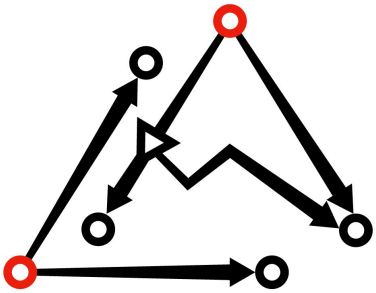

Least Authority

PRIVACY MATTERS

Vesting Smart Contracts

Final Security Audit Report

Tezos

Report Version: 16 March 2019

Table of Contents

Overview

The Tezos Foundation has requested that Least Authority perform a security audit of their Vesting Smart Contracts, in preparation for the upcoming betanet and mainnet launches of Tezos.

The security audit was performed from June 8 - 14, 2018, by Johan Kjaer and Hannes Mehnert of the Robur team with support from Ramakrishnan Muthukrishnan of the Least Authority team. This initial report was issued on June 14, 2018. An updated report following the verification phase has been delivered on December 14, 2018. A final report following a secondary verification conducted based on the responses provided by Tezos *(Tezos Foundation Comments on the Least Authority Vesting Smart Contracts Audit Report, 26 February 2019)* was issued on March 16, 2009.

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Tezos Vesting Smart Contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code repositories are in scope:

- https://gitlab.com/tezos/tezos/merge_requests/368
- https://gitlab.com/tezos/tezos/tree/vesting-contracts

Specifically, we examined the Git revisions:

`73e87e7a42a9137dd132e5958fa727471e0d9ca1`

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

Manual Code Review

In manually reviewing all of the contract code, we looked for any potential issues with code logic, error handling, and interaction with contracts that are dependencies. We also considered areas where more defensive programming could reduce the risk of future issues and speed up future audits. Although our primary focus was on the contract code, we examined some dependency code and behavior when it was relevant to a particular line of investigation.

Our investigation focused on the following areas:

- Any attack that impacts funds, such as the draining or manipulating of funds.
- Other ways to exploit contracts, including potential ways to disrupt the execution of the contract (Denial of Service).

We also manually went through the **vesting-multisig** contract step-by-step, using the description of Michelson to infer the current stack, verified the input handling, and that the comments documenting the behaviour and invariants of the contract corresponded to the actual code.

Findings

Code Quality

The stack-based DSL Michelson is concise, and well documented, although we found some inconsistencies between documentation and implementation, as noted in *Suggestion D*.

The cost model of how many instructions a contract may execute and how much storage it may use does not seem to be documented. In contrast to earlier smart contract languages, Michelson avoids partial executions of contracts, which causes issues in Ethereum. For example, by specifying that a contract acts like an atomic transaction, the results either in failure or in a list of "internal operations" to be performed - dealing with transfer of tez etc. - and storage. These internal operations are executed in a batch once the contract execution finished.

Vesting Contract

A vesting contract is a contract in which funds can be deposited. A vesting balance - that increases at a configurable vesting interval by a configurable vesting amount - can be transferred if a threshold of signatures sign such an operation.

Both the threshold and the keys are configurable, and there are two levels: **N** out of **M** key lists have to sign an operation, where each key list requires **P** signatures out of **Q** keys.

Another operation is "**pour**", where a single key can be configured that is allowed to transfer funds to a single configured contract. While the vesting amount and interval can only be specified at origination time of the contract, the **pour** information, keys and thresholds, and the delegate can be changed dynamically. Each of these changes require sufficient group signatures.

A single operation, "**vest**", can be executed without any signature. "**Vest**" checks whether the current timestamp is greater than the stored next vesting timestamp, and increases the vesting balance by the vesting amount, and the next vesting timestamp by the vesting interval.

The implementation of vesting contracts uses a replay counter (initially zero), stored in storage, and incremented after each successful signed operation (pour, delegate, change keys, change pour). If the contract execution fails, due to gas limit, insufficient signatures, etc., the replay counter is not incremented.

The signatures are applied to the hash of the operation, e.g. transfer is a signature over:

```
(Pair (Left (Left (Pair receiver amount)) (Pair contract_address replay))
```

Other operations have a different structure - "`(Left (Right ..))`" - but always include the second pair.

We consider this a safe method to sign operations without the possibility to reuse a signature for a different operation.

Before a transfer, the amount to be transferred is compared to be smaller than or equal to the current balance of the contract, and also to be smaller than or equal to the current vesting balance. If one of the checks fail, contract execution is aborted.

Michelson Implementation

Michelson is implemented as a domain-specific language in OCaml, a memory-safe programming language. The Michelson operations leverage OCaml's GADT mechanism to provide strong typing of the

parameters. This is an excellent design decision for a contract language since it provides a clear specification of the interface.

Michelson's integers and natural numbers use the multiple precision arithmetic library **gmplib** (via the **zarith** OCaml library), and thus do not overflow/underflow, but instead may lead to out of memory exception. Using zarith is a good decision, since the built in integer types in OCaml have silent underflow/overflow semantics[1], which can cause issues in applications where correct results are important.

Tezos Implementation

The Tezos code base makes use of a limited interface, **tezos-stdlib**, to the OCaml standard library.

This is a very sensible decision, since the default compiler-provided standard library itself contains a plethora of functions that raise exceptions and behave unpredictably.

Notably Tezos avoids *polymorphic comparison*, a functionality widely regarded as problematic by the OCaml developer community since it can lead to violations of type abstractions and is a common source of bugs in OCaml applications.

The code is well organized and concise; following Ethereum best practices and avoids known bugs such as re-entrancy; and it does not depend particularly on economic assumptions about what is or is not in the rational self interest of traders.

Issues

We list the issues we found below.

ISSUE / SUGGESTION	STATUS
Issue A: Rejection of Valid Data	*Unresolved*
Issue B: Possible to Create Invalid Contracts	*Unresolved*
Issue C: Incorrect Error Locations	*Unresolved*
Suggestion 1: Documentation Inconsistencies	*Partially Resolved*

Issue A: Rejection of Valid Data

Synopsis

A vesting contract which is executed with **N** / **M** signatures, where an additional invalid signature is provided, fails to execute, since any invalid signature directly leads to failure.

Impact

Aborting upon encountering an invalid signature de facto permits a single signer to "veto" the execution.

Preconditions

The "attacker" must be able to provide an invalid signature (i.e. be part of **M**).

[1] http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#1_Integerarithmetic

Technical Details

An example test script is provided as **rejecting-nplusx-signatures.sh**, which attempts a transfer with two valid and one invalid signature, where the requirement is 2 valid signatures.

Mitigation

While it might be useful to warn about invalid signatures, the contract should execute as long as the threshold is met.

Tezos Response

This issue does exist, but since the fault is attributable and since it is possible to re-submit the request without the invalid signature, it is not a viable vector for denial of service. This issue would be more significant for contracts involving a larger number of potential signers.

Status

The change request was closed without the changes being merged (https://gitlab.com/tezos/tezos/merge_requests/368/) into the master branch of the Tezos codebase. The code appears to have been abandoned. While we understand the Tezos development team feels this is not a significant issue, we still recommend this be resolved.

Verification

Unresolved.

Issue B: Possible to Create Invalid Contracts

Synopsis

It is possible to create contracts that can be baked, but always fail on execution.

Impact

This can lead to issues where invalid contracts are mistakenly activated, especially if they are automatically generated by external tools such as web services or compilers from high-level languages.

If the problem with the contract is only identified at a later stage, this can lead to "frozen funds" that cannot be recovered.

Technical Details

1. An instance of this issue can be triggered (see **ill-formed-signature-list-for-transfer.sh**) by a long key list, which is accepted by the initial bake, but any attempt to execute an operation leads to **gas limit exceeded**.
2. Another example is provided in **ill-formed-timestamp.sh** in which the vesting interval is set to **40962844799** seconds (which is the maximum range of the timestamp type due to usage of the calendar library) and after one vesting operation, any other vesting leads to "**Fatal error: ill-formed data**".

Mitigation

Upon origination of a vesting contract, the required gas for verifying the multiple signatures could be extrapolated and if above a certain threshold, the contract could be rejected.

To mitigate the **ill-formed timestamp example**, it should be ensured that data written to the store can always be read and interpreted correctly.

Tezos Response

This is more interesting because it is valid for all smart contracts. The mitigation is more complex. It is a decision of Tezos to build a Turing complete language for its smart contracts. This allows our users to create almost any contract they can imagine, at the cost of making it harder for us to prove properties before testing. Determining statically the properties of the program before running it is an open research problem, and we are constantly working in this domain through formal verification and static analysis to prove the desired properties.

Status

The change request was closed without the changes being merged (https://gitlab.com/tezos/tezos/merge_requests/368/) into the master branch of the Tezos codebase. The code appears to have been abandoned. The Tezos development team has communicated that it has plans to address this issue through formal verification, but have not yet completed the process at this time.

Verification

Unresolved.

Issue C: Incorrect Error Locations

Synopsis

When a vesting operation fails due to the next vesting timestamp being greater than the current time, the error location is reported incorrectly.

Impact

This can lead to confusion of the developer who attempts to debug their contract.

Technical Details

The attached **incorrect-error-loc.sh** exercises this behaviour and reports:

At line 131 characters 17 to 21, script reached FAIL instruction.

The contract dump including line numbers shows a **SWAP** operation at line 131.

It is more likely that the **ASSERT** in line 127 caused the **FAIL**.

Mitigation

Tracking error locations more precisely while parsing and interpreting contracts should be done.

Tezos Response

There is a bug in that the actual michelson parser does not track macro locations correctly. It is currently being rewritten by a developer who is aware of this issue and will fix it.

Status

The change request was closed without the changes being merged (https://gitlab.com/tezos/tezos/merge_requests/368/) into the master branch of the Tezos codebase. According to the Tezos development team, the Michaelson parser is in the process of being rewritten, but has not yet been completed.

Verification

Unresolved.

Suggestions

Suggestion 1: Documentation Inconsistencies

While reading the documentation for Michelson (whitedoc/michelson.rst), we noticed some shortcomings of the specifications. The Tezos development team has reported to us that the documentation is in the process of being updated and we've noted specifics in this section.

We list the inconsistencies reported and the current status in order of appearance:

Section V – Operations

The **ITER** construct (here on lists, but the same applies for set and map) is described as:

> **ITER** body: Apply the body expression to each element of a list. The body sequence has access to the stack.
>
> ```
> :: (list 'elt) : 'A -> 'A
>
> iff body :: [ 'elt : 'A -> 'A ]
>
> > ITER body / { a ; <rest> } : S => body ; ITER body / a : {
> <rest> } : S
>
> > ITER body / {} : S => S
> ```

In the first case, we read this that body is executed with the stack `a : { <rest> } : S`, while the implementation executes body with the stack `a : S`, and again with the head of `<rest>` until `<rest>` is the empty list.

Section V – Status

Our suggestion remains that this should be elaborated on in the Michelson Whitepaper to prevent such confusion from arising.

Section V – Verification

Unresolved.

--

Section VI - Domain specific data types

While the **tez** unit is described early in the specification, the **mutez** unit (1 / 1,000,000th of a **tez**) is not described, and only briefly mentioned in the section that gives examples of value specifications. We expected it to be described in this section.

Section VI – Status

The following commits address this item by changing the mention of to the "tez" unit throughout the document to reference the "mutez" unit instead:

`15c8c7af869a9e6b0c8bee15f642e7c8eae01f2f`

`15c8c7af869a9e6b0c8bee15f642e7c8eae01f2f`

`5e4bd12d3bbebf523c6b0ab1413586662ff3c867`

`bf75c86e0ad6b86b36aceb999de885c4ab67dc89`

Section VI – Verification

Resolved.

--

Section VII - Domain specific operations

1. Under **"Operations on Tez"** it is explained that **"Tez are internally represented by a 64 bit signed integer."**

   In the implementation, Tez are internally represented as **mutez**, which are in turn represented using only the positive part of OCaml's **int64** data type (which is a 64-bit signed integer).

   It follows that it would be more accurate to explain Tez as backed by a signed 43-bit unsigned integer, ie $\mathbf{log_2(2^{63} / 1{,}000{,}000)}$.

2. The **INT** operation does not seem to be documented. From reading the implementation it looks like this the documentation should explain that the INT operation transforms the type of the NAT value at the top of the stack to an INT type.

Section VII – Status

1. Addressed by commit `15c8c7af869a9e6b0c8bee15f642e7c8eae01f2f` by correcting "Tez" -> "Mutez".

2. Commit `15c8c7af869a9e6b0c8bee15f642e7c8eae01f2f` removes the mention of the "INT" operation from the syntax definition at "XI - Full grammar", but as of commit 24b0ab4b (master on November 27th 2018) "INT" is still referenced as an instruction in "X - Annotations" under the "Variable Annotations" subsection (line 2125).

   It also seems to still be mentioned in docs/api/rpc_proposal.rst, and to be implemented in src/proto_alpha/lib_protocol/src/script_ir_translator.ml and michelson_v1_primitives.ml in the same directory.

   Similarly the "Variable Annotations" section mentions a "IS_NAT" instruction, but that is not mentioned anywhere else.

   Yet it seems to implemented as "ISNAT" (without the underscore) in src/proto_alpha/lib_protocol/src/michelson_v1_primitives.ml and in use by several test cases (cps_fact.tz, slices.tz) and mentioned in docs/api/rpc_proposal.rst.

Section VII – Verification

Partially resolved.

--

Section IX - Concrete Syntax (of Michelson)

1. It seems like there is a typo in this sentence: **"1,234,567.0" means 12345678<u>9</u> tez**. We believe it should read: **"1,234,567.0" means 1234567 tez**.

2. It would seem like the author intended to write "**RFC 3339**", but left out a "**3**" in this sentence: "**timestamps are written using RFC 339 notation**".

   Being precise when referencing external technical specifications is important since it eases the task of understanding the project.

Section IX – Status

1. Addressed by commit `15c8c7af869a9e6b0c8bee15f642e7c8eae01f2f`
2. Addressed by commit `5693ac2fb`

Section IX – Verification

Resolved.

Other Observations

The following section details observations that were not in the scope of this audit, but that we nonetheless made, and believe should be mentioned.

Insecure RPC Between Tezos-Client and Tezos-Node

The communication between **tezos-client** and **tezos-node** is done via a HTTP+JSON API, to achieve this the OCaml libraries **cohttp** (HTTP) and **conduit** (transport-agnostic communication channels) are used.

Conduit supports TCP and TLS - either using OCaml's TLS library or OCaml's binding to OpenSSL - and **tezos-client** and **tezos-node** have a flag to enable TLS. By default, TCP is used for this communication. When conduit is used on the server side, a provided private key and X.509 certificate are used, while the conduit client side does not verify the server certificate. This means the communication between **tezos-client** and **tezos-node** can be compromised by a man-in-the-middle attacker.

We recommend that Tezos implements certificate verification, either a *chain of trust* or verification of certificate/public-key fingerprints.

A workaround for an initial release might be to restrict the **tezos-client** to only communicate with **localhost**.

Tezos Response

We have mitigated this issue to some extent by the documentation change identified, recommending that communications be restricted to localhost. We are waiting on cohttp to support TLS authentication, but are investigating other RPC mutual authentication.

Status

The temporary workaround suggested in our initial report (restricting the listener to the localhost loopback interface) was added to the documentation in commit id `f626e9d6ae7d83c6f6f4af2cfeb3a84b17dfcd28`. This document recommends to protect the tezos-node control interface from external connections, but does not prevent local applications running in different execution contexts from accessing it.

Our recommendation remains that Tezos makes mutually authenticated RPC mandatory, which would both protect against same-host attackers and re-enable the possibility to have the node listen on the external network interfaces if Tezos should wish to do so.

Verification

Partially resolved.

Insecure Installation Process

The installation instructions for Tezos currently contains a number of insecure steps where the user is vulnerable to man-in-the-middle attacks.

These include accessing cloning code from **gitlab.com** without verifying the authorship, and installing dependencies from OPAM (the OCaml package manager) which does not verify authorship of the code it installs. This leaves the users of Tezos in its current state vulnerable to compromise by malicious attackers, and we recommend that this be remedied before a Tezos version intended for public consumption be released.

We recommend that the Tezos developers start signing their committed code cryptographically using the code signing capabilities built into the **git** source control utility, this can be done with **git-tag --sign** for tags/releases, and **git-commit --gpg-sign** for commits. The signing keys used by trusted developer should be published in a way that makes updates to the list of keys transparent, for example on keybase.io[2], or in an append-only log backed by a reputable blockchain.

While an early draft proposal for package signing for OPAM exists, in the meantime, the OCaml dependencies could be declared in a way that allows users to install them from their operating system's package manager (from Debian's package repositories, for example), or they could be vendored inside the Tezos repository as **git subtrees** pinned to specific revision hashes.

The commits updating these pinnings could then be signed with **git** by the Tezos developers, establishing a "chain of trust."

The OCaml compiler itself presents another problem since their releases are not cryptographically signed. This unfortunately makes usage of OCaml inherently insecure, since there is no safe way to install the OCaml compiler, but using packages signed by OS vendors would at least make targeted attacks harder.

Tezos Response

This is an area of active improvement. We already started signing our commits, on release, we have the intention to sign tags and downloadable archives, and we will improve our handling of external dependencies. The opam repository has been pinned with sha256. Additionally, we are investigating other ecosystem-level ways to handle code signing, audit reporting, and interoperability.

Status

Most of the git commits to the master branch are now signed by authors of the Tezos project using OpenPGP keys, as can be witnessed starting from commit `4a436c2f18ffb84b87e8cd85ed45eedb584efa2c`.

We did not see signs of the documentation having been updated with instructions to validate these signatures, nor did we find a list of contributors whose keys are supposed to be trusted to submit commits. We would expect this information to be available in e.g. docs/introduction/howtoget.rst which details the installation process.

[2] https://keybase.io/

The build process invokes a script called `install_build_deps.sh` that sources another script called `versions.sh` that specifies a commit hash of the opam repository that tezos controls with all the build dependencies. When it is necessary to change the dependency versions, the admin manually refreshes that repository and updates the commit hash variable.

Verification

Partially resolved.

Recommendations

We recommend that the unresolved *Issues* and *Suggestions* stated above be addressed as soon as possible in addition to continuing discussions on all of the *Other Observations* listed above.

In future security audits, we strongly suggest that audit results are responded to in a more timely manner and verification is performed within a reasonable time frame of the initial review and delivery of the initial audit report. Over time, it becomes increasingly difficult for auditing teams to contribute value if the focus becomes on the code diffs of a significantly changed codebase (due to further development over time) and not the security issues themselves. Prompt responses can better facilitate meaningful auditor input throughout future development releases.