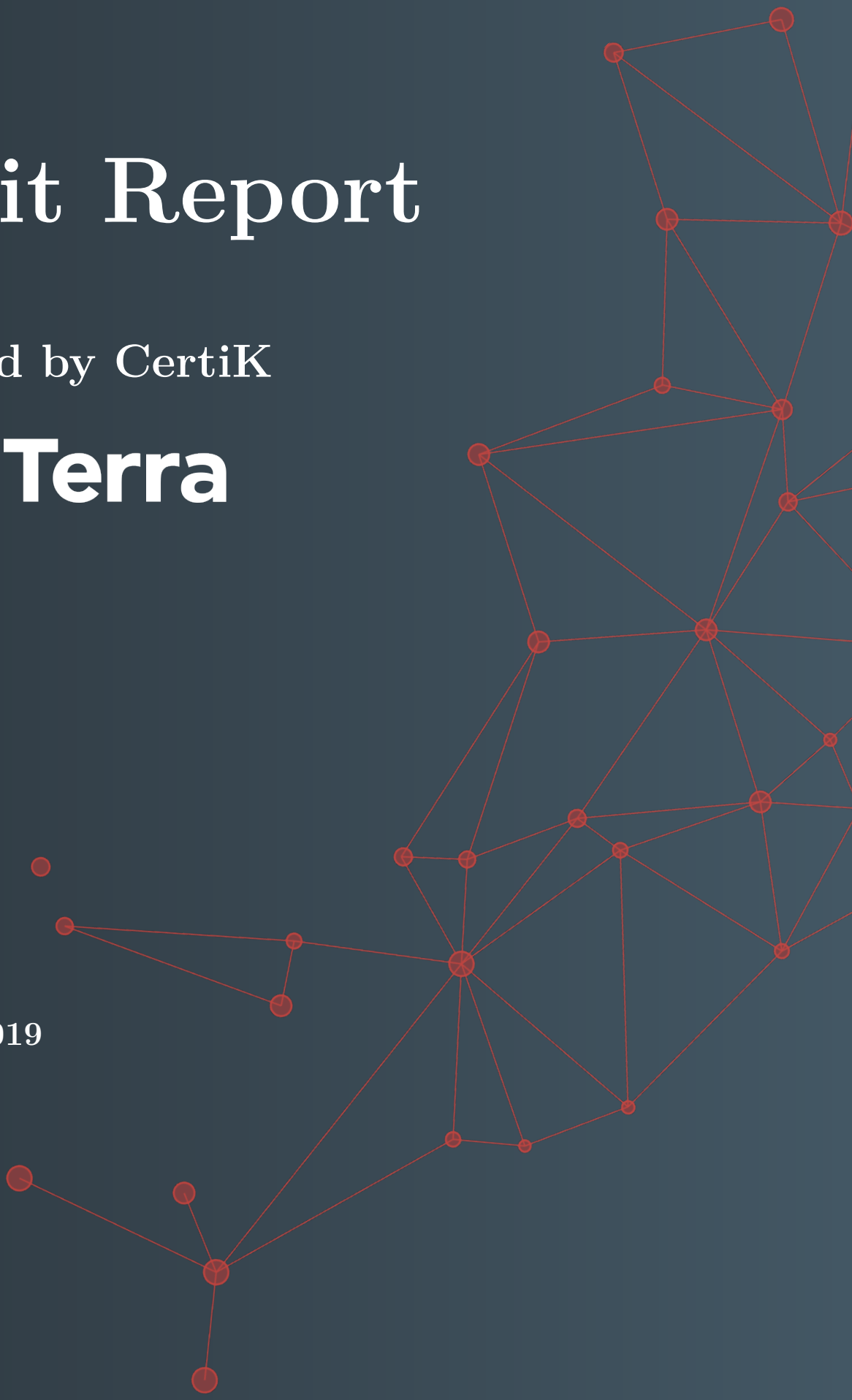
CERTIK

Audit Report

Produced by CertiK

for Terra

May 6th, 2019



Disclaimer

This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and Terra(the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This Report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets.

For more information: https://certik.org/

Manual Review Notes

Summary

CertiK has been performing a security audit of Terra, a blockchain providing stable-coin cryptocurrency via automated macro-economics style currency adjustment and incentives. The audit work is done by economists, security researchers and security engineers from CertiK, with collaboration from the Terra research and development team. The audit work initially started in January 2019 and wrapped up in early May 2019.

The auditing work, including those done and to be done, including the following aspects.

- First, CertiK studied the Terra whitepaper as well as other research, analysis, and design documents from Terra, to comprehend the blockchain features and complete semantics the Terra protocol is supposed to achieve. From economics theory's standpoint, CertiK validated the correctness of the mathematical reasoning, the model's robustness against potential high-level ill-intentioned currency manipulations, and also noted all the model parameters, especially those of empirical values or prune to later dynamic modifications once the mainnet is up. For cases not immediately obvious, CertiK asked Terra for either explanation of empirical study leading to them or advised Terra to protect them and pay special attention in their future modifications.
- Secondly, CertiK audited the entire blockchain code base of Terra, which is written in Go language and constructed using the Cosmos Blockchain SDK / Tendermint Consensus Engine. The implementation consists of an ABCI application, several Cosmos SDK plugins (keepers), and a few command-line binaries. The main focus of the audit is on validating the implementation of each plugin conforms to and fully implements aspects of the theoretical model of Terra protocol. Special attention is also put on making sure the implementation is well-protected against possible edge scenarios and attacks. Common blockchain / distributed systems' safety, correctness, liveness, and security problem patterns are also checked for the implementation. Wherever making sense, CertiK also checked and commented on detailed design, modularity, and future-proof code patterns. CertiK also looked at the tests of Terra and checked for their coverage.

Overall, CertiK found Terra's theoretical model as well as the Go implementation to be well designed and executed. While CertiK does not comment on the final performance of the Terra blockchain, the modeling and mathematical reasoning are considered sound. The implementation is clean and well-structured and demonstrated a very good command of Go programming patterns as well as common good programming practices. CertiK has seen reasonably good test coverage, too.

Scope of Work

- The audit work was initiated basing on an older version of Terra code base and documents and later moved on to newer versions. The code and documents this

report is based on is from the May 1, 2019 snapshot with change set hash $d6341ad6c9c2fd5149b3001c7d8567701ac3aed3$.

- Only non-test Go source code under `terra-project/core` are audited. Test Go code, as well as dependent libraries' code, are read and analyzed by need for the auditing, but no special manual effort has been spent on establishing their security, correctness, or effectiveness. Rather, those are assumed to hold for the purpose of this audit.
- Since most of the non-keeper code, such as those located under folders like `app`, `client`, `cmd`, `server`, and version directories, are either forked from CosmosSDK or mostly boilerplate, they received less attention during the audit than the keeper code.
- Among the keeper code (located under `/x/<keeper>`) , more attention is spent on the keeper modules themselves, rather than the keeper client code (located under `/x/<keeper>/client`), which are mostly wrapper calls to the keeper modules.
- As there have been interactions with Terra during the auditing process, and as the audit target has changed its version over time, not all audit opinions or comments are necessarily included in this report.

General Assumptions and Comments

1. Terra's economic theory is based on simulating real-world currency governance, in order to achieve stable-coin behaviors anchored to real-world currency behaviors, as well as to stimulate planned growth of the chain expecting to face similar economic cycles as real-world economy.
2. Of the two goals above, the stable-coin behavior is important for end users of the chain, while the planned growth is important for stakeholders of the chain.
3. Terra will assume majority control of the chain stake in the beginning and will only gradually increase the diversity of chain governance over time. Decentralization is not a top priority of the chain. Hence most of diverse-stake attack models applicable to some other blockchains will not be an issue for Terra.
4. Terra design is theory-heavy. Most of the crucial design decisions are in the Terra economics. The actual implementation architecture is rather straight-forward, comparatively, and follows good Cosmos-SDK usage practices in general.

Correctness Comments

1. In `/x/budget/params.go`, function `validateParams()`, the following code and its message do not match:

```
if params.ActiveThreshold.LT(sdk.ZeroDec()) {
  return fmt.Errorf("budget active threshold should be greater
  ↪  than 0, is %s", params.ActiveThreshold.String())
}
```

It seems like the error should be thrown on LTE() instead of LT(), if the message is correct.

2. In `/x/pay/handler.go`, function `handleMsgMultiSend()`, tax is being paid serially for all inputs, but might bail out if one of them fails to pay. So when a later processed input fails and bails out, all previously paid taxes for other inputs will still take effect. Plus, all unprocessed input will not pay the tax here.

```
for _, input := range msg.Inputs {
  taxErr := payTax(ctx, k, tk, fk, input.Address, input.Coins)
  if taxErr != nil {
    return taxErr.Result()
    ...
  }
  ...
}
```

It is not entirely clear if such a partial tax-paying state is possible for the chain. In the case it is not, there should be atomic checking of sufficient funds before calling this function, and the tax paying error should be a panic instead, as it is unexpected and indicates bug somewhere else. In the case it is yes, the right way to do it seems to be either:

(a) making sure there is enough fund for all tax-paying before starting to pay any of them, or

(b) accumulate tax paying errors but do not bailed out until all inputs are processed.

We prefer solution (a), as in solution (b) one will also need to figure out how to accumulate and return multiple errors, when multiple tax pay actions fail.

3. In both `/x/mint/keeper.go` and `/x/treasury/keeper.go` we are seeing unnecessary recursive calls, such as

```
func (k Keeper) GetIssuance(...) (issuance sdk.Int) {
  if bz := store.Get(keyIssuance(denom, day)); bz != nil {
    ...
  } else {
    ...
    if day.LTE(sdk.ZeroInt()) {
      ...
    } else {
      // Fetch the issuance snapshot of the previous epoch
      issuance = k.GetIssuance(ctx, denom, day.Sub(sdk.OneInt()))
    }
    ...
  }
}
```

As the depth of recursion corresponds to the history of chain, this not only constantly increases the worst-case execution time of these functions, but also might eventually face stack size limit and cause the chain to crash one day, especially if the epoch span is shortened.

The solution is to convert this to a loop straight-forwardly.

Performance and Style Comments

1. In `/types/claimpool.go`, function `Sort()` is actually merely doing merging rather than sorting. A better name would avoid future confusions.

2. In `/x/budget/keeper.go`, `programID` is parsed out from the key elements. It is not clear why Program.ProgramId cannot be used directly instead, which would simplify code and avoid costly parsing operations.

3. In both `/x/mint/keeper.go` and `/x/treasury/keeper.go`, `GetIssuance()`, `GetRewardWeight()`, and `GetTaxRate()` are all storing newly calculated results into the persistent state at current epoch, to serve as cached results for future inquiry. It is not certain if they need to be put into the persistent state, which is costly in the long-run, or they would be served well with some transient stores instead. We think it is the second case.

Go / CosmosSDK Keeper Typical Issues Audited

1. Issue: Usage of Go `unsafe` statement, which potentially can cause ill-typed data, illegal pointers, buffer overruns, arithmetic overflows, as well as compromised correctness.

   - Result: No usage is found.

2. Issue: Usage of Go type assertion without success checking, e.g., `v := x.(T)` instead of `v, ok := x.(T)`, which can cause unexpected panic if the type assertion fails.

   - Result: All usage properly check for the type assertion success.

3. Issue: Concurrent usage of Go maps, which is neither atomic nor thread-safe.

   - Result: No usage is found.

4. Issue: Usage of Go slice slicing and append features without being careful with slice aliasing or zero-value "initialization" of slice elements. Go slices are actually reference types that might alias or unalias silently during slicing or growing. Also, `[len:cap]` area can be filled with previous non-zero values, hence slice elements after in-place growth may contain unexpected "non-zero" values, which can be against the intuition from the Go guarantee of zero-value initialization (of new allocation).

   - Result: All slice slicing / append usages are safe and correct.

5. Issue: Usage of Go slice copying on aliased slices. This would cause the classic overlapping `memcpy()` problem.

   - Result: All slice copying usages are non-aliasing and correct.

6. Issue: Usage of Go feature of silently ignoring return value on errors: for example, suppose we have `func foo() (int, error)`, have calling statement such as `foo()` or `v, _ := foo()`, instead of `v, err := foo()`, will result in the returned error being ignored silently. This potentially can cause incorrect result or unexpected panics.

   - Result: Some non-critical errors are found to be ignored.

```
/cmd/init/gentx.go: L185: ip, _ := server.ExternalIP()
/cmd/init/gentx_test.go: L21: valPubKey, _ :=
↪  sdk.GetConsPubKeyBech32(...)
/cmd/init/init_test.go: L63: r, w, _ := os.Pipe()
/cmd/terracli/main.go: L180: _, _ = w.Write(...)
/server/util.go: L78: conf, _ = tcmd.ParseConfig()
/server/util.go: L92: terraConf, _ := config.ParseConfig()
/x/budget/end_blocker_test.go: L95: claims, _ :=
↪  EndBlocker(...)
/x/budget/end_blocker_test.go: L100: claims, _ :=
↪  EndBlocker(...)
/x/budget/end_blocker_test.go: L127: claims, _ :=
↪  EndBlocker(...)
/x/budget/end_blocker_test.go: L137: claims, _ :=
↪  EndBlocker(...)
/x/market/msg_test.go: L10: _, addrs, _, _ :=
↪  mock.CreateGenAccounts(...)
/x/oracle/ballot_test.go: L134: _, addrs, _, _ :=
↪  mock.CreateGenAccounts(...)
/x/oracle/end_blocker_test.go: L166: input, _ := setup(t)
/x/oracle/end_blocker_test.go: L229: rewardees, _ :=
↪  EndBlocker(...)
/x/oracle/end_blocker_test.go: L237: rewardees, _ :=
↪  EndBlocker(...)
/x/oracle/keeper_test.go: L35: price, _ =
↪  ....GetLunaSwapRate(...)
/x/oracle/msg_test.go: L10: _, addrs, _, _ :=
↪  mock.CreateGenAccounts(...)
```

7. Issue: Unused or test-only variables, struct fields, methods, or functions, which not only increases the binary size, but also blurs the true intention or semantics of them and their associated constructs.

   - Result: Some unused declarations are found

```
/x/budget/end_blocker.go: L16: voteCount := 0
/x/treasury/indicator.go: L66: func SRL(...) sdk.Dec
/x/treasury/indicator.go: L71: func MRL(...) sdk.Dec
```

8. Issue: Unwarranted panics, such as those for non-bug errors, especially when there already exists an error returning path, can potentially cause unexpected program exits or false negative in tests.

   - Result: Several such panics are found.

   ```
   /server/util.go: L67: panic(err)
   /x/budget/querier.go: L52: panic("could not marshal result to
   ↪  JSON")
   ```

9. Issue: Usage of stateless argumentless function to construct constants (via global variables), which not only incurs both execution time and footprint, but also makes it less obvious the constant and stateless nature of those variables.

   - Result: Several such functions are found.

   ```
   /x/budget/errors.go: L25: func ErrInvalidTitle() sdk.Error
   /x/budget/errors.go: L30: func ErrInvalidDescription()
   ↪  sdk.Error
   /x/budget/errors.go: L46: func ErrVoteNotFound() sdk.Error
   ```

10. Issue: Arithmetic overflows.

    - Result: None is found. This is because all arithmetic operations are done via Cosmos SDK.

11. Issue: Usage of Go routine in chain-processing logic, which potentially can cause race conditions, unexpected behavior of non-atomic data types (such as map), dangling execution context and resources belonging to it, as well as non-deterministic execution order and hence non-deterministic persistent state.

    - Result: No usage is found.

12. Issue: Usage of Go random libraries in chain-processing logic, which potentially can cause non-deterministic computation result and persistent state.

    - Result: No usage is found.

13. Issue: Usage of Go time libraries in chain-processing logic, which potentially can cause non-deterministic computation result and persistent state.

    - Result: No usage is found.

14. Issue: Usage of Go channel in chain-processing logic, which should not be needed for serialized execution of the logic, potentially can cause program hanging if used in a blocking fashion with poorly implemented synchronizations, and potentially can cause non-deterministic execution order and hence non-deterministic persistent state.

    - Result: No usage is found.

15. Issue: Usage of Go synchronization libraries in chain-processing logic, which should not be needed for serialized execution of the logic, potentially can cause program hanging if used with poorly implemented synchronizations, and potentially can cause non-deterministic execution order and hence non-deterministic persistent state.

    - Result: No usage is found.

Reference Documentation

CertiK used the following source of truth to enhance the understanding of the Terra system:

1. Terra Whitepaper[1]
2. Terra Developer Document[2]
3. Terra Protocol Source Code[3]

All listed sources act as a specification. For any inconsistency discovered within the actual code behavior, we consulted with the Terra team for further discussion and confirmation.

[1]Terra Website & Whitepaper: `https://terra.money/`
[2]Terra Developer Document: `https://docs.terra.money/`
[3]Terra Protocol GitHub Repository: `https://github.com/terra-project/core`