Storj Token Audit

JUNE 23, 2017 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

The Storj team asked us to review and audit their new Storj Token (STORJ) code. We looked at their contracts and now publish our results.

The audited contracts can be found in their storj-contracts repo. The version used for this report is commit `2bdeb27c0216d2f0889b6e7363d8a84b54cd7f39`.

Code quality is very good. Functionality is properly modularized, and most lines of code and nearly all functions have accompanying comments stating their purpose and/or reasoning.

Here's our assessment and recommendations, in order of importance.

EDIT: Most problems were addressed in the latest version of the code.

Severe

Problems with PaymentForwarder's pay function

`pay` function in line 44 of PaymentForwarder.sol has two severe problems.

1. The `pay` function accepts zero-valued payments. This is not desirable, per se, but in this case it brings a greater problem.

The code assumes that all payments are nonzero in the calculation of the variable `customerCount` (see lines 53 to 55). If two calls are made to `pay` with the same `customerId` and the first one has `msg.value == 0`, `customerCount` will be incremented both times because `weiAmount` will be 0 in line 57.

Add a precondition to ensure `msg.value != 0`.

EDIT: Addressed in https://github.com/Storj/storj-contracts/commit/40b8dd82ec58ad795c21b1f37e11ebccdb10a6c8.

1. Anyone will be able to call the `pay` function with arbitrary data. We don't have access to the rest of the crowdsale code and couldn't determine what the desired behavior is here. For example, who should be allowed to call the `pay` function is not clear. Consider making the function only callable from a specific address if that's the intended behavior.

EDIT: This was intended behavior.

Potential problems

Lack of safeguards in PaymentForwader

For such an important component, `PaymentForwarder` is lacking in safety measures. All math operations are unchecked for overflows. The comments state: "We trust Ethereum amounts cannot overflow uint256". Although this seems to be true, it's better to be safe than sorry. An overflow could leave the contract in an inconsistent state, and removing the possibility is worth the cost. This is part of what we call security in depth. Consider making all math operations safe by using `SafeMath` as with other contracts.

EDIT: SafeMath added in https://github.com/Storj/storj-contracts/commit/40b8dd82ec58ad795c21b1f37e11ebccdb10a6c8.

PaymentForwarder can be unhalted

The Haltable contract is designed for the use case of halting a working contract in case of an emergency. This can be used, for example, to fix something manually before resuming normal operation.

The functionality required by PaymentForwarder (as documented), on the other hand, is to completely finalize the processing of payments at the end of the crowdsale. We recommend to remove the `unhalt` feature so that the contract can't resume receiving payments after finalization, and to use more appropriate variable and modifier names.

EDIT: This was intended behavior.

Anyone can burn tokens

`CentrallyIssuedToken` inherits from `BurnableToken`, which provides a function allowing anyone to burn a certain token amount, destroying them and reducing `totalSupply`. Storj needs this to burn part of their token holdings, according to their crowdsale plans, but as a side effect anyone can burn their tokens.

Please make sure this is expected, or consider making the `burn` function only callable from a Storj address. Note that any ERC20 token allows owners to burn tokens by transferring them to the zero address or any random one, but this doesn't modify the `totalSupply`.

EDIT: This was intended behavior.

Discrepancy between comment and code

`CentrallyIssedToken` has a constructor parameter named `_owner`, which is used as the `upgradeMaster`. The comment in line 29 states an intent to allocate the initial balance to the *owner*; however, the balance is allocated to `msg.sender`. If it's an error, use the owner variable; otherwise, update the comment.

Shadowing of inherited state variable

The state variable `originalSupply` is declared in the UpgradeAgent contract, and redeclared in TestMigrationTarget (which inherits from the former). This is a case of variable shadowing that'll cause problems: functions implemented in UpgradeAgent will use their own state variable of the same name. Remove the declaration in TestMigrationTarget.

Warnings

Use safe math

There are many unchecked math operations in the code. It's always better to be safe and perform checked operations. Consider using OpenZeppelin's safe math library, or performing pre-condition checks on any math operation.

EDIT: SafeMath added in https://github.com/Storj/storj-contracts/commit/40b8dd82ec58ad795c21b1f37e11ebccdb10a6c8.

Use the latest version of the Solidity compiler

Current code is written for older versions of solc (some contracts specify 0.4.6, others 0.4.8). We recommend changing the solidity version pragma to the latest version (`pragma solidity ^0.4.11`) to enforce the use of an up to date compiler.

Undocumented interface of UpgradeAgent

Compared to the rest of the code, UpgradeAgent contract is lacking in documentation and clarity. Since this is intended to be used by future developers, we believe it's specially important that it be very well documented.

Our understanding is that a future upgraded token contract should implement an `upgradeFrom` function that receives a token owner and an amount of tokens (within the owner's balance) as parameters to transfer from the older to the new token. This function is called by the older token as seen in line 72 of UpgradeableToken.sol.

Consider renaming the parameters to `owner` and `amount`, which are more representative of their meaning. Also, consider declaring `upgradeFrom` as an `external` function, as it seems to be meant only callable from the old token.

Program against interfaces when appropriate

The `token` state variable in Issuer is declared of type `StandardToken`, which is a concrete implementation of an ERC20 token. Although it makes no difference in the generated code, consider changing the variable's type to `ERC20`.

Overloading of the Transfer event

The BurnableToken contract uses both a Burned event and the Transfer event (with a *to* field set to the zero address) to log the burning of tokens. The latter is overloading the semantics of the `Transfer` event of the ERC20 standard, which might not be a good idea as other software (such as wallets or exchanges) could be relying on these semantics. This is an ongoing discussion in the ERC20 standardization process, but our stance on it is that a different event should be used.

A comment in the code says the use of the Transfer event is to "keep exchanges happy". We're not aware of any exchange that relies on this behavior, but if this were the case, please note that there are balance modifications not logged as Transfers when upgrading tokens. If Transfer needs to be used in this way, be sure to be consistent about it.

EDIT: This was done for EtherScan.io compatibility. Unfortunately, this could not be addressed very well in a standard-compliant manner, as there is no standard behavior for this.

Types are not checked on runtime

Although the check in line 95 of UpgradeableToken.sol is a good idea, and will potentially catch errors when upgrading a token, we'd like to clarify that it's not enough to ensure that the `upgradeAgent` address corresponds to a contract inheriting from `UpgradeAgent`.

Solidity types are only checked/enforced on compile time. Once the contract is deployed, and transactions are executed, there's no guarantees that types will be correct. A malicious `upgradeMaster` could potentially call the `setUpgradeAgent` function with a contract that has an implementation of `isUpgradeAgent` and no implementation of `upgradeFrom`.

We have no recommendations here, but wanted to clarify on the matter.

Additional information and notes

- Good job using OpenZeppelin!
- There are no modifiers defined or used in the code written by Storj, but we think they could be helpful for improved code clarity. At this stage of the development process, however, it might be safer not to undergo such a refactoring.
- The comment in line 77 of UpgradeableToken.sol is incomplete.
- The comment in line 10 of BurnableToken.sol is incomplete.
- The comment in line 124 of UpgradeableToken.sol has a typo: says "begun" instead of "begin".
- The comment in line 60 of PaymentForwarder.sol explains the rationale for the `paymentsByBenefactor` state variable: being able to "construct contributions solely based on blockchain data". It should be noted, though, that events are *also* "blockchain data", and the emitted `PaymentForwarded` event would be sufficient to reconstruct all contributions. Consider removing `paymentsByBenefactor`.
- The contract Issuer.sol seems to be unnecessary, as it can be implemented in an off-chain script with a database.
- The throwing fallback function in line 41 of TestMigrationTarget.sol is not needed in Solidity ≥0.4.0. If a payable fallback function is not declared, payments to the fallback function are rejected.

Conclusions

One severe security issue was found, along with recommendations on how to fix it. Some additional changes were proposed to follow best practices and reduce potential attack surface.

EDIT: Most problems were addressed in the latest version of the code.

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the STORJ Token contract. We have not reviewed the related Storj project. The above should not be construed as investment advice or an offering of tokens. For *general information about smart contract security, check out our thoughts here.*

Security Audits

- If you are interested in smart contract security, you can continue the discussion in our forum, or even better, join the team 🚀
- If you are building a project of your own and would like to request a security audit, please do so here.

RELATED POSTS

SECURITY AUDITS

Metal Token Audit

The Metal team asked us to review and audit their new Metal Token contract code. We looked at their...

READ MORE

by OpenZeppelin Security

SECURITY AUDITS

Enigma Token Audit

The Element Group team asked us to review and audit their Enigma Token contract. We looked at the...

READ MORE

by OpenZeppelin Security

SECURITY AUDITS

Everus Token Audit

The Everus team asked us to review and audit their Everus Token (EVR) contract. We looked at the...

READ MORE

by OpenZeppelin Security