig) String() string {
        return fmt.Sprintf("NodeURL: %s, APIKey: %s", cc.NodeURL,
cc.APIKey)
}
```

This is then called when initializing the program:

© Coinspect 2024

```
        logger.Info("Running with configuration: chain: %s, database: %s", cfg.Chain, cfg.DB.Database)
```

Recommendation

Do not print sensitive information.

Status

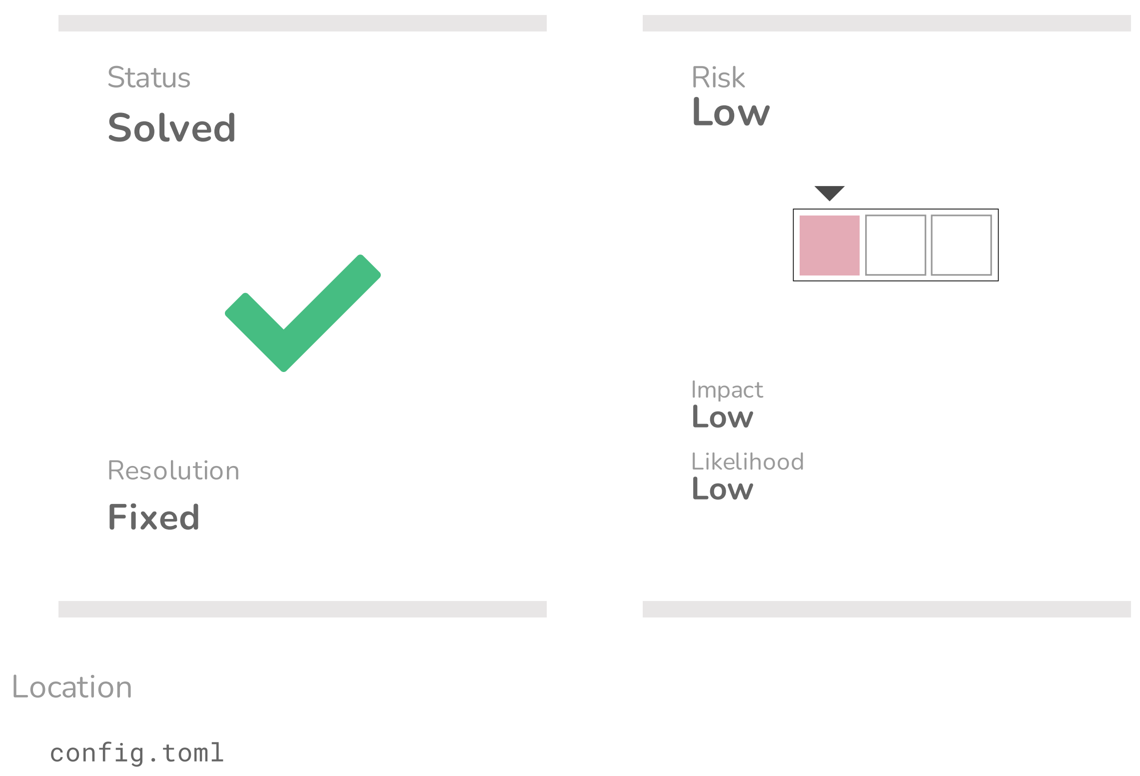
Fixed on commit `4698f7f76ef963c677dce0ec9e8d2654693e80cb`.

The log exposing sensitive keys was removed.

© Coinspect 2024

FCCI-004

Secrets are stored in configuration files



Description

Attackers positioned inside the host can fetch database credentials and API keys from the configuration files while the operators have no way to audit access to these secrets and rotate them in case of compromise.

The program expects database credentials to be in the `config.toml` file:

```
func connect(ctx context.Context, cfg *config.DBConfig) (*gorm.DB,
error) {
	// Connect to the database
	dbConfig := mysql.Config{
		User:                 cfg.Username,
		Passwd:               cfg.Password,
		Net:                  tcp,
		Addr:                 fmt.Sprintf("%s:%d", cfg.Host,
cfg.Port),
		DBName:               cfg.Database,
```

© Coinspect 2024

```
                AllowNativePasswords: true,
                ParseTime:            true,
        }
```

This architecture makes it harder for operators to use a secret manager such as AWS Secrets Manager or Hashicorp.

It is also worth noting that the configuration file is committed to the repository, making it more likely for leaks to occur.

Recommendation

Make the program able to read database credentials from the environment. Secrets managers usually have features to inject secrets into environment variables, making it easier for operators to use them.

Status

Fixed on commit `0c159a90003d3b682a9f533b3eee9c0f598737fa`.

Secrets can now be retrieved from environment variables.

© Coinspect 2024

6. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any on-chain systems or frontends that communicate with the network, nor the general operational security of the organization that developed the code.

© Coinspect 2024