Ubinetic's checker for Youves

Tezos Foundation

Independent security assessment report

inference

Report version: 1.0 / date: 18.08.2023

Table of contents

Version / Date	Description
1.0 / 18.08.2023	Final version

Summary

Inference AG was engaged by the Tezos Foundation to perform an independent security assessment of Ubinetic's checker version.

Inference AG performed the security assessment based on the agreed scope, following our approach and activities as outlined in the "Project overview" chapter between 1st September 2022 and 18th August 2023. Feedback from Ubinetic was received and Inference performed a follow-up assessment.

Based on our scope and our performed activities, our security assessment revealed several issues. Additionally, different observations were also made which, if resolved with appropriate actions, may improve the quality of Ubinetic's checker.

This report only shows remaining open or partly resolved security issues and observations.

Overview on issues and observations

Details for each reported issue or observation can be obtained from the "Security issues" and "Observations" sections.

Row header	Severity / Status
Issues	
S-UCH-001: Oracle price	unknown / partially closed
Observations	
O-UCH-001: No two-step procedure to replace admin	- / open
O-UCH-002: Testing	- / open
O-UCH-003: function abs() used to convert from INT to NAT	- / open
O-UCH-004: Documentation	- / open

Project overview

Scope

The scope of the security assessment was Ubinetic's checker smart contract[1].

All files for the checker smart contract were made available via the source code repo https://gitlab.papers.tech/papers/ubinetic/youves-checker/ and our initial security assessment considered commit "570a3b5c460a80e609a0c7ca18bbf9d44ac6fd00".

Additionally, we also had access to the following documentation in order to build up our understanding of checker: https://checker.readthedocs.io/[2]

Our reassessment considered commit: 5e091f62eaa398fa65c2be8dc9adf41504a83c65
and the compiled smart contract in Michelson on Ghostnet:
KT196c1Nx81DQ1QxX3W8FwuaxW5B5VbBCoTh[3]

[1] Ubinetic's checker smart contract is a fork with changes from https://github.com/tezos-checker/checker
[2] https://github.com/tezos-checker/checker/tree/d06c341cfdabecada2d9e201950b62d54e06a95f
[3] See last bullet point in section scope limitation

Scope limitations

Our security assessment is based on the following key assumptions and scope limitations:

- Any potential adversarial activities conducted by the administrator of the contract and operational errors by administrators were out of scope.
- Smart contracts for tokens not in scope were considered trusted (ctez).
- Content of metadata and token metadata was out of scope.
- Also out of scope were: the impact assessment to checker in case of changes to the blockchain's characteristics (e.g. block times) cases where the blockchain is stalled or progressing slowly. Our security assessment took place while Lima was the active protocol on the Tezos mainnet.
- The economic model of checker, including the appropriate settings for thresholds, factors, indexes, etc. was out of scope. Additionally, potential impacts of significant changes in the economic environment were not assessed. These may have an impact on checker since checker often uses constants which cannot be changed after smart contract deployment.
- Due to tool limitations we were unable to review the complete Michelson code for the entrypoint "touch"[4].

Methodology

Inference's methodology for smart contract security assessments on Tezos is a combination of a source code review of the smart contract source code in the high-level language (e.g. Ligo or SmartPy), and the resulting compiled code in Michelson. This approach provides additional assurance that the compiler producing the Michelson code is correct, and does not introduce any security issues. Furthermore, this approach fosters a better understanding for smart contract users of what has been reviewed and what has been effectively deployed on-chain.

In order to ensure a high quality in our security assessments, Inference is using subject matter experts having a high adversarial scenario mindset to spot potential issues in smart contracts under review. Additionally, we apply checklists derived from good practices and commonly known issues based on the Tezos smart contract assessment checklist to document our work and to ensure good issue coverage.

[4] However, since we have not identified any security issues during the course of the Michelson code review we have high confidence that the unreviewed code for the entrypoint "touch" is also correct.

Furthermore, Inference maintains regular communications with the smart contract development team to ensure a correct understanding of the smart contract solution and environment, but also to make teams aware of any observations as soon as possible.

Inference's internal quality assurance procedures ensure that results of security assessments are challenged for completeness and appropriateness by a second independent expert.

Objectives

The objectives are the identification of security issues with regards to the assessed smart contracts and their conceptual design and specification. The security assessment also focuses on adversarial scenarios on specific use cases which have been listed in appendix Adversarial scenarios. These were identified together with the Ubinetic developers and checked during our security assessment.

Activities

Our security assessment activities for the defined scope were:

- Source code review of smart contract code in OCaml
- Source code review of the smart contracts in Michelson

We applied the checklist for smart contract security assessments on Tezos, version 1.1 obtained from https://github.com/InferenceAG/TezosSecurityAssessmentChecklist. We applied the following security checklist tables:

- System / Platform
- Storage
- Gas issues and efficiency
- Code issues
- Transactions
- Entrypoint
- On-chain views
- Admin / Operator functions
- Other topics & test cases

Our activities for the follow-up assessment were:

- Source code review of changes in the smart contract code in OCaml
- Source code review of changes in the smart contracts in Michelson
- Reassessing security issues and observations from initial assessment in case they are claimed to be resolved

Security issues

S-UCH-001: Oracle price

When calling the “touch” entrypoint the checker smart contract initially uses the locally stored price to calculate the protected index and then updates the locally stored price at the end by calling the Oracle smart contract using a callback scheme.

The checker smart contract has not implemented any protection to ensure that the locally stored price in the checker smart contract is not updated by third parties between the execution of two calls of the “touch” entrypoint.

Thus third parties may observe Oracle prices after a “touch” entrypoint has been executed and update the price as soon as the observed Oracle reaches a new maximum or minimum price.

Probability/Impact/Severity - Unknown. See section “recommendation”.

Recommendation:
We recommend analysing this issue whether third parties would benefit in certain market situations if they are allowed to deliberately initialise an update of the Oracle price between two consecutive executions of the “touch” entrypoint?
In order to prevent prices being set by third parties we recommend using views or ensuring that the checker “receive_price” callback entrypoint can only be called if the callback has been initialised by the “touch” entrypoint .

Comment from Ubinetic:
The oracle contract will only allow calls from the checker contract.

Results from reassessment:
We updated the status from “open” to “partially closed”. The solution implemented by Ubinetic works only, since Ubinetic controls the Oracle smart contract. Thus, we have not closed the issue to raise awareness of this situation among potential adapters of checker smart contract code. Another viable solution, in cases where there is no direct control to implement the security measures within the Oracle contract, is to create a proxy contract guaranteeing that price updates can only be triggered by the checker contract.

Observations

O-UCH-001: No two-step procedure to replace admin

The “set_admin” entrypoint allows setting a new address for the admin role in the checker smart contract. After verification that the current admin initiated “set_admin”, the “set_admin” entrypoint directly updates the admin role with the newly provided address.

This poses a risk that the address for the admin role is set to a wrong or non-working address.

Recommendation:
We recommend implementing a two-step procedure to change any privileged addresses. In the first step the current privileged address proposes a new address, followed by a second step in which the newly proposed address accepts the proposal. The change of the critical privileged address is then done after the acceptance at the second step.

Comment from Ubinetic:
No two-step procedure to replace admin - Checker contract is controlled by the youves DAO (DAO is the admin) and it is unlikely that this will change. If a new DAO will be set as admin, the process is thorough and requires an extensive review from the community and setting the wrong admin is unlikely.

O-UCH-002: Testing

We identified that the following scenarios have not be covered with test cases:

- Testing whether checker is correctly behaving if the "completed_auctions" queue contains more than one finished auction in it.
- Testing whether checker is correctly behaving in the case where the AVL tree is at its maximum size.

Recommendation:
We recommend implementing these two test cases.

Comment from Ubinetic:
The first scenario was tested by our PM and no issues were found with it. The second scenario is hard to test in practice, but we had taken care of possible gas cost issues that might arise, by limiting the number of processed slices in a transaction.

O-UCH-003: function abs() used to convert from INT to NAT

Checker uses the "abs" function in several places to convert between type INT and type NAT.

Using the "abs" function for converting numbers could potentially lead to a vulnerability since, in certain scenarios, the conversion of negative numbers would be wrong.

However, for the reviewed code in scope, we have not identified a scenario to exploit the usage of abs to convert between INT and NAT types. Checks performed before the conversion prevent negative numbers from being converted.

Recommendation:
We recommend, as a good practice, to use the function "is_nat" to convert from type INT to NAT.

Comment from Ubinetic:
New version of the code contains this check, but an already deployed version is used without this check. The necessary checks were done and no situations where the value converted is less than 0 will arise in practice.

O-UCH-004: Documentation

Reading the documentation https://checker.readthedocs.io/ for our understanding we recognized the following relevant information is missing:

- Calling the “touch” entrypoint provides “kits” as a reward to the caller of the entrypoint. Additionally, the documentation is also different with regards to how the “protected_index” is calculated. The implementation uses a linear function as the approximation for the min and max boundaries. Thus, if the “touch” entrypoint is not regularly executed, values differ more and more from the documentation’s specification. Therefore, the incentive to call “touch” should be documented and appropriately advertised.
- Ubinetic’s checker variant assigns a stake to users. However, the documentation is not mentioning this and we also have not received any specific documentation related to Ubinetic’s checker variant.

Recommendation:
We recommend updating the documentation.

Comment from Ubinetic:
Our PM is taking care of the documentation regarding the stakes for the governance token. About the touch entrypoint needed to be called often, I added documentation in the code regarding it.

Observation will be addressed by updating the documentation.

Disclaimer

Appendix

Adversarial scenarios

The following adversarial scenarios have been identified together with the Ubinetic developers and checked during our security assessment.

Scenario	Assessment result
Create a “burrow” contract without paying the “burrow” creation deposit (create_deposit).	**Ok** Nothing identified.
Delete / overwrite an existing “burrow” (e.g. using create_burrow entrypoint).	**Ok** Nothing identified.
Increase collateral/deposit without providing any collateral.	**Ok** Nothing identified.
Withdraw collateral when overborrowed.	**Ok** Nothing identified.
Extract ”burrow” creation deposit (create_deposit).	**Ok** Nothing identified.
Mint more kits than the provided collateral would grant.	**Ok** Nothing identified.
Get more collateral back than the burned kits would provide.	**Ok** Nothing identified.
Increase bid for liquidation less than the defined “bid_improvement_factor”.	**Ok** Nothing identified.
Influence the Oracle price which is provided to the checker.	**Ok** Nothing identified. However, we have not assessed the Oracle itself.
Influence the observed price in CFMM used by checker.	**Ok** Price can be influenced when liquidity in CFMM is low or somebody has lots of tokens available. Nothing identified in addition to these liquidity related possibilities.

Prevent a (warranted) liquidation. (Note: Unwarranted liquidations can be cancelled by the “burrow” owner.)	**Ok** Nothing identified.
Prevent higher bids from being provided.	**Ok** Nothing identified.
Obtain more LP tokens in CFMM than the equivalent amount of liquidity would grant.	**Ok** Nothing identified.
Obtain more liquidity in CFMM than the equivalent amount of LP tokens would grant.	**Ok** Nothing identified.
Obtain more tokens in a CFMM exchange than the CFMM would grant.	**Ok** Nothing identified.
Extract more rewards than intended when calling the entrypoint “touch”.	**Ok** Nothing identified.
Extract more rewards in a “mark for liquidation” than intended.	**Ok** Nothing identified.
Activate a “burrow” without paying the “burrow” deposit.	**Ok** Nothing identified.
Locking user assets in a “burrow”.	**Ok** Nothing identified.

Risk rating definition for smart contracts

Severities are quantified with two dimensions, roughly defined as follows, whereas the examples have to be regarded as indication only:

Probability of occurrence / materialisation of an issue
(bullets for a category are linked with each other with “and/or” condition.)

- Low:
  - A trusted / privileged role is required.
  - Contract may end up in the issue if other conditions, which are also unlikely to happen, are required.
- Medium:
  - A specific role or contract state is required to trigger the issue.
  - Contract may end up in the issue if another condition is fulfilled as well.
- High:
  - Anybody can trigger the issue.
  - Contract’s state will over the short or long term end up in the issue.

Impact:
(bullets for a category are linked with each other with “and/or” condition.)

- Low:
  - Non-compliance with TZIP standards
  - Unclear error messages
  - Confusing structures
- Medium:
  - A minor amount of assets can be withdrawn or destroyed.
- High:
  - Not inline with the specification
  - A non-minor amount of assets can be withdrawn or destroyed.
  - Entire or part of the contract becomes unusable.

Severity:

	Low impact	Medium impact	High impact
High probability	**High**	**Critical**	**Critical**
Medium probability	**Medium**	**High**	**Critical**
Low probability	**Low**	**Medium**	**High**

Glossary

Term	Description
Ligo	High level smart contract language. Website: https://ligolang.org/
OCaml	High level smart contract language. Website: https://ocaml.org/
Origination	Deployment of a smart contract
SmartPy	High level smart contract language. Website: https://smartpy.io/
TZIP	Tezos Improvement Proposal