Least Authority
PRIVACY MATTERS

BTG Pactual

Security Audit Report

Tezos Foundation

Final Report Version: 13 March 2020

Table of Contents

Overview

Background

Tezos Foundation has requested that Least Authority perform a security audit of the following:

- **ReitBZ Security Token** (real estate backed token on the Ethereum protocol)
  - Smart contract implementing the FA1.2 standard interface, including additional functions for whitelisting token holders and distributing dividends to them. The contract must have an Administrator, which is the generic.tz multisig contract, requiring 3 out of 5 signatures for admin operations.
- **Token Management Dashboard**
  - BTG administrators interact with the token contract and its multisig admin through the Token Management Dashboard. The dashboard collects signatures from Ledger hardware wallets through the browser, stores them until 3 are collected, and uses a "gas wallet" through Amazon Lambda / secrets manager to sign operations and send them to the chain. The dashboard shows all relevant token information, such as past operations, and manages user accounts / PKHs.

Project Dates

- **November 25 - December 31**: Initial Review *(Completed)*
- **January 3**: Initial Audit Report delivered *(Completed)*
- **February 24 - 27:** Verification Review *(Completed)*
- **February 28:** Final Audit Report delivered *(Completed)*
- **March 13:** Updated Final Audit Report delivered *(Completed)*

Review Team

- Ramakrishnan Muthukrishnan, Security Researcher and Engineer
- Sajith Sasidharan, Security Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Nathan Ginnever, Security Researcher and Engineer
- Emery Rose Hall, Security Researcher and Engineer
- Alex Leitner, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the ReitBZ Security Token and Token Management Dashboard followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

- **ReitBZ:** Tezos Foundation shared `tezos-btg.tar.gz` with Least Authority on 7 November 2019
- **Token Management Dashboard:** https://gitlab.com/obsidian.systems/tq/tezos-token-dashboard

For the ReitBZ Contract, we examined the version contained within `tezos-btg.tar.gz` with a SHA256SUM:

```
fc22493c2f9256b14ef50c7fb8d12470f8d5dc14d34ad9e0eb11e664957868c7
```

For our follow up verification of the ReitBZ Contract, we examined the version contained within `tezos-btg-master-d216ad2deb70af83b9b39926102453638e4e0e97.zip` with SHA256SUM:

`2c32eaa35dc8d837ff1e3b7f58bf10d4053cee5ea2d1ffca8e672ab33127544a`

For the Token Management Dashboard, we examined the Git revision:

`c52e7f71ef3cdd2c3ea107d32b76248bc490c8a8`

For our follow up verification of the Token Management Dashboard, we examined the Git revision:

`ffec42d9161b919d253c865392a5530675ea46bd`

All file references in this document use Unix-style paths relative to the project's root directory.

Supporting Documentation

The following documentation was available to the review team:

- BTG Pactual documentation (contained within `tezos-btg.tar.gz`)
  - dividends.pdf
  - BTG-contract.md
  - specification.md

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the smart contracts;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Economic incentives: ensure token economics (monetary incentives to punish bad behavior and reward good behavior) are included and functional;
- DoS/security exploits that would impact the contracts intended use or disrupt the execution of the contract;
- Vulnerabilities in the smart contracts code;
- Protection against malicious attacks and other ways to exploit contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

ReitBZ Security Token

Overall, we found that the code is well written. The Tezos FA1.2 standard covers similar functionality as the Ethereum ERC-20 standard, which has been well-researched and implemented in various conditions. As a result, there is less of an attack surface and the tokens that do not diverge from standard ERC-20 concepts are generally more resilient to attacks. Tokens that are more vulnerable do not use safemath or forget a modifier.

We discovered that the ReitBZ sales contracts implements unique functionality in the dividends payment contract and by adding a custom whitelist to the standard ERC-20 functions, thus diverging from functionality present in most sale contracts which include minting, recording transactions, and checking against the whitelist. As a result, the dividends payment contract warranted closer evaluation than other parts of the sales system. We found that the contract's implementation in a typed language and the Michelson runtime's rejection of poorly typed programs provides further assurance against vulnerabilities.

We also reviewed the code for known and documented ERC-20 vulnerabilities documented in a paper surveying attacks on Ethereum smart contracts and a compiled list of solidity vulnerabilities. During our audit, we identified no overflow errors in the `totalSupply`. The code appears well handled by Michelson itself, along with the gas and storage costs. In addition, we checked the Token White List family of entry points against the Tezos specifications and found no flaws in this particular area. Since Michelson is designed in a way that integers and natural numbers are arbitrary precision and size is only limited by fuel, we did not deeply investigate and test overflows.

Although we did a cursory review of the Dividend Management entry points against the Tezos specifications and found they appear to behave as described, we believe this could be an area at risk for vulnerabilities. Because of the low level nature of the Michelson language, it is difficult to reason about non-trivial programs, and this presents a challenge when reviewing the implementation.

Token Management Dashboard

We reviewed the backend by manual reading of the code line by line. In particular, we checked logging and whether any secrets are being leaked and investigated for timing attacks (such as comparisons). In general, timing attacks are less likely to be an issue in a lazy language like Haskell where it is difficult to predict when a particular expression is evaluated. As a result, we did not find any timing issues as such.

Overall, we found the dashboard code to be difficult to review and build. In particular, the manner in which the frontend JavaScript is compiled made it difficult to comprehensively evaluate. Given that this includes the driver code for Ledger devices and handling of user input, we believe this to be an area of concern and poses a significant risk. Building the dashboard took hours on an AWS EC2 VM (with a four-core CPU and 32 GB RAM) and while much of that build time is attributable to Nix's way of building dependencies, the code complexity adds unnecessary risk.

We encourage a rework of this area of the project to follow more idiomatic web development practices and patterns. This would make the code more comprehensible and conducive to security evaluations, therefore reducing the risk of vulnerabilities that go unnoticed.

Specific Issues

We list the issues we found in the code in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: [ReitBZ Security Token] `setAdministrator` Function Does Not Handle Typos	Resolved
Issue B: [ReitBZ Security Token] Unbounded Amount of Minted Coins	Resolved

Issue C: [ReitBZ Security Token] Potential for Front Running	Resolved
Issue D: [ReitBZ Security Token] Looping Could Cause Loss of Gas Funds	Invalid Issue
Issue E: [ReitBZ Security Token] Multisig Parameters Could be Improperly Constructed	Resolved
Issue F: [ReitBZ Security Token] Lack of Generic Multisig Tests	Partially Resolved
Suggestion 1: [ReitBZ Security Token] Further Review of Entry Points	Partially Resolved
Suggestion 2: [Dashboard] Hardcoded Callback Addresses	Resolved
Suggestion 3: [Dashboard] Use Scrubbed Memory for Secrets	Resolved
Suggestion 4: [Dashboard] Improve Documentation	Resolved

Issue A: [ReitBZ Security Token] `setAdministrator` Function Does Not Handle Typos

Location

btg-ReitBZ-token/src/Lorentz/Contracts/BTG/Token/Impl.hs#L94

Synopsis

A safeguard against typos in the `setAdministrator` function does not exist.

Impact

If the admin is changed to an invalid or unknown address, the contract becomes essentially useless.

Preconditions

We have not identified functionality in the dashboard to call `setAdministrator` so we assume that a custom multisig package or byte string will be created to initialize the functionality that may contain a bad address.

Technical Details

The `setAdministrator` function does not double check the validity of the new admin. If a typo or uncontrolled address is inserted, the contract is permanently lost. Extended features are gas-expensive and are prone to error as well (specifically checking the data length of provided address).

Remediation

It has been suggested by Ethereum that the client does proper checks or that there be some VM level checking on format of address.

Consider creating a two step update process that would propose an update to the ownership of the contract. After that transaction is confirmed, the token could require the proposed address to accept the new ownership. This would catch any updates to incorrect or malformed state.

Status

The two step update process has been implemented by Tezos.

Verification

Resolved.

Issue B: [ReitBZ Security Token] Unbounded Amount of Minted Coins

Location

btg-ReitBZ-token/src/Lorentz/Contracts/BTG/Token/Impl.hs#L100

Synopsis

The `mint` function allows for an unbounded amount of minted coins.

Impact

This “money” is only received by the baker, while everyone else will need to store the large contract.

Technical Details

At the very least, the contract can be arbitrarily large in size (TB) and only bounded by the gas required for minting.

Remediation

Modify the `mint` function to bound the amount of minted coins.

Status

A bound on the total amount of tokens that can be minted and modified has been implemented. Both `mint` and `mintBatch` functions check against that bound and fail if it is exceeded.

Verification

Resolved.

Issue C: [ReitBZ Security Token] Potential for Front Running

Location

btg-ReitBZ-token/src/Lorentz/Contracts/BTG/Token/Impl.hs#L66

Synopsis

The FA1.2 API for the `Approve` function can lead to ambiguity on behalf of the approver of their approved amount of tokens.

For example, approver A allows a spender B ten tokens and goes offline. At some point in the future, B spends five of the approved tokens and A does not witness this. A then decides to only allow the spending of six tokens to B and calls the approve function to set the balance to zero before calling it again to allow B to spend six tokens. If A did not witness the spend of five tokens from B at some point, then B will have spent a total of eleven tokens while A thinks they were only approved to spend six. B may also witness the incoming reset of the approve to zero tokens transaction and bribe a miner to include a spend of ten tokens before this happens without A knowing.

Impact

As discussed in ERC20 API: An Attack Vector on Approve/TransferFrom Methods, this can still lead to front running attacks where it is not clear via blockchain references to the sender if the previous approval had spent any of the total amount.

Preconditions

This exists in all standard FA1.2 tokens that implement `Approve`.

Feasibility

This attack requires that the Approver does not have a “good view” of the current state of a token contract. This could happen if their client is out of date, but it is not highly likely that an Approver will not see an amount already spent. It is also possible for an approved address to bribe a miner to include their spend before a reduction of their proved amount gets into the blockchain, however, this is also unlikely.

Technical Details

The FA1.2 Token standard implemented in this contract adds a layer of security to prevent a front running attack on the `Approve` method. This is done by checking that the approval of any amount that is not already zero is set to zero before adding more approved tokens. As documented by the FA1.2 spec, an error is provided: “UnsafeAllowanceChange - attempt to change approval value from non-zero to non-zero was performed. The error will contain nat :previous value, where previous stands for the allowance value upon the contract call.”

Mitigation

Keeping state to handle this would be a better approach but may be out of the scope of the project. This would represent the atomic solution mentioned in the document above. A miner being bribed to spend the approval before revoking could still happen, but the approver’s transaction would then fail to notify them that the change cannot happen and an amount was already spent.

Status

An `approveCAS` function has been added and the current balance can be checked against what is expected.

Verification

Resolved.

Issue D: [ReitBZ Security Token] Looping Through Inputs

Location

btg-ReitBZ-token/src/Lorentz/Contracts/BTG/Dividends/Impl.hs#L58

Synopsis

There are a few places that the contract is allowed to loop through an array and conduct some operations. A Solidity equivalent reference to this can be found in this documentation on Gas Limits and Loops. While attempts can be seen to reduce the gas cost within these loops, the hard limit remains and there is the potential of a stalled state when a transaction runs out of gas if this behaves the same as Ethereum. Tezos transactions have a hard limit of 400,000 units of gas, and we found that each storage call to the batched whitelist function requires 496 units. At 154 addresses, the contract call could stall if the client does not reject this before submission to the network. Longer lists will lead to larger transactions that may be harder to include in blocks if there is a heavy load on the network, and a larger fee may need to be applied to fit the transaction in on time.

Impact

A stalled transaction could leave the state of updates using loops unknown for the client. The client may believe that all updates were applied when not all of them made it into the contract state or attempt to reapply the same update twice.

Feasibility

The client that is providing the list of users to be updated may be aware in advance by testing if a transaction will run out of gas so this may not be necessary. The Solidity version of the contract code mentions iterating off-chain to avoid this.

Mitigation

We suggest writing a unit test that attempts to enter 154 addresses into the `updateWhiteListBatched` function and dividend disbursement test the behavior of hitting the 400,000 gas unit limit.

Another approach would be to call a function that updates one account at a time in a loop off-chain, with some state on-chain that marks if an account has been updated such that an off-chain loop can call many times the same function without ever hitting a gas limit. If the off-chain client calls an update to the function more than once, the contract state will prevent it from updating. This could function like a nonce for preventing multisig replay of stale state.

Status

Invalid Issue. Tezos has responded that they have considered the loops thoroughly and have confirmed that this issue has been handled due to the loops being bound and verifiable ahead of time. Incomplete state updates are also not an issue with Tezos as it is with Ethereum smart contracts, since either a call succeeds or fails leaving no partial updates when gas depletes.

Verification

Not Applicable.

Issue E: [ReitBZ Security Token] Multisig Parameters Could Be Improperly Constructed

Location

https://gitlab.com/obsidian.systems/tq/tezos-token-dashboard/blob/c52e7f71ef3cdd2c3ea107d32b76248bc490c8a8/backend/test/Main.hs

Synopsis

We did not identify any tests for the multisig wallet that could be used to control sensitive operations on the token. If a generic multisig wallet is used, then signers may unknowingly process an unexpected transaction.

Impact

The parameters may be improperly constructed before signing which could lead to signatures on transactions that were not intended (i.e. an improper update address or call to an unexpected function). If the multisig contract owns the token, this could cause total loss of control.

Technical Details

The contract allows for the “generic multisig” contract to control execution of generic functions on the contract system. The inputs to this multisig signing will be Michelson code snippets that are hard to read.

We did not find any client side tools that will help the signers understand the code that is being proposed to the multisig wallet.

Mitigation

The Tezos client offers basic commands to construct transactions like transfer and approve but custom calls will need their own implementation. The clients should have a human readable method to construct the multisig transactions. Consider using a multisig wrapper contract that only allows signing specific parameters.

Status

The generic multisig contract has been replaced with a specialized wrapper, the `MultisigWrapper.hs`.

Verification

Resolved.

Issue F: [ReitBZ Security Token] Lack of Generic Multisig Tests

Location

https://gitlab.com/obsidian.systems/tq/tezos-token-dashboard/blob/c52e7f71ef3cdd2c3ea107d32b76248bc490c8a8/backend/test/Main.hs

Synopsis

No tests that confirm the intended functionality of the generic multisig contract itself were present in the code. Since there are no tests on the multisig contract itself, it is assumed that tests exist elsewhere. It is difficult to understand the functionality of the multisig contract without proper tests. There could be a case where the contract is updated to null list, or a list with below the threshold of necessary signers.

Impact

The multisig contract will be the controlling address of the token and unexpected behavior can cause loss of control of the token. If there is not a proper modifier to limit the ability of ownership change to prevent the case where there are no owners of the multisig, then control will be lost.

Mitigation

We suggest writing proper tests on the multisig contract.

Status

The generic multisig has been replaced with a specific multisig wrapper that has been integrated into the client. Most, but not all, of the properties that hold true for the generic multisig contract will hold true for the wrapper as well. It is suggested that the difference be tested for correctness with either further audits or formal verification.

Verification

Partially Resolved.

Suggestions

Suggestion 1: [ReitBZ Security Token] Further Review of Entry Points

Location

All contracts.

Synopsis

We encourage further investigation of entry points. At this time, we simply do not have enough knowledge on all possible security issues with Michelson the language, its runtime, Tezos, and this contract in particular, but that does not indicate that vulnerabilities do not exist.

Mitigation

Viable long-term approaches could include:

- Complementing audit with static analysis and/or formal verification;
- Writing property tests against the contract, including testing these entry points against unusual situations;
- Re-implementing the contract using a high-level language that compiles to Michelson. Some of them exist at present (LIGO, Juvix, Lamtez), although they are at various stages of development.

Status

Regarding the suggested potential mitigation strategies:

- Tezos states that formal verification is currently out of scope. We recommend that static analysis and / or formal verification be reconsidered in the future.
- Unit tests for contract specifications have been implemented.
- The ReitBZ Security Token has been implemented in Lorentz, a Haskell eDSL for Michelson contracts. It should be noted, however, that Lorentz does not offer a substantially higher level of abstraction than Michelson itself, and as a result, we recommend reimplementing the contract in a high-level language be reconsidered.

Verification

Partially Resolved.

Suggestion 2: [Dashboard] Hardcoded Callback Addresses

Location

https://gitlab.com/obsidian.systems/tq/tezos-token-dashboard/blob/c52e7f71ef3cdd2c3ea107d32b76248bc490c8a8/backend/src/Backend/Token.hs#L142

https://gitlab.com/obsidian.systems/tq/tezos-token-dashboard/blob/c52e7f71ef3cdd2c3ea107d32b76248bc490c8a8/backend/src/Backend/Token.hs#L159

Synopsis

`src/Backend/Token.hs` has the address `KT1GSuYsHtr38FwN2KpG1e9jcvyh49zJ6Wkr` hardcoded in the source code. These appear to be addresses for the callback contract to hold the return values.

Mitigation
Consider deploying a callback contract first when the user selects an option that invokes an entry point that needs a callback contract address to be passed and show a cue to the user in the UI. Alternatively, deploy a callback contract when the web dashboard is initially run.

Status
A new identifier targetContract has been introduced into every function that previously implicitly operated on a hardcoded contract. A config structure has been added to carry around these address and which come from the files: `config/backend/result-receiver-contract-for-*`.

Verification
Resolved.

Suggestion 3: [Dashboard] Use Scrubbed Memory for Secrets

Location
Various use of secret like passwords in:

https://gitlab.com/obsidian.systems/tq/tezos-token-dashboard/-/tree/c52e7f71ef3cdd2c3ea107d32b76248bc490c8a8/backend/src/Backend

Synopsis
Using ByteString or Text for storing passwords and other secrets like keys may cause accidental leakage into logs. In addition, a less critical concern is that these memory areas may be collected by the garbage-collector and reallocated for other use and leak these secrets. However, this is not very easy to exploit since it is not possible to use a piece of memory without initialization in Haskell.

Mitigation
Use a library like securemem instead of ByteString or Text for storing secrets in order to prevent it from getting logged accidentally and from being scrubbed after use.

Status
All the account related functions in the backend have been updated to use scrubbed memory (using the securemem library).

Verification
Resolved.

Suggestion 4: [Dashboard] Improve Documentation

Location
README.md

Synopsis
We found there to be a lack of documentation (i.e. a step-by-step walkthrough of the configurable items) regarding the building and deployment of the dashboard.

Mitigation

We recommend including documentation that would allow new contributors and reviewers to understand the dashboard more easily and efficiently, and reduce some of the existing complexity in building the dashboard.

Status

Additional information has been added to the Dashboard's README.md. It now includes additional instructions on Nix caching for faster build, instructions on Obelisk, configuration information, information on the proxy contracts for the view endpoints, and information on the multisig deployment with the new multisig-wrapper.on using Nix and Obelisk.

Verification

Resolved.

Recommendations

We recommend that the partially resolved *Issues* and *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team. We also recommend that future development releases continue to apply security best practices and that additional security audits be conducted to address any potential issues and vulnerabilities.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.