Security assessment checklist

Smart contracts on Tezos

Version / Date	Description
1.0 / 01.02.2022	Version 1.0 for publication

Purpose

This checklist has been created to foster knowledge and experience sharing. The checklist has the goal to be one framework element ensuring high security on Tezos.

The security checklist is mainly addressed to Tezos smart contract developers and assessors.

Disclaimer

This checklist does not claim to be complete or to cover any possible scenario, but only provides a basic checklist of common known potential security issues and pitfalls.

Applying this checklist does not ensure all potential weaknesses can be found. Users of the checklist have to think outside of this checklist in the role of a malicious adversary in order to identify security issues not (directly) covered by this checklist.

How to use

Smart contract developers

The security assessment checklist helps Tezos smart contract developers to learn about potential security issues, but also to cross-check their developed code.

Smart contract security assessors

Smart contract security assessors may use this checklist for their security assessments to ensure the obvious has not been left behind, but also to document work.

The suggested way on how to use this checklist is to refer in the report which version of the checklist has been used, but also to clearly outline which checklist tables have been used. If a particular checklist table has been used, none of the checks in the table can be deleted. All checks of the particular checklist table have to be executed, resp. marked as “not

assessed", if not done. See also chapter "result states" about possible states for each check.

Most importantly, the checklist should not be seen as a pure tick box exercise, but as a checklist to ensure the well known potential weaknesses and pitfalls are not missed. Security assessors are highly encouraged to understand the smart contract's use cases and to think about security issues not explicitly listed in the checklist. As such the security assessor has to think outside the box in order to identify potential security issues. Each checklist table contains in the last row a general entry to document additional notable performed checks, results, and observations.

For instance, a security assessor, if using this checklist, can include it in their report something like:

Our security assessment used the Tezos security checklist version 1.0, which can be found on https://github.com/InferenceAG/TezosSecurityAssessmentChecklist. We applied the following security checklist tables:

- *General tests*
- *System/Platform*
- *Storage*
- *Entrypoint*

Contribution

Please contribute to the checklist[1] to foster a collaborative ecosystem with a high security of the Tezos blockchain and its products running on.

[1] *https://github.com/InferenceAG/TezosSecurityAssessmentChecklist*

Security assessment paradigms / frame of reference

Understanding of underlying system's mechanics

Knowledge

In order to prevent security issues it is paramount to understand the mechanics of the underlying system. Here, some links to information which may act as a starting point in order to understand mechanics:

- https://tezos.gitlab.io/active/michelson.html
- https://tezos.gitlab.io/developer/michelson_anti_patterns.html
- https://ligolang.org/docs/tutorials/security/security
- https://gitlab.com/tezos/tezos/-/blob/52a074ab3eb43ad0087804b8521f36cb517f7c28/docs/whitedoc/gas_consumption.rst

Testing

Since the underlying system's mechanics on purpose (e.g. protocol upgrade) or mistakenly (e.g. bug or unknown effect introduced in an update) can change, the underlying system's mechanic has to be regularly and independently checked. This allows developers and security assessors when developing or assessing a smart contract to reliably base their work on this underlying system's mechanics.

Thus, as a companion to this checklist, we started to create a Tezos security baseline checking framework including test cases to validate whether the underlying system's mechanics are still the way we know it respectively expect it.

This framework is still a very basic one and an initial set of test cases has been written. We highly encourage everybody to get familiar with the already defined test cases and to add additional missing test cases.

The Security baseline checking framework for Tezos can be found here:
https://github.com/InferenceAG/TezosSecurityBaselineChecking

Assumptions

Making assumptions in security is very dangerous. Made assumptions could be wrong/flawed or get inappropriate due to various developments over time.

Any assumptions should be clearly documented by the developers, peer-reviewed by other developers, and challenged by a security assessor. Security assessors should discuss assumptions with developers and they have to be documented by the developers in the official documentation and/or in the security assessment report.

Common flawed basic assumptions done by developers and security assessors are:

- Assuming different actors / adversaries are distinct
- Assuming different actors do coordinate among themselves
  (e.g. in order to agree on security framework and parameters between interfaces)
- Assuming different adversaries do not coordinate among themselves
  (e.g. in order to collaborate in order to exploit a solution)

Developers and security assessors have to review the code unbiased from such assumptions. Special care has to be applied where smart contracts interact with third-party contracts or include code from third-parties. For instance, by including code in a high-level smart contract language or Michelson (e.g. using “Global Constants”).

Security checklist

The checklist for smart contracts on Tezos consists of different checklist tables covering different topics.

General Tezos checklist tables

System / Platform

Description	Result
System / platform documentation & specification with regards to Tezos Check documentation and specification for the system / platform with regards to Tezos, especially with regards to the smart contracts. The documentation/specification should at least provide information on how the system / platform is working, the use cases with regards to the smart contracts, and the intended behavior of the entire system / platform, especially with regards to the smart contracts.	
Glossary Check whether there is a glossary with specific domain wording in order to clearly define used words and improve common understanding.	
Documentation Check whether the same wording/definitions are used throughout the code and its documentation. No mix of different wordings/definitions. Example: rewards, bonus, interest. This improves readability and comprehension of code.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Storage

Description	Result
Storage documentation & specification Check documentation and specification for the storage. The documentation/specification should at least provide information about: - Meaning and types of storage parameters.	
Storage content Check whether storage only contains data subject to change.	
Storage parameters types Check storage parameters whether they have the appropriate type. *Example:* A number which is always positive by design/specification such as e.g. a token amount should be of type NAT and not INT.	

Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Gas issues and efficiency

Description	Result
Gas exhaustion (TSCD-009) Check for opportunities where operations fail because storage or loaded code/external calls get too big. E.g.: • Creation of contract call loops or on-chain view loops • Insertion of big numbers / values • Not using lazily deserialized storage such as e.g. big_maps • Iteration over lists which possibly could grow infinite. • No overuse of accessing and writing of complex data structure	
SmartPy sp.local() Check (in Michelson) whether there are repeated calculations of the same variables. If so, suggest the use of “sp.local()” in SmartPy.	
Early checks & fails Check whether the entrypoint returns/fails fast in order to save gas.	
Error strings Suggestion to use short strings (error mnemonics) or even numbers as error codes, which then can be looked up in a code error mapping table in the documentation. Note: It is recommended to throw errors (at least error codes) in order to provide hints to users and applications, why/where an execution failed.	
Repetition of code in Michelson Check whether lambda functions are used for repeating code parts. E.g. for “check_admin”.	
Useless checks Check whether code contains useless or repeated same checks. They potentially could be removed in order to save storage/gas.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Code issues

Description	Result
Duplication of code in high-level smart contract languages Check whether duplication of code is avoided in high-level smart contract languages (e.g. Ligo or SmartPy) by e.g. making use of functions.	
Constants in code Check whether constants are defined in high-level smart contract languages in a central location / file.	
Unused variables and code (TSCD-010) check whether any unused variables and code elements are removed from code.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Contract origination and initialisation

Description	Result
Improper initialisation parameters Check the initialization parameters to ensure the parameters are appropriately chosen and defined. Parameters have to be set in a way, which does not make (part of) the contract unusable directly after initialisation or soon after after a few interactions.	
Race conditions (TSCD-022) Analyze whether the process of contract origination and initialisation opens any opportunity for race conditions.	
Contract size Check contract size and gas limits.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Transactions

Description	Result
Transaction ordering (TSCD-017) Analyze whether a smart contract solution is susceptible to any beneficial attacks by transaction ordering by third parties (mempool observers, meta transaction relayers, bakers, etc.) and whether appropriate countermeasures are in place.	
Transaction delays (TSCD-020) Analyse whether a smart contract solution creates a disadvantage for its users, if a transaction is delayed by nature or on purpose (meta transaction relayers, bakers, etc.).	

0tez transaction to implicit address Analyse whether the smart contract may forges any transaction with 0tez to an implicit address. Note: Any transaction with 0tez to an implicit account will fail.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Entrypoint specific

The following checklist tables have to be applied per entrypoint.

General

Description	Result
Entrypoint documentation & specification Check documentation and specification for this entrypoint. The documentation/specification should at least provide information about: - Description what the entrypoint is for and what it should do - Meaning and types of input parameters / arguments - Pre conditions (e.g.who can call the entrypoint) - Post conditions - Returned operations (transfers, contract origination, etc.)	
Entrypoint and entrypoint parameters (TSCD-001) Check entrypoint parameters for its purpose and whether the entrypoint and all of its parameters are really required. Also check whether parameters can be used in a way different to the specification of the entrypoint. *Example:* A “from” parameter is not required if the entrypoint is always sending tez from “sender” or “source” address to a destination address.	
Entrypoint parameter types Check entrypoint parameters whether they have the appropriate type. *Example:* A number which is always positive by design/specification such as e.g. a token amount should be of type NAT and not INT.	
Authorization check (TSCD-004, TSCD-006) Check entrypoint implementation whether an authorization check is implemented and check whether implementation is correctly performing according to the entrypoint’s specification. *Consider:* “sender” vs “source” instruction to obtain the correct address.	
Handling of transferred tez (TSCD-011) Check whether the entrypoint is correctly dealing with tez transferred to and check whether this is in line with the entrypoint’s specification. *Additional note:* In general, the entrypoint should fail, if the entrypoint does not expect any tez. If the entrypoint is accepting tez, ensure that the tez is correctly handled by checking e.g. that any balance tracking values are correctly updated.	
Callbacks: Information obtained from other contracts (TSCD-014) Check if entrypoint’s code is correctly handling information obtained from other contracts using callbacks.	

In case the entrypoint is a “setter” entrypoint in a callback scheme, also analyze whether appropriate checks are implemented to ensure only intended contracts within the correct operation flow can call this setter function. Additional note: Due to Tezos message passing style the operation is executed, after the entrypoint’s code has been executed. Thus, values called via an operation are not immediately available. Additional note: In general, avoid obtaining information from other contracts, if possible. Consider implementing and using “on-chain views”.	
Calls to untrusted addresses (TSCD-009 & TSCD-013) In case the entrypoint is crafting operations to untrusted contracts, carefully analyse for any security risk/issues, since the called addresses may not act in the way the smart contract coder assumes. Potential security risks are: ● Gas exhausting attacks ● DoS attacks ● Providing wrong answers / values ● Reentrancy and call ordering attacks ● Replacement of code (using lambda functions stored in big_maps) ● Destination contract can make the whole transaction to fail *Consider and investigate:* ● Contract allows the registration of arbitrary addresses - implicit, but also smart contract addresses. ● Any tez payout functions. Note: Generally it is recommended to implement a solution where users have to claim their payout instead that the contract pays tez out. ● Consider implementing a check that excludes that callee can be a smart contract (SmartPy: “sp.sender <= "tz3jfebmewtfXYD1Xef34TwrfMg2rrrw6oum") See also the token integration checklist: https://gitlab.com/camlcase-dev/dexter/-/blob/develop/docs/token-integration.md	
Operation ordering (TSCD-015) Check whether the operations crafted by the entrypoint are in the correct logical order and in line with the entrypoint’s specification.	
Processing with wrong values (check in Michelson) Check whether the entrypoint is correctly processing values. *Example:* ● Subsequent code takes an initial storage or entrypoint value instead of an updated value. ● Subsequent code takes an already calculated value instead of the original value provided via a storage or an entrypoint parameter.	
Storing of wrong values (check in Michelson) Check whether the entrypoint’s resulting storage has wrong (updated) values in. *Example:*	

The values in the output storage may not be the final calculated values, but the input values or some intermediary calculated values due to wrong storage handling by the entrypoint's code. Lookout for Michelson instructions such as: Update, Update n, and Pair.	
Calculations Include "calculations" test cases in case the entrypoint has any calculations in.	
Testing Check whether the on-chain view is appropriately covered with testing. Consider coverage with unit testing, integration testing, and mutation testing.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

SmartPy

The following test cases are related to SmartPy. However, both should be already detected, if Michelson code is reviewed in the entrypoint test cases above.

Description	Result
SmartPy: Inadvertent meta-programmation with control statements (TSCD-021) Check control statements in SmartPy code whether they are correctly used and produce the desired/specified programmatic flow.	
SmartPy. Using sp.local together with maps / big_maps (TSCD-025) Check whether big_map / map values stored and modified in sp.local derived variables are correctly written back to the map / big_map.	

On-chain view specific

The following checklist table has to be applied per each declaration of a on-chain view in own code, but also for called on-chain views:

Description	Result
On-chain view documentation & specification Check documentation and specification for this on-chain view. The documentation/specification should at least provide information about: - Description what the on-chain view is for and what it should do - Pre conditions - Post conditions - Meaning and types of input parameters / arguments	
On-chain view input parameters Check on-chain view input parameters for its purpose and whether parameters are really required. Also check whether parameters can be used in a way different to the specification of the on-chain view.	
On-chain view input parameters types Check on-chain view input parameters whether they have the appropriate type. *Example:* A number which is always positive by design/specification such as e.g. a token amount should be of type NAT and not INT.	
Calculation with wrong values(check in Michelson) Check whether the on-chain view is correctly processing values. *Example:* ● Subsequent code is taking an initial storage or on-chain view value instead of an updated value. ● Subsequent code is taking an already calculated value instead of the original value provided via storage or on-chain view parameter.	
Storing of wrong values (check in Michelson) Check whether the on-chain view's result has the wrong values in.	
Calling untrusted on-chain views In case the code is calling an untrusted on-chain view, potential security risks have to be carefully assessed. Potential security risks are: ● Gas exhausting attacks ● DoS attacks ● Providing wrong answers / values ● Replacement of code (using of lambda functions stored in big_maps)	
Calculations Include "calculations" test cases in case the entrypoint has any calculations in.	
Testing	

Check whether the on-chain view is appropriately covered with testing. Consider coverage with unit testing, integration testing, and mutation testing.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Calculations

The checklist table “calculations” has to be applied in case there are calculations done within an entrypoint or within an on-chain view.

Description	Result
Euclidian division and bit shifts (TSCD-016) Check every division and bit shift to the right whether in-accuracy of calculation can not be exploited. *Additional note:* Avoid using divisions and use multiplications for comparisons, if possible. Example: A user has “a” tokens of a token where in total “A” tokens exist in the pool, but the user also has b tokens of B tokens. Of which token does the user have a bigger share? Instead of comparing “a/A” with “b/B”, you can prevent a division by comparing a * B with b * A. (Note: only valid in case A and B are both positive or negative numbers).	
Imprecise numbers (Euclidian division and bit shifts) (TSCD-016) Check whether (internal) used and/or stored numbers are sufficiently precise to reflect the desired design. *Example:* Calculating and compounding of a yearly interest rate of 0.9%: A user having 100 tokens would never earn any interests, since 100n * 9n/1000n = 0tokens. Thus, based on the specification / definition this may be inappropriate, since accumulation of interest (compounding) would not be done, if this rate is applied all the time with an imprecise value to store the token.	
Use of BALANCE instruction (TSCD-023) Check whether BALANCE instruction is appropriately used.	
Mutez overflow/underflow (TSCD-002) Mutez values may overflow, since they are 64bit unsigned integers. *Additional note:* Overflow/underflow creates a runtime error. Thus, mutez overflows/underflows may lead to unusable smart contracts.	
Variable types Check locally used parameters whether they have the appropriate type. *Example:* A number which is always positive by design/specification such as e.g. a token amount should be of type NAT and not INT.	
Explicit usage of Euclidean divisions (SmartPy) Check whether explicit usage of “//” (or sp.ediv) for Euclidian divisions is used instead of “/”. *Additional note:* This is only a suggestion to be more explicit in coding, which may also increase coders' awareness about Euclidean divisions.	

Type conversion using “abs” (TSCD-018) Check whether code, when using the function “abs” to convert a value of type “INT” to a value of type “NAT”, is ensuring that the value to be converted is not negative.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Specific checklist tables

Admin/operator functions

Description	Result
Admin/operator lockout Check in multi-admin/-operator setups whether the contract has a built-in check that the last admin/operator in the list can not be removed without adding a new one. *Additional notes:* Sometimes this is done by design, since teams may want to fully remove admin/operators at a later stage.	
Replacing of admin/operator accounts (TSCD-019) Replacing an admin/operator should be a two-step process to prevent a working admin/operator account from being replaced with a non-working one.	
Correct functioning of multi-voting/approval solutions Check whether the current voting / approval procedure is in line with the design / specification. In particular, check whether: • Quorum / threshold is correctly verified ensuring enough votes / approval are required. • Wrong crafted votes/approvals lead to a failure of the execution (veto right)	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Signature replay (TSCD-008)

Description	Result
Cross contract Check whether the signed data includes the contract destination.	
Same contract Check whether the signed data includes a counter variable.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Compiler

Description	Result

Compiler version (TSCD-003) Check whether the used compiler is up to date and does not have any known bugs/vulnerabilities.	
Deprecated functions (HL languages) (TSCD-005) Check whether the code is using deprecated functions.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Other topics & test cases

Description	Result
Private, secret, or sensitive data stored on-chain (TSCD-012) Analyze whether the smart contract transacts and/or stores any private, secret, or sensitive data and if doing, whether this is appropriately done.	
Using of time (TSCD-024) Analyze whether the smart contract is appropriately using any approximation to time.	
Using randomness (TSCD-007) Analyze whether the smart contract is appropriately sourcing/deriving random values.	
Signature malleability Analyze whether the smart contract is checking signatures and analyze whether the check is vulnerable to signature malleability. Additional information: Signature malleability may be possible where parameters for signature creation are unsafely combined and can be influenced by third parties. Simple example: sign(hash(concat(“ABC”, “DEF”))) = sign(hash(concat(“AB”, “CDEF”))).	
TZIP standards Check whether the smart contract is compliant with TZIP standards claimed by the design.	
Metadata - immutability of data / Verification of data (hashs) Check whether metadata is stored at a location where the metadata is immutable or where at least changes to the metadata can be detected.	
Other checks and observations Placeholder for adding checks and observations, which are not in the default checklist.	

Result states

State	Description
n/a	Not applicable.
Ok	Result of check, which is as expected.
Not ok	Result of check, which is not as expected. Any check flagged as “not ok” will be listed in the report either as an issue or a suggestion.
Note	Note to be considered by the reader.
Note (not reported)	Note to be considered by the reader of the checklist. This note has not been addressed in the security assessment report.
Not assessed	Check has not been assessed.
TODO	Check not yet done or completed.