Protocol Audit Report

Version 1.0

Ardeshir Gholami

March 7, 2024

Protocol Audit Report

Ardeshir Gholami

7 march 2024

Prepared by: Ardeshir Lead Auditors: - Ardeshir Gholami :)

Table of Contents

    * [L-1] Returning Total Supply Instead of Request ID in `snek_raffle::request_raffle_winner` May Cause confusion
    * [L-2] Missing Event Emission for Requested Raffle Winner in `snek_raffle::request_raffle_winner`
  - Informational
    * [I-1] Unused Error Definition `snek_raffle::ERROR_TRANSFER_FAILED` for Transfer Failure

Protocol Summary

PasswordStore is a test protocole which saves a password and lets the only owner read it anytime. # Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash: https://github.com/Cyfrin/2024-03-snek-raffle

```
## Scope
```

–| contracts |– snek_raffle.vy

~~~
## Roles

# Executive Summary

## Issues found

| Severity | number of issues found |
| -------- | ---------------------- |
| High     | 3                      |
| Medium   | 0                      |
| Low      | 2                      |
| Info     | 1                      |
| Total    | 6                      |

# Findings
## High
### [H-1] Incorrect Token URI for Legendery Snek

**Description:**
The smart contract assigns a token URI to each minted token, which is intended to point to a JSON file containing metadata for the token. However, for tokens of legendary snek rarity, the token URI is incorrectly set to point to an image file of rare snek instead of a JSON file.

**Impact:**
 This will lead to minting a wrong NFT for the legendary snek.



**Proof of Concept:**

Upon opening the `RARE_SNEK_URI` IPFS link, a JSON file is retrieved that contains metadata for a rare snek token, including its name, description, image, and attributes. The JSON file is as follows:
```json
{
    "name": "Jungle Snek",
    "description": "An adorable rare jungle snek!",
    "image": "ipfs://QmRujARrkux8nsUG8BzXJa8TiDyz5sDJnVKDqrk3LLsKLX",
    "attributes": [
        {
            "trait_type": "cuteness",
            "value": 100
        }
    ]
~~~

```
}
```

However, the `LEGEND_SNEK_URI` IPFS link also points to the `"image"` ipfs of the `RARE_SNEK_URI`, indicating that the wrong content is being used for legendary snek tokens. This discrepancy is evident in the following code snippet:

```
COMMON_SNEK_URI: public(constant(String[53])) = "ipfs://
    QmSQcYNrMGo5ZuGm1PqYtktvg1tWKGR7PJ9hQosKqMz2nD"
RARE_SNEK_URI: public(constant(String[53])) = "ipfs://
    QmZit9nbdhJsRTt3JBQN458dfZ1i6LR3iPGxGQwq34Li4a"
LEGEND_SNEK_URI: public(constant(String[53])) = "ipfs://
    QmRujARrkux8nsUG8BzXJa8TiDyz5sDJnVKDqrk3LLsKLX"
```

Recommended Mitigation: To resolve this issue, the contract should ensure that each type of snek token has a unique IPFS URI that points to its specific metadata and image. This can be done by updating the `LEGEND_SNEK_URI` constant to point to the correct JSON file for legendary snek tokens.

[H-2] Incorrect Function Selector for Chainlink VRF Coordinator in `snek_raffle::rawFulfillRandomWords`

Description: The `rawFulfillRandomWords` function is designed to be called back by the Chainlink VRF Coordinator to provide the random words. However, the function signature does not match the expected signature for the Chainlink VRF Coordinator's callback function. The Chainlink VRF Coordinator expects a function with the signature `rawFulfillRandomWords(uint256,uint256[])`, but the provided function signature is `rawFulfillRandomWords(uint256,uint256[1])`. This will work while using the VRFCoordinatorMock contract but when working with the real implementation this wont work.

Impact: This issue will cause the Chainlink VRF Coordinator's callback not to go through, and we would never get a winner. Because the rawFulfillRandomWords function, which chooses the winner, is never called by the VRF Coordinator contract. This failure to call back to the rawFulfillRandomWords function is a critical issue that prevents the raffle from being completed, as the winner selection process is central to the functionality of the raffle. Additionally, because of this, the raffle will always stay in calculating mode and will be unusable. This situation significantly impacts the usability and functionality of the raffle, making it impossible for participants to engage with the raffle process as intended.

Proof of Concept: 1. Running this command in Foundry-chisel we can calculate the function Selector for `rawFulfillRandomWords(uint256, uint256[])` which is the correct signature:

```
-> bytes4(keccak256("rawFulfillRandomWords(uint256,uint256[])"))
Type: bytes4
| Data: 0
    x1fe543e300000000000000000000000000000000000000000000000000000000
```

2. But When running the signature which is the equivalent of the `snek_raffle::rawFulfillRandomWords` which is `rawFulfillRandomWords(uint256,uint256[1])` we get:

```
-> bytes4(keccak256("rawFulfillRandomWords(uint256,uint256[1])"))
Type: bytes4
| Data: 0
    x4e04d70800000000000000000000000000000000000000000000000000000000
```

3. It is obvious that the `0x1fe543e3` correct selector is diffrent with the one we are sending `0x4e04d708`

Recommended Mitigation: When deploying to Mainnet we change the function defenition to correct function signature:

```
@external
-def rawFulfillRandomWords(requestId: uint256, randomWords: uint256[
    MAX_ARRAY_SIZE]):
+def rawFulfillRandomWords(requestId: uint256, randomWords: DynArray[
    uint256, MAX_ARRAY_SIZE]):
     """The function the VRF Coordinator calls back to to provide the
        random words."""
     assert msg.sender == VRF_COORDINATOR.address, ERROR_NOT_COORDINATOR
     self.fulfillRandomWords(requestId, randomWords)
```

The `snek_raffle::fulfillRandomWords` function should also be changed as below:

```
@internal
-def fulfillRandomWords(request_id: uint256, random_words: uint256[
    MAX_ARRAY_SIZE]):
+def fulfillRandomWords(request_id: uint256, _random_words: DynArray[
    uint256, MAX_ARRAY_SIZE]):
.
.
.
```

[H-3] Rarity for Every Snek is Set to 33 Instead of the Suggested Rarity, Causing the Rare and Legendary NFTs to be as Common as Common Ones

Description: The rarity of the NFTs is set using the random word in the `snek_raffle::fulfillRandomWords` function, but the value is set by modding the random word by 3 instead of 100. This results in a rarity distribution that does not reflect the intended rarity levels, making rare and legendary NFTs as common as common ones.

```
@internal
```

```
def fulfillRandomWords(request_id: uint256, random_words: uint256[
    MAX_ARRAY_SIZE]):
    index_of_winner: uint256 = random_words[0] % len(self.players)
    recent_winner: address = self.players[index_of_winner]
    self.recent_winner = recent_winner
    self.players = []
    self.raffle_state = RaffleState.OPEN
    self.last_timestamp = block.timestamp
@>  rarity: uint256 = random_words[0] % 3
    self.tokenIdToRarity[ERC721._total_supply()] = rarity
    log WinnerPicked(recent_winner)
    ERC721._mint(recent_winner, ERC721._total_supply())
    send(recent_winner, self.balance)
```

Impact: This issue significantly undermines the rarity factor, making the NFTs less valuable and less desirable to collectors.

Proof of Concept: First to proove this concept we must change `chainlinkvrfMock.vy` to actually produce a random number: 1. Change the `chainlinkvrfMock.vy` so it returns the block.timestamp as random number

```
@external
def fulfillRandomWords(requestId: uint256, consumer: address):
    """Returns an array of random numbers. In this mock contract, we
        ignore the requestId and consumer.

    Args:
        requestId (uint256): The request Id number
        consumer (address): The consumer address to
    """
    # Default to 77 as a mocking example
-    words: uint256[MAX_ARRAY_SIZE] = [77]
+    words: uint256[MAX_ARRAY_SIZE] = [block.timestamp]
    self.fulfillRandomWordsWithOverride(requestId, consumer, words)
```

2. Add the following Script to the test suit `snek_raffle_test.py`: Note that the random library is used here to move the time so the vrfcoordintorMock actually produce a randomNumber:

```
import random
from collections import Counter

def test_rarity(
    raffle_boa_entered, vrf_coordinator_boa, entrance_fee
):
    rarities = []

    for j in range(100):
        additional_entrants = 10
        for i in range(additional_entrants):
            player = boa.env.generate_address(i)
            boa.env.set_balance(player, STARTING_BALANCE)
            with boa.env.prank(player):
                raffle_boa_entered.enter_raffle(value=entrance_fee)
        starting_balance = boa.env.get_balance(USER)
        rand = int(random.randrange(1,100))
        boa.env.time_travel(seconds=INTERVAL +1+rand)

        raffle_boa_entered.request_raffle_winner()

        # Normally we need to get the requestID, but our mock ignores
            that
        vrf_coordinator_boa.fulfillRandomWords(0, raffle_boa_entered.
            address)

        recent_winner = raffle_boa_entered.get_recent_winner()
        winner_balance = boa.env.get_balance(recent_winner)
        prize = entrance_fee * (additional_entrants + 1)

        rarities.append(raffle_boa_entered.get_token_Rarity(j))
    item_counts = Counter(rarities)
    print(item_counts)
    assert(rarities.count(2) >= 5)
```

3. The printed output will be something like this:

```
tests/snek_raffle_test.py ..Counter({2: 47, 0: 30, 1: 23})
```

which indicates that more than 5 percent of the NFTs are actully rare.

Recommended Mitigation: To address this issue, the rarity determination should be adjusted to reflect a more balanced distribution. This can be achieved by modding the random word by 100 instead of 3, allowing for a more granular distribution of rarities. Additionally, the rarity distribution should be clearly defined and implemented in the tokenURI function to ensure that each rarity category is represented accurately.

Here's how the fulfillRandomWords function can be revised to implement the recommended mitigation:

```
@internal
def fulfillRandomWords(request_id: uint256, random_words: uint256[
    MAX_ARRAY_SIZE]):
    index_of_winner: uint256 = random_words[0] % len(self.players)
    recent_winner: address = self.players[index_of_winner]
    self.recent_winner = recent_winner
    self.players = []
    self.raffle_state = RaffleState.OPEN
    self.last_timestamp = block.timestamp
-   rarity: uint256 = random_words[0] % 3
```

```
+    rarity: uint256 = random_words[0] % 100
     self.tokenIdToRarity[ERC721._total_supply()] = rarity
     log WinnerPicked(recent_winner)
     ERC721._mint(recent_winner, ERC721._total_supply())
     #q misshandling of eth? pull over push
     send(recent_winner, self.balance)
```

And the tokenURI function should be updated to reflect the new rarity distribution:

```
@external
@view
def tokenURI(token_id: uint256) -> String[53]:
    rarity: uint256 = self.tokenIdToRarity[token_id]
+   if rarity < LEGEND_RARITY:
+       return self.rarityToTokenURI[2]
+   elif rarity < RARE_RARITY:
+       return self.rarityToTokenURI[1]
+    else:
+       return self.rarityToTokenURI[0]
-   return self.rarityToTokenURI[rarity]
```

This approach ensures that the rarity distribution is more balanced, with each rarity category having a distinct chance of being selected. This will make the NFTs more valuable and desirable to collectors, addressing the issue of rare and legendary NFTs being as common as common ones.

Low

[L-1] Returning Total Supply Instead of Request ID in `snek_raffle::request_raffle_winner` May Cause confusion

Description: The `request_raffle_winner` function is designed to request a random winner from the VRF Coordinator after a raffle has completed. However, instead of returning the `request_id` obtained from the VRF Coordinator, the function returns the total supply of the ERC721 tokens. This discrepancy could lead to confusion and potential issues in tracking and verifying the request for a random winner.

Impact: Returning the total supply instead of the request_id could have several implications for the raffle process: Verification Difficulty: For participants or external observers, verifying the outcome of the raffle becomes more challenging. They would need to rely on the total supply of tokens to infer the outcome, which is not a reliable or straightforward method. Transparency Issues: This approach could lead to transparency issues, as it does not provide a clear and direct way to track the request for a random winner. This could affect the trust and confidence of participants in the raffle process. Operational Complexity: For the raffle organizers, this could introduce operational complexity. They would need to manage and track the total supply of tokens separately from the actual raffle process, which could increase the administrative overhead.

Proof of Concept:

```
@external
def request_raffle_winner() -> uint256:
    .
    .
    .
    request_id: uint256 = VRF_COORDINATOR.requestRandomWords(
        GAS_LANE,
        SUBSCRIPTION_ID,
        REQUEST_CONFIRMATIONS,
        CALLBACK_GAS_LIMIT,
        NUM_WORDS
    )
@>  return ERC721._total_supply()
```

Recommended Mitigation: To address this issue, the function should return the request_id obtained from the VRF Coordinator. This change ensures that the function's return value accurately reflects the outcome of the request for a random winner, making it easier to track and verify the request. Here's an example of how this could be implemented:

```
@external
def request_raffle_winner() -> uint256:
    .
    .
    .
    request_id: uint256 = VRF_COORDINATOR.requestRandomWords(
        GAS_LANE,
        SUBSCRIPTION_ID,
        REQUEST_CONFIRMATIONS,
        CALLBACK_GAS_LIMIT,
        NUM_WORDS
    )
-   return ERC721._total_supply()
+   return request_id
```

[L-2] Missing Event Emission for Requested Raffle Winner in `snek_raffle::request_raffle_winner`

Description: The `RequestedRaffleWinner` event is defined in the events definition section of the smart contract. However, the `request_raffle_winner` function does not emit any events when the random number is requested.

Impact: The event definition is excessive, and there is no easy access to requestIds. This could lead to difficulties in tracking the request for a random winner, especially for external observers or participants who might be interested in the outcome of the raffle.

Proof of Concept:

```
# Events
@> event RequestedRaffleWinner:
      request_id: indexed(uint256)
    .
    .
    .
@external
def request_raffle_winner() -> uint256:
    """Request a random winner from the VRF Coordinator after a raffle
        has completed."""
    is_open: bool = RaffleState.OPEN == self.raffle_state
    time_passed: bool = (block.timestamp - self.last_timestamp) >
        RAFFLE_DURATION
    has_players: bool = len(self.players) > 0
    has_balance: bool = self.balance > 0
    assert is_open and time_passed and has_players and has_balance,
        ERROR_NOT_ENDED

    self.raffle_state = RaffleState.CALCULATING
    request_id: uint256 = VRF_COORDINATOR.requestRandomWords(
        GAS_LANE,
        SUBSCRIPTION_ID,
        REQUEST_CONFIRMATIONS,
        CALLBACK_GAS_LIMIT,
        NUM_WORDS
    )
@>  #@audit No event emission here!
    return ERC721._total_supply()
```

Recommended Mitigation: To address this issue, the request_raffle_winner function should emit the RequestedRaffleWinner event to signal that a raffle winner has been requested. This change ensures that the function's actions are clearly communicated and can be easily tracked. Here's an example of how this could be implemented:

```
@external
def request_raffle_winner() -> uint256:
    """Request a random winner from the VRF Coordinator after a raffle
        has completed."""
    is_open: bool = RaffleState.OPEN == self.raffle_state
    time_passed: bool = (block.timestamp - self.last_timestamp) >
        RAFFLE_DURATION
    has_players: bool = len(self.players) > 0
    has_balance: bool = self.balance > 0
    assert is_open and time_passed and has_players and has_balance,
        ERROR_NOT_ENDED
    self.raffle_state = RaffleState.CALCULATING
    request_id: uint256 = VRF_COORDINATOR.requestRandomWords(
        GAS_LANE,
        SUBSCRIPTION_ID,
        REQUEST_CONFIRMATIONS,
        CALLBACK_GAS_LIMIT,
        NUM_WORDS
    )
+   log RequestedRaffleWinner(request_id)
    return ERC721._total_supply()
```

Informational

[I-1] Unused Error Definition `snek_raffle::ERROR_TRANSFER_FAILED` for Transfer Failure

Description: The smart contract defines an error `ERROR_TRANSFER_FAILED` with the message "SnekRaffle: Transfer failed". This error is intended to be used when an Ether transfer fails. However, the error is not used anywhere in the contract, and must be removed.

Impact: The absence of error handling for failed Ether transfers could lead to silent failures, making it difficult to diagnose and resolve issues related to Ether transfers. This could impact the integrity of the raffle process, as participants might not receive their rewards if the transfer fails.

Proof of Concept: The `send` function is used to transfer Ether to the `recent_winner`. While the function is used, there is no explicit error handling for the case where the transfer fails. The defined error `ERROR_TRANSFER_FAILED` is not utilized in the contract.

Recommended Mitigation: Given that the send function in Vyper does not return any value for success or failure and just reverts in case of failure, it's best to remove the defined error ERROR_TRANSFER_FAILED. This simplifies the contract's logic and relies on the inherent safety features of the send function.

```
ERROR_NOT_ENDED: constant(String[25]) = "SnekRaffle: Not ended"
-ERROR_TRANSFER_FAILED: constant(String[100]) = "SnekRaffle: Transfer
    failed"
ERROR_SEND_MORE_TO_ENTER_RAFFLE: constant(String[100]) = "SnekRaffle:
    Send more to enter raffle"
ERROR_RAFFLE_NOT_OPEN: constant(String[100]) = "SnekRaffle: Raffle not
    open"
ERROR_NOT_COORDINATOR: constant(String[46]) = "SnekRaffle:
    OnlyCoordinatorCanFulfill"
```