Store Labs

Store Token

Smart Contract Security Assessment

Version: 1.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the `STORE` token smart contract. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the `STORE` token contract contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/-closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the `STORE` token smart contract.

Overview

`STORE` is a Blockchain-powered platform that relies on a novel protocol (BlockfinBFT) to provide a leaderless, asynchronous and Byzantine Fault-Tolerant consensus algorithm to address scalability and decentralization challenges.

The `STORE` tokens have the following features:

- The `STORE` token implements the ERC20 standard as defined in `EIP-20`.
- Privileged users (i.e. Administrators) cannot modify account balances.
- The total `STORE` supply is capped at 1 billion tokens.
- The deployer of the `STOREToken` contract inherits the entire token supply and thus has the responsibility of distributing tokens.
- The `STOREToken` contract utilises a burn address by implicitly allowing the sender to set the recipient address in `transfer()` and `transferFrom()` to the zero address.

Security Assessment Summary

This review was conducted on the file `store.sol`, for which the SHA256 signature is `ab306e9856b03d1144b5ae61000b36c93b89249e54e28a532b0376a398c88a5f`.

The complete list of contracts contained in `store.sol` is as follows:

```
└── store.sol
    ├── STOREToken
    ├── SafeMath
    └── ERC20Interface
```

This security assessment targeted exclusively the `STOREToken` contract.

The manual code review section of the reports, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focuses on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

The testing team identified a total of three (3) issues during this assessment, all classified as informational.

These issues have all been acknowledged by the development team.

To support this review, the testing team used the following automated testing tools:

- Rattle: https://github.com/trailofbits/rattle
- Mythril: https://github.com/ConsenSys/mythril
- Slither: https://github.com/trailofbits/slither
- Surya: https://github.com/ConsenSys/surya

Output for these automated tools is available upon request.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the `STORE` token smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including comments not directly related to the security posture of the token contract, are also described in this section and are labelled as *"informational"*.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
STOR-01	Missing Ownership Mechanism	Informational	Open
STOR-02	ERC20 Implementation Vulnerable to Front-Running	Informational	Open
STOR-03	Miscellaneous General Statements	Informational	Open

STOR-01	Missing Ownership Mechanism
Asset	`store.sol`
Status	Open
Rating	Informational

Description

The deployer of the `STOREToken` smart contract is responsible for the distribution of the `STORE` tokens as the entire token supply is assigned to `msg.sender` (i.e. the Ethereum address who called the `constructor()` function during deployment).

If this particular account's private key is lost after the deployment, it will become impossible to allocate the `STORE` tokens to any other users. Similarly, if this private key is leaked, malicious actors can control the entire supply.

Recommendations

Consider implementing additional roles to manage access control rules for the `STORE` token contract. Refer to this contract for an example implementation.

STOR-02	ERC20 Implementation Vulnerable to Front-Running
Asset	`store.sol`
Status	Open
Rating	Informational

Description

Front-running attacks [3, 4] involve users watching the blockchain for particular transactions and, upon observing such a transaction, submitting their own transactions with a greater gas price. This incentivises miners to prioritise the later transaction.

The ERC20 implementation is known to be affected by a front-running vulnerability, in its `approve()` function.

Consider the following scenario:

1. Alice approves Malory to spend 1000 STORE, by calling the `approve()` function on the STORE token contract;
2. Alice wants to reduce the allowance, from 1000 STORE to 500 STORE, and sends a new transaction to call the `approve()` function, this time with `500` as a second parameter.
3. Malory watches the blockchain transaction pool and notices that Alice wants to decrease the allowance. Malory proceeds with sending a transaction to spend the 1000 STORE, with a higher gas price, which is then mined before Alice's second transaction;
4. Alice's second `approve()` happens, effectively providing Malory with an additional allowance of 500 STORE;
5. Malory spends the additional 500 STORE. As a result, Malory spent 1500 STORE, when Alice wanted her to only spend 500 STORE.

Recommendations

Be aware of the front-running issues in `approve()` and potentially add extended approve functions which are not vulnerable to the front-running vulnerability.

A further method typically used to address this attack vector is to force users to change the allowance to 0 first, before updating it to the desired value (requires two separate transactions). See the `safeApprove()` function in OpenZeppelin's `SafeERC20` contract.

STOR-03	Miscellaneous General Statements
Asset	`store.sol`
Status	Open
Rating	Informational

Description

This section describes general observations made by the testing team during this assessment that do not have direct security implications:

1. The `STORE` token contract unnecessarily uses `safeAdd()` in `transfer()` and `transferFrom()` when the total supply of `STORE` tokens fits easily into a `uint256`.
2. There are some comments with either typos or misleading information (see Recommendations section below for more details).
3. The ERC20 standard does not prevent users from inadvertently sending tokens to smart contracts addresses that don't have a way to withdraw/transfer those tokens. This can lead to significant value loss as described here;
4. Interfaces are usually built to handle `ERC20` tokens that use "18" decimals. Thus, there may be compatability issues with interfaces that integrate the `STORE` token in the future.
5. The variable representing the token supply for the `STORE` token contract is `public`, yet it is prefixed with the `_` character (`_tokenSupply`), which is conventionally used for `private` variables.
6. Using a decimal representation for large numbers (i.e. `_totalSupply = 100000000000000000`) can be prone to errors;
7. The `totalSupply()` function returns the number of tokens initially created, substracted by the token balance sent to the zero address (`0x0000000000000000000000000000000000000000`). It is worth noting that there are other ways of *burning* tokens, for example by sending them to the `0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead` address.
8. The `STORE` token contracts do not use the latest version of the Solidity compiler (version in use: `^0.5.0`, latest version available: `0.7.1`);
9. Variable names used throughout the contract don't strictly match the `ERC20` standard. Note that functions and events signatures are compliant with the `ERC20` standard.
10. The `uint` type is used extensively in the smart contract. It is an alias for the `uint256` type.

Recommendations

1. Consider removing the use of `safeAdd()` in `transfer()` and `transferFrom()` for gas saving purposes;
2. Consider updating the following comment changes:
   (a) Change `Constrctor` to `Constructor` in `store.sol:L51` ;
   (b) Update "1Billion in Edisons" to "1Billion in $Store" in `store.sol:L59` .

   It is worth noting that the supply for `ERC20` tokens is expressed in the smallest unit, meaning that by defining a token supply of $100,000,000,000,000,000$*, with decimals set to* 8*, the contract effectively creates* $1,000,000,000$ *STORE tokens. Please ensure this is the intended behaviour.*
3. Consider implementing `ERC777` (or `ERC223`) which both prevent users from sending tokens to contracts that fail to implement a `tokensReceived()` function;
4. Consider changing `decimals = 8` to `decimals = 18` (as suggested in the comment on line [43]);
5. Consider renaming `_tokenSupply` to `tokenSupply` ;
6. Consider using the scientific notation (i.e `10**17`) when defining the token supply;
7. Make sure the behaviour of the `tokenSupply()` function is understood (i.e. the value returned might not be adequate);
8. Consider upgrading the version of the Solidity compiler to use the latest version available.
9. Consider renaming the `tokens` variable to `value` to match the `ERC20` standard.
10. Consider replacing the `uint` type occurences with the `uint256` type, for clarification purposes.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `pytest` framework was used to perform these tests and the output is given below.

```
tests/test_approvals.py::test_transfer_approval                              PASSED  [12%]
tests/test_approvals.py::test_transfer_approval_insufficient_allowance       PASSED  [25%]
tests/test_deploy.py::test_deploy                                            PASSED  [37%]
tests/test_events.py::test_transfer_event                                    PASSED  [50%]
tests/test_events.py::test_approval_event                                    PASSED  [62%]
tests/test_transfers.py::test_valid_transfers                                PASSED  [75%]
tests/test_transfers.py::test_transfer_to_zero_address                       PASSED  [87%]
tests/test_transfers.py::test_transfer_insufficient_balance                  PASSED  [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

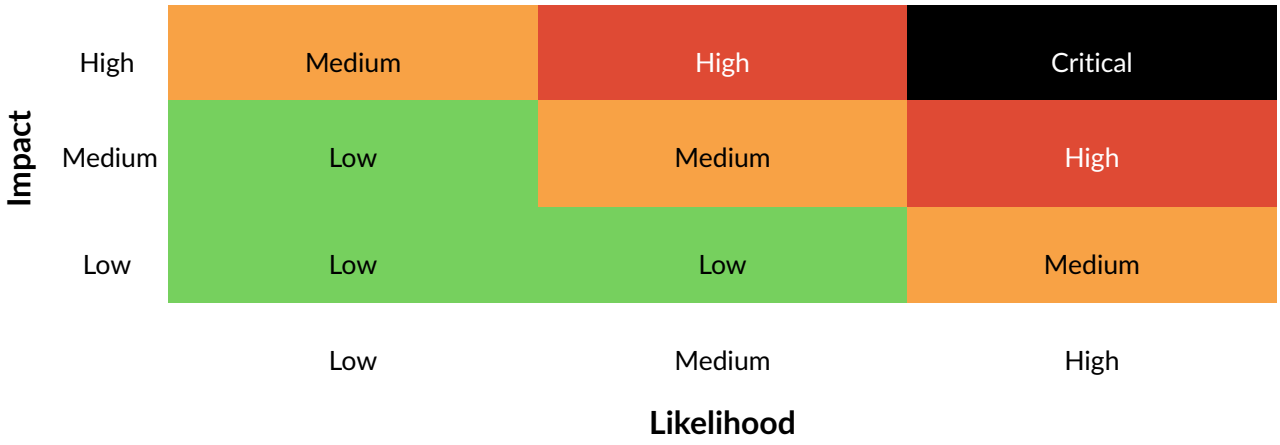

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

[3] Sigma Prime. Solidity Security - Front Running. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#race-conditions. [Accessed 2018].

[4] NCC Group. DASP - Front Running. Website, 2018, Available: http://www.dasp.co/#item-7. [Accessed 2018].