Larva Labs Meebits

2021-06-30

Overview

About C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the code contest outlined in this document, C4 conducted an analysis of Larva Labs' Meebits smart contract system written in Solidity. The code contest took place between April 29 and May 1, 2021.

Wardens

9 Wardens contributed reports to the Meebits code contest:

- 0xRajeev
- 0xsomeone
- Thunder
- s1m0
- jvaqa
- Jmukesh
- GalloDaSballo
- cmichel
- gpersoon

This contest was judged by Joseph Delong.

Final report assembled by ninek and moneylegobatman.

Summary

The C4 analysis yielded an aggregated total of 15 unique vulnerabilities. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity, 3 received a risk rating in the category of MEDIUM severity, and 6 received a risk rating in the category of LOW severity.

C4 analysis also identified 8 non-critical recommendations.

Scope

The code under review can be found within the C4 code contest repository and comprises 1 smart contract written in the Solidity programming language.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings

[H-00] ERC-721 Enumerable Spec mismatch for index of `tokenByIndex()` function

Index starts at 0 for token array, but the implementation here requires index to be greater than 0. This will prevent querying of tokens at index 0.

See reference implementation.

This will impact compatibility with NFT platforms that expect full conformity with ERC-721 specification.

Recommend accepting 0 index by changing to `require(index >= 0 && index < TOKEN_LIMIT);`.

dangerousfood (Meebits) commented: > Beebots indexes by 1 for whatever reason

[H-01] Signature malleability of EVM's `ecrecover` in `verify()`

EVM's `ecrecover` is susceptible to signature malleability, which allows replay attacks, but that is mitigated here by tracking accepted offers and canceling them (on L645) specifically to prevent replays. However, if any application logic changes, it might make signature malleability a risk for replay attacks.

See reference.

Recommend using OpenZeppelin's ECDSA library

[H-02] Arbitrary Transfer of Unowned NFTs

Due to how the market functions are structured, it is possible to arbitrarily transfer any NFT that is not owned by any address.

The function in question is the `tradeValid` function invoked by `acceptTrade` before the trade is performed. It, in turn, validates the signature of a trade via `verify`, which does not account for the behavior of `ecrecover`.

When `ecrecover` is invoked with an invalid signature, the zero-address is returned by it, meaning that `verify` will yield `true` for the zero-address as long as the signature provided is invalid.

This can be exploited to transfer any NFT whose `idToOwner` is zero, including NFTs that have not been minted yet.

Recommend an additional check be imposed within `verify` that ensures the signer is not the zero-address which will alleviate this check. For more details, consult the EIP721 implementation by OpenZeppelin.

[H-03] `Beebots.TradeValid()` Will Erroneously Return True When Maker Is Set To `Address(0)` and `makerIds` Are Set To The `TokenIds` of Unminted Beebot NFTs

`Beebots.TradeValid()` will erroneously return true when `maker` is set to `address(0)` and `makerIds` are set to the `tokenIds` of unminted beebot NFTs.

`Beebots.verify()` returns true no matter what signature is given when signer is set to `address(0)`. This means that `BeeBots.tradeValid()` will erroneously return true when `maker` is set to `address(0)`.

Finally, before an NFT has even been minted at all, it is assumed to have an owner of `address(0)` due to the `idToOwner` mapping being initialized to zero for all uninitialized slots, so an attacker can call `tradeValid()` with `maker` set to `address(0)` and `makerIds` set to the `tokenIds` of any unminted `nftIds`, and `tradeValid()` will erroneously return true.

- (1) `Beebots.verify()` returns true no matter what signature is given when signer is set to `address(0)`.
  - (1a) `BeeBots.verify()` does not check to ensure that signer is not `address(0)`.
  - (1b) This is a problem because `ecrecover` fails silently if the signature does not match and returns zero.
  - (1c) So if an attacker passes in `address(0)` as the signer, then verify will return true no matter what signature is provided, since `ecrecover` will return `address(0)`, and the signer is `address(0)`, so verify will pass.
  - (1d) This means that `BeeBots.tradeValid()` will erroneously return true when maker is set to `address(0)`.
- (2) Before an NFT has even been minted at all, it is assumed to have an owner of `address(0)` due to the `idToOwner` mapping being initialized to zero for all uninitialized slots
  - (2a) Solidity initializes all mappings to 0 for all slots that have not yet been set.
  - (2b) So for any NFT ID that has not yet been minted, the corresponding owner in the mapping `BeeBots.idToOwner` is `address(0)`, even though that NFT should not even exist.
  - (2c) This means that an attacker can call `tradeValid()` with maker set to `address(0)` and makerIds set to any unminted nftIds, and `tradeValid()` will erroneously return true.

(1) Recommend adding this check to `Beebots.verify()`: `require(signer != address(0), "Cannot verify signatures from 0x0");`

(2) Recommend adding this check to `Beebots.tradeValid()`: `require(maker != address(0), "Maker 0x0 not allowed");`

dangerousfood (Meebits) commented: > Wow, this exploit is absolutely stunning.

[H-04] function `tokenByIndex` treats last index as invalid

NFT indexes start from 0:

```
// Don't allow a zero index, start counting at 1
return value.add(1);
```

So if there are 30 tokens, indexes would be 1-30. However, function `tokenByIndex` sets such boundaries:

```
require(index > 0 && index < TOKEN_LIMIT);
```

This means that the last token (with index 30 in this case) will not be valid.

Recommend using:

```
require(index > 0 && index <= TOKEN_LIMIT);
```

dangerousfood (Meebits) commented: > Beebots is indexing by 1

[H-05] NFT can be minted for free after sale ended

The `getPrice()` function returned 0 after the sale ended and (`SALE_LIMIT - numSales`) NFT can be minted for free.

Without documentation, it's not clear if this is the expected behavior or not. If it's unexpected, it is recommended to revert instead of returning 0. If it's expected behavior, it's possible to create a smart contract and claim all the remaining NFT front-running the regular users.

Medium Risk Findings

[M-00] Legacy Function Usage

The `withdraw` function utilizes the `transfer` invocation, which has a fixed gas stipend and can fail, especially beyond the Berlin fork, which increased the gas costs for first-time invocations of a transfer.

The EIP should be sufficient.

Recommend using a safe wrapper library, such as the OpenZeppelin `Address` library's `sendValue` function, which forwards sufficient gas for the transfer regardless of the underlying OPCODE gas costs.

[M-01] `randomIndex` is not truly random - possibility of predictably minting a specific token Id

'randomIndex' is not random. Any miner has access to these values:

```
uint index = uint(keccak256(abi.encodePacked(nonce, msg.sender, block.difficulty, block.time
```

Non-miner attackers could also test the minting random condition until they get the ID they are looking to access.

The internal variable `indices` seems to be used to avoid this type of collision.

While this makes it less straightforward, there is still the possibility of minting a token with a specific ID.

That said, `_addNFToken` is checking if the token is already owned by an address, ensuring a token can't be stolen.

Refactoring as suggested below will save gas, make code easier to read and prevent reverts in rare unfortunate occasions of clashes.

Recommend not generating random IDs and instead using counters. It makes the code more predictable and easier to read, avoids clashing of IDs, and reduces the need to track minted tokens.

[M-02] instead of `call()` , `transfer()` is used to withdraw the ether

```
function withdraw(uint amount) external {
  require(amount <= ethBalance[msg.sender]);
  ethBalance[msg.sender] = ethBalance[msg.sender].sub(amount);
  msg.sender.transfer(amount);
  emit Withdraw(msg.sender, amount);
}
```

To withdraw ETH, it uses `transfer()`, this transaction will fail inevitably when:

1. The withdrawer smart contract does not implement a payable function.
2. Withdrawer smart contract does implement a payable fallback which uses more than 2300 gas unit.
3. The withdrawer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

Recommend using `call()` to send ETH.

Low Risk Findings

[L-00] Atypical contract structure affects maintainability and readability

A typical/recommended contract structure has the variable declarations followed by events instead of the other way around. This affects readability/maintainability and may introduce/persist security issues.

Recommend considering restructuring the contract to place the variable declarations before events.

[L-01] Mint can be front-run

The price of an NFT falls over time which creates a dynamic that one potential buyer wants to wait for the price to drop but not wait too long to avoid hitting the max sale cap.

However, any such `mint` call can be observed on public blockchains, and attackers can wait until another person decides to buy at the current price and then frontrun that person.

Legitimate minters can be frontrun and end up with a failed transaction and without the NFT as the max sale limit is reached: `require(numSales < SALE_LIMIT, "Sale limit reached.");`.

Front-running is hard to prevent; maybe an auction-style minting process could work where the top `SALE_LIMIT` bids are accepted after the sale duration.

[L-01] Missing parameters in `SalesBegin` event of critical `startSale()` function

Consider including `salesStartTime` and `salesDuration` as parameters in the `SaleBegins` event to allow off-chain tools to track sale launch time and duration, especially given that sale price depends on the time elapsed in the sale.

Recommend adding `salesStartTime` and `salesDuration` as parameters in the `SaleBegins` event of `startSale()` function.

[L-02] Incorrect Implementation

The `tokenByIndex` function appears not to perform correctly as it simply checks its input argument and returns it.

It is recommended this function be adequately fleshed out or omitted from the codebase to avoid redundant bytecode.

[L-03] Missing error messages in require statements of various function

The use of informative error messages helps troubleshoot exceptional conditions during transaction failures or unexpected behavior. Otherwise, it can be misleading and waste crucial time during exploits or emergency conditions.

While many require statements have descriptive error messages, some are missing them.

For reference, see Note 2 in OpenZeppelin's Audit of Compound Governor Bravo.

Recommend using meaningful error messages which specifically describe the conditional failure in all require statements.

[L-04] Missing event in critical `devMint()` function

The dev/deployer is allowed to mint an unlimited quantity of NFTs without paying arbitrary recipients. This reduces the token balance and affects token

availability for other sale participants, and therefore is significant enough to warrant its own event.

Recommend adding an event for `devMint` and emit at the end of `devMint()` function.

[L-05] SafeMath library asserts instead of reverts

The implementation of SafeMath performs an `assert` instead of a `revert` on failure. An `assert` will consume all the transaction gas, whereas a `revert/require` releases the remaining gas to the transaction sender again. Usually, one wants to try to keep the gas cost for contract failures low and use `assert` only for invariants that should always be true.

Recommend using `require` instead of `assert`.

Gas Optimizations

[G-00] Explicit initialization with zero not required for `numTokens`

Explicit initialization with zero is not required for variable declaration of `numTokens` because uints are 0 by default. Removing this will reduce contract size and save a bit of gas.

Recommend removing explicit initialization with zero.

[G-01] Numerous Gas Optimizations

This finding is dedicated to the numerous gas optimizations that can be applied across the codebase.

- The `tradeValid`, `cancelOffer`, and `acceptTrade` functions should have their `memory` arrays declared as `calldata`, significantly reducing the gas costs of the functions.
- The `require` statements of L629 and L650 are redundant as the usage of `SafeMath` inherently guarantees them.
- The `deployer`, `beta`, `alpha`, and `beneficiary` variables can all be declared as `immutable` since they are assigned only once during the contract's `constructor`.
- The `SafeMath` statements of L333, L337, L349, and L385 are redundant as they are guaranteed to be safe due to surrounding `require` and `if` clauses.
- The `abi.encodePacked` invocations of L539 and L541 are redundant given that the elements of the arrays cannot be tight packed since they each occupy a full 256-bit slot.

[G-02] state variables that could be declared constant

These state variables can be declared constants to save the gas: `nftName` and `nftSymbol`.

[G-03] public function that could be declared external

public function that could be declared external to save gas

```
1.totalSupply()
2.tokenByIndex(uint256)
3.hashToSign(address,address,uint256,uint256[],uint256,uint256[],uint256,uint256)
```

[G-04] Optimizations storage

Suggestions provided here.

[G-05] `creatorNftMints` is assigned only 0 or 1

It is unclear why this mapping points to uint: `mapping (uint256 => uint256) public creatorNftMints;` the only values it could get is either 0 or 1, so a boolean type might be more suitable here.

Recommend using true/false values if the intention was that 0 means false and 1 means true:

```
mapping (uint256 => boolean) public creatorNftMints;
```

[G-06] `Require()` not needed

On line 650, `require(amount <= ethBalance[msg.sender]);` is not needed because it's implicitly checked when making the subtraction in the following line.

Recommend removing the `require()`.

dangerousfood (Meebits) commented: > Fantastic catch imo.

[G-07] `PauseMarket()` can be optimized

The function `pauseMarket()` on line 230 can be optimized.

Recommend not using an argument and set `marketPaused = !marketPaused`.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable

security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.