Audit Report

THORWallet Governance Token and Vesting Smart Contracts

Produced for Defi Suisse AG

Produced by Dominik Spicher, 21 Analytics AG

20.07.2021

21 ANALYTICS

Contents

1 Executive Summary

We have found the token and vesting smart contracts to be generally of high quality. The review has uncovered no issues of high severity, two easily fixed issues of medium severity, and eight issues of low severity. We also list eight minor suggestions of how the contracts could be improved.

2 Scope

The review was performed on the `0.9.0` release[1] of the `THORWallet/smartcontract` Git repository[2].

In particular, the review pertained to the following smart contract source files, specified here with their `sha256` hashes:

```
tgt.sol          8332511952947f79924a9a90bf0280da97cad7e64d80b41b78d832e63027e721
vesting.sol      b666eda41f32acac0d2d640561fc73752982c2e2efd019cc481ad03a302f897f
```

and the following auxiliary test and specification files:

```
Readme.md          d054caa64f0c892ff453852c7159b7599fe2dd8759a824feca689ce9a58fa398
supply-curve.ods   a7fa488c7ff36a6bcc7bd55dc237710e30c127a633afbc308e98c08510289118
tgt-test.js        f92125663eec1f639db96bb972db9122690dd7a051c093f26d41d07772ef3b33
ERC20.allowance.js a3237bffe6b2d784199b89a787f04f014062329b431a3f572e5a7e6ae476d00a
ERC20.behavior.js  912b97366686e241ee16f05200c03a0a32923f91a451358dfb4d3797b1bd5a00
```

Imported contracts from well-known libraries such as Open-Zeppelin were not part of the review.

The review was constrained to the Solidity source files, low-level assembly code generated thereof was not inspected.

Additionally, the code was checked against information in the following pitch deck, which is not part of the above repository:

```
Thorwallet Pitch_v1.6.pdf
685f46f640ba14a033790e82b8f85f4906f146ed6f20b04dd233f929a6ec1be9
```

[1] Git revision `a8f4d5b1894f3a1f11bf52efe190c53caae3c059`
[2] https://github.com/THORWallet/smartcontract

3 Methodology

The review consisted of the following steps:

- Check for compliance with smart contract development best practices
- Check for compliance with specification, where available and applicable
- Manual inspection and analysis of the smart contract and test code
- Usage of static analysis tools

4 System description

4.1 Contracts

4.1.1 TGT

The THORWallet Governance Token (TGT) is an ERC20 token smart contract. It goes through two life-cycle stages: In a first, non-live stage, transfers are disallowed until exactly 460 million tokens have been minted by the contract owner to arbitrary addresses. Afterwards, the contract can be transitioned by the owner into a live stage where transfers are allowed. In this stage, the contract issues tokens in a monthly rhythm to a reserve address according to a predefined issuance schedule lasting nine years until all billion tokens have been issued.

4.1.2 Vesting

The Vesting contract keeps track of vested TGT token allocations. A vesting allocation includes an initially claimable amount,[3] as well as an amount that vests linearly over a given period of time. Holders of vested tokens can claim tokens at any given time. If the claimed amount is in agreement with the vesting schedule, the tokens are transferred from the vesting contract to the holder.

4.2 Roles

All contracts have standard contract owners who get privileged access to a subset of functions. Owners are initially set to the contract deployment account. The token smart contract includes functionality to transfer ownership to a different account.

5 Best practices checklist

Non-adherence to the characteristics listed in the below list does not represent a security issue itself. However, following best practices is a good indicator of care and attention to detail, it

[3] This is somewhat confusingly called a “cliff amount”, even though a vesting cliff usually refers to a future date until which no shares can be vested.

allows the review process to be more focused and efficient, thus making it more likely for issues to be uncovered.

- ✓ The code was provided as a Git repository
- ○ There exists specification, covering the most important aspects of smart contract functionality (only the issuance schedule is specified)
- ✓ The development process is understandable through well-delineated, atomic commits
- ✓ Code duplication is minimal
- ✓ The smart contract source files are provided in an unflattened manner
- ✓ The code compiles with a recent Solidity version
- ✓ The code is consistently formatted
- ✓ Code comments are in line with the associated code (mostly)
- ✓ There are tests
- ✓ Tests are easy to run
- ✓ Tests provide a reasonably high coverage (mostly, see Appendix A)
- ✓ The code is well documented
- ✓ There is no commented code (mostly)
- ✓ There is no unused code (mostly)
- ○ The code follows standard Solidity naming conventions
- ○ Smart contract functions are grouped according to Solidity guidelines

6 Findings

We rank our findings according to their perceived severity (high, medium, low), where an issue's severity is understood to be the product of its likelihood of being triggered and the impact of its consequences. Where deemed appropriate, we also report issues that are not security-relevant per-se but impact things like transaction-cost effectiveness or user experience.

See section 6.4 for suggestions for improvements which have no discernible impact on any of the above.

6.1 High severity

No issues found.

6.2 Medium severity

6.2.1 Possibly unrecoverable state due to over-minting of TGT token in pre-live phase

Advancing the TGT contract into the live state via the `mintFinish()` function requires exactly `INIT_SUPPLY` tokens to have been minted. This introduces the danger that the owner mints too

many tokens, whereupon the contract would end up in an unrecoverable state. This is because the supply cannot be decreased through burning in the pre-live stage.

Recommendation: It is recommended to `require()` at the end of the `mint()` function that `INIT_SUPPLY` has not been surpassed.

6.2.2 Token issuance schedule misses first month of first year

The variable `_lastEmitMAt` keeps track of which month (counting from the date where the contract went into the live stage) tokens were last emitted in. At the beginning of `emitTokensInternal()` it is checked how many full months[4] have passed since the token went live, and tokens are only emitted if this number is bigger than the current value of `_lastEmitMAt`. Because `_lastEmitMAt` is default-initialized to zero, the first tokens are only emitted after the first full month has passed, thus missing the very first month. The total number of tokens emitted in the first year will therefore equal 11*15'000'000 = 165'000'000, deviating from the advertised schedule by 15'000'000 tokens.[5]

Unfortunately, this is not caught by the test case. There, in tgt-test.js:L171, a comment erroneously claims that an intermediate month is skipped and the running total of the first year will run 15 million tokens short (which is checked). In fact, it has been the first month that was missed.[6] See also issue 6.3.8.

The final token amount will still be correct after the full issuance schedule of ten years, because in the very end tokens are issued according to the remaining discrepancy towards the maximum supply of one billion. Nevertheless, this issue has been deemed medium severity because it is guaranteed to occur, and because the deviation from the advertised schedule is significant in both the number of tokens and time.

Recommendation: It is recommended to make `_lastEmitMAt` a signed integer and initialize it to minus one. This will restore issuance during the first month.

6.3 Low severity

6.3.1 Dangerous use of `unchecked {}`

At three different locations in the TGT contract (L134, L225, L262), over- and underflow checks (introduced in Solidity 0.8.0) are deactivated through the use of `unchecked` code blocks. Presumably, this is done in order to save gas costs. However, the gas savings associated with

[4] It's full months because Solidity division rounds towards zero.
[5] To complete the argument, note that when `timeInM=12`, `timeInY` will be equal to `1 > 0 = _lastEmitYAt`, which will cause `_curveSupply` to be increased (with an incorrectly high yearly amount). This establishes that during the first year, 15'000'000 tokens will only be emitted eleven times.
[6] To see that no intermediate month is skipped, note that `60*60*24*30` on L156 equals an extension of `60*60*24*30*i` on L173 to `i=1`.

this are minuscule[7], especially in light of issue 6.3.2. This is only a potential issue because all three locations have immediately preceding `require`-checks which make sure that the operation cannot underflow (note that `unchecked` is not inherited in function calls, thus limiting the effect to the evaluation of function arguments). However, the danger is that these checks are inadvertently removed or changed due to human error before deployment. This is especially pertinent in `_transfer()` where a balance is decreased, and an underflow would be catastrophic.

Recommendation: It is recommended to remove the `unchecked` blocks.

6.3.2 Unnecessary gas costs and deteriorated user-experience due to token issuance

During the live phase, tokens should be issued every month. The smart contract checks whether it is necessary to do so right now in `_transfer()` and `_approve()` by calling `emitTokensInternal()`. Thus, `emitTokensInternal()` ends up being called by the following public functions:

- `permit()`
- `approve()`
- `transferFrom()` (two times)
- `increaseAllowance()`
- `decreaseAllowance()`
- `transfer()`
- `burn()`

This is highly suboptimal, because only the vast minority of those calls will actually be necessary (i.e. tokens will actually be issued). Also, this leads to unpredictable gas costs for users using those public functions.

Quantifying this issue, our experiments suggest the following gas costs for an ERC20 transfer:

- Without calling `emitTokens()`: 39’551
- With `emitTokens()` exiting on line L171 (i.e. no tokens issued): 42’386
- With `emitTokens()` executing fully and actually emitting tokens: 78’541

Thus, the expected constant unnecessary overhead of the above calls is expected to be roughly 3’000 gas (6’000 for `transferFrom()`), whereas the rare unexpected transaction cost for the user is an additional 39’000 gas, roughly doubling an ERC20 transfer cost.

Recommendation: Even with the current implementation, token administrators still need to observe contract activity in case none of the above functions are called during a month.[8]

[7] Our experiments suggest them to be 160 gas. Compared to a typical ERC20 transfer cost of 40’000 gas, this represents well below half a percent.
[8] Assuming it is their desire to adhere to the token schedule and not miss any months.

Therefore, it is recommended to remove all above internal invocations of `emitTokens()` and instead call the function directly each month. This will incur some additional transaction cost on behalf of the token administrator.[9] However, it significantly enhances the user experience for users of the smart contract.

6.3.3 Checks on `msg.sender` which would be better handled by modifiers

Both the TGT and Vesting contracts contain multiple checks where `msg.sender` is compared to a stored `_owner` address. These checks would be better covered by an `onlyOwner()` modifier, increasing the readability and simplicity of the smart contracts. Alternatively, the Open-Zeppelin Ownable[10] contract could be used.[11]

This concerns the following locations:

- TGT: L64, L70, L78
- Vesting: L44

6.3.4 TGT liveliness checks would be more robustly handled by modifiers

The TGT contract contains numerous checks that require the contract to be either live or not yet live. Readability of the contract could be improved if `isLive()` and `isNotLive()` modifiers were used instead. This is especially pertinent here because the two checks only differ in one symbol (== vs. `!=`).

This concerns the following locations: L149, L162, L211, L257, L276.

6.3.5 Non-transferable ownership of vesting contract

The `_owner` variable of the vesting contract can not be updated, and thus ownership not transferred to other addresses. This is risky because it leaves the contract in a vulnerable state should the owner's private key be compromised.

Recommendation: It is recommended to introduce a means of transferring ownership, either through a custom method similar to the one used for the TGT contract, or the Open-Zeppelin `Ownable` contract (see issue 6.3.3).

6.3.6 Erroneous vesting parameters can not be corrected

In the vesting contract, the check on L52 ensures that vesting parameters can not be submitted for an address twice. This is reasonable, as it is very unclear how vesting parameters for identical addresses could be merged, as the comment immediately above suggests. However, this also has the consequence that erroneously submitted parameters can not be corrected. A

[9] However, see comment 7.1 for a possible mitigation
[10] https://docs.openzeppelin.com/contracts/4.x/access-control#ownership-and-ownable
[11] The Open-Zeppelin implementation would also already provide means of transferring ownership, thus allowing `transferOwner()` to be removed.

potential error, say because the administrator did not include the cliff amount in the total amount,[12] can thus not be fixed.

Recommendation: In order to allow mistakes to be fixed without being forced to define what merging vesting schemes means, we suggest introducing the possibility to delete vesting parameters for a given address. On a technical level, this is easily done by removing the corresponding mapping entry. It would then be the administrator's responsibility to take into account tokens already claimed by the beneficiary under the old vesting scheme.

6.3.7 Unnecessary gas costs by using public functions instead of external

If public functions are not used in the smart contract, it is more efficient to declare them `external` instead. This is the case for the following functions:

- TGT
  - `totalSupply(), balanceOf(), transfer(), allowance(), approve(), setCurve(), transferOwner(), setReserve(), live(), name(), decimals(), burn(), mint(), emitTokens(), mintFinish(), increaseAllowance(), decreaseAllowance(), transferAndCall(), permit(), nonces()`
- Vesting
  - `vest(), canClaim(), vestedBalance(), vestedBalanceOf(), claim()`

6.3.8 Poorly structured tests

The tests contained in tgt-test.js provide a reasonable, although improvable, test coverage (see Appendix A). However, and more importantly, the vesting and token issuance tests are not unit-tests testing a properly defined test-case but instead try to cover a range of conditions and states, covering large parts of the life-cycles involved. This makes it much harder to reason about the test cases, and increases the likelihood that the testing code contains mistakes. This is acknowledged by the implementers, as demonstrated by comments on L78 and L134.

A perfect example of this danger is issue 6.2.2, whose presence is not uncovered by the test, because the faulty behaviour is hidden by a later part of the test whose effects are misunderstood by the implementer.

Recommendation: It is recommended to split up the tests for each of them to test specific aspects, and label them clearly with an expected outcome through appropriate Mocha test labels.

6.4 Comments

In this section we list suggestions for minor improvements that are not deemed severe enough to be called an issue, or whose resolution requires ambiguous tradeoffs.

[12] The smart contract registers `amounts[i] - cliffAmounts[i]` as the amount that vests over time

6.4.1 Adherence to the token issuance schedule requires a monthly transaction

If the token issuance schedule is to be adhered to, a transaction must call one of the public functions listed in issue 6.3.2 or `emitTokens()` directly each month. One potential adverse effect could be that the token administrator is forced to pay a temporarily high transaction fee because the transaction needs to be confirmed within a certain timeframe.

Instead, `emitTokens()` could be adapted to take into account months which have previously been missed and mint tokens for each of them. Care would need to be taken if this span of month spans multiple yearly periods and thus multiple issuance amounts. Due to this increased implementation complexity, no recommendation is given as to how to resolve this trade-off.

6.4.2 Confusing “cliff” nomenclature

As mentioned in section 4.1.1, the vesting scheme includes a token amount which can be claimed immediately by the beneficiary. In the code, this is called a “cliff” amount. This is somewhat at odds with the common vocabulary of vesting agreements where a cliff refers to a future date until which no vesting is possible.[13] A better name for the use-case at hand would be “unvested” because this amount is not part of a vesting dynamic at all (as is alluded to in the comment on L33).

6.4.3 Unused `IERC677Receiver` interface in `vesting.sol`

The `IERC677Receiver` interface is unused and can be removed.

6.4.4 Confusing variable name and unnecessarily complicated code when computing the claimable amount

In the vesting `claimableAmount()` function, the amount that has vested according to the vesting schedule is computed as follows (L69 - 76, comments omitted):

```
uint256 timeUnlocked = 0;
if(v.vestingDuration < currentDuration) {
      timeUnlocked = v.vestingAmount;
} else {
      uint256 vestingFraction = v.vestingDuration / currentDuration;
      timeUnlocked = v.vestingAmount / vestingFraction;
}
```

[13] See e.g. https://en.wikipedia.org/wiki/Vesting or https://www.investopedia.com/ask/answers/09/what-is-cliff-vesting.asp

There are two minor issues with this code. First, `vestingFraction` is not really a fraction but a multiple. Instead, its inverse is a fraction between zero and one. Second, the code can be simplified substantially by replacing it with the following line:[14] [15]

```
uint256 timeUnlocked = v.vestingAmount * min(1, currentDuration /
v.vestingDuration);
```

6.4.5 Uncovered vesting obligations on the vesting contract

The vesting contract honors its token vesting obligations by transferring tokens to the beneficiary once he claims them. From the beneficiary's perspective, it is completely unknown whether the contract will ever be able to do so, given that he has no guarantees about the token balance of the vesting contract. Furthermore, an insufficient token balance could lead to a race between vesting beneficiaries who each try to claim the available balance.

The assurances towards vesting beneficiaries could be improved if the smart contract only allows new vesting schemes to be added if the vesting contract itself already has the necessary token balance.

6.4.6 No events on the vesting smart contract

Smart contract events are a useful tool to keep track of a set of well-defined and parameterized occurrences in smart contract logic. This is very often used for audit and book-keeping purposes. Additionally, many third-party services offer efficient and convenient ways to extract events according to various criteria.

The vesting contract does not make use of this possibility. Due to the sensitive nature of the functionality (and its associated accounting needs), it is recommended to include events for the most important actions, for example:

```
event VestingAdded(address indexed account, uint256 amount, uint256
cliffAmount, uint64 vestingDuration);

event Vested(address indexed account, address indexed to, uint256 amount);
```

6.4.7 Issuance schedule is not committed to

Normally, the issuance schedule is an important piece of an agreement between issuer and investor, as token scarcity and putative value can be influenced greatly by it. From this perspective, it is problematic that the issuance schedule can be changed at will by the contract

[14] Assuming the Open-Zeppelin min function has been made available (https://docs.openzeppelin.com/contracts/4.x/api/utils#Math-min-uint256-uint256-)

[15] This assumes that no zero vesting durations are passed by the administrator, which is unclear given the comment on line L71. Note, however, that a zero vesting period is not really needed because it is already covered by the "cliff" amount, which essentially represents vesting under a zero period.

owner through the `setCurve()` function, without any restrictions. For example, he could call the function with an empty array argument, immediately causing the rest of the billion tokens to be emitted. From the investor's perspective, this could be improved by posing certain conditions on the newly provided issuance schedule, or by completely removing the possibility to change it.

6.4.8 Presumably unclear account nature of the token reserve

The commented code in the TGT smart contract L:200-203 suggests that the reserve account may be a contract account, instead of an externally owned account. In this case, it would be highly recommended to indeed call a token receiving function on the reserve contract, as is alluded to in the commented code.[16] This mitigates the risk somewhat that the tokens end up stuck on a contract that was not designed to receive ERC20 tokens, and has no possibility to send them on.

7 Limitations

Even though the code has been reviewed carefully on a best-effort basis, undiscovered issues can not be excluded. This report does not consist in a guarantee that no undeclared issues remain, nor should it be interpreted as such.

Appendix A: Test Coverage[17]

```
--------------|----------|----------|----------|----------|
File          |  % Stmts | % Branch |  % Funcs |  % Lines |
--------------|----------|----------|----------|----------|
 contracts/   |     77.4 |       53 |    67.57 |    77.33 |
  tgt.sol     |    71.68 |       50 |    63.33 |    71.79 |
  vesting.sol |    96.97 |    61.54 |    85.71 |    96.97 |
--------------|----------|----------|----------|----------|
All files     |     77.4 |       53 |    67.57 |    77.33 |
--------------|----------|----------|----------|----------|
```

[16] This should then also be done further up in the context of L179, in case the reserve account has been changed immediately preceding final issuance.
[17] Generated with the “solidity-coverage" hardhat plugin (version 0.7.16)