Audit Report
GATE Token

Summary and Scope

17	1	0	0
Notes	**Warnings**	**Bugs**	**Vulnerabilities**

This audit covers **Global Stock Exchange Group's GATE EIP-20 Token**, and its OpenZeppelin ERC20 implementation. During the audit, attention was paid to design, overall code quality, best practices, business logic and security.

The token is deployed at **0x9d7630adf7ab0b0cb00af747db76864df0ec82e4** on Ethereum Mainnet. The source code of the deployed contract has been verified to be HEAD **dd49e22d1fa9e0420ab77f71a0d020b0bcdadbb4** in the official GitHub repository.

The token is built upon OpenZeppelin's `ERC20Burnable`, using version 3.3.0. No previous audit for the burnable token of this particular version has been found.

Since the official name of the token standard is EIP-20, and OpenZeppelin's implementation is named ERC20 (for historical reasons), we refer to the standard itself as EIP-20, and OpenZeppelin's implementation as ERC20.

Line numbers will refer to the source on Etherscan.

Design

The GATE token contract is an EIP-20 compatible token with few additional features from OpenZeppelin: atomically incrementing and decrementing allowances, and token burning functionality.

Anyone can burn their tokens

Despite this being the usual way of implementing token burning, it's worth of noting that:

- Burning decreases the `totalSupply`, which is usually static. And since anyone can burn, it's possible during large market dips that a malicious actor could buy a lot of tokens cheaply, only to burn them later, as an act of market manipulation.
- **Exchanges and wallet providers should expect that `totalSupply` will decrease in the future**.

EIP-20 compatibility

OpenZeppelin's ERC20 is EIP-20 compatible, but with some liberties:

- Token transfers to (or from) `address(0)` are not possible. This is not specified in the token standard, but can be considered a best practice.
- On error, OpenZeppelin code calls `revert`, instead of returning `false`. The OpenZeppelin team considers this "conventional and does not conflict with the expectations of ERC20 applications" (see line 294).

Used tools and components

The contract was deployed on 11th of January 2021. The contract was compiled using Solidity 0.7.1 which was at the time over 4 months old. Since the Solidity team does not support previous versions, newest 0.8 or 0.7.6 should have been used. The contract is designed for OpenZeppelin 3.3.0, which was the latest version at the time of deployment. However, **the components seem to have no critical security issues at the time of writing**. Solidity compiler's errata was also reviewed, and no critical issues affecting this particular contract was found.

Contracts

The GATE token is composed of 4 contracts compiled into one smart contract by inheritance, creating the GATE token contract.

Context

OpenZeppelin code heavily utilizes this Context contract for GSN support. This contract is not GSN compatible, and hence has this dummy Context. While this contract is useless in this contract, OpenZeppelin code relies on its existence.

ERC20

Basic EIP-20 token, supports the core EIP-20 standard, additional conventional functions `name`, `symbol` and `decimals`, and OpenZeppelin specific `increaseAllowance()` and `decreaseAllowance()` for changing user's allowances atomically.

ERC20Burnable

This contract amends the ERC20 contract with `burn()` and `burnFrom()`, giving token holders a possibility to burn the tokens they own. It's worth noting, that `totalSupply` will be decreased accordingly.

GateToken

The main contract, only minting the initial supply to the deployer, as per OpenZeppelin convention.

Implementation

Generally, the code is of high quality, easy to read and understand, and consistent coding style is used throughout the codebase.

The Positive

- **Optimizer runs are greater than default**: the 1500 runs used, although unconventional, suits this particular purpose well (token contracts are generally called frequently).
- **OpenZeppelin is used**: OpenZeppelin is widely considered to be the best Solidity library available, built by industry's top experts.
- **Encapsulation design pattern is used**: encapsulation makes auditing easier.
- **SafeMath library is used**: although SafeMath is not needed from Solidity 0.8 onwards, using this library with earlier Solidity versions is considered a best practice.

The Neutral

- **Restrictions regarding `address(0)`**: tokens cannot be transferred to `address(0)`. This is not wrong per se (and is actually considered a best practice), but this behavior is not described in the EIP-20 standard. Conversely, token transfers to the token contract itself are not restricted (see *Notes* below).
  - Checks for `address(0)` could reside in respective `public` (external) functions: this way the `internal` functions could still be used to work with `address(0)`

addresses, while external accidental usage of `address(0)` would not be possible.

- **No `Burn` or `Mint` events**: separate events (in addition to the current ones) would make event scanning more straightforward, especially if the internal `address(0)` restrictions were to be removed.

The Negative

- **Old Solidity is used**: Solidity team does not support older versions, so rolling release scheme is used. Effectively this means that always the newest version should be used. With one minor modification (see *Notes* below), one can get the code compiled on 0.8.
- **Non-standard NatSpec is used**: although this is the OpenZeppelin way, NatSpec should stay structured as per the specification, so it could be used by other components.
- **No asserts used**: some security tools (such as symbolic execution, and basic formal verification) expect asserts, and would not work without them.
- **Commenting is not consistent**: some EIP-20 quirks are documented (like `Transfer` event's `value` can be 0), and some are not (like `transfer()` can be called with 0 `value`).

Vulnerabilities (0)

Vulnerabilities are bugs which have serious consequences, such as loss of capital. No vulnerabilities were found.

Bugs (0)

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. No bugs were found.

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.

Line number	**Warning**
486	GATE tokens can be sent to this token contract. Implement `_beforeTokenTransfer()` in order to prevent sending GATE tokens to this contract. It's understandable why this is not done in ERC20, since it could be used in applications where sending tokens to the token contract is desired. Preventing this is considered a best practice.

Notes (17)

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

Line number	Note
21	`Context` not really needed: we are not GSN compatible, and as the comment says, we are not a “intermediate, library-like contract” either. However, OpenZeppelin code relies on `_msgSender()` provided by this contract. `_msgData()` on the other hand, is unused.
23	Address could be explicitly cast to `payable` by `payable()` for clarity. This is also a requirement from Solidity 0.8 onwards, since `msg.sender` isn’t `address payable` anymore. This is the only change needed to get the code compile on Solidity 0.8.
27	`this;`, although a clever way to trick the compiler, is not needed anymore: from Solidity 0.7 onwards the compiler does not complain about the `view` visibility modifier, since `msg.data` is now `view`.
131	If Solidity 0.8 would be used, `SafeMath` would not be needed, since by default all arithmetic operations would be safe. Getting rid of `SafeMath` would greatly reduce the code footprint. Also, most of the functions here are not actually used.
142	`add()` should support an optional error message (like the rest of the functions). See `sub()` for an implementation example.
190	`mul()` should support an optional error message (like the rest of the functions). See `sub()` for an implementation example.
235	Always remove dead code instead of commenting it out.
326	This is not true: `decimals` can be set later by `_setupDecimals()` at any time. It should be immutable after initialization (as per the original design), though.
329	Move all the variable initializations to their own setters (`_setupDecimals()`), which would be callable only when contract is under initialization (unlike `_setupDecimals()`, which is callable at all times). Also consider making `name` and `symbol` mutable (for future uses of the ERC20 code): these are purely for display use, and would not have any impact on exchanges, for example, unlike with `decimals`.

329	`public` visibility modifier for constructors is deprecated in Solidity 0.7.
332	Instead of direct value assignment, use setter `_setupDecimals()`. This would conform to the encapsulation design pattern (private variables with getters and setters).
603	Per ERC20 convention (and taking into account that other OpenZeppelin non-EIP-20 functions in ERC20 contract also conform to this), make `burn()` to return `bool`, as other ERC20 functions do.
618	Per ERC20 convention (and taking into account that other OpenZeppelin non-EIP-20 functions in ERC20 contract also conform to this), make `burnFrom()` to return `bool`, as other ERC20 functions do.
619	Wrong error message prefix: should be `ERC20Burnable` instead of `ERC20`, per the OpenZeppelin convention of using the contract's name as the error message prefix. This would make troubleshooting easier.
630	Since `GateToken` is not a library contract (and therefore not designed to be inherited), use a strict compiler version.
637	Use NatSpec's `@dev`, like with the rest of the contracts.
644	For consistency, use `_msgSender()`, like OpenZeppelin is doing.

Conclusion

✅ **No critical issues with the smart contract code found during the audit**. Despite the feedback above, no changes to the source code are needed. Audits are not definite proof of bug free code: smart contracts are still experimental technology, and can harbor unforeseen 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 worked full time with smart contracts: developing and auditing for Ethereum and EOSIO based platforms. His previous Ethereum smart contract audit was of DAWN.