Competitive Security Assessment

ParaSpace P2P Pair Staking

Jan 20th, 2023

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

Project Detail

Project Name	ParaSpace P2P Pair Staking
Platform & Language	Solidity
Codebase	• https://github.com/para-space/paraspace-core • audit commit - cf85f96d53b66eee72cdd2f168092d1d1a1aa167 • final commit - 03d0af3f145a43263755e3b6ea9996db88e96b0d
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

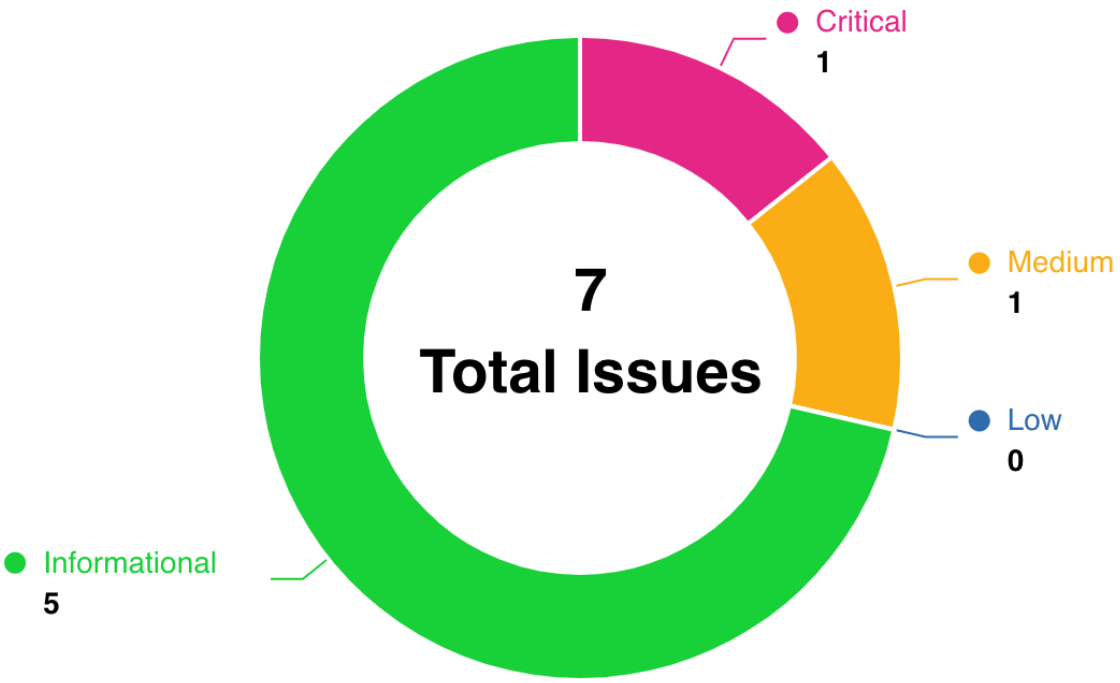
Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	1	0	1	0	0	0
Medium	1	0	0	1	0	0
Low	0	0	0	0	0	0
Informational	5	0	3	2	0	0

Audit Scope

File	Commit Hash
contracts/interfaces/ICApe.sol	cf85f96d53b66eee72cdd2f168092d1d1a1aa167
contracts/interfaces/IP2PPairStaking.sol	cf85f96d53b66eee72cdd2f168092d1d1a1aa167
contracts/misc/P2PPairStaking.sol	cf85f96d53b66eee72cdd2f168092d1d1a1aa167

Code Assessment Findings



ID	Name	Category	Severity	Status	Contributor
PSP-1	Centralization risk	Privilege Related	Informational	Acknowledged	Hupixiong3
PSP-2	Fetch the length of array before the loop to save gas	Gas Optimization	Informational	Fixed	Hupixiong3, Xi_Zi
PSP-3	Missing Protection against Signature Replay Attacks in `P2PPairStaking` contract	Signature Forgery or Replay	Critical	Acknowledged	thereksfour, Xi_Zi
PSP-4	NFT's owner can prevent other matchers from calling breakUpMatchedOrder	Logical	Medium	Fixed	thereksfour

PSP-5	**The user's assets are at risk of being permanently locked in the contract**	**Logical**	**Informational**	**Acknowledged**	**xfu**
PSP-6	**`P2PPairStaking` Gas Optimization by using custom errors**	**Gas Optimization**	**Informational**	**Acknowledged**	**Xi_Zi**
PSP-7	**`P2PPairStaking` Gas Optimization for constant hashes**	**Gas Optimization**	**Informational**	**Fixed**	**Xi_Zi, xfu**

PSP-1:Centralization risk

Category	Severity	Code Reference	Status	Contributor
Privilege Related	Informational	• code/contracts/misc/P2PPairStaking.sol#L606-L617	Acknowledged	Hupixiong3

Code

```
606:    function claimCompoundFee(address receiver) external onlyOwner {
607:        this.claimCApeReward(receiver);
608:    }
609:
610:    function rescueERC20(
611:        address token,
612:        address to,
613:        uint256 amount
614:    ) external onlyOwner {
615:        IERC20(token).safeTransfer(to, amount);
616:        emit RescueERC20(token, to, amount);
617:    }
```

Description

Hupixiong3 : The owner can arbitrarily take the user's income CAPEcoin .

Recommendation

Hupixiong3 : By merging `claimCompoundFee()` and `resuceERC20()`, and limiting the number of `CAPEcoins` that can be transferred by the `resuceERC20()` function.

Consider below fix in the `rescueERC20()` function

```
function rescueERC20(
     address token,
     address to,
     uint256 amount
) external onlyOwner {
     If(token==cApe){
         this.claimCApeReward(to);
     }else{
         IERC20(token).safeTransfer(to, amount);
         emit RescueERC20(token, to, amount);
      }
    }
```

Client Response

We think it's Ok, we use an upgradable way to deploy P2P, it has much more centralization risk. And if we take your recommendationand user sends his cApe to P2Pcontract accidentally, we can't rescue for him.

PSP-2:Fetch the length of array before the loop to save gas

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	Informational	• code/contracts/misc/P2PPairStaking.sol#L344	Fixed	Hupixiong3, Xi_Zi

Code

```
344:        for (uint256 index = 0; index < orderHashes.length; index++) {
```

Description

Hupixiong3 : Cache array length into memory to reduce GAS consumption
Xi_Zi : In the function claimForMatchedOrderAndCompound, when judging the range of i in the for loop, use i < tradeDetails.length; to proceed. In this way, every time the loop statement is executed, the length method of tradeDetails will be called, and the gas consumption will be large.

```
function claimForMatchedOrderAndCompound(bytes32[] calldata orderHashes)
        external
        nonReentrant
    {//@audit
        for (uint256 index = 0; index < orderHashes.length; index++) {
            bytes32 orderHash = orderHashes[index];
            _claimForMatchedOrderAndCompound(orderHash);
        }
    }
```

Recommendation

Hupixiong3 : Cache array length into memory
Consider below fix in the `claimForMatchedOrderAndCompound()` function

```
function claimForMatchedOrderAndCompound(bytes32[] calldata orderHashes)
        external
        nonReentrant
{
        Uint256 Length=orderHashes.length;
        for (uint256 index = 0; index < Length; index++) {
            bytes32 orderHash = orderHashes[index];
            _claimForMatchedOrderAndCompound(orderHash);
        }
}
```

Xi_Zi : It is recommended to define a variable which `uint256 length = orderHashes.length;` Then in the for loop, i < length is used to determine. Sample code is as follows:

```
uint256 length = orderHashes.length;
for (uint256 i = 0; i < length; ) {
    ...
}
```

Client Response

Fixed.

PSP-3:Missing Protection against Signature Replay Attacks in `P2PPairStaking` contract

Category	Severity	Code Reference	Status	Contributor
Signature Forgery or Replay	Critical	• code/contracts/misc/P2PPairStaking.sol#L493-L520	Acknowledged	thereksfour, Xi_Zi

Code

```
493:     function _validateOrderBasicInfo(ListingOrder calldata listingOrder)
494:         internal
495:         view
496:     {
497:         require(
498:             listingOrder.startTime <= block.timestamp,
499:             "ape order not start"
500:         );
501:         require(listingOrder.endTime >= block.timestamp, "ape offer expired");
502:
503:         bytes32 orderHash = getListingOrderHash(listingOrder);
504:         require(!isListingOrderCanceled[orderHash], "order already canceled");
505:
506:         if (
507:             msg.sender != listingOrder.offerer && msg.sender != matchingOperator
508:         ) {
509:             require(
510:                 _validateOrderSignature(
511:                     listingOrder.offerer,
512:                     orderHash,
513:                     listingOrder.v,
514:                     listingOrder.r,
515:                     listingOrder.s
516:                 ),
517:                 "invalid signature"
518:             );
519:         }
520:     }
```

Description

thereksfour : When a user matches orders and staking, the orders are checked and staking is performed in matchPairStakingList/matchBAKCPairStakingList.

```
    function matchBAKCPairStakingList(
        ListingOrder calldata apeOrder,
        ListingOrder calldata bakcOrder,
        ListingOrder calldata apeCoinOrder
    ) external nonReentrant returns (bytes32 orderHash) {
        //1 validate all order
        _validateApeOrder(apeOrder);
        _validateBakcOrder(bakcOrder);
        _validateApeCoinOrder(apeCoinOrder);
```

When the order is checked in _validate*Order, the signature (v, r, s) needs to be provided if the caller is not the offerer of the order.

```
        if (
            msg.sender != listingOrder.offerer && msg.sender != matchingOperator
        ) {
            require(
                _validateOrderSignature(
                    listingOrder.offerer,
                    orderHash,
                    listingOrder.v,
                    listingOrder.r,
                    listingOrder.s
                ),
                "invalid signature"
            );
        }
```

However, when the order passes the check, no changes are made to the order state, resulting in the signature being reusable. For BAYC/MAYC/BAKC, once the order is matched, the NFT is sent to the contract and since the _validate*Order requires owner == the offerer of the order, the ApeOrder/BakcOrder does not pass the check when the signature is reused.

```
    function _validateApeOrder(ListingOrder calldata apeOrder) internal view {
        _validateOrderBasicInfo(apeOrder);

        address expectedToken = bayc;
        if (
            apeOrder.stakingType == StakingType.MAYCStaking ||
            apeOrder.stakingType == StakingType.MAYCPairStaking
        ) {
            expectedToken = mayc;
        }
        require(apeOrder.token == expectedToken, "ape order invalid token");
        require(
            IERC721(expectedToken).ownerOf(apeOrder.tokenId) ==
                apeOrder.offerer,
            "ape order invalid owner"
        );
    }
```

However, for ApeCoinOrder, as long as the offerer of the order has enough ApeCoin and has approved P2PPairStaking, ApeCoinOrder can be matched multiple times by signature reuse.

```
    function _validateApeCoinOrder(ListingOrder calldata apeCoinOrder)
        internal
        view
    {
        _validateOrderBasicInfo(apeCoinOrder);
        require(apeCoinOrder.token == apeCoin, "ape coin order invalid token");
    }
```

And once the matching order is successful, a malicious user can use another vulnerability that I submitted "NFT's owner can prevent other matchers from calling breakUpMatchedOrder" to prevent ApeCoin from being withdrawn

Xi_Zi : Sometimes it is necessary to perform signature verification in smart contracts to achieve better usability.A secure implementation must protect against Signature Replay Attacks by, for example, keeping track of all processed message hashes and only allowing new message hashes.When the contract uses `SignatureChecker.verify`, there is no keeping track of all processed message hashes, it is necessary to increase the nonce or process the already used message hash to reduce the risk of signature replay.

```
function _validateOrderSignature(
        address signer,
        bytes32 orderHash,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) internal view returns (bool) {
        return
            SignatureChecker.verify(
                orderHash,
                signer,
                v,
                r,
                s,
                DOMAIN_SEPARATOR
            );
    }


function verify(
        bytes32 hash,
        address signer,
        uint8 v,
        bytes32 r,
        bytes32 s,
        bytes32 domainSeparator
    ) internal view returns (bool) {
        // \x19\x01 is the standardized encoding prefix
        // https://eips.ethereum.org/EIPS/eip-712#specification
        bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator, hash));
        if (Address.isContract(signer)) {
            // 0x1626ba7e is the interfaceId for signature contracts (see IERC1271)
            return IERC1271(signer).isValidSignature(digest, abi.encodePacked(r, s, v)) ==
0x1626ba7e;
        } else {
            return recover(digest, v, r, s) == signer;
        }
    }
```

Recommendation

thereksfour : Consider using a variable to record whether an ApeCoinOrder is matched or not, and the matched ApeCoinOrder cannot pass the order check. And add the order hash of the ApeCoinOrder to the MatchedOrder structure,

and when break up the matched order, set the match status of the ApeCoinOrder to false

Xi_Zi : 1. Store every message hash that the smart contract has processed. When new messages are received, check against the already existing ones and only proceed with business logic if it's a new message hash. 2. Include nonce into signed message.

Client Response

We intend to do this. This can allow user just list an order to match as many as possible. And user can cancel his order to invalid this order and break up his matched order freely.

PSP-4:NFT's owner can prevent other matchers from calling breakUpMatchedOrder

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	• code/contracts/misc/P2PPairStaking.sol#L319-L334	Fixed	thereksfour

Code

```
319:         IERC721(order.apeToken).safeTransferFrom(
320:             address(this),
321:             order.apeOfferer,
322:             order.apeTokenId
323:         );
324:         IERC20(apeCoin).safeTransfer(
325:             order.apeCoinOfferer,
326:             order.apePrincipleAmount
327:         );
328:         if (order.stakingType >= StakingType.BAYCPairStaking) {
329:             IERC721(bakc).safeTransferFrom(
330:                 address(this),
331:                 order.bakcOfferer,
332:                 order.bakcTokenId
333:             );
334:         }
```

Description

thereksfour : When an order is matched, all matchers have permission to call the breakUpMatchedOrder function to break up the matched order.

```
    function breakUpMatchedOrder(bytes32 orderHash) external nonReentrant {
        MatchedOrder memory order = matchedOrders[orderHash];

        //1 check owner
        require(
            msg.sender == matchingOperator ||
                msg.sender == order.apeOfferer ||
                msg.sender == order.bakcOfferer ||
                msg.sender == order.apeCoinOfferer,
            "no permission to break up"
        );
```

In the breakUpMatchedOrder function, BAYC/MAYC/BAKC/APECoin will be transferred to the corresponding matcher. However, when transferring BAYC/MAYC/BAKC, breakUpMatchedOrder calls the safeTransferFrom function,

```
        IERC721(order.apeToken).safeTransferFrom(
            address(this),
            order.apeOfferer,
            order.apeTokenId
        );
        IERC20(apeCoin).safeTransfer(
            order.apeCoinOfferer,
            order.apePrincipleAmount
        );
        if (order.stakingType >= StakingType.BAYCPairStaking) {
            IERC721(bakc).safeTransferFrom(
                address(this),
                order.bakcOfferer,
                order.bakcTokenId
            );
        }
```

which allows the recipient of BAYC/MAYC/BAKC to prevent the contract from sending BAYC/MAYC/BAKC to himself in the onERC721Received function, thus prevent other matchers from calling the breakUpMatchedOrder function

```
    function _checkOnERC721Received(address from, address to, uint256 tokenId, bytes memory _data)
        private returns (bool)
    {
        if (to.isContract()) {
            try IERC721Receiver(to).onERC721Received(_msgSender(), from, tokenId, _data) returns
(bytes4 retval) {
                return retval == IERC721Receiver(to).onERC721Received.selector;
```

Consider a BAKC holder who sends a BAKC to a smart contract with the following code:

```
function setFlag(bool f) onlyOwner{
      flag = f;
}
function  onERC721Received( op, from, tokenId, data) returns(bytes){
     if(flag) return this.onERC721Received.selector;
     else return "0x00";
}
```

The contract's owner can control whether the contract is allowed to receive NFTs sent via safeTransferFrom by setting the flag.

The holder of the BAKC is matched to a holder of BAYC and APECoin to stake. The shares are (BAYC:70 BAKC:15 APECoin: 15) The BAYC holder then finds a match with 80 shares and wants to call the breakUpMatchedOrder function to end the current matched order. However, since BAKC is currently set to not receive NFTs, the breakUpMatchedOrder call fails, i.e. the other matchers cannot break up the matched order

Recommendation

thereksfour : Consider wrapping the call to safeTransferFrom in the breakUpMatchedOrder function with a try catch statement. When the NFT send fails, set a map variable to allow the user to claim his NFT later

Client Response

We do have this problem before, but since now we only support supplied NFT from Lending pool, this problem disappeared.

PSP-5:The user's assets are at risk of being permanently locked in the contract

Category	Severity	Code Reference	Status	Contributor
Logical	Informational	• code/contracts/misc/P2PPairStaking.sol#L283	Acknowledged	xfu

Code

```
283:         _claimForMatchedOrderAndCompound(orderHash);
```

Description

xfu : Consider below POC contract

```
function breakUpMatchedOrder(bytes32 orderHash) external nonReentrant {
        ...

        //2 claim pending reward
        _claimForMatchedOrderAndCompound(orderHash);

        ...
    }

function _claimForMatchedOrderAndCompound(bytes32 orderHash) internal {
        ...

        if (order.stakingType < StakingType.BAYCPairStaking) {
            uint256[] memory _nfts = new uint256[](1);
            _nfts[0] = order.apeTokenId;
            if (order.stakingType == StakingType.BAYCStaking) {
                apeCoinStaking.claimSelfBAYC(_nfts);
            } else {
                apeCoinStaking.claimSelfMAYC(_nfts);
            }
        } else {

            ...

            if (order.stakingType == StakingType.BAYCPairStaking) {
                apeCoinStaking.claimSelfBAKC(_nfts, _otherPairs);
            } else {
                apeCoinStaking.claimSelfBAKC(_otherPairs, _nfts);
            }
        }

        uint256 shareBefore = ICApe(cApe).sharesOf(address(this));
        IAutoCompoundApe(cApe).deposit(address(this), rewardAmount);
        uint256 shareAfter = ICApe(cApe).sharesOf(address(this));

        ...
    }
```

The implementation of `breakUpMatchedOrder` depends on `_claimForMatchedOrderAndCompound`, `_claimForMatchedOrderAndCompound` depends on apeCoinStaking's claim implementation and IAutoCompoundApe's deposit implementation. The third-party implementation may have the risk of throwing an exception, causing the `breakUpMatchedOrder` function to fail to execute the subsequent asset withdrawal logic.

Recommendation

xfu : Add emergency withdrawal function The emergency withdrawal function only realizes the withdrawal from ApeCoinStaking assets, and then returns the assets to the original users. The authority of the emergency withdrawal function should be executed by the contract owner.

Client Response

We think it's ok, because cApe was used with an upgradable way to deploy and it is fully under our control.

PSP-6: `P2PPairStaking` Gas Optimization by using custom errors

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	Informational	• code/contracts/misc/P2PPairStaking.sol#L107 • code/contracts/misc/P2PPairStaking.sol#L109 • code/contracts/misc/P2PPairStaking.sol#L124 • code/contracts/misc/P2PPairStaking.sol#L128 • code/contracts/misc/P2PPairStaking.sol#L132 • code/contracts/misc/P2PPairStaking.sol#L196 • code/contracts/misc/P2PPairStaking.sol#L200	Acknowledged	Xi_Zi

Code

```
107:         require(msg.sender == listingOrder.offerer, "not order offerer");
109:         require(!isListingOrderCanceled[orderHash], "order already cancel");
124:         require(
128:         require(
132:         require(
196:         require(
200:         require(
```

Description

Xi_Zi : When Solidity v0.8.4 introduced the ability to define custom errors, developers gained a better way to handle transactions that revert. As stated in Solidity's v0.8.4 release, the use of custom errors can reduce runtime and deployment costs.It is recommended to use custom errors instead of require to save gas.

```
function _validateApeOrder(ListingOrder calldata apeOrder) internal view {
        _validateOrderBasicInfo(apeOrder);

        address expectedToken = bayc;
        if (
            apeOrder.stakingType == StakingType.MAYCStaking ||
            apeOrder.stakingType == StakingType.MAYCPairStaking
        ) {
            expectedToken = mayc;
        }
        require(apeOrder.token == expectedToken, "ape order invalid token");
        require(
            IERC721(expectedToken).ownerOf(apeOrder.tokenId) ==
                apeOrder.offerer,
            "ape order invalid owner"
        );
    }

    function _validateBakcOrder(ListingOrder calldata bakcOrder) internal view {
        _validateOrderBasicInfo(bakcOrder);

        require(bakcOrder.token == bakc, "bakc order invalid token");
        require(
            IERC721(bakc).ownerOf(bakcOrder.tokenId) == bakcOrder.offerer,
            "bakc order invalid owner"
        );
    }
```

Recommendation

Xi_Zi : It is recommended to use custom errors instead of require to save gas. Reference link: https://blog.openzeppelin.com/defining-industry-standards-for-custom-error-messages-to-improve-the-web3-developer-experience/

Client Response

We think it's ok.

PSP-7: `P2PPairStaking` Gas Optimization for constant hashes

Category	Severity	Code Reference	Status	Contributor
Gas Optimization	Informational	• code/contracts/misc/P2PPairStaking.sol#L27-L39	Fixed	Xi_Zi, xfu

Code

```
27:     bytes32 public constant LISTING_ORDER_HASH =
28:         keccak256(
29:             "ListingOrder(uint8 stakingType,address offerer,address token,uint256 tokenId,uint256 share,uint256 startTime,uint256 endTime)"
30:         );
31:     bytes32 public constant MATCHED_ORDER_HASH =
32:         keccak256(
33:             "MatchedOrder(uint8 stakingType,address apeToken,address apeOfferer,uint32 apeTokenId,uint32 apeShare,address bakcOfferer,uint32 bakcTokenId,uint32 bakcShare,address apeCoinOfferer,uint32 apeCoinShare,uint256 apePrincipleAmount)"
34:         );
35:     bytes32 internal constant EIP712_DOMAIN =
36:         keccak256(
37:             "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
38:         );
39:
```

Description

Xi_Zi : In Solidity, using bytes32 public constants to store hashed strings saves gas. This is because constants are pre-hashed at compile time and don't need to be hashed at runtime. And if the hash operation is performed at runtime, it will consume more gas.

```
bytes32 public constant LISTING_ORDER_HASH =
        keccak256(
            "ListingOrder(uint8 stakingType,address offerer,address token,uint256 tokenId,uint256
share,uint256 startTime,uint256 endTime)"
        );
    bytes32 public constant MATCHED_ORDER_HASH =
        keccak256(
            "MatchedOrder(uint8 stakingType,address apeToken,address apeOfferer,uint32
apeTokenId,uint32 apeShare,address bakcOfferer,uint32 bakcTokenId,uint32 bakcShare,address
apeCoinOfferer,uint32 apeCoinShare,uint256 apePrincipleAmount)"
        );
    bytes32 internal constant EIP712_DOMAIN =
        keccak256(
            "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
        );
```

xfu : Consider below POC contract

```
contract P2PPairStaking is
    Initializable,
    OwnableUpgradeable,
    ReentrancyGuardUpgradeable,
    IP2PPairStaking
{
    ...

    bytes32 public constant LISTING_ORDER_HASH =
        keccak256(
            "ListingOrder(uint8 stakingType,address offerer,address token,uint256 tokenId,uint256
share,uint256 startTime,uint256 endTime)"
        );
    bytes32 public constant MATCHED_ORDER_HASH =
        keccak256(
            "MatchedOrder(uint8 stakingType,address apeToken,address apeOfferer,uint32
apeTokenId,uint32 apeShare,address bakcOfferer,uint32 bakcTokenId,uint32 bakcShare,address
apeCoinOfferer,uint32 apeCoinShare,uint256 apePrincipleAmount)"
        );
    bytes32 internal constant EIP712_DOMAIN =
        keccak256(
            "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
        );

    ...

    function initialize() public initializer {
        ...

        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                EIP712_DOMAIN,
                keccak256("ParaSpace"),
                keccak256(bytes("1")),
                block.chainid,
                address(this)
            )
        );

        ...
    }
```

```
  }
```

Recommendation

Xi_Zi : Calculate the hash of the string first, then use the constant to store it, and explain the string corresponding to the hash in the form of a comment。
xfu : Consider below fix in the function

```
contract P2PPairStaking is
    Initializable,
    OwnableUpgradeable,
    ReentrancyGuardUpgradeable,
    IP2PPairStaking
{
    ...

    // "ListingOrder(uint8 stakingType,address offerer,address token,uint256 tokenId,uint256
share,uint256 startTime,uint256 endTime)"
    bytes32 public constant LISTING_ORDER_HASH =
"0x227f9dd14259caacdbcf45411b33cf1c018f31bd3da27e613a66edf8ae45814f";
    // "MatchedOrder(uint8 stakingType,address apeToken,address apeOfferer,uint32 apeTokenId,uint32
apeShare,address bakcOfferer,uint32 bakcTokenId,uint32 bakcShare,address apeCoinOfferer,uint32
apeCoinShare,uint256 apePrincipleAmount)"
    bytes32 public constant MATCHED_ORDER_HASH =
"0xea1b65f1d4f5d13139950e2a2c8d6c55617bb881b2252e16928249b0c015d0fb";
    // "EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
    bytes32 internal constant EIP712_DOMAIN =
"0x8b73c3c69bb8fe3d512ecc4cf759cc79239f7b179b0ffacaa9a75d522b39400f";

    ...

    function initialize() public initializer {
        ...

        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                EIP712_DOMAIN,
                // keccak256("ParaSpace")
                "0x88d989289235fb06c18e3c2f7ea914f41f773e86fb0073d632539f566f4df353",
                // keccak256(bytes("1"))
                "0xc89efdaa54c0f20c7adf612882df0950f5a951637e0307cdcb4c672f298b8bc6",
                block.chainid,
                address(this)
            )
        );

        ...
    }
```

```
  }
```

Client Response

Fixed.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3’s prior written consent in each instance.

This report is not an “endorsement” or “disapproval” of any particular project or team. This report is not an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3’s position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.