GStar Token and Crowdsale Smart Contracts Audit by Ambisafe Inc.
May, 2018
Oleksii Matiiasevych, Alexey Shendrick, Ulyana Gromova

1. **Introduction.** GStar requested Ambisafe to perform an audit of the contracts implementing their token and crowdsale. The contracts in question are hosted at: https://github.com/GStar-AI/crowdsale-smart-contracts/tree/9a9d26b68728ebb84a054c4f6b87e8798a893efc/contracts

   Contracts in scope are (and their parents):
   - GStarToken
   - GStarCrowdsale

2. **Disclaimer.** The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts for any specific purpose, or their bugfree status. The audit documentation below is for internal management discussion purposes only and should not be used or relied upon by external parties without the express written consent of Ambisafe.

3. **Executive Summary.** Contracts are **safe** to use as is. No breaking issues were identified. SafeMath used to protect calculations from overflows. Sale contracts can be paused by contract owner at any moment, which might be useful in an emergency situation. GStarCrowdsale is aiming to raise 76000 ETH combining received ETH and private contributions made by other means, contract will not allow to receive anything beyond that, though it cannot verify the validity of supplied **private contributions** value. Purchased tokens are not distributed immediately, but rather after the owner calls **releaseTokens()**, which implies that only **owner** of the contract decides when tokens become available for transfers. Contributors must be whitelisted before trying to do the purchase. 1.6 billion tokens will be created. **ERC20.totalSupply()** will always return 1.6 billion tokens, as it is not affected by the burning. GStarToken contract is **fully** compliant with the finalized ERC20 standard. There is **no** known compiler bugs for the specified compiler version and above that affect contracts in scope.

4. **Critical Bugs and Vulnerabilities.** No places in code were identified as critical issues.

5. Line By Line Review.

5.1. Note: it's not possible to foresee which solidity version will be used in the deployed contracts. Consider specifying strict compiler version.

5.2. Crowdsale.sol, Line 84. Note: **_processPurchase()** execution is commented out, which might break the child contracts logic if they rely on it. Consider uncommenting it, and leaving it up to a child to override it or not.

5.3. Crowdsale.sol, Line 131. Note: **_deliverTokens()** execution is commented out, which might break the child contracts logic if they rely on it. Consider uncommenting it, and leaving it up to a child to override it or not.

5.4. BasicToken.sol, Line 36. Note: **SafeMath.sub** is excessive here, because overflow can never happen due to a check on the line 33. Consider either removing the check or using simple subtraction here to optimize gas usage.

5.5. StandardToken.sol, Line 30. Note: **SafeMath.sub** is excessive here, because overflow can never happen due to a check on the line 27. Consider either removing the check or using simple subtraction here to optimize gas usage.

5.6. StandardToken.sol, Line 32. Note: **SafeMath.sub** is excessive here, because overflow can never happen due to a check on the line 28. Consider either removing the check or using simple subtraction here to optimize gas usage.

5.7. GStarCrowdsale.sol, Line 23. Note: **startTime** can be made **constant** because it never changes. This will save gas on deployment and reading.

5.8. GStarCrowdsale.sol, Line 24. Note: **endTime** can be made **constant** because it never changes. This will save gas on deployment and reading.

5.9. GStarCrowdsale.sol, Line 31. Note: **MINIMUM_PURCHASE_AMOUNT_IN_WEI** can be made **constant** because it never changes. This will save gas on deployment and reading. Consider using **0.1 ether** instead of **10**17** to make it easier to read.

5.10. GStarCrowdsale.sol, Line 37. Note: **fundingGoal** can be made **constant** because it never changes. This will save gas on deployment and reading.

5.11. GStarCrowdsale.sol, Line 74. Note: **isWhitelisted()** modifier is applied twice in this function. First in the function itself, then again on the line 78 (inside of the **WhitelistedCrowdsale** parent contract). Consider removing the modifier from here, to save gas on execution of every purchase.

5.12. GStarCrowdsale.sol, Line 102. Note: discount rate is hardcoded and is not affected by the **rate** set in constructor. We assume that **rate** is also known already

and can be hardcoded in the constructor, or the discount rate should be calculated based on the **rate** variable.

5.13. GStarCrowdsale.sol, Line 121. Note: contrary to the naming of **_postValidatePurchase()** function, this line doesn't do any validation. We recommend moving this action to **_updatePurchasingState()** or even better to **_processPurchase()** (see 5.2 and 5.3), as we think it is a more appropriate place for it.

5.14. GStarCrowdsale.sol, Line 172. Note: balance check is excessive here, because GStarToken will not allow to do the transfer with not enough balance. Consider removing the check to optimize gas usage.

5.15. GStarCrowdsale.sol, Line 188: Minor: it is possible to start Crowdsale after it has been closed, which will allow new payments to be received. Consider not allowing that.

5.16. GStarToken.sol, Line 15. Note: naming style inconsistency. In other files, state variables are named with **camelCase** style, while this only variable is named using **snake_case** style.

5.17. GStarToken.sol, Line 39. Note: **SafeMath.sub** is excessive here, because overflow can never happen due to a check on the line 36. Consider either removing the check or using simple subtraction here to optimize gas usage.

5.18. GStarToken.sol, Line 40. Note: **SafeMath.sub** is excessive here, because overflow can never happen due to a check on the line 36. Consider using simple subtraction here to optimize gas usage.

5.19. Note: Code style is inconsistent across the contracts. Some internal functions are with underscore prefix while others are not. Some parameters in functions are with underscore prefix while others are not. We recommend using a unified code style across the project.

Oleksii Matiiasevych