Protector Guild

—

Namespace Security Review

—

Security Researchers
33audits, Solidity Security Researcher
bLnk, Solidity Security Researcher
10ambear, Solidity Security Researcher
maxgrok, Solidity Security Researcher

Report prepared by: Max Goodman
April 12th, 2024

Contents

1 About Protector Guild

Protector Guild is a cybersecurity firm that specializes in solidity smart contract security reviews. Learn more about us at protectorguild.co

2 Introduction

Namespace is a secondary market for ENS subdomain names.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Namespace, according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

3.1 Impact

- **Critical** - leads to a loss of a significant portion (>50%) of assets in the protocol, or significant harm to all users.
- **High** - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- **Medium** - global losses <10% or losses to only a subset of users, but still unacceptable.
- **Low** - losses will be annoying but bearable–applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Action required for severity levels

- **Critical** - Must fix as soon as possible (if already deployed)
- **High** - Must fix (before deployment if not already deployed)
- **Medium** - Should fix
- **Low** - Could fix

4 Executive Summary

Over the course of 5 days in total, Namespace Tech DOO engaged with Protector Guild to review Namespace. In this period of time, a total of 4 issues were found.

Summary

Project Name	Namespace Tech DOO
Repository	thenamespace/namespace-contractsv2
Commit	dfef130f
Type of Project	Solidity Smart Contracts, ENS
Time Allotted for Audit	April 1st - April 5th
Methods	Manual Review, Foundry

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	3
Gas Optimizations	0
Informational	0
Total Issues	**4**

5 Findings

5.1 Medium Risk

5.1.1 transferFunds function doesn't refund excessive msg.value

Description

If a user sends excess ETH in order to mint a subname, the excess funds could get stuck in the contract due to the following:

- The require statement in the `_transferFunds` function: `require(msg.value >= mintFee + mintPrice, "Insufficient funds");` (NamespaceMinting.sol#L163). This does not require that the msg.value sent is the exact price, merely that it is greater than the mintFee and the mintPrice.
- The transfers in the `_transferFunds` function `payable(paymentReceiver).transfer(mintPrice);` (NamespaceMinting.sol#L165) & `payable(treasury).transfer(mintFee);` (NamespaceMinting.sol#L166) do not factor in the potential for extra ETH stored in the contract as a result of this `_transferFunds` function.

The protocol does not include a mechanism to handle excess native ETH sent by the user other than manually withdrawing the funds to the treasury, then sending it individually to the user who overpaid. The current mechanism could incur additional administrative work and cost to refund excess ETH to users as they cannot recover the funds themselves. In addition, this harms the user experience of the overall protocol as well as could harm the reputation of the protocol itself if there is not a mechanism in place for refunding excess ETH sent by users accidentally.

Reference

NamespaceMinting.sol#L163-166

Remediation

Consider adding a mechanism to return any potential extra ETH to the caller within the `_transferFunds` function.

We suggest the following remediation within _transferFunds :

```
  function _transferFunds(address paymentReceiver, uint256
mintPrice, uint256 mintFee) internal {
        require(msg.value >= mintFee + mintPrice, "Insufficient
funds");
++ uint256 amountToRefund = msg.value - mintFee - mintPrice;
        payable(paymentReceiver).transfer(mintPrice);
        payable(treasury).transfer(mintFee);
++      (bool successRefund, ) =
payable(msg.sender).call{value:amountToRefund(“”);
++       require(successRefund, “Refund failed”);
    }
```

Fixed: **Commit 643ab**

5.2 Low Risk

5.2.1 Single Step Ownership

Description

The `setTreasury` function makes use of OpenZeppelin's `onlyOwner` modifier to ensure that only an owner can update the address of the treasury. If, for some

reason, the team switched addresses to one not controlled by Namespace Team, then it would not be possible to recover the ownership of the contracts.

Reference

controllers/Controllable.sol#L4-L6

Remediation

To prevent this from happening, we recommend OpenZeppelin's `Ownable2Step` library, which adds a two-step mechanism for transferring ownership of the contract. This layer of friction assists in preventing common mistakes in ownership transfers.

We recommend implementing a replacement of the OpenZeppelin's Ownable library with an implementation of `Ownable2Step` : (access/Ownable2Step.sol)

Fixed: Commit 643ab

5.2.2 Transfer being used instead of call to transfer ETH

Description

The use of `transfer` function in the `NamespaceMinter` could have unintended consequences in the future. This is due to the function being reliant on a perceived "fixed" gas cost (currently 2300 gas), which has changed in the past.

Due to the immutable nature of smart contracts, dependency on gas costs (an inherently fluid concept), this could lead to Denial of Service (DDoS).

Reference

NamespaceMinting.sol#L165-L166

NamespaceMinting.sol#L175

(Consensys Diligence "Stop Using Solidity's Transfer Now" (2019)

Remediation

Use `call` to transfer native ETH to avoid the aforementioned Denial of Service issue:

```
function _transferFunds(address paymentReceiver, uint256
mintPrice, uint256 mintFee) internal {
    require(msg.value >= mintFee + mintPrice, "Insufficient
funds");
--  payable(paymentReceiver).transfer(mintPrice);
--  payable(treasury).transfer(mintFee);
++  (bool successReceiver, ) =
payable(paymentReceiver).call{value: mintPrice}(“”);
++   require(successReceiver, “Payment receiver call failed”);
++   (bool successTreasury, ) = payable(treasury).call{value:
mintFee}("");
++   require(successTreasury, "Treasury call failed");
}
```

NamespaceMinting.sol#L165-L166

```
function withdraw() external onlyOwner {

     require(address(this).balance > 0, "No funds present.");

--   payable(treasury).transfer(address(this).balance);
++   (bool success, ) = payable(treasury).call(value:
address(this).balance);
++   require(success, "Withdraw failed");
}
```

NamespaceMinting.sol#L175

Fixed: Commit 643ab

5.2.3 Absent address(0) Check

Description

The `setTreasury` function within the NamespaceMinting.sol contract and TreasuryProxy.sol contract does not validate the `_treasury` parameter when a new treasury address is set. If the `_treasury` parameter is set incorrectly by the owner, it could result in lost fees.

Reference

```
   function setTreasury(address payable _treasury) external
onlyOwner {
        treasury = _treasury;
    }
```

contracts/NamespaceMinting.sol#L169-L171

contracts/treasury/TreasuryProxy.sol#L22-L24

Remediation

Just to make sure there is a valid address set to the treasury, we recommend the following remediations within the `setTreasury` functions in both contracts/NamespaceMinting.sol#L169-L171 and contracts/treasury/TreasuryProxy.sol#L22-L24.

```
   function setTreasury(address _treasury) external onlyOwner {
++    if(_treasury == address(0)){revert();}
```

```
        treasury = _treasury;
    }
```

contracts/NamespaceMinting.sol#L169-L171

```
   function setTreasury(address payable _treasury) external
onlyOwner {
++    if(_treasury == payable(address(0))){revert();}
        treasury = _treasury;
    }
```

contracts/treasury/TreasuryProxy.sol#L22-L24

Fixed: **Commit 643ab**