OPEN GRANTS

Open Grants

Smart Contract Security Assessment

Version: 2.0

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the `Open Grants` smart contracts. 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 `Open Grants` smart contracts 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 `Open Grants` smart contracts.

Overview

The `Open Grants` smart contracts specify a standard for users to propose, fund, and manage the distribution of grants. It allows for `grantees` to be listed and receive payments through a single smart contract (`UnmanagedStream.sol`). The owner of the contract is able to specify the mechanism of funding, whether this be through some “raise all or none” method or through a typical vesting scheme as implemented in `EtherVesting.sol`

These smart contracts would enable the greater Ethereum community to participate in projects through voting and the awarding of funds.

The `Open Grants` smart contracts have the following features:

- `EtherVesting.sol`:
  - The contract correctly releases funds based on the vesting schedule and is not easily manipulated by miners and other adversarials for a sufficiently long vesting duration.

- `UnmanagedStream.sol`:
  - The contract distributes funds evenly, where the last `grantee` receives as most `n - 1` additional wei due to rounding for `n grantees`.
  - There is a safe maximum of 40 `grantees` that the contract can handle when deploying and splitting payments.

Security Assessment Summary

This review was conducted on the files `UnmanagedStream.sol` and `EtherVesting.sol`, which have SHA256 signatures `3efd75892a61a11aacb98027c06ca7c8ae19ecceaf211d810301e8a045ba497a` and `c0694eebbf0925c31430933a1b0d20b170fff9d2153992bceeaca0c3d43ebc08` respectively.

These files were hosted on the open-grants repository and were assed at commit dab1878.

The complete list of contracts contained in `UnmanagedStream.sol` and `EtherVesting.sol` are as follows:

```
└── UnmanagedStream.sol
    ├── ReentrancyGuard
    ├── BaseGrant
    ├── GranteeConstructor
    └── Funding

└── EtherVesting.sol
    ├── Ownable
    ├── ReentrancyGuard
    └── Funding
```

Note: the OpenZeppelin and ABDK libraries and dependencies were excluded from the scope of this assessment.

This security assessment targeted exclusively the `UnmanagedStream` and `EtherVesting` contracts.

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 six (6) issues during this assessment, of which:

- One (1) is classified as low risk,
- Five (5) are classified as informational.

Retesting activities targeted commit (0098749).

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 `Open Grants` 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
OG-01	Grantee DoS By Reverting	Low	Open
OG-02	`EtherVesting` Start May Be Arbitrarily Set	Informational	Resolved
OG-03	Storage Operations Gas Savings	Informational	Resolved
OG-04	Overlap in the Naming of Events	Informational	Resolved
OG-05	Contracts Do Not Implement Safe Ownership Transfer Pattern	Informational	Open
OG-06	Miscellaneous General Statements	Informational	Resolved

OG-01	Grantee DoS By Reverting		
Asset	`UnmanagedStream.sol` & `EtherVesting.sol`		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The contract `UnmanagedStream` will disperse funds, when they are received, to a list of recipients (i.e. grantees).

The following code shows the transfer of an amount, `eligibilePortion`, to a grantee.

```
(bool success, ) = currentGrantee.call{ value: eligiblePortion}("");
require(
    success,
    "fallback::Transfer Error. Unable to send eligiblePortion to Grantee."
);
```

There is a potential denial of service that can occur if a grantee were to deliberately `revert` or otherwise return `success = false`. If the `currentGrantee.call` were to fail for a single grantee then the entire transaction will be reverted and none of the grantees will receive any funds, effectively rendering the contract locked.

A grantee may trivially create a contract address which has a `receive()` function that always reverts or alternatively reads a mutable state variable that may revert when the owner has set this variable. Thereby, giving them control over when the `UnmanagedStream` may distribute funds.

Note that the grantree would be preventing themself from receiving funds and so this is not an economically viable attack.

The same attack vector exists in `EtherVesting.revoke()`, if the `_beneficiary` were set to be a contract which may arbitrarily revert at the beneficiary's will it could prevent the `revoke()` function from being called.

Recommendations

Consider implementing a two stage withdrawal pattern where `receive()` will update the amount grantees can withdraw then the grantee make a withdrawal transaction from the contract.

Resolution

A fix has been applied in to `EtherVesting` in commit 0098749 which will continue the `revoke()` function even if the transfer to the `_beneficiary` fails. However, this will also transfer any unreleased amount to the owner which should be left in the contract for the `_beneficiary`.

The issue has been left open as no fix has yet been applied to the function `receive()` in `UnmanagedStream`.

OG-02	`EtherVesting` Start May Be Arbitrarily Set
Asset	`EtherVesting.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The contract `EtherVesting` allows the start date to be anytime in the past or future as long as the end date $(start + duration)$ is in the future.

A side effect from allowing the start date to be in the future is that the function `revoke()` is unable to be called until after the start date.

`revoke()` will call the function `_vestedAmount()` which will attempt to do `block.timestamp.sub(_start)`. This will underflow, which reverts when using safe math, if `_start > block.timestamp`. That is it will revert when the current time is before the start date.

Recommendations

Consider modifying the function to return zero if the current time is before the start date to allow the `revoke()` to execute.

Resolution

The function `_vestedAmount()` has been modified to return zero for the case where the present time is before the start time in commit b66a46f.

OG-03	Storage Operations Gas Savings
Asset	`contracts/storage/`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Significant gas savings can be made by converting the storage operations from `external` visibility to `public` visibility, then making internal function calls (i.e. jumps) rather than external function calls.

Executing `this.function()` performs an external function call and costs significantly more gas than an internal function call `function()`. There are multiple external calls made in `UnmanagedStream` such as `this.getGranteeReferenceLength()`.

An anaysis of gas usage when using external function calls vs internal function calls can be seen in the table below.

Number of Grantees	External Call Gas	Internal Call Gas
1	78923	66005
5	175448	132818
10	296108	216338

Note when implementing an interface which declares functions with an `external` visibility, the implementation can use a `public` visibility.

Recommendations

Consider changing the visibility to `public` and making internal function calls rather than external function calls.

Resolution

The functions in question have updated the visibility to `public` and now use internal calls in commit b66a46f.

OG-04	Overlap in the Naming of Events
Asset	`IBaseGrant.sol` & `IFunding.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Both interfaces `IBaseGrant` and `IFunding` have an event with the name `LogFunding`.

The event `LogFunding(address indexed donor, uint256 value)` appears in both files with the same name and inputs.

Recommendations

We recommend changing the name of one or both events to avoid confusion and overlap.

Resolution

The duplicate events have been removed in commit b66a46f.

OG-05	Contracts Do Not Implement Safe Ownership Transfer Pattern
Asset	`Ownable.sol`
Status	Open
Rating	Informational

Description

The current transfer of ownership pattern calls the function `transferOwnership(address newOwner)` which instantly changes the owner to the `newOwner`. This allows the current owner of the contracts to set an arbitrary address (excluding the 0 address).

If the address is entered incorrectly or set to an unowned address, the owner role of the contract is lost forever. Thus, a user would not be able to pass the `onlyOwner` modifier.

Similarly, the function `renounceOwnership()` allows an owner to remove themself as owner and prevent any future owners. Again this will cause any `onlyOwner` modifiers to always fail.

Recommendations

Transferring owner privileges can be mitigated by implementing a `transferOwnership` pattern. This pattern is a two-step process, whereby a new owner address is selected, then the selected address must call a `claimOwnership()` before the owner is changed. This ensures the new owner address is accessible.

OG-06	Miscellaneous General Statements
Asset	`contracts/`
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

- In `GranteeConstructor.sol`, the `require` statement on line [54] cannot be reached. If `currentGrantee == address(0)` then the previous check will fail `currentGrantee > lastAddress`.
- In `EtherVesting.sol` the function `revoke()` may attempt to make a transfer when `refund == 0`. Consider only transfering if `refund > 0`.
- The functions `release()` and `revoke()` in `EtherVesting` are only called externally hence the visibility may be changed from `public` to `external`.
- A gas saving can occur by using `numGrantees` in the `for` loop definition rather than `this.getGranteeReferenceLength`.
- There is inconsistent usage of the `_` prefix. Typically constructor variables are the only variables that have the `_` prefix.

Recommendations

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

Resolution

All comments have been acknowledged or addressed by the development team. The comments were addressed in commit b66a46f.

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_deploy.py::test_deploy                                    PASSED    [8%]
tests/test_ether_vesting.py::test_release                            PASSED    [16%]
tests/test_ether_vesting.py::test_revoke_not_owner                   PASSED    [25%]
tests/test_ether_vesting.py::test_user_cannot_revoke                 PASSED    [33%]
tests/test_ether_vesting.py::test_user_can_revoke                    PASSED    [41%]
tests/test_ether_vesting.py::test_receive_no_value                   PASSED    [50%]
tests/test_unmanaged_stream.py::test_max_grantees                    PASSED    [58%]
tests/test_unmanaged_stream.py::test_max_ether                       PASSED    [66%]
tests/test_unmanaged_stream.py::test_grantee_constructor             PASSED    [75%]
tests/test_unmanaged_stream.py::test_grantee_getters                 PASSED    [83%]
tests/test_unmanaged_stream.py::test_gas_usage_external_calls        PASSED    [91%]
tests/test_unmanaged_stream.py::test_receive_no_value                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.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

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].

σ'