our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Vallarok

SECURITY REVIEW

Date: 24 April 2024

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach working with the best talents in the space.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Vallarok

Vallarock is an AI-powered survival game in a post-apocalyptic Viking open world. The game utilises NFT functionality that allows character upgrading.

Vallarock's `RunaAGI` contract is the gateway to the entire game functionality, as it tackles the minting of the NFTs.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 1 day with a total of 12 hours dedicated to the audit by the core Shieldify team.

The codebase does not contain testing of any kind. Shieldify's security review identified problems of various severity, primarily about signature malleability, as well as replay and reentrancy attacks, among other less severe vulnerabilities. There are no tests.

5.1 Protocol Summary

Project Name	**Vallarok**
Repository	vallarok
Type of Project	ERC721C collection
Audit Timeline	1 day
Review Commit Hash	23066dc1c56a59cf84e971b678aa7b5da98be0ab
Fixes Review Commit Hash	b737b62ac5c72060442cb81ddc06ddcc22f04a76

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
contracts/RunaAGI.sol	187
Total	**187**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **2**
- **Medium** issues: **3**
- **Low** issues: **1**

ID	Title	Severity	Status
[H-01]	Malicious User Can Mint Unlimited Amount of NFTs Due to Reentrancy in `whitelistMint()` and ‘FCFSMint()‘	High	Fixed
[H-02]	`FCFSMint()` Can Be Called By Anyone, Not Only By Whitelisted Users	High	Acknowledged
[M-01]	Signature Malleability in `RunaAGI.sol`	Medium	Fixed
[M-02]	When The Owner Uses `mintToWallet()`, `saleSupplyMinted` Is Not Incremented	Medium	Fixed
[M-03]	Large Centralization Risk	Medium	Acknowledged
[L-01]	`call()` Should Be Used Instead of `transfer()` on an `address payable`	Low	Fixed

7. Findings

[H-01] Malicious User Can Mint Unlimited Amount of NFTs Due to Reentrancy in `whitelistMint()` and `FCFSMint()`

Severity

High Risk

Description

The `whitelistMint()` and `FCFSMint()` functions are vulnerable to reentrancy. We can see that the function fails to apply a reentrancy modifier.

When a malicious user calls them, they can do a reentrancy attack via the `_safeMint()` function that is called. We can see that absolutely all variables are updated afterwards. Thus, a malicious user can mint as many NFTs as he wants.

Location of Affected Code

File: contracts/RunaAGI.sol

```
function whitelistMint(uint256 _salt, bytes32 _msgHash, bytes memory
    _signature) external payable {
// code
     _safeMint(msg.sender, _tokenIdCounter.current());
     _tokenIdCounter.increment();
     _totalSupply++;
     saleSupplyMinted++;
}
```

```
function FCFSMint(uint256 _salt, bytes32 _msgHash, bytes memory
    _signature) external payable {
// code
     _safeMint(msg.sender, _tokenIdCounter.current());
     _tokenIdCounter.increment();
     _totalSupply++;
     saleSupplyMinted++;
     fcfsMinted[msg.sender] += 1;
}
```

Recommendation

Consider adding a reentrancy modifier on `whitelistMint()` and `FCFSMint()`.

Team Response

Fixed as suggested.

[H-02] `FCFSMint()` Can Be Called By Anyone, Not Only By Whitelisted Users

Severity

High Risk

Description

From the ReadMe, we can see that the `FCFSMint()` function should be called only by whitelisted users:

> Ability to participate in the FCFS phase where every whitelisted wallet can mint two additional NFTs

But as we can see in the functions thus verification is completely missing.

Location of Affected Code

File: contracts/RunaAGI.sol#L135

```
function FCFSMint(uint256 _salt, bytes32 _msgHash, bytes memory
    _signature) external payable {
      require(isFCFSMintingActive, "FCFS minting is not active");
      require(saleSupplyMinted + 1 <= saleSupply, "Maximum supply reached")
         ;
      require(msg.value == salePrice, "Incorrect ether amount");
      require(fcfsMinted[msg.sender] + 1 <= 2, "Limit reached");

      bytes32 msgHash = keccak256(abi.encodePacked(msg.sender, _salt));
      bytes32 signedMsgHash = msgHash.toEthSignedMessageHash();

      require(signedMsgHash == _msgHash, "Invalid message hash!");
      require(_msgHash.recover(_signature) == signer, "Invalid signer!");
      signatures[_signature] = true;

      _safeMint(msg.sender, _tokenIdCounter.current());
      _tokenIdCounter.increment();
      _totalSupply++;
      saleSupplyMinted++;
      fcfsMinted[msg.sender] += 1;
}
```

Recommendation

Consider adding a check if the user calling the function is whitelisted.

Team Response

Acknowledged.

[M-01] Signature Malleability in `RunaAGI.sol`

Severity

Medium Risk

Description

In `RunaAGI.sol` the functions `whitelistMint()`, `claimPreSale()` and `FCFSMint()` used `ECDSA.recover` from OpenZeppelin.

We can see from `package.json` that the version used is `4.6.0`.

The functions `ECDSA.recover` and `ECDSA.tryRecover` are vulnerable to a kind of signature malleability due to accepting EIP-2098 compact signatures in addition to the traditional 65-byte signature format. This is only an issue for the functions that take a single `bytes` argument, and not the functions that take `r`, `v`, `s` or `r, v,s` as separate arguments. Reference

The functions `whitelistMint()`, `claimPreSale()` and `FCFSMint()` used `_salt` variable. This variable plays an essential role in the cryptographic scheme, primarily to ensure that each signature is unique, even if the same user requests it multiple times.

The `_salt` makes it harder for attackers to generate valid signatures unless they know the exact parameters, including the salt, that were used to create the signature. If a malicious actor sees the `_salt` in the Mempool, they could attempt to submit a similar transaction with the same parameters but a higher gas price to have it processed before the original.

Impact

The potentially affected contracts are those that implement signature reuse or replay protection by marking the signature itself as used rather than the signed message or a nonce included in it. A user may take a signature that has already been submitted, submit it again in a different form, and bypass this protection.

Location of Affected Code

File: contracts/RunaAGI.sol

```
require(_msgHash.recover(_signature) == signer, "Invalid signer!");
// code
require(_msgHash.recover(_signature) == signer, "Invalid signer!");
// code
require(_msgHash.recover(_signature) == signer, "Invalid signer!");
```

Recommendation

Upgrade `@openzeppelin/contracts` to the latest version or a version that is newer or at least 4.7.3.

Team Response

Fixed as suggested.

[M-02] When The Owner Uses `mintToWallet()`, `saleSupplyMinted` Is Not Incremented

Severity

Medium Risk

Description

The `mintToWallet()` function in `RunaAGI.sol` is used only by the owner to mint NFT to the wallet but in the for loop where NFTs are minted, it is forgotten to add a `saleSupplyMinted` update.

The function increases `_totalSupply` each time a token is minted but does not adjust the `saleSupplyMinted` counter. This counter tracks how many tokens have been sold or allocated during specific sale events.

Additionally, because of this, the admin can accidentally set the sale supply lower than the minted supply.

Impact

By not increasing `saleSupplyMinted` when minting tokens through `mintToWallet`, the contract might inaccurately represent how many tokens are left for sale under the sale-specific conditions.

Location of Affected Code

File: contracts/RunaAGI.sol#L159

```
function mintToWallet(address _to, uint256 _amount) external onlyOwner {
    require(_totalSupply + _amount <= maxSupply, "Exceeds maximum supply"
       );
    for (uint256 i = 0; i < _amount; i++) {
        _safeMint(_to, _tokenIdCounter.current());
        _tokenIdCounter.increment();
        _totalSupply++;
    }
}
```

Recommendation

Consider including a `saleSupplyMinted` increase by 1 within the loop to ensure that each minting operation reflects on the sale supply counter.

Team Response

Fixed as suggested.

[M-03] Large Centralization Risk

Severity

Medium Risk

Description

The `withdrawFunds()` Function within the RunaAGI.sol file poses a significant centralization risk. This function permits the contract owner to withdraw the entire balance of funds to any specified address. This capability grants the owner excessive control over the contract's funds, potentially leading to misuse or exploitation.

Location of Affected Code

File: contracts/RunaAGI.sol

```
function withdrawFunds(address payable _to) external onlyOwner {
    uint256 balance = address(this).balance;
    _to.transfer(balance);
    emit FundsWithdrawn(_to, balance);
}
```

Recommendation

Consider implementing a Timelock mechanism. This mechanism introduces a delay between the initiation of a withdrawal request and its execution, providing an opportunity for stakeholders to review and potentially intervene in case of any suspicious or unauthorized withdrawal attempts.

Team Response

Acknowledged.

[L-01] `call()` Should Be Used Instead of `transfer()` on an `address payable`

Severity

Low Risk

Description

The `transfer()` and `send()` functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase in the SLOAD instruction.

Location of Affected Code

File: contracts/RunaAGI.sol#L190

```
function withdrawFunds(address payable _to) external onlyOwner {
    uint256 balance = address(this).balance;
    _to.transfer(balance);
    emit FundsWithdrawn(_to, balance);
}
```

Scenario

Additionally, the use of the deprecated `transfer()` function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a payable function.
- The claimer smart contract does implement a payable fallback which uses more than 2300 gas units.
- The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through a proxy, raising the call's gas usage above 2300.
- Additionally, using more than 2300 gas might be mandatory for some multisig wallets.

Recommendation

Use `call()` instead of `transfer()`, but be sure to respect the CEI pattern and/or add re-entrance guards, as several hacks already happened in the past due to this recommendation not being fully understood.

```diff
function withdrawFunds(address payable _to) external onlyOwner {
    uint256 balance = address(this).balance;
-   _to.transfer(balance);
+   (bool success,) = payable(_to).call{value: balance}("");
+   require(success, "Transfer failed.");
    emit FundsWithdrawn(_to, balance);
}
```

Team Response

Fixed as suggested.

our shielding · Your smart contracts, our shielding · Your smart

shieldify

Thank you!