

Audit Report

GATE Token Staking Platform v2 second audit report

Summary and Scope

19	1	5	2
Notes	Warnings	Bugs	Vulnerabilities

This audit covers the second iteration of **Global Stock Exchange Group's GATE Token Staking Platform smart contracts**. During the audit, attention was paid to design, overall code quality, best practices, business logic and security. Math was independently verified by the GATE Team.

The staking platform consists of one smart contract: [Compound.sol](#). Also OpenZeppelin components [IERC20.sol](#), [Ownable.sol](#) and [Context.sol](#) were audited.

Git repository head [f1520f2d2cc810017041af6c0e2c20b8ec9bad50](#) was audited.


The GATE Team released remedies for all the critical problems encountered. However, while the auditor reviewed the changes, these changes are out-of-scope of this audit. See “Remedies” chapter for more details.

Purpose

This is the second iteration of GATE Token Staking Platform: a simple stake-and-get-rewarded smart contract for the [GATENet token](#).

Design

The contract is a simple autocompounding staking smart contract: users stake their GATENet tokens for a pre-defined time period, and get tokens vested to the contract in return. This is good



design: always keep the on-chain code footprint minimum. However, some of the complexity of autocompounding without loops have been replaced with unbounded loops, making some of the operations costly, and risky. There are also multiple view-only functions (which are not used internally) which should be implemented off-chain (and in other contracts, if needed), instead of on-chain.

Unbounded loops

The design relies on unbounded loops, which is [dangerous in the context of smart contracts](#): in addition to making execution costly, there is a risk of hitting the block gas limit, making the execution impossible to complete within the given gas limit. Always avoid loops, especially unbounded ones in smart contracts.

Checks-Effects-Interactions pattern is honored

Checks-Effects-Interactions pattern is carefully followed, this is a [best practice](#): this effectively eliminates re-entrancy attacks (the “DAO hack”). Although the contract is communicating only with a trusted address (the GATENet token), this is always the design pattern to follow.

Ownerless design

Ownerless design is good smart contract designing: all the involved parties should be equal. This is achieved with the current design.

Used tools and components

Solidity version 0.8.9 was used, which is the oldest safe Solidity, this is considered a best practice. Solidity compiler’s [errata](#) was also reviewed, and no critical issues affecting this particular contract were found. Optimizer is not used, and optimizer rounds are therefore not defined, this can be considered a bad practice.

OpenZeppelin Solidity library version 4.4.0 is used. While this is not the most recent version of the 4.4.x series, none of the found security issues [seem to affect this contract](#). Except regarding EIP-20 interface definition, OpenZeppelin is not really used. Ownable.sol is compiled in, but never used, adding only code footprint, and an extra attack vector. However, no problems regarding Ownable.sol found during the audit.

Contracts

The whole business logic fits well into one file of 284 lines of code, [Compound.sol](#). This can be considered a best practice: on-chain code should be as minimal as possible, and distributing functionality this simple across multiple contracts would be impractical and confusing. However, code footprint could be smaller, because some view-only functions could be implemented off-chain instead (for more information see bugs #2, #3, #4 and #5).

Implementation

The code follows mostly common Solidity practices and style, and is mostly consistent.

The Positive

- Solidity version is 0.8.9, which is the oldest safe Solidity, and Solidity version is defined [strictly to be 0.8.9](#). Both of these are considered a best practice.
- Ownerless design well implemented.
- Events are well used, and are descriptive.
- External functions are marked `external`. Although in this case it does not bring gas benefits, marking functions designed to be called only externally as `external` for semantic purposes is a good idea.
- `require()`s are used well, with meaningful error messages to the user.

The Negative

- `Ownable` is not actually used. While this is not a problem per se, always [remove dead code](#).
- `excess` is not needed, treat it as [dead code](#), and remove as such. Instead, invoke `transferFrom()` with the correct `amount` (matching the shares).
- Incomplete commenting.
 - NatSpec not used: even the source code might live “forever”, having NatSpec will help automatic handling of the source code in the future.
- Global mapping `fees[]` is not really needed, delete the logic as [dead code](#).
- No interface used for `Compound`. Use interfaces for robust integration with other smart contracts in the future.
- Asserts not used. This makes symbolic execution tools (such as Mythril) [practically useless](#), since there is no way to differentiate between normal errors (`require()`) and programming errors (`assert()`).
- Coding style is not always consistent: follow [Solidity style guidelines](#), and OpenZeppelin style.
- Encapsulation pattern not followed: consider using the OpenZeppelin’s [encapsulation pattern](#) for safer inheritance for possible future development.

Vulnerabilities (2)

Vulnerabilities are bugs which have serious consequences, such as loss of capital.

File:Line number	Problem and solution
Compound.sol:109	Might reach the block gas limit for some power users, making it impossible to withdraw user funds, effectively locking the user funds.
Compound.sol:266	Although the loop is technically capped, it will reach the current block gas limit in a year, if the contract stays untouched. If this loop reaches the block gas limit, it would make calling staking and unstaking impossible, effectively locking all the funds, until (if ever) the transaction fits into a block..

Bugs (5)

Bugs are programming errors which are not serious enough to cause serious consequences, such as loss of capital, but can be problematic in other ways.

File:Line number	Problem and solution
Compound.sol:69	<code>calculateFees()</code> might reach block gas limit for some power users, making it impossible to cash rewards. Original user funds are not affected, though.
Compound.sol:191	Unbounded loops might hit the block gas limit for some power users, failing the transaction. Although <code>view</code> functions are not usually mined on-chain (and therefore not affected by the block gas limit), this might happen if this function is used by some other smart contract. Remove this functionality, and implement it off-chain.
Compound.sol:204	Unbounded loops might hit the block gas limit for some power users, failing the transaction. Although <code>view</code> functions are not usually mined on-chain (and therefore not affected by the block gas limit), this might happen if this function is used by some other smart contract. Remove this functionality, and implement it off-chain.
Compound.sol:223	Unbounded loops might hit the block gas limit. Although <code>view</code> functions are not usually mined on-chain (and therefore not affected

	by the block gas limit), this might happen if this function is used by some other smart contract. Remove this functionality, and implement it off-chain.
Compound.sol:240	Unbounded loops might hit the block gas limit for some power users, failing the transaction. Although <code>view</code> functions are not usually mined on-chain (and therefore not affected by the block gas limit), this might happen if this function is used by some other smart contract. Remove this functionality, and implement it off-chain.

Warnings (1)

Warnings are features which work in a way that could differ from its original purpose, but are not bugs. These might have unintended consequences.

File:Line number	Problem and solution
Compound.sol:84	<code>shares</code> would be 0 if the <code>amount</code> is smaller than <code>shareWorth</code> , losing user funds (although the funds are still in <code>excess</code> , they cannot be cashed out, since <code>shares</code> is 0) Make sure <code>shareWorth</code> would never reach <code>MINIMUM_STAKE</code> , or add a check to <code>revert</code> gracefully on such an occasion.

Notes (19)

Notes are not problems per se, but instead points to pay attention to.

File:Line number	Problem and solution
Compound.sol:8	Maybe there would be a more describing name than <code>Compound</code> ? Maybe <code>AutoCompound</code> or <code>Staking</code> ?
Compound.sol:8	<code>Ownable</code> is not used. Remove unused code.
Compound.sol:18	Set this in the constructor (and take it as an argument), for maximum code reusability.
Compound.sol:19	Set this in the constructor (and take it as an argument), for maximum code reusability.
Compound.sol:45	<code>Staked</code> not emitted. Remove or emit.

Compound.sol:47	Rename <code>block</code> as <code>timestamp</code> . This could be also removed, since the timestamp information is available from the block of the transaction emitting this event.
Compound.sol:48	<code>Unstaked</code> not emitted. Remove or emit.
Compound.sol:62	Why is a static value set here? Either set it in the declaration, or take it as a constructor argument.
Compound.sol:69	Set this at the variable declaration, or take it as an constructor argument.
Compound.sol:70	This is useless: <code>fees[msg.sender]</code> is always 0.
Compound.sol:91	Extra parenthesis are unnecessary and confusing.
Compound.sol:119	<code>fee[]</code> logic is unnecessary (it's always 0): use a variable instead to save gas (<code>fee[]</code> resides in expensive storage).
Compound.sol:131	Revert instead, so the user could predict that they should not send this transaction, and pay gas for nothing, especially when there is the for loop making it potentially expensive.
Compound.sol:134	Add parenthesis for clarity.
Compound.sol:148	Unbounded loops are a bad idea. See bug #1.
Compound.sol:150	Skip expensive storage operations if <code>shares</code> is 0.
Compound.sol:162	Will <code>revert</code> if trying to invoke after <code>endDate</code> . While failing itself is fine in this situation, use <code>require()</code> with an error message to fail gracefully.
Compound.sol:163	Extra parenthesis are unnecessary and confusing.
Compound.sol:176	Although this seems to do the trick, it's always safer to compare the actual variable instead, in this case <code>totalShares</code> .

Remedies


The GATE Team introduced significant remedies for vulnerabilities #1 and #2, bug #4 in commit [e8c6fd86566cae311eb72f5b3b1d37199160fd7c](#). The following remedies were introduced:

Vulnerability #1	A fallback function <code>withdraw()</code> was introduced as a way to claim single
------------------	---

	stakes.
Vulnerability #2	Loop iterations were greatly reduced by changing the 1 hour interval to 1 day, making the loops cheaper, and making reaching block gas limit extremely improbable.
Bug #4	Loop iterations were greatly reduced by changing the 1 hour interval to 1 day, making the loops cheaper, and making reaching block gas limit extremely improbable.

Although these changes are out-of-scope of this particular audit, the auditor conducted a regular code review for these changes.

Conclusion

 **Critical issues with the smart contract code found during the audit.** Although the auditor reviewed the fixes, the recommendation is to audit the code again after fixing the critical issues.

This audit was conducted in accordance with the auditor's best skills and knowledge, but audits are not definite proof of bug free code. Smart contracts are still experimental technology, and can harbor unexpected problems.

About the Author

[Ville Sundell](#) has been involved with smart contracts since Bitcoin's "Script", stumbled upon Ethereum in 2015 and published his first Solidity smart contract in 2016. Since his first audit in [2017](#), he has audited tokens listed on world's leading exchanges such as [MATRYX](#), [DAWN](#) and [GATENet](#). His previous Ethereum smart contract audit was of [PayRue Staking Contract](#).