Urbit

Stardust

Smart Contract Security Review

Version: 1.0

September, 2021

Contents

Introduction

Sigma Prime was commercially engaged by Tlon Corportation to perform a time-boxed security review of a set of smart contracts related to the Stardust project. 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 smart 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 Urbit smart contract.

Overview

Urbit is designed as a peer-to-peer personal operating system running on cloud infrastructures. **Azimuth** is one of the core subsystems in Urbit that manages identities or address space. It is essentially a public key infrastructure built on top of smart contract-based blockchains. Its structure is similar to how IPv4 is designed, with dedicated terms such as galaxies, stars, planets, and moons, that identify the granularity of the identity, similar to classful addressing in IPv4. While the Azimuth contract is used to store all identity information, all operations to Azimuth must go through the `Ecliptic` contract, which owns Azimuth.

The **Stardust** project is a set of smart contracts developed by community members allowing owners of Azimuth identities to deposit their related ERC-721 tokens and obtain a fungible token (**StarToken**, an ERC-777 token) representing this ownership. Specifically, the `Treasury` contract allows a star owner to deposit the star and receive $1e18$ `$STAR` tokens. Anyone can also redeem a star from the `Treasury` contract for a fix amount of $1e18$ `$STAR` tokens. `$STAR` tokens are therefore only minted when a star is deposited, and burned when a star is redeemed.

Security Assessment Summary

This review was conducted on the `Treasury` contract hosted on the Stardust repository, assessed at commit c446b1f.

Note: the Azimuth, Ecliptic and StarToken contracts were excluded from the scope of this assessment but used extensively during testing. Additionally, OpenZepplin contracts and libraries were also excluded from the scope of this review.

The manual code review section of the report, 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 focused 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].

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

- 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.

Findings Summary

The testing team identified a total of 9 issues during this assessment. Categorized by their severity:

- Low: 2 issues.
- Informational: 7 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the scope of this review. 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 gas optimisations, 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
UTR-01	`Treasury` Cannot Recover `Star` Mistakenly Transferred To It	Low	Open
UTR-02	`Treasury` Cannot Recover `StarToken` Tokens Mistakenly Transferred To It	Low	Open
UTR-03	Multiple `Treasury` and `StarToken` Contracts Allowed	Informational	Open
UTR-04	Potentially Unclaimable Star Upon Losing Tokens	Informational	Open
UTR-05	Lack of Return Value in `redeem()` Function	Informational	Open
UTR-06	Different Star Valuations Motivates Arbitrage and Flashloan	Informational	Open
UTR-07	Operator Not Supported and Redundant Checks on Function `deposit()`	Informational	Open
UTR-08	ERC-777 Related Reentrancy Considerations	Informational	Open
UTR-09	Miscellaneous `Treasury` General Comments	Informational	Open

UTR-01	`Treasury` Cannot Recover `Star` Mistakenly Transferred To It		
Asset	`Treasury.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A user can mistakenly transfer ownership of a star to the `Treasury` contract. Consider the following case:

1. Alice spawns a star;
2. Instead of setting `Treasury` as a proxy through `Ecliptic.setSpawnProxy()`, Alice calls a different function, `Ecliptic.transferPoint()`. The star ownership changes from Alice to `Treasury`;
3. `Treasury` was not notified of such transfer and therefore the star is not recorded in the `assets`;
4. Alice cannot take back the mistakenly transferred star.

Recommendations

Consider adding a function to account for transferred stars into the `assets` array. For this purpose, a mapping can help check whether the star has been included in `assets`:

```
mapping(uint16 => bool) assetMap;
```

Then, the following function can be used.

```
function addToAsset(uint16 _star) public {
 require(assetMap[_star] == false, "Treasury: The star is in the assets");
 assets.push(_star);
 assetMap[_star] = true;
}
```

Notice that additional instructions are necessary on the functions `deposit()` and `redeem()`. In the `deposit()` function, the following instruction can be added on line [111]:

```
assetMap[_star] = true;
```

While in the `redeem()` function, the following instruction can be added on line [130]:

```
delete assetMap[_star];
```

UTR-02	`Treasury` Cannot Recover `StarToken` Tokens Mistakenly Transferred To It		
Asset	`Treasury.sol`		
Status	**Open**		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

A `Treasury` contract is coupled with a `StarToken` (ERC-777 standard) token. `StarToken` tokens are minted to a user who deposits a star to the `Treasury` contract and burned when a user redeems a star. The `Treasury` contract does not have a function to sweep over `Startoken` tokens mistakenly sent to it.

Consider the following scenario:

1. Alice deposits a star and receives $1e18$ `StarToken`;
2. Alice mistakenly transfers some of her `StarToken` tokens, say 100 tokens, to `Treasury` contract;
3. Alice cannot recover the mistakenly transferred tokens, while the `Treasury` contract also cannot recover the tokens.

Recommendations

Consider introducing a function where a privileged account (e.g. a governance contract, say `Governor`), can recover any excess tokens:

```
function recoverTokens(address _to) onlyGovernor public {
 uint256 tokenAmount;
 if(assets.length > 0)
 tokenAmount = startoken.balanceOf(address(this)) % (assets.length * oneStar);
 else
 tokenAmount = startoken.balanceOf(address(this));
 if(tokenAmount > 0) {
  startoken.transferFrom(address(this), _to, tokenAmount);
 }
}
```

Note that this would require the governor to have an allowance on behalf of the `Treasury` contract in the `StarToken` contrat

UTR-03	Multiple `Treasury` and `StarToken` Contracts Allowed
Asset	`Treasury.sol`
Status	Open
Rating	Informational

Description

The `Treasury` contract deploys a new `StarToken` contract during contract creation, as indicated on line [26]:

```
StarToken public startoken = new StarToken(0, new address[](0));
```

It is possible to deploy multiple `Treasury` contracts. It also means that there can be multiple `StarToken` contracts, where tokens in one `StarToken` contract cannot redeem a star in other `Treasury` contracts. This can potentially be confusing or misleading for users.

Recommendations

Make sure this behaviour is intended and consider deploying a dedicated `StarToken` contract and hardcoding its address in the `Treasury` contract.

UTR-04	Potentially Unclaimable Star Upon Losing Tokens
Asset	`Treasury.sol`
Status	**Open**
Rating	Informational

Description

The `Treasury` contract requires a fixed amount of $1e18$ `STAR` tokens to redeem a star asset. Since `STAR` tokens are only minted when users deposit stars, the amount of tokens held by the `Treasury` must be $1e18$ multiplied by `Treasury`'s asset count (equal to the number of deposited stars). That way, when all stars are redeemed, the total balance of `STAR` equals to zero.

This coupling between `Treasury` and `STAR` tokens means that if even one token is lost, i.e. sent to an address without a known related private key, then there should be at least one unclaimable star in the `Treasury`'s asset.

Recommendations

Make sure this behaviour is intended.

UTR-05	Lack of Return Value in `redeem()` Function
Asset	`Treasury.sol`
Status	Open
Rating	Informational

arb:market

Description

The `Treasury` contract can be interacted with from *"externally owned accounts"* (EOAs) and other smart contracts. When calling the `redeem()` function, an EOA's UI can track which `_star` it receives from `Treasury` contract through emitted events.

However, a contract account cannot read events data, and therefore, cannot necessarily easily work out what `_star` it receives.

What a *redeeming* contract can do is predict what `_star` it will receive from `Treasury` by retrieving the last item in the `Treasury.assets` array.

Recommendations

Consider updating the `redeem()` function to introduce a return value:

```
function redeem() public returns (uint16) {
 // must have sufficient balance
 require(startoken.balanceOf(_msgSender()) >= oneStar);

 // there must be at least one star in the asset list
 require(assets.length>0);

 // remove the star to be redeemed
 uint16 _star = assets[assets.length-1];
 assets.pop();

 // check its ownership
 require(azimuth.isOwner(_star, address(this)));

 // burn the tokens
 startoken.ownerBurn(_msgSender(), oneStar);

 // transfer ownership
 IEcliptic ecliptic = IEcliptic(azimuth.owner());
 ecliptic.transferPoint(_star, _msgSender(), true);

 emit Redeem(azimuth.getPrefix(_star), _star, _msgSender());

 return _star;
}
```

Such that a calling contract may catch the return value as follows:

```
event eStarRedeem(uint16 point);

function redeemOneStar() onlyOwner public {
 require(treasury.startoken().balanceOf(address(this)) >= ONE_STAR,
 "Recipient: STAR balance not enough");
 uint16 _star = treasury.redeem();
 emit eStarRedeem(_star);
}
```

Note that the snippets above does not account for other suggestions in this report, i.e. changing `_star` data type from `uint16` to `uint32` for code consistency.

The recommended change's impact on gas costs is insignificant, that is, from $230,327$ gas in the original function to $230,398$ gas in the modified function, or 0.03% gas increase.

UTR-06	Different Star Valuations Motivates Arbitrage and Flashloan
Asset	`Treasury.sol`
Status	**Open**
Rating	Informational

Description

Urbit's stars are NFTs that do not necessarily have equal value, as can be seen on Opensea where stars are traded at different prices (at the time of writing, the highest asking price for an Urbit star is 888 ETH and the lowest is 1.8 ETH).

On the other hand, the `Treasury` contract treats all stars equally from a valuation perspective which opens up two potentially interesting scenarios:

- A user deposits a star with a market price less than $1e18$ $STAR tokens. In this case, the user profits from the difference between the market value and the received $STAR tokens. This activity is similar to an arbitrage and can be applied to the DeFi context (would simply require a market pair with $STAR token on a decentralised exchange such as Uniswap).
- A user finds a star worth more than $1e18$ $STAR tokens in the `Treasury` contract's `assets` inventory. The user knows that the inventory release is done in *LIFO* (Last In First Out) fashion. To take out the wanted star from the inventory, say in *n-th* position from the last, the user takes a $n * 1e18$ $STAR tokens loan from the market, calls the `redeem()` function n times, keeps the star they want, then returns $n - 1$ stars to the `Treasury` contract by calling the `deposit()` function $n - 1$ times. The user then sells the acquired star on the market and pays the loan interest by using profits.

Recommendations

Make sure this behaviour is understood and intended.

UTR-07	Operator Not Supported and Redundant Checks on Function `deposit()`
Asset	`Treasury.sol`
Status	Open
Rating	Informational

Description

The `Ecliptic` contracts allows for an operator to perform actions on behalf of a `point` (or in this context, `star`) owner, for example, to call the function `transferPoint()`. This is reflected on line [457] of `Ecliptic.sol` as follows.

```
require(azimuth.canTransfer(_point, msg.sender));
```

While `Azimuth.canTransfer()` is implemented on line [1098-1108] of `Azimuth.sol` as follows:

```
function canTransfer(uint32 _point, address _who)
 view
 external
 returns (bool result)
 {
  Deed storage deed = rights[_point];
  return ( (0x0 != _who) &&
  ( (_who == deed.owner) ||
  (_who == deed.transferProxy) ||
  operators[deed.owner][_who] ) );
 }
```

The snippets above indicate that, other than the owner and the transfer proxy, an appointed operator should be allowed to transfer `point` ownership through the function `Ecliptic.transferPoint()`.

This capability is useful if the real owner (EOA) wants to manage their `point`s in a management contract and wishes to interact with the `Treasury` contract through that management contract instead of directly from the EOA.

However, the following code path on line [86-90] in the `Treasury` contract prevents an operator from depositing a spawned `star`.

```
if (
 azimuth.isOwner(_star, _msgSender()) &&
 azimuth.getSpawnCount(_star) cd== 0 &&
 azimuth.isTransferProxy(_star, address(this))
 ) {
```

Recommendations

If there is no specific reason why only spawned `star` owners are allowed to call the function `deposit()`, we recommend adjusting the `Treasury` contract to enable interactions with appointed operators by changing the condition checking in line [87] and line [89].

```
if (
  azimuth.getSpawnCount(_star) == 0
 ) {
```

Note that depositing an unspawned `star` requires direct interaction with the star's `prefix` owner or its spawn proxy as specified in `Azimuth.sol` on line [979–988] below, and therefore the operator cannot deposit unspawned `star`.

```
function canSpawnAs(uint32 _point, address _who)
 view
 external
 returns (bool result)
 {
  Deed storage deed = rights[_point];
  return ( (0x0 != _who) &&
  ( (_who == deed.owner) ||
  (_who == deed.spawnProxy) ) );
 }
```

Also, a set of checks have been conducted in Function `Ecliptic.spawn()`, and therefore redundant checks can be removed from `Treasury.deposit()`. A simplified version of the `deposit() function` could be as follows:

```
function deposit(uint16 _star)
 public
 {
  require(azimuth.getPointSize(_star) == IAzimuth.Size.Star);
  IEcliptic ecliptic = IEcliptic(azimuth.owner());

  // case (1):
  if (azimuth.getSpawnCount(_star) == 0) {
   // transfer ownership of the _star to :this contract
   ecliptic.transferPoint(_star, address(this), true);
  }
  // case (2):
  else
  {
   ecliptic.spawn(_star, address(this));
  }

  //  update state to include the deposited star
  //
  assets.push(_star);

  //  mint star tokens and grant them to the :msg.sender
  //
  startoken.mint(_msgSender(), oneStar);
  emit Deposit(azimuth.getPrefix(_star), _star, _msgSender());
 }
```

UTR-08	ERC-777 Related Reentrancy Considerations
Asset	`StarToken.sol`
Status	Open
Rating	Informational

Description

The `StarToken` contract implements the ERC777 token standard using the related OpenZeppelin library. One of the most distinguishable features of the ERC777 standard compared to ERC20 is that if the token receiver is a contract, then the contract can implement the `ERC777Recipient` interface which defines the function `tokensReceived()`. This function is a hook that will be triggered every time the contract receives a token. To enable the hook trigger, the receiving contract must register itself to the ERC1820 contract registry.

This unique ERC777 feature can be weaponised to trigger a **reentrancy** [3] condition. To be successful, this attack would require the victim contract to be affected by a security flaw as the result of not abiding by the recommended Checks-Effects-Interactions pattern.

This reentrancy condition can allow attackers to use two stars to switch for any other star in the list (see this relevant issue).

Say we have `assets = [a, b, c]` and we have a balance of $2e18$ `StarTokens`. The standard `redeem()` workflow would normally only allow us to redeem the Star `c`. However, consider the following:

1. `redeem()` **(1)**:
   - `pop(c)` -> `assets = [a, b]`
   - `ownerBurn(1e18)` - reenter on `tokensToSend()` before we've burnt our balance of $2e18$
2. `redeem()` **(2)**:
   - `pop(b)` -> `assets = [a]`
   - `ownerBurn(1e18)` - reenter on `tokensToSend()` before we've burnt our balance of $2e18$
3. `redeem()` **(3)**:
   - `pop(a)` -> `assets = []`
   - `ownerBurn(1e18)` -> `balance = 1e18`
   - `transferPoint(a, attacker)`
4. continue `redeem(2)`:
   - `ownerBurn(1e18)` -> `balance = 0`
   - `transferPoint(b, attacker)`
5. `deposit(b)`:
   - `transferPoint(b, treasury)`

- `push(b)` -> `assets =[b]`
- `mint(1e18, attacker)` -> `balance =` $1e18$

6. continue `redeem(3)`:
   - `ownerBurn(1e18)` -> `balance =` 0
   - `transferPoint(c, attacker)`
7. `deposit(c)`:
   - `transferPoint(c, treasury)`
   - `push(c)` -> `assets = [b, c]`
   - `mint(1e18, attacker)` -> `balance =` $1e18$

The attacker managed to own star `a` and only spend $1e18$ for it, bypassing the LIFO (*"Last In First Out"*) queue.

The testing team could not identify an exploitable attack vector for reentrancy on the `Treasury` contract. However, the testing team notes that the `ERC777` and `ERC1820` contracts were not included in the scope of this review. As a result, the testing team cannot express any opinions related to the security posture of these contracts.

Recommendations

ERC777 token contracts do significantly increase the attack surface available to malicious users, compared to simpler ERC20 token contracts. Consider whether the added complexity is worth the extra features provided.

UTR-09	Miscellaneous `Treasury` General Comments
Asset	`Treasury.sol`
Status	**Open**
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **No Clear Revert Message**

   The `Treasury` contract does not provide clear revert messages upon reverting. Therefore, users or developers may find it hard to track why a transaction fails.

   We suggest introducing the following revert messages:

   - line [81]: *"Treasury: Must be a star"*.
   - line [122]: *"Treasury: Not enough balance"*.
   - line [125]: *"Treasury: No star available to redeem"*.
   - line [132]: *"Treasury: Treasury does not own the star asset to redeem"*.

2. **Redundant Star Ownership Check**

   `Treasury` contract acquires stars from users who called function `deposit()`. In this function, the deposited stars' ownership is transferred from the users to `Treasury` contract and stored in `assets` list.

   Function `redeem()` redeems a star by burning 1e18 `STAR` tokens and transfers the ownership of a star from `assets` list to the redeeming user. However, there is an extra star ownership check in line [132] which is not necessary.

   As a recommendation, the code on line [132] can safely be removed.

3. **Data Type Unmatched: `uint16` for `_star` and `uint32` for `point`**

   Variable `_star` used in `Treasury` contract is a `uint16`. This variable matches the `point` variable in the `Azimuth` and `Ecliptic` contracts. `_star` and `point` are of type `uint16` and `uint32` respectively. Although a `_star` address is represented in 16 bits data, we recommend keeping the `_star` data type to `uint32` for code consistency.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Test Suite

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

```
test_const                                    PASSED  [3%]
test_mint                                     PASSED  [6%]
test_burn                                     PASSED  [9%]
test_approve                                  PASSED  [12%]
test_authorizeOperator                        PASSED  [15%]
test_revokeOperator                           PASSED  [18%]
test_operatorBurn                             PASSED  [21%]
test_operatorSend                             PASSED  [24%]
test_ownerBurn                                PASSED  [27%]
test_renounceOwnership                        PASSED  [30%]
test_send                                     PASSED  [33%]
test_transfer                                 PASSED  [36%]
test_transferFrom                             PASSED  [39%]
test_transferOwnership                        PASSED  [42%]
test_constructor                              PASSED  [45%]
test_constructor_multi_deployments            PASSED  [48%]
test_initial_state                            PASSED  [51%]
test_deposit_spawned                          PASSED  [54%]
test_deposit_spawned_proxy_not_set            PASSED  [57%]
test_deposit_spawned_transfer_to_treasury     PASSED  [60%]
test_deposit_unspawned                        PASSED  [63%]
test_deposit_lost_token_redeem_failed         PASSED  [66%]
test_redeem_owner                             PASSED  [69%]
test_redeem_alice                             PASSED  [72%]
test_redeem_extrabalance                      PASSED  [75%]
test_redeem_no_star                           PASSED  [78%]
test_redeem_many_stars                        PASSED  [81%]
test_deposit_redeem_recipient                 PASSED  [84%]
test_deposit_redeem_recipient_mod             PASSED  [87%]
test_deposit_redeem_recipient_malicious       PASSED  [90%]
test_azimuth                                  PASSED  [93%]
test_startoken                                PASSED  [96%]
test_gldToken                                 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.

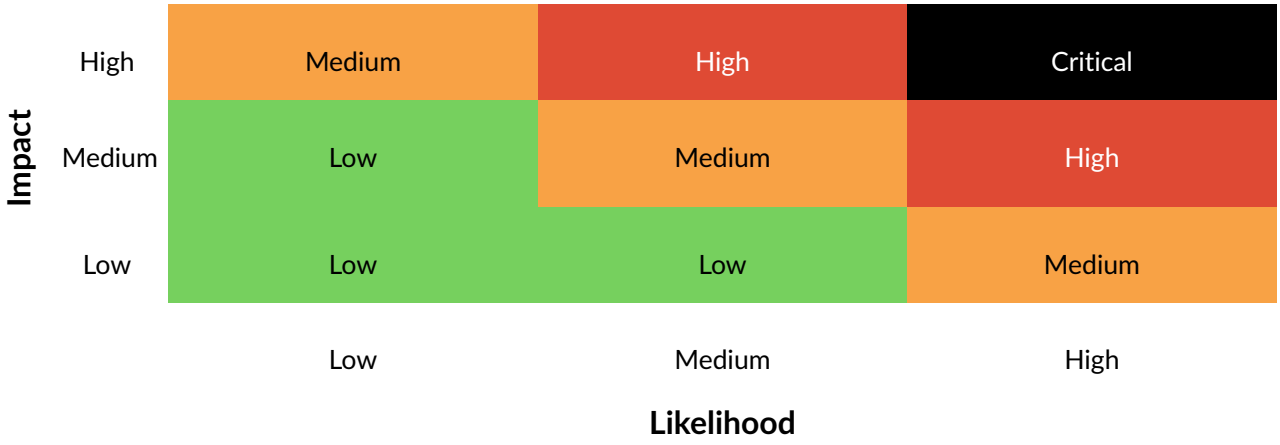

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

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 - Reentrancy. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html#reentrancy. [Accessed 2018].