Flare

Security Review

Top Level Client

coinspect

Top Level Client

Source Code Security Review

Version: v240515 | Prepared for: Flare | May 2024

Source Code Security Review

1. Executive Summary

    2.2 Findings where caution is advised

    2.3 Solved issues & recommendations

3. Scope

4 Assessment

    4.1 Security assumptions

    4.2 Testing

    4.3 Additional Fixes

5. Detailed Findings

    TOP-01 - Evil voter can vote in rounds with no weight

© Coinspect 2024

TOP-02 - Rogue voter can vote for all merkle roots
TOP-03 - Voters are incentivized to withhold their signatures
TOP-04 - Malicious voter can always be a finalizer
TOP-05 - Different signing policies have the same hash
TOP-06 - Private keys are not sufficiently protected
6. Disclaimer

© Coinspect 2024

1. Executive Summary

In **January 2024**, **Flare** engaged Coinspect to perform a source code review of its **Top Level Client**. The objective of the project was to evaluate the security of the application, which is a critical off-chain component of the Flare Systems Protocol.

The **Top Level Client** is in charge of requesting data from each subsystem in the protocol and forwarding it to the blockchain, as well as monitoring the on-chain submission of signatures and relaying those once the threshold has been met. It also needs to handle registration for voters.

	Solved	Caution Advised	Resolution Pending
High	2	0	0
Medium	1	1	0
Low	1	1	0
No Risk	0	0	0
Total	**4**	**2**	**0**

`TOP-01` describes how a malicious voter can vote in rounds where they have no weight. `TOP-02` shows how the same voter can sign for different Merkle roots, highlighting how this strategy undermines the objectives of the system. `TOP-03` and `TOP-04` outline key incentive problems within the system that result in honest participants being deprived of rewards and subvert the system's incentives. `TOP-05` demonstrates how different

signing policies can result in identical hashes. `TOP-06` addresses concerns related to private key management.

2. Summary of Findings

2.2 Findings where caution is advised

These issues have been addressed, but the risk they pose 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
TOP-05	Different signing policies have the same hash	Medium
TOP-03	Voters are incentivized to withhold their signatures	Low

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
TOP-01	Evil voter can vote in rounds with no weight	High
TOP-02	Rogue voter can vote for all merkle roots	High
TOP-04	Malicious voter can always be a finalizer	Medium
TOP-06	Private keys are not sufficiently protected	Low

© Coinspect 2024

3. Scope

The scope was set to be the repository at flare-system-client on commit `436f7cd55c4f79f7e8213ede57731360fe53be74`.

It is worth noting that the **Top Level Client** is one part of a more complex system, which involves an indexer, several off-chain subsystems which provide data to the top level client and a set of smart contracts. This document focuses only in the **Top Level Client**.

4 Assessment

The **Top Level Client** has three well defined modules, each in charge of fulfilling one of the keys responsibilities of the software:

1. The `finalizer` module: Handles the collection of signatures and their submission to the `Relay` contract.
2. The `protocol` module: Manages communication with subprotocols and submits data in the form of `calldata` to the `Submission` contract.
3. The `registration` module: Oversees the registration of new voters in the system.

Note that interactions with the **Top Level Client** from an attacker's perspective are limited, as the system only reads data from an indexed database containing blockchain data. This means that an attacker would need to insert their payloads in the blockchain itself. Nevertheless, it is important to acknowledge that this is fairly easy for an attacker, as the system relies on `calldata` information to coordinate its actions. That means that the smart contracts themselves avoid doing any validation when possible.

With that in mind, Coinspect considered the main and highest priority threat to be an *evil voter*, as the voter set if fully open to anyone with sufficient stake. An evil voter might want to:

1. Gain unfair advantage over other voters to claim bigger rewards or claim them more often
2. Crash or slow down other voters so as to prevent finalization of data

Additionally, Coinspect considered issues related to secrets management, cryptographic operations, and software bugs that could impair the functionality of the service.

4.1 Security assumptions

As the **Top Level Client** is simply a part of a broader system, it assumes other components work correctly in order to function. In particular, Coinspect assumed that:

1. The indexer provides accurate and up-to-date data.
2. The subprotocol clients work reliably and provide accurate data.

3. The majority of the staking-vote submits accurate data and voters do not collude
4. The majority of the staking-vote adequately protects its private keys
5. The signing-policy is set by a trusted entity which submits the signing-policy in time and with correct data

Subprotocols must be aware of certain behaviors of the top level client and the protocol as a whole. In particular, Coinspect found that subprotocols must be specially careful with `additionalData`, as this data is malleable by anyone even when it is part of a bigger structure associated with a particular signer.

4.2 Testing

Coinspect found almost no unit tests on the project, with only two modules having significant coverage: `voters` and `merkle`.

```
; top-level-client (436f7cd) $ go test -coverprofile cov.out ./...
?       flare-tlc/client/config [no test files]
?       flare-tlc/client/context        [no test files]
?       flare-tlc/client/cronjob        [no test files]
?       flare-tlc/client/finalizer      [no test files]
?       flare-tlc/client/main   [no test files]
?       flare-tlc/client/protocol       [no test files]
?       flare-tlc/client/registration   [no test files]
?       flare-tlc/client/runner [no test files]
?       flare-tlc/client/scheduler      [no test files]
?       flare-tlc/client/shared [no test files]
?       flare-tlc/config        [no test files]
?       flare-tlc/database      [no test files]
?       flare-tlc/logger        [no test files]
?       flare-tlc/utils/chain   [no test files]
?       flare-tlc/utils/contracts/registry      [no test files]
?       flare-tlc/utils/contracts/relay [no test files]
?       flare-tlc/utils/contracts/submission    [no test files]
?       flare-tlc/utils/contracts/system        [no test files]
ok      flare-tlc/client/shared/voters  0.009s  coverage: 79.7% of
statements
ok      flare-tlc/utils 0.001s  coverage: 7.3% of statements
ok      flare-tlc/utils/merkle  0.007s  coverage: 93.9% of statements
```

4.3 Additional Fixes

On April 8, 2024, Coinspect reviewed additional changes and features as of commit `c6096b0492b780c207f7e760b34bf2b3892dfffb`.

Flare indicated that the most relevant changes made from the original commit were:

- Gas price and gas limits can now be configured
- Votes for future epochs are now skipped
- It is possible to specify a separate sender for submitting signatures
- Value 0 for `starting_voting_round` configuration option now means the current voting round
- The system allows specifying data fetch retries and timeouts for all submitters in configuration files
- Skip sending submit transactions if no payload was received from a client
- Added dry-run for sending certain transactions

© Coinspect 2024

5. Detailed Findings

TOP-01

Evil voter can vote in rounds with no weight

Location

```
client/finalizer/finalizer_client.go
```

Description

An attacker can call `submitSignatures` for a voting round that does not yet exist, and the client will treat it as a vote using the last signing policy. This loophole allows an evil voter to cast votes for future data under the current signing policy.

The full impact of this vulnerability varies by subprotocol and depends on how far in advance realistic merkle roots for a future round can be prepared by a voter. A generalized version of this attack involves inspecting the blockchain and front-

running votes for the next `roundId` if the `signing policy` is about to change and is unfavorable to the attacker.

To exploit this vulnerability, an attacker needs to call `submitSignatures` with a `votingRoundId` that is higher than the current one. The client periodically checks transactions to `submitSignatures` and will invoke `ProcessSubmissionData` as soon as it detects a new transaction.

`ProcessSubmissionData` will attempt to retrieve the signing policy for the `votingRoundId` specified in the `payloadItem` and, if successful, add the item to the `submissionStorage`:

```
		// ! Try to find the signing policy for the payload of
voting round
		sp :=
c.signingPolicyStorage.GetForVotingRound(payloadItem.votingRoundId)
		if sp == nil {
			first := c.signingPolicyStorage.First()
			if first != nil && payloadItem.votingRoundId <
first.startVotingRoundId {
				// This is a submission for an old
voting round, skip it
				continue
			}
			return fmt.Errorf("no signing policy found for
voting round %d", payloadItem.votingRoundId)
		}
		addResult, err :=
c.submissionStorage.Add(payloadItem.payload, sp)
```

The issue lies in the implementation of `GetForVotingRound`. This method calls `findByVotingRoundId`, which has a quirk: it returns `nil` only if the passed `votingRoundId` is *smaller* than all the possible `votingRoundIds`. If not, it returns the last element of the list.

```
func (s *signingPolicyStorage) findByVotingRoundId(votingRoundId
uint32) *signingPolicy {
	i, found := sort.Find(len(s.spList), func(i int) int {
		return cmp.Compare(votingRoundId,
s.spList[i].startVotingRoundId)
	})
	if found {
		return s.spList[i]
	}
	if i == 0 {
		return nil
	}
	return s.spList[i-1]
}
```

© Coinspect 2024

Even though the `SigingPolicy` for that round does not exist yet, it will be used to consider the weight of the voter, as seen in the `Add()` method for the `submissionStorage`:

```
        message.weight += sp.voters.VoterWeight(voterIndex)
```

Recommendation

Do not accept votes for rounds in the future. This can be implemented easily by making `findByVotingRoundId` return `nil` always when the item was not found.

If this needs to be implemented to avoid posting the same `signingPoliciy` when it has not changed, delete votes for rounds to-come when a new signing policy is received.

Status

Fixed on commit `32a6c3a3a6cfe0b65390c8b65f29ea7baa67bf9c` of the https://github.com/flare-foundation/flare-system-client repository.

The signing process now retrieves the threshold of the active signing policy and ignores future voting rounds. Additionally, when a voting round is beyond the expected end of the last registered epoch, the client attempts to get more than the 60% of the voting weight before sending the data.

Proof of concept

```
func TestFindByVotingRoundId(t *testing.T) {
        newSp := func(rewardEpochId uint32) *signingPolicy {
                return &signingPolicy{
                        rewardEpochId:      int64(rewardEpochId),
                        startVotingRoundId: rewardEpochId + 100,
                        threshold:          200,
                        seed:               big.NewInt(42),
                        rawBytes:           []byte{0x42},
                        blockTimestamp:     99999,
                        voters: voters.NewVoterSet(

[]common.Address{common.HexToAddress("0xc2f249642d3c7bcf1380ccc20374c0f
516d2f8fe")},
                                []uint16{uint16(rewardEpochId)},
                        ),
```

```
                }
        }

storage := newSigningPolicyStorage()
        for i := 100; i < 400; i++ {
                err := storage.Add(newSp(uint32(i)))
                if err != nil {
                        t.Error(err)
                }
        }

// Note that findByVotingRoundId "lies" and will tell us that a signing
policy
        // is non-nil for a policiy that does not exist yet.
        // This will make the SubmissionStorage use this policy for the
submission
        // while counting the weight for a p.message.votingRoundId
which is in the future
        expected := 500
        result := storage.findByVotingRoundId(uint32(expected))
        if result.startVotingRoundId != uint32(expected) {
                t.Errorf("expected: %d, got: %d", expected,
result.startVotingRoundId)
        }
}
```

TOP-02

Rogue voter can vote for all merkle roots

Location

`client/finalizer/submission_storage.go`

Description

A malicious voter can exploit the `submitSignature` function to vote for *each* merkle root that appears, thereby ensuring they receive rewards for participating in the protocol. This not only guarantees rewards for the malicious voter but also incentivizes all voters to vote for each message indiscriminately. Such behavior complicates reaching a consensus that accurately reflects true facts.

To better understand the issue, it's important to recall the process by which voters send data to `submitSignature`. The data is encoded into a `messageData` struct by the **Top Level Client**:

```
type messageData struct {
        payload       []*signedPayload
        weight        uint16
```

```
        signingPolicy *signingPolicy
}
```

The vector of `messageData` is stored in a map called `vrMap` in the `submissionStorage`, which is a nested map from `(votingRoundId, votingRoundKey, messageHash) -> messageData`.

```
type votingRoundKey struct {
        protocolId  byte
        messageHash common.Hash
}

type votingRoundItem struct {
        msgMap map[votingRoundKey]*messageData
}

type submissionStorage struct {
        // Map from voting round id to voting round item, a map from
(protocol id, message hash) to message data
        // We use two maps instead of one to make it easier to remove a
voting round
        vrMap map[uint32]*votingRoundItem

// mutex
        sync.Mutex
}
```

To tally up the votes and trigger finalization, the weight of each message is counted. A crucial invariant that must be held is that voters should not be able to duplicate their voting weight by voting for the same `(votingRoundId, votingRoundKey, messageHash)` more than once. This is enforced through the following lines:

```
        voterIndex := sp.voters.VoterIndex(p.signer)
        if voterIndex < 0 {
                return addPayloadResult{}, fmt.Errorf("signer %s is not
a voter", p.signer.Hex())
        }
        if message.payload[voterIndex] != nil {
                return addPayloadResult{added: false}, nil // already
added
        }

p.index = voterIndex
    thresholdAlreadyReached := message.thresholdReached()
        message.payload[voterIndex] = p
```

The code snippet checks the index of the signer in the `signedPayload` according to the signing policy. If the signer is a valid voter, their index is used to determine whether their payload has already been added to `message.payload`. If not already added, the payload `p` is assigned to `message.payload[voterIndex]`.

However, this system does not consider votes for other messages: a voter can vote for several `messageHash` entries under the same `(votingRoundId, votingRoundKey)` without any restrictions, allowing them to vote multiple times for different messages with no penalty.

Recommendation

Ensure that when a voter casts a new vote, their voting weight is removed from any previous vote they have made. This prevents the accumulation of influence from multiple votes within the same voting round.

Status

Fixed.

This issue was addressed at the reward calculation layer, which is beyond the scope of this review. To further mitigate the problem, implementing penalties for double signing during reward allocation has been recommended.

© Coinspect 2024

TOP-03

Voters are incentivized to withhold their signatures

Location

`client/finalizer/submission_storage.go`

Description

A malicious voter can exploit the `relay` function by withholding their signature until their contribution causes the combined signature set to reach the threshold for finalization. Although this constitutes merely a griefing attack with minimal impact (since finalization would occur regardless), it encourages high-stake voters to delay publishing their signatures to avoid being preempted and to potentially secure more rewards. Rewards are distributed based on the stake of the first caller to `relay`, thus subverting the intended incentives: high-stake voters are motivated to wait as long as possible to minimize the risk of being outmaneuvered.

To illustrate this attack, let `V` be a malicious voter:

1. `V`, a low-stake voter within the voting set selected to finalize the current round, participates normally in subprotocols, executing `submit1` and `submit2` as required.
2. `V` monitors `submitSignatures` but does not reveal their own signatures.
3. `V` waits until the sum of the weights of the collected signatures plus the weight of their own signature is sufficient to exceed the threshold for finalization.
4. At this critical point, `V` calls `relay` with everyone's signatures, including their own, thus securing the first caller advantage.

This behavior is currently not subject to penalties as the `relay` method cannot verify whether signatures were previously disclosed via `submitSignatures`—this information exists solely in calldata.

Moreover, even if a mechanism to impose penalties were devised, `V` could circumvent it using a smart contract that combines the submission and relay steps into a single transaction:

```
function attack(submitSignaturePayload bytes, relayPayload bytes)
public {
    submissionContract.call(submitSignaturePayload);
    relayContract.call(relayPayload);
}
```

In fact, this attack strategy is particularly effective as it allows the attacker to earn rewards both for submitting a signature for the root that is going to be finalized and for being the one to finalize it.

While the attack itself does not pose a substantial immediate risk, its long-term consequences are significant: it can undermine the overall protocol. As mentioned earlier, higher-stake voters are motivated to maximize their rewards. Consequently, they tend to delay publishing their signatures as long as possible, which in turn delays the finalization process.

Recommendation

Consider implementing a single-leader scheme to mitigate manipulation by delaying signature submission. Additionally, revising the reward system so that it does not depend on who deposits first could reduce the incentive for this type of gaming.

Status

© Coinspect 2024

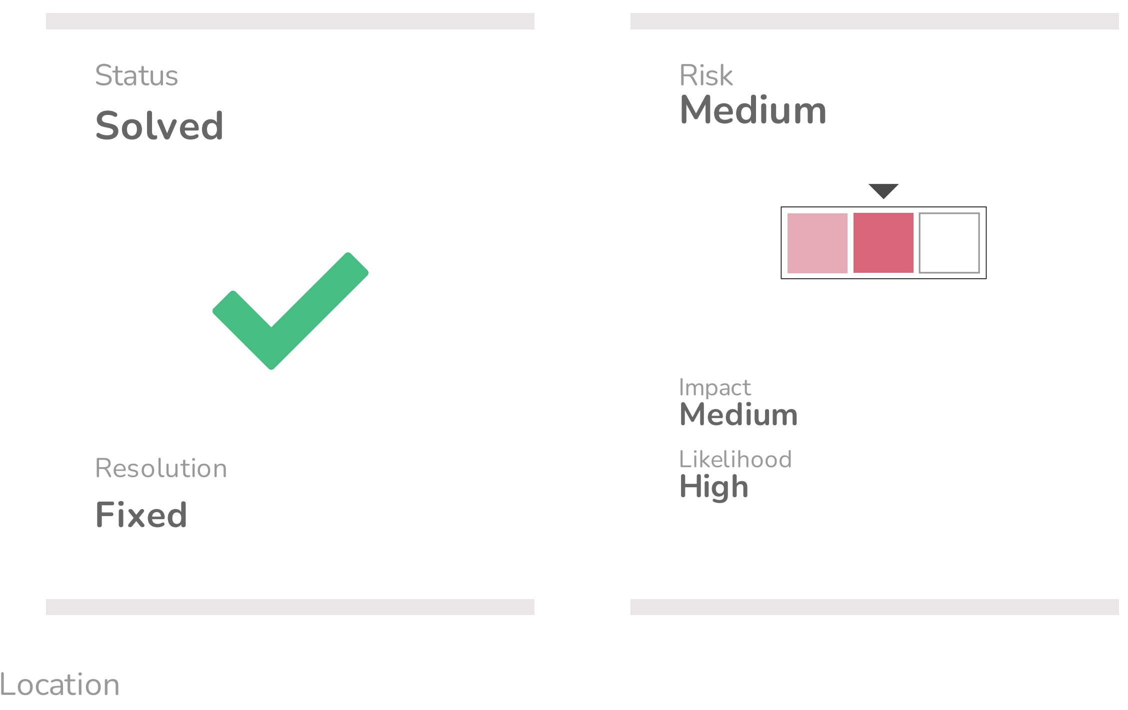
Acknowledged.

Flare has stated that the risk is mitigated through their reward calculation system. According to them, finalizers will receive special rewards only if they make their call after a designated grace period, which should discourage premature and strategic submissions.

© Coinspect 2024

TOP-04

Malicious voter can always be a finalizer



Location

`client/shared/voters/voter_set.go`

Description

A voter aiming to maximize their rewards can manipulate their position in the voter list to be selected as a finalizer more frequently. This is achieved by calculating which voter indexes are likely to be selected to finalize rewards, then brute-forcing addresses until one achieves a desired position when sorted lexicographically. The voter can then update their signing address to this strategically advantageous address.

To understand how this vulnerability can be exploited, consider two key facts:

1. **Selection of Finalizers**: Privileged finalizers are selected using an algorithm whose output can be predicted but not directly influenced by voters. This algorithm utilizes a `hash(protocolId, votingRoundId)` to select specific indexes from the voting set.

© Coinspect 2024

2. **Updatable Voting Set**: The voting set consists of signing addresses that are sorted l exicographically and can be updated by the voters' identity addresses, as per the specification:

```
voters - the list of eligible voters in canonical order (lexicographic
order). These are the signing addresses of voter entities.
```

Given these factors, the algorithm's predictability allows a voter to anticipate which indexes will be chosen. Although the algorithm itself cannot be directly influenced, the set it selects from can be manipulated. A malicious voter can thus gain an unfair advantage by calculating the advantageous indexes, creating addresses until one falls into the correct position, and updating their signing address accordingly for each reward epoch.

Additionally, consider how an attacker might combine this strategy with the vulnerabilities described in **TOP-03** to further exploit the system.

Recommendation

Use a Verifiable Random Algorithm to generate the indexes of the voters that are selected to be finalizers in this round. This would ensure that the random value is not liable to be predictable.

Consider implementing limits in how often signing addresses can be updated.

Status

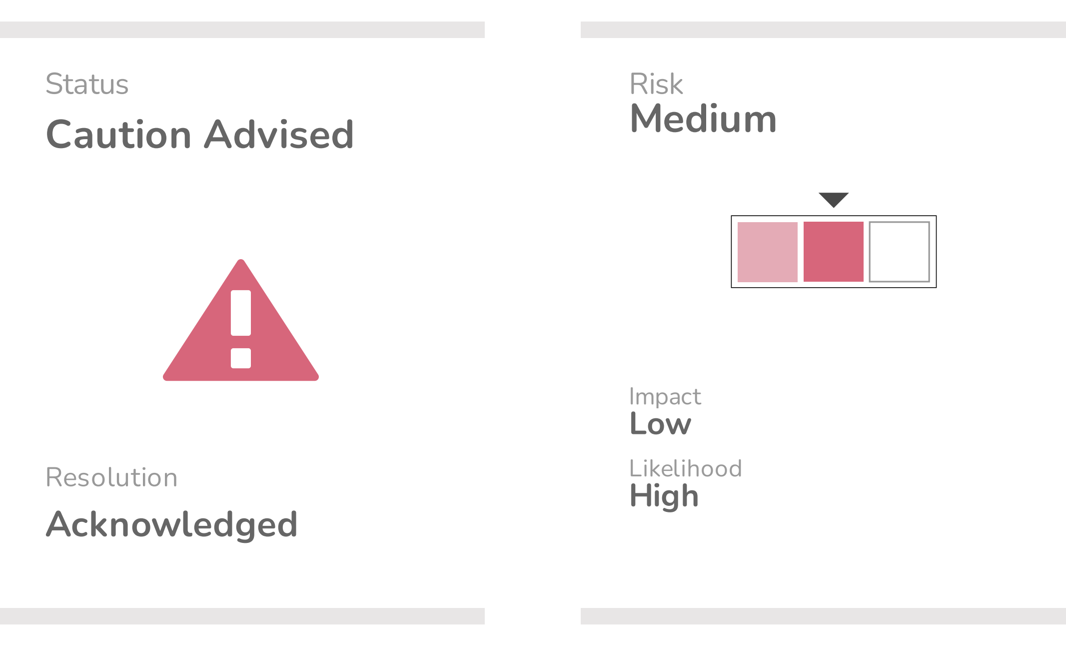
Fixed.

The finalizer subset of voters is generated based on the secure random seed of the current signing policy. As long as the random number provided by the contracts is secure, there is no exploit scenario for this issue.

© Coinspect 2024

TOP-05

Different signing policies have the same hash



Location

```
client/registration/system_manager_utils.go:96
```

Description

Because of how the `SigningPolicy` hash is constructed, different signing policies result in the same hash. This is known as a existencial forgery vulnerability. The precise impact of this vulnerability is hard to gauge, as it depends on how the hash and the resulting signature constructed from it are interpreted by smart contracts. Nevertheless, by itself it represents a misconstruction of the hashing function, as a core property for hashes it that finding collisions should be hard.

The root cause can be found in the function `SigningPolicyHash()` which calculates a hash of the `signingPolicyBytes` byte array.

It applies 0-padding to the input data to a 32-byte boundary using `0x00` bytes appended at the end of the array in this way:

© Coinspect 2024

```
if len(signingPolicy)%32 != 0 {
        signingPolicy = append(signingPolicy, make([]byte, 32-
len(signingPolicy)%32)...)
}
```

Due to this 0-padding process, by appending `0x00` to the initial input data and hashing it, you may create multiple (up to 32) signing policies sharing the same hash by appending zeroes. Since this hash serves as the basis for policy signing, one signature could potentially validate numerous distinct signing policies.

Recommendation

Avoid adding padding to your input data since the library functions already handle input padding correctly. If possible, opt for a standard hash function such as `crypto.Keccak256()`, rather than implementing custom hashing methods like `SigningPolicyHash()`. This way, you can ensure that the entire `signingPolicy` byte array gets properly hashed.

Status

Acknowledged.

The Flare Team stated that in general terms this could be an addressable issue. However, since signing policies are not blindly hashed but verified by the `Relay` smart contract, the case shown does not happen.

Proof of concept

```
func TestSigningPolicyHashCollision(t *testing.T) {
        signingPolicy := []byte{
                0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
                0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
                0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
                0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
                0x42, 0x00,
        }

signingPolicy2 := []byte{
                0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
```

```
0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
                0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
                0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
                0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
                0x42,
        }

hash1 := SigningPolicyHash(signingPolicy)
        hash2 := SigningPolicyHash(signingPolicy2)

if reflect.DeepEqual(hash1, hash2) {
                t.Error("different payloads should not result in same
hash")
        }

}
```

TOP-06

Private keys are not sufficiently protected

Location

Description

Attackers can leverage the fact that the private key guarding voting weights are not correctly protected, and even operators willing to support safer guards are not allowed to do so with the current implementation.

By leveraging the fact that the key is stored in text files, attackers that get access to certain amount of stake can trivially post erroneous data to the protocol.

While the system correctly considers keys intended for cold storage (`identity keys`) and those meant for hot storage, the hot storage keys do not support basic security measures such as being able to be stored in a secure secret storage.

It is worth noting that the separation between cold storage and hot storage is discouraged by the protocol itself, as keys intended for day-to-day communication with the blockchain are, by default, set to the cold-storage key according to the specifications:

© Coinspect 2024

```
In addition, a voter can set prioritized submission addresses that are
used for communication with blockchain.
These include submit addresses and a submit signatures address. By
default, prioritized addresses are set to the identity address.
```

Recommendation

Make sure that keys can be stored as safely as possible while fullfiling their roles. For hot-storage keys, that means at least supporting safer secret storage providers such as `AWS Secret Manager` or `Hashicorp`. This can easily be achieved by allowing keys to be read from the environment.

Additionally, make sure that no key for normal operations is set to be the one intended for cold-storage. Reject registrations that try to reuse the identity address for other purposes.

It is worth noting that when dealing with private keys there is **always** some risk of compromise, but following this guidelines minimizes the risk and impact of a compromise while at the same time remaining practical to implement.

Status

1. For private keys, this was fixed on commit `d8e8f61f9dc21f9cefba5ecfa12bf59445ca3056` of the https://github.com/flare-foundation/flare-system-client repository, as the keys are now retrieved from env variables.

Nevertheless, the `X-API-Key` protecting the communications with the `ftso-scaling` server is missing these protections, as can be seen in the `protocols.go` file.

```
XApiKey     string `toml:"x_api_key"` // Value of the X-API-KEY header
```

2. Fixed on commit `2d3d97ccf102b3bec3e4f9f8ca115fa850585091` of the https://github.com/flare-foundation/flare-system-client.

The server's API key is now retrieved from env variables.

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