Chain Cryptographic and Blockchain Assessment

Crypto.org

April 22, 2021 – Version 1.0

Prepared by
Paul Bottinelli
Ava Howell
Aleksandar Kircanski
Eric Schorn

Executive Summary

Synopsis

Starting in December 2020, Crypto.org engaged NCC Group's Cryptography Services team to conduct a cryptographic and blockchain security assessment of the Crypto.org Chain. The Chain application is built using the Cosmos SDK and the underlying Tendermint consensus engine. The engagement took places over multiple phases, starting in December 2020 and concluding in March 2021. The application was reviewed for any bugs related to transaction processing and integration with Tendermint via the Cosmos SDK. Though most of the critical logic is contained within the SDK itself as well as Tendermint, Crypto.org has implemented 3 custom SDK modules: `chainmain`, `subscription`, and `supply`. These modules were reviewed for any possible security issues resulting from their handling of state-changing message contained in transactions, as well as for any issues with their front ends (defined in the `cli/` directories).

The final phase focused on the implementation of `subscription` module at https://github.com/crypto-org-chain/chain-main/tree/master/x/subscription/spec during March 2021 and two person-days were devoted to this phase.

Scope

The assessment was primarily focused on the three custom Cosmos SDK modules implemented by Crypto.org, which implement the chain logic:

- **chainmain** The primary custom SDK module. Defines a number of CLI commands for interacting with the Crypto.org chain.
- **subscription** A module for creating/stopping "subscription plans" on the Crypto.org chain
- **supply** A module for querying token supply.

In addition, a holistic review of the entire chain repository (located at https://github.com/crypto-org-chain/chain-main) was conducted, including reviewing configuration files such as the genesis configuration file.

Testing was performed using code review on a local instance.

Key Findings

The assessment uncovered a number of findings. Initially, one high severity finding was discovered, which was later discovered to be a false positive due to the inclusion of a custom patch which fixed the issue in `go.mod`. The remaining findings were all low severity, including (but not limited to):

- **Insufficient Input Validation of Signed Integer CLI Arguments**: Insufficiently validated signed-integers extracted from CLI arguments may allow negative values to trigger undesired behavior in downstream logic.
- **Empty Genesis Validation Function**: Missing validation operations may allow attackers to introduce malicious data into the system, potentially negatively affecting downstream logic. In the case of the Genesis state, inconsistent information may result in consensus breaches.

Dashboard

Target Metadata

Name	chain-main
Type	Cosmos SDK Application
Platforms	Go
Environment	Local Instance

Engagement Data

Type	Code Review
Method	Code-assisted
Dates	2020-12-29 to 2021-02-26
Consultants	3
Level of Effort	10 person-days

Targets

Chain Repository	https://github.com/crypto-com/chain-main

Finding Breakdown

Critical issues	0	
High issues	1	🟥
Medium issues	0	
Low issues	4	🟨🟨🟨🟨
Informational issues	3	⬜⬜⬜
Total issues	**8**	

Category Breakdown

Configuration	2	🟨🟨
Data Validation	3	🟨🟨⬜
Other	2	⬜⬜
Patching	1	🟥

Key

🟥 Critical 🟥 High 🟧 Medium 🟨 Low ⬜ Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 17.

Title	Status	ID	Risk
Use of Cosmos SDK v0.41.x	Fixed	007	High
Insufficient Input Validation of Signed Integer CLI Arguments	New	001	Low
Empty Genesis Validation Function	New	002	Low
Extraneous CLI Flag Option	New	003	Low
Genesis Configuration Notes	New	006	Low
Potential for Missing Unicode Normalization	New	004	Informational
Genesis File is Updated Unsafely	Fixed	005	Informational
Error Ambiguity in `BitSet.Test`	New	008	Informational

Finding Details

Finding	**Use of Cosmos SDK v0.41.x**
Risk	**High** Impact: High, Exploitability: High
Identifier	NCC-CRYX002-007
Status	Fixed
Category	Patching
Location	`go.mod`
Impact	Due to a bug in the `v0.41.x` handling of validator slashing evidence, a validator will not be able to be successfully slashed on the Crypto.org chain.

Description

Cosmos SDK version 0.41.x contains a bug which prevents chains which uses a non-standard `Bech32PrefixConsAddr` from correctly decoding validator addresses in the handling of evidence. This bug is noted in the release notes for 0.41.x:

```
IMPORTANT: Due to a bug in the v0.41.x series with how evidence handles validator
→ consensus addresses (#8461), SDK based chains that are not using the default
→ bech32 prefix (cosmos, aka all chains except for the Cosmos Hub) should not u
→ se this release or any release in the v0.41.x series. Please see #8668 for tr
→ acking & timeline for the v0.42.0 release, which will include a fix for this
→ issue.
```

The Crypto.org chain as of commit `66b350397c16aa788dad99afb4f44c604e3e0d84` was found to use 0.41.x:

```
// ...snip.. (go.mod)
require (
    github.com/confluentinc/bincover v0.1.0
    **github.com/cosmos/cosmos-sdk v0.41.0**
    github.com/cosmos/ledger-go v0.9.2 // indirect
    github.com/gogo/protobuf v1.3.3
    github.com/golang/protobuf v1.4.3
    github.com/gorilla/mux v1.8.0
    github.com/grpc-ecosystem/grpc-gateway v1.16.0
    github.com/imdario/mergo v0.3.11
    github.com/rakyll/statik v0.1.7
    github.com/spf13/cast v1.3.1
    github.com/spf13/cobra v1.1.1
    github.com/stretchr/testify v1.7.0
    github.com/tendermint/tendermint v0.34.7
    github.com/tendermint/tm-db v0.6.4
    google.golang.org/genproto v0.0.0-20210114201628-6edceaf6022f
    google.golang.org/grpc v1.35.0
)
```

Since the Crypto.org chain uses the Bech32 prefix of `cro`, this bug applies.

Recommendation Upgrade to `v0.42.x` before releasing the Crypto.org chain.

Retest Results As of the most recent commit (`7a49be9`), the Cosmos SDK has been updated and as such this issue is resolved.

Finding Insufficient Input Validation of Signed Integer CLI Arguments

Risk Low Impact: Medium, Exploitability: Low

Identifier NCC-CRYX002-001

Status New

Category Data Validation

Location
- Lines 85 and 97 of chain-main/x/chainmain/client/cli/genaccounts.go
- Lines 100, 195 and 199 of chain-main/x/chainmain/client/cli/testnet.go

Impact Insufficiently validated signed-integers extracted from CLI arguments may allow negative values to trigger undesired behavior in downstream logic.

Description Insufficient input validation is one of the most common cause of software vulnerabilities. The code at the bottom of the `genaccounts.go` source file excerpted below describes a number of values that may be passed via command line flags. The latter two values involving vesting `Start` and `End` can be either a RFC-3339 date or a unix timestamp. A valid unix timestamp can never be negative.

```
207 cmd.Flags().String(flags.FlagHome, defaultNodeHome,
     ➜  "The application home directory")
208 cmd.Flags().String(flags.FlagKeyringBackend, flags.DefaultKeyringBackend,
     ➜  "Select keyring's backend (os|file|kwallet|pass|test)")
209 cmd.Flags().String(flagVestingAmt, "", "amount of coins for vesting accounts")
210 cmd.Flags().String(flagVestingStart, "0", "schedule start time (RFC-
     ➜  3339 format or unix timestamp) for vesting accounts")
     ➜
211 cmd.Flags().String(flagVestingEnd, "0",
     ➜  "schedule end time (RFC-3339 format or unix timestamp) for vesting accounts")
212 flags.AddQueryFlagsToCmd(cmd)
```

The parsing logic for the `vestingStart` argument is shown below, where the value is parsed as a **signed** integer on line 85.

```
81 vestingStartStr, errstart := cmd.Flags().GetString(flagVestingStart)
82 if errstart != nil {
83     return fmt.Errorf("failed to parse vesting start: %w", errstart)
84 }
85 vestingStart, errstart := strconv.ParseInt(vestingStartStr, 10, 64)
86 if errstart != nil {
87     vestingStart, errstart = ParseTime(vestingStartStr)
88     if errstart != nil {
89         return fmt.Errorf("failed to parse vesting start: %w", errstart)
90 }   }
```

While the subsequent switch statement on lines 128-137 perform non-zero checks, the argument is never checked for a negative value prior to use. Thus, inappropriate values may be sent into downstream logic.

The above scenario pertains to the `vestingStart` and `vestingEnd` arguments in the `genaccounts.go` source file, and the `numValidators`, `vestingStart` and `vestingEnd` arguments in the `testnet.go` source file.

Recommendation Add checks for negative CLI argument values preferably alongside their parsing logic, or in the subsequent switch statements (which are currently performing the non-zero check only).

Finding	**Empty Genesis Validation Function**
Risk	**Low** Impact: Medium, Exploitability: Low
Identifier	NCC-CRYX002-002
Status	New
Category	Data Validation
Location	• Lines 13 to 15 of chain-main/x/chainmain/types/genesis.go • Lines 10 to 12 of chain-main/x/supply/types/genesis.go
Impact	Missing validation operations may allow attackers to introduce malicious data into the system, potentially negatively affecting downstream logic. In the case of the Genesis state, inconsistent information may result in consensus breaches.

Description The `GenesisState` structure represents the state of the blockchain at creation. In order to ensure its validity and consistency, a `Validate()` method is defined. In its current state, this function does not perform any validation and always succeed, by returning `nil` (even though it states that an error will be returned upon failure). This can be seen in the following code excerpt taken from the relevant files.

```go
// Validate performs basic genesis state validation returning an error upon any
// failure.
func (gs GenesisState) Validate() error {
        return nil
}
```

The `Validate()` function is called in multiple places, such as in the `ValidateGenesis()` functions (both in `x/supply/module.go` and `x/chainmain/module.go`), as can be seen below.

```go
// ValidateGenesis performs genesis state validation for the capability module.
func (AppModuleBasic) ValidateGenesis(cdc codec.JSONMarshaler,
  config client.TxEncodingConfig, bz json.RawMessage) error {
  // ...
  return genState.Validate()
}
```

Additionally, it is also called in the `InitTestnet()` function in `x/chainmain/client/cli/testnet.go` or in the `AddGenesisAccountCmd` function from `x/chainmain/client/cli/genaccounts.go`. Therefore, in all these instances, no validation is actually performed.

Recommendation Update the `Validate()` function to perform stricter validation of the Genesis state.

Finding **Extraneous CLI Flag Option**

Risk **Low** Impact: Low, Exploitability: Low

Identifier NCC-CRYX002-003

Status New

Category Configuration

Location chain-main/x/chainmain/client/cli/genaccounts.go

Impact An extraneous CLI flag option may indicate mismatched functional, documented or operational expectations. In this case, a `memory` keyring backend can be supplied, which will cause the contents to be non-persistent.

Description The CLI flag help text near the end of the `genaccounts.go` source file excerpted below suggests that valid options for the keyring backend includes `os`, `file`, `kwallet`, `pass` and `test` per line 208.

```
207 cmd.Flags().String(flags.FlagHome, defaultNodeHome,
    ➜  "The application home directory")
208 cmd.Flags().String(flags.FlagKeyringBackend, flags.DefaultKeyringBackend,
    ➜  "Select keyring's backend (os|file|kwallet|pass|test)")
209 cmd.Flags().String(flagVestingAmt, "", "amount of coins for vesting accounts")
210 cmd.Flags().String(flagVestingStart, "0", "schedule start time (RFC-
    ➜  3339 format or unix timestamp) for vesting accounts")
    ➜
211 cmd.Flags().String(flagVestingEnd, "0",
    ➜  "schedule end time (RFC-3339 format or unix timestamp) for vesting accounts")
212 flags.AddQueryFlagsToCmd(cmd)
```

However, logic in the same source file excerpted below simply extracts the command line flag value on line 56 and passes it directly to `keyring.New()` on line 63.

```
56 keyringBackend, kerr := cmd.Flags().GetString(flags.FlagKeyringBackend)
57 if kerr != nil {
58     return kerr
59 }
60
61 // attempt to lookup address from Keybase if no address was provided
62 // nolint: govet
63 kb, err := keyring.New(sdk.KeyringServiceName(), keyringBackend,
   ➜  clientCtx.HomeDir, inBuf)
```

The `keyring.New()` function can be found in the Cosmos SDK `crypto/keyring/keyring.go` source file where the backend value is 'decoded'. The first option on line 164 pertains to a `memory` configuration which configures subsequent behavior.

Thus, the `memory` option can be passed via the Chain CLI which will affect keyring backend operation outside of that specified in the flag's help text.

Recommendation Adapt the flag help text to include the `memory` option, or filter out the `memory` option prior to passing the backend value into `keyring.New()`.

Finding	**Genesis Configuration Notes**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-CRYX002-006
Status	New
Category	Configuration
Impact	Improper consensus configuration may result in network disruption.

Description The testnet genesis configuration for the Crypto.org cosmos chain specifies the following parameters:

```
    "block": {
      "max_bytes": "22020096",
      "max_gas": "-1",
      "time_iota_ms": "1000"
    },
    "evidence": {
      "max_age_num_blocks": "100000",
      "max_age_duration": "172800000000000",
      "max_num": 50
    },
```

As part of the review of the Crypto chain, NCC Group evaluated the configuration in the genesis file. Note that the maximum gas, the transaction cost, is unlimited. This may cause issues where users of the network are able to submit very costly to validate transactions which slow down the network (at a cost).

Additionally, the maximum block size is very large, at 22MB. At a block time of 1 second, this implies that to participate in the consensus protocol in a timely manner and avoid missing blocks in all cases, a validators must have a bandwidth capacity with each other of at least 22MB/s. This is a rather intense requirement, and may result in a higher degree of validator centralization.

Finally, the `max_age` parameter is set such that slashing evidence essentially never expires. This may disrupt long-term network incentives, for reference the Cosmos Hub currently sets `max_age` at 100000 blocks.

Recommendation Reach out to the Tendermint team for guidance on the optimal settings for the parameters noted in this finding. Consider reviewing the genesis files for Cosmos Hub, Terra, and other Cosmos zones.

Finding	**Potential for Missing Unicode Normalization**
Risk	**Informational** Impact: High, Exploitability: Medium
Identifier	NCC-CRYX002-004
Status	New
Category	Data Validation
Location	Documentation specifying `mint_denom` and `bond_denom` as strings at: https://crypto.org/docs/chain-details/parameters.html
Impact	A missing unicode normalization step on string identifiers may allow seemingly duplicate entries, unexpected downstream miscomparisons and/or user confusion.

Description This is a proactive finding intended to raise awareness of a potential pitfall involving Golang strings and Unicode normalization. As no code risks have been specifically identified, this finding is categorized as Informational.

UTF-8 is the de facto character encoding used on the modern web[1] and in Golang applications. However, a Golang string is no more than simply a slice of bytes. The official Golang blog[2] states:

> It's important to state right up front that a string holds arbitrary bytes. It is not required to hold Unicode text, UTF-8 text, or any other predefined format. As far as the content of a string is concerned, it is exactly equivalent to a slice of bytes.

Unicode characters with accents or other modifiers can have multiple correct encodings. For example, the Á (a-acute) glyph can be encoded as a single character U+00C1 (the "composed" form) or as two separate characters U+0041 then U+0301 (the "decomposed" form). In some cases, the order of a glyph's combining elements is significant and in other cases different orders must be considered equivalent.[3] Normalization[4] is the process of standardizing string representation such that if two strings are canonically equivalent and are normalized to the same normal form, their byte representations will be the same. Only then can string comparison and ordering operations be relied upon.

The Chain documentation defines the `mint_denom` and `bond_denom` parameters as strings. Further, this finding is applicable to all string identifiers within Chain. The code repository shows no indication of Unicode normalization functionality.

String normalization is significant for Chain because participants may establish `mint_denom` or `bond_denom` identifiers involving these type of characters. Without string normalization, different participants may establish identifiers that appear identical but are in fact distinct such as Bank of Álpha and Bank of Álpha. From the developer perspective, downstream logic may attempt to perform string operations under the assumption of correctness (potentially involving Golang string literals which are always properly encoded UTF-8 bytes) with undesired results such as panics. Golang provides a string validation utility function[5] that can validate proper encoding. Note that Golang version 1.13 and above provides `strings.ToValidUTF8` [6] which removes invalid runes but does not perform normalization. There are four standard

[1] https://w3techs.com/technologies/details/en-utf8
[2] https://blog.golang.org/strings
[3] http://unicode.org/reports/tr15/tr15-22.html
[4] https://blog.golang.org/normalization
[5] https://golang.org/pkg/unicode/utf8/#ValidString
[6] https://golang.org/pkg/strings/#ToValidUTF8

normalization forms,[7] of which NFKC[8] is the most popular and is most suitable for the Chain application.

More information about Unicode and UTF-8 attacks can be found in this Black Hat presentation: https://www.blackhat.com/presentations/bh-usa-09/WEBER/BHUSA09-Weber-UnicodeSecurityPreview-SLIDES.pdf.

Recommendation Utilize the `strings.ToValidUTF8` function or the `ValidString()` function from the Golang `utf8` package, to ensure correct character encoding immediately after byte deserialization. Further, perform string normalization to form NFKC on both deserialization and serialization to ensure a consistent string representation. The Golang `norm` package[9] provides suitable functionality for this purpose.

Another alternative is to ensure strings consist of only ASCII. This would need to be clearly documented.

[7] http://unicode.org/reports/tr15/#Norm_Forms
[8] See question 2 of https://unicode.org/faq/normalization.html
[9] https://godoc.org/golang.org/x/text/unicode/norm

Finding	**Genesis File is Updated Unsafely**
Risk	**Informational** Impact: None, Exploitability: None
Identifier	NCC-CRYX002-005
Status	Fixed
Category	Other
Location	`cmd/chain-maind/app/app.go`
Impact	If a machine running the Crypto.org chain experiences a sudden fault, such as a power outage, while running the `init` command, the genesis file will be corrupted and may result in an inability to sync with the rest of the network.

Description The client code for the Crypto.org chain defines an `init` command which merges an existing `genesis.json` file, used to bootstrap the blockchain state, with a statically defined map:

```
	initCmd := genutilcli.InitCmd(app.ModuleBasics, app.DefaultNodeHome)
	initCmd.PostRunE = func(cmd *cobra.Command, args []string) error {
		genesisPatch := map[string]interface{}{
			"app_state": map[string]interface{}{
				// ...snip..., hard-coded genesis states
			},
		}
		file, err := os.OpenFile(cleanedPath, os.O_RDWR, 0600)
		if err != nil {
			return err
		}
		defer func() {
			if closeErr := file.Close(); closeErr != nil {
				fmt.Printf("Error closing file: %s\n", closeErr)
			}
		}()
		var genesis map[string]interface{}
		if err := json.NewDecoder(file).Decode(&genesis); err != nil {
			return err
		}

		if err := mergo.Merge(&genesis, genesisPatch,
		 ➜ mergo.WithOverride); err != nil {
			return err
		}
		if err := file.Truncate(0); err != nil {
			return err
		}
		if _, err := file.Seek(0, 0); err != nil {
			return err
		}
		return json.NewEncoder(file).Encode(&genesis)
```

Note that this method of updating the genesis file is not *atomic*: the genesis file can be left in a corrupted state if the machine running the `init` command experiences a fault while writing the genesis file (such as a power outage). This corruption most likely would result in the genesis file simply needing to be re-downloaded as the JSON will be invalid, however corruption of state within the genesis file may also occur.

Each time a command is invoked, a file called the Genesis JSON file is mutated and saved.

This procedure isn't done safely: if the machine running the command crashes during saving, the JSON file may be corrupted.

They should instead safely save the file, by writing to a temporary output, syncing, renaming to the final file, and syncing again.

Recommendation Update the code which writes to the Genesis json file such that the file is updated *atomically*. This can be accomplished by writing the file to a temporary output (such as `genesis.json.tmp`), syncing that output using `file.Sync()`, calling `os.Rename` to rename the `genesis.json.tmp` to `genesis.json`, and finally applying a final `Sync()`.

Finding **Error Ambiguity in `BitSet.Test`**

Risk **Informational** Impact: Low, Exploitability: None

Identifier NCC-CRYX002-008

Status New

Category Other

Location https://github.com/crypto-org-chain/chain-main/blob/35c802b53e4b69512d13614b9cfae4949077abfd/x/subscription/types/bitset.go#L50

Impact In future code revisions, an out of bounds bit test error may disregarded, resulting in incorrect computation.

Description To support validation of certain message types, `subscription/types/bitset.go` defines a number of common bit operations on 64-bit words. Consider how `BitSet.Set` and `BitSet.Clear` functions are defined:

```
// Set bit i to 1
// if i >= 64, return an error
func (b *BitSet) Set(i uint) error {
        if i >= wordSize {
                return errorOutOfBound
        }
        b.set |= 1 << i
        return nil
}

// Clear bit i to 0
// if i >= 64, return an error
func (b *BitSet) Clear(i uint) error {
        if i >= wordSize {
                return errorOutOfBound
        }
        b.set &^= 1 << i
        return nil
}
```

In both cases, if the supplied bit position is larger than the word size, an error is returned. Now, consider the `BitSet.Test` method:

```
// Test whether bit i is set.
// if i >= 64, return false
func (b *BitSet) Test(i uint) bool {
        if i >= wordSize {
                return false
        }
        return b.set&(1<<i) != 0
}
```

This function conflates errors and legitimate bit testing results. Once the function returns `false`, there's no way for the caller to know whether the passed index was out of bounds or the bit was not set. It is assumed that the development process took a shortcut in this case, since currently the `BitSet.Test` function is only invoked in `CompiledCronSpec.Roundup` and the parameter cannot be greater than 64. It would make sense to rewrite this function to avoid any potential mistakes in future code iterations.

Recommendation	Rewrite the function to return a boolean *and* an error. If the supplied argument is out of bounds, set the error.

Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

Appendix B: Engagement Notes and Observations

1. Overview

This informational section highlights selected portions of the engagement methodology used, a sample of priority concerns investigated, and a few observations that do not warrant security-related findings but may lead to topics of discussion. The primary strategy for this project relied on manual source code inspection supported by limited execution of the included test cases. Priority was given to inspecting the code's conformance to specification, the correctness of cryptographic algorithms and implementation, along with general secure coding practices that could potentially impact legitimate operation.

Note: this initial content will be extended throughout the duration of the full engagement.

2. Scope and Related References

The primary sources of material used for the engagement included:

- Commit `a6f3afc` of https://github.com/crypto-com/chain-main
- Initial focus on `chain-main/x/chainmain` and `chain-main/x/supply` subdirectories
- Sampled focus on other non-test components of `chain-main`.
- Chain documentation
- Tendermint core documentation

3. Initial Survey

This subsection is a basic review of important low-level implementation characteristics performed after an initial documentation review. While the entire code base is surveyed, only pertinent/interesting points noted.

- The latest Golang toolchain `1.15` is specified and primary dependencies are (very closely) up to date, via: `go list -u -m -json all | go-mod-outdated -update -direct`
- Sources and uses of randomness were considered.
- The appropriateness of the selected ciphersuites was reviewed, e.g., ed2519 and secp256k1.
- Deserialization and serialization were reviewed at length:
  - Protocol buffers
  - CLI flags, see finding NCC-CRYX002-001 on page 6.
  - Good to see `0o600` permissions set on file write.
- Unicode string normalization was reviewed across code and documentation.
- Validation functionality/opportunities were reviewed.

4. Observations

- In the functions `TotalSupplyHandlerFn()` and `LiquidSupplyHandlerFn()` located in the file `x/supply/client/rest/query.go`, the client context at a given height (`clientCtx`) is first obtained by calling the function `ParseQueryHeightOrReturnBadRequest()`. After performing their respective queries (`QueryWithData`), these two functions obtain the client context at the height returned, with a call to `clientCtx = clientCtx.WithHeight(height)` (on lines 34 and 61, respectively). Unless the height returned from `QueryWithData()` is different, it seems that this operation might be redundant, since the function `ParseQueryHeightOrReturnBadRequest()` already returns the client context at the given height.
- A number of deprecated and legacy functions are still in use throughout the codebase. Consider deleting or updating these functions. For example, the NCC Group team noted usages of the deprecated[10] `AminoCodec` in the file `x/supply/types/codec.go` (this also applies to the file `x/chainmain/types/codec.go`). These are called by the method

[10] https://docs.cosmos.network/master/core/encoding.html

`RegisterLegacyAminoCodec` of `AppModuleBasic` in `x/supply/module.go`.

- Similarly, consider removing functions that are not used, such as the method `LegacyQuerierHandler()` of `AppModule` defined in `x/chainmain/module.go`.
- Most of the functions in `x/chainmain/module.go` are either empty, or perform default behavior.
- Similarly, the function `InitGenesis()` in `x/chainmain/genesis.go` is empty and the function `ExportGenesis()` of the same file returns the default genesis.
- The function `RegisterRoutes()` in `x/chainmain/client/rest/rest.go` is also empty.