Competitive Security Assessment

ParaSpace V1.4 P3

Mar 26th, 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 V1.4 P3
Platform & Language	Solidity
Codebase	• https://github.com/para-space/paraspace-core • audit commit - b0a957fc7b6df9109a8a617d7dddce102088d43c • final commit - 629e07165cbbf6679727d0f83fa8f72598d09d16
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

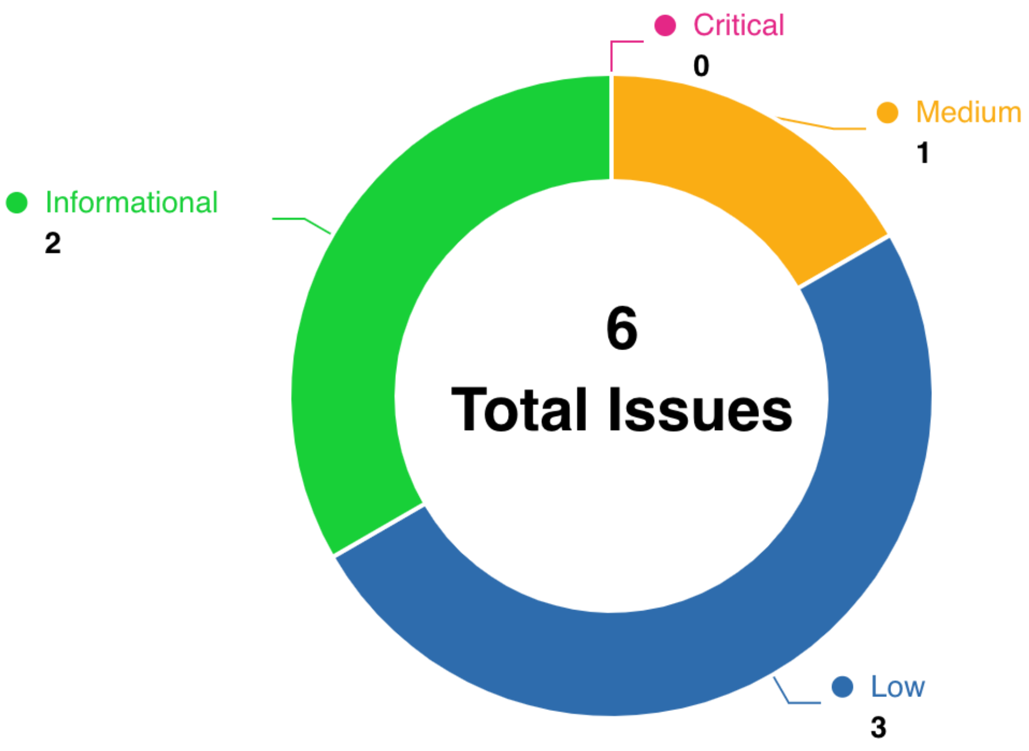
Code Vulnerability Review Summary

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

Audit Scope

File	Commit Hash
contracts/protocol/tokenization/libraries/MintableERC721Logic.sol	b0a957fc7b6df9109a8a617d7dddce102088d43c
contracts/protocol/tokenization/base/MintableIncentivizedERC721.sol	b0a957fc7b6df9109a8a617d7dddce102088d43c
contracts/protocol/libraries/logic/LiquidationLogic.sol	b0a957fc7b6df9109a8a617d7dddce102088d43c
contracts/protocol/libraries/logic/GenericLogic.sol	b0a957fc7b6df9109a8a617d7dddce102088d43c
contracts/protocol/tokenization/NToken.sol	b0a957fc7b6df9109a8a617d7dddce102088d43c
contracts/protocol/libraries/logic/MarketplaceLogic.sol	b0a957fc7b6df9109a8a617d7dddce102088d43c

Code Assessment Findings



ID	Name	Category	Severity	Status	Contributor
PSV-1	MIN_TRAIT_MULTIPLIER should be 1e18 not 0e18	Logical	Low	Declined	thereksfour
PSV-2	MarketplaceLogic._checkAllowance approves marketplace to use uint256.max amount of creditToken, which allows the user to deplete the tokens in the POOL.	Logical	Medium	Declined	thereksfour
PSV-3	NFT listing price could be incorrectly calculated	Logical	Informational	Acknowledged	jayphbee

PSV-4	**_buyWithCredit/_acceptBidWithCredit no longer supports filling NFT orders with any creditToken.**	**Logical**	**Low**	**Fixed**	**thereksfour**
PSV-5	**`executeBurnMultiple` should validate `user` is not address 0**	**Logical**	**Low**	**Declined**	**comcat**
PSV-6	**safetransferfrom doesn't have the callback check**	**Logical**	**Informational**	**Acknowledged**	**comcat**

PSV-1:MIN_TRAIT_MULTIPLIER should be 1e18 not 0e18

Category	Severity	Code Reference	Status	Contributor
Logical	Low	• code/contracts/protocol/tokenization/libraries/MintableERC721Logic.sol#L78-L79 • code/contracts/protocol/tokenization/libraries/MintableERC721Logic.sol#L683-L689	Declined	thereksfour

Code

```
78:    uint256 internal constant MIN_TRAIT_MULTIPLIER = 0e18;
79:

683:    function _checkTraitMultiplier(uint256 multiplier) private pure {
684:        require(
685:            multiplier >= MIN_TRAIT_MULTIPLIER &&
686:                multiplier < MAX_TRAIT_MULTIPLIER,
687:            Errors.INVALID_AMOUNT
688:        );
689:    }
```

Description

thereksfour : In MintableERC721Logic, MAX_TRAIT_MULTIPLIER and MIN_TRAIT_MULTIPLIER represent the upper and lower bounds of the trait multiplier and are used in _checkTraitMultiplier. Since MIN_TRAIT_MULTIPLIER is 0, it means that the admin can set the trait multiplier to less than 1e18, and since 1e18 is the floor price, the trait multiplier should not be less than 1e18. This makes _checkTraitMultiplier unable to limit the trait multiplier.

```
     * @dev This constant represents the maximum trait multiplier that a single tokenId can have
     * A value of 20e18 results in 20x of price
     */
    uint256 internal constant MAX_TRAIT_MULTIPLIER = 20e18;
    /**
     * @dev This constant represents the minimum trait multiplier that a single tokenId can have
     * A value of 1e18 results in no price multiplier
     */
    uint256 internal constant MIN_TRAIT_MULTIPLIER = 0e18;
...
    function _checkTraitMultiplier(uint256 multiplier) private pure {
        require(
            multiplier >= MIN_TRAIT_MULTIPLIER &&
                multiplier < MAX_TRAIT_MULTIPLIER,
            Errors.INVALID_AMOUNT
        );
    }
```

Recommendation

thereksfour :

```
-    uint256 internal constant MIN_TRAIT_MULTIPLIER = 0e18;
+    uint256 internal constant MIN_TRAIT_MULTIPLIER = 1e18;
```

Client Response

Since percentDiv is dividing a value that is less than 1, current implementation actually provid better precision

PSV-2:MarketplaceLogic._checkAllowance approves marketplace to use uint256.max amount of creditToken, which allows the user to deplete the tokens in the POOL.

Category	Severity	Code Reference	Status	Contributor
Logical	Medium	• code/contracts/misc/marketplaces/LooksRareAdapter.sol#L68-L75 • code/contracts/dependencies/looksrare/contracts/LooksRareExchange.sol#L218-L225 • code/contracts/protocol/libraries/logic/MarketplaceLogic.sol#L615-L620 • code/contracts/protocol/libraries/logic/MarketplaceLogic.sol#L688-L707	Declined	thereksfour

Code

```
        consideration[0] = ConsiderationItem(
            itemType,
            token,
            0,
            makerAsk.price, // TODO: take minPercentageToAsk into account
            makerAsk.price,
            payable(takerBid.taker)
        );

        _transferFeesAndFundsWithWETH(
            makerAsk.strategy,
            makerAsk.collection,
            tokenId,
            makerAsk.signer,
            takerBid.price,
            makerAsk.minPercentageToAsk
        );

    function _checkAllowance(address token, address operator) internal {
        uint256 allowance = IERC20(token).allowance(address(this), operator);
        if (allowance == 0) {
            IERC20(token).safeApprove(operator, type(uint256).max);
        }
    }

    function _validateAndGetPrice(
        DataTypes.ExecuteMarketplaceParams memory params,
        MarketplaceLocalVars memory vars
    ) internal pure returns (uint256 price) {
        for (uint256 i = 0; i < params.orderInfo.consideration.length; i++) {
            ConsiderationItem memory item = params.orderInfo.consideration[i];
            require(
                item.startAmount == item.endAmount,
                Errors.INVALID_MARKETPLACE_ORDER
            );
            require(
                item.itemType == ItemType.ERC20 ||
                    (vars.isETH && item.itemType == ItemType.NATIVE),
                Errors.INVALID_ASSET_TYPE
            );
            require(
                item.token == params.credit.token,
            Errors.CREDIT_DOES_NOT_MATCH_ORDER
        );
        price += item.startAmount;
```

Description

thereksfour : Take Looksrare for example. In _buyWithCredit, the taker needs to provide makerAsk.price amount of creditToken.

```
        consideration[0] = ConsiderationItem(
            itemType,
            token,
            0,
            makerAsk.price, // TODO: take minPercentageToAsk into account
            makerAsk.price,
            payable(takerBid.taker)
        );
...
    function _validateAndGetPrice(
        DataTypes.ExecuteMarketplaceParams memory params,
        MarketplaceLocalVars memory vars
    ) internal pure returns (uint256 price) {
        for (uint256 i = 0; i < params.orderInfo.consideration.length; i++) {
            ConsiderationItem memory item = params.orderInfo.consideration[i];
            require(
                item.startAmount == item.endAmount,
                Errors.INVALID_MARKETPLACE_ORDER
            );
            require(
                item.itemType == ItemType.ERC20 ||
                    (vars.isETH && item.itemType == ItemType.NATIVE),
                Errors.INVALID_ASSET_TYPE
            );
            require(
                item.token == params.credit.token,
                Errors.CREDIT_DOES_NOT_MATCH_ORDER
            );
            price += item.startAmount;
```

But in LooksRareExchange.matchBidWithTakerAsk, the actual takerAsk.price will be used as the deal price.

```
        _transferFeesAndFunds(
            makerBid.strategy,
            makerBid.collection,
            tokenId,
            makerBid.currency,
            makerBid.signer,
            takerAsk.taker,
            takerAsk.price,    // @audit: deal price
            takerAsk.minPercentageToAsk
        );
```

If takerAsk.price > makerAsk.price (in Looksrare, the maker can use any strategy to make it valid), the taker can use the token in POOL to fill the order since _checkAllowance approves the marketplace to use uint256.max amount of creditToken.

```
    function _checkAllowance(address token, address operator) internal {
        uint256 allowance = IERC20(token).allowance(address(this), operator);
        if (allowance == 0) {
            IERC20(token).safeApprove(operator, type(uint256).max);
        }
    }
```

Consider the following scenario. POOL has 100 creditTokens. alice's makerAsk.price == 100 on Looksrare. bob offers takerAsk.price = 200 to fill the order. Since makerAsk.price == 100, bob only needs to provide 100 creditTokens and Looksrare will send all 200 (100+100) creditTokens in the POOL to alice. Since POOL is not designed to keep tokens, it should be medium risk

Recommendation

thereksfour : Consider only approving the marketplace to use the vars.price amount of creditTokens in _checkAllowance. Change to

```
    function _checkAllowance(address token, address operator, uint256 amount) internal {
            IERC20(token).safeApprove(operator, 0);
            IERC20(token).safeApprove(operator, amount);
    }
```

Client Response

We allow trait multiplier to be <1 for specific void NFTs

PSV-3:NFT listing price could be incorrectly calculated

Category	Severity	Code Reference	Status	Contributor
Logical	Informational	• code/contracts/protocol/libraries/logic/MarketplaceLogic.sol#L688	Acknowledged	jayphbee

Code

```
688:    function _validateAndGetPrice(
```

Description

jayphbee : NFT listing price is calculated in `_validateAndGetPrice`

```
    function _validateAndGetPrice(
        DataTypes.ExecuteMarketplaceParams memory params,
        MarketplaceLocalVars memory vars
    ) internal pure returns (uint256 price) {
        for (uint256 i = 0; i < params.orderInfo.consideration.length; i++) {
            ConsiderationItem memory item = params.orderInfo.consideration[i];
            require(
                item.startAmount == item.endAmount,
                Errors.INVALID_MARKETPLACE_ORDER
            );
            require(
                item.itemType == ItemType.ERC20 ||
                    (vars.isETH && item.itemType == ItemType.NATIVE),
                Errors.INVALID_ASSET_TYPE
            );
            require(
                item.token == params.credit.token,
                Errors.CREDIT_DOES_NOT_MATCH_ORDER
            );
            price += item.startAmount;
        }
    }
```

The `itemType` must be `ERC20` or `NATIVE`. `ERC20` and `NATIVE` could have different decimals, but `price` is accumulated without differentiating it. Thus NFT listing price could be incorrectly calculated.

Recommendation

jayphbee : check ERC20 decimals to scale up or scale down.

Client Response

It's intended for optimizing gas cost.

PSV-4:_buyWithCredit/_acceptBidWithCredit no longer supports filling NFT orders with any creditToken.

Category	Severity	Code Reference	Status	Contributor
Logical	Low	• code/contracts/protocol/libraries/logic/MarketplaceLogic.sol#L440-L465	Fixed	thereksfour

Code

```
440:    function _flashSupplyFor(
441:        DataTypes.PoolStorage storage ps,
442:        DataTypes.ExecuteMarketplaceParams memory params,
443:        MarketplaceLocalVars memory vars,
444:        address seller
445:    ) internal {
446:        if (vars.isETH) {
447:            return; // impossible to supply ETH on behalf of the
448:        }
449:
450:        DataTypes.ReserveData storage reserve = ps._reserves[vars.creditToken];
451:        DataTypes.UserConfigurationMap storage sellerConfig = ps._usersConfig[
452:            seller
453:        ];
454:        DataTypes.ReserveCache memory reserveCache = reserve.cache();
455:        uint16 reserveId = reserve.id; // cache to reduce one storage read
456:
457:        reserve.updateState(reserveCache);
458:
459:        uint256 supplyAmount = Math.min(
460:            IERC20(vars.creditToken).allowance(seller, address(this)),
461:            vars.price.percentMul(DEFAULT_SUPPLY_RATIO)
462:        );
463:        if (supplyAmount == 0) {
464:            return;
465:        }
```

Description

thereksfour : In the previous implementation of _buyWithCredit/_acceptBidWithCredit, when creditAmount == 0, the user could use creditToken that did not have a corresponding PToken to fill the order, which made it possible because when creditAmount == 0, there was no need to borrow any PToken. However, in the current implementation, even if creditAmount == 0 and supplyAmount == 0, _flashSupplyFor will try to get the PToken corresponding to the creditToken.

```
    function _flashSupplyFor(
        DataTypes.PoolStorage storage ps,
        DataTypes.ExecuteMarketplaceParams memory params,
        MarketplaceLocalVars memory vars,
        address seller
    ) internal {
        if (vars.isETH) {
            return; // impossible to supply ETH on behalf of the
        }

        DataTypes.ReserveData storage reserve = ps._reserves[vars.creditToken];
        DataTypes.UserConfigurationMap storage sellerConfig = ps._usersConfig[
            seller
        ];
        DataTypes.ReserveCache memory reserveCache = reserve.cache();
        uint16 reserveId = reserve.id; // cache to reduce one storage read

        reserve.updateState(reserveCache);

        uint256 supplyAmount = Math.min(
            IERC20(vars.creditToken).allowance(seller, address(this)),
            vars.price.percentMul(DEFAULT_SUPPLY_RATIO)
        );
        if (supplyAmount == 0) {
            return;
        }
```

And if the creditToken does not have a corresponding PToken, _flashSupplyFor fails, thus reverting the entire transaction.

```
    function cache(DataTypes.ReserveData storage reserve)
        internal
        view
        returns (DataTypes.ReserveCache memory)
    {
        DataTypes.ReserveCache memory reserveCache;

        reserveCache.reserveConfiguration = reserve.configuration;
        reserveCache.xTokenAddress = reserve.xTokenAddress;

        (, , , , DataTypes.AssetType reserveAssetType) = reserveCache
            .reserveConfiguration
            .getFlags();                                // @audit: revert here
```

While the protocol should encourage the user to use the creditToken supported by the protocol, this should not be mandatory and should allow the user to use other creditTokens to fill NFT orders if the collateral is sufficient

Recommendation

thereksfour : Consider that the PToken corresponding to the creditToken is no longer obtained when supplyAmount == 0 in _flashSupplyFor()

```
    function _flashSupplyFor(
        DataTypes.PoolStorage storage ps,
        DataTypes.ExecuteMarketplaceParams memory params,
        MarketplaceLocalVars memory vars,
        address seller
    ) internal {
        if (vars.isETH) {
            return; // impossible to supply ETH on behalf of the
        }
+       uint256 supplyAmount = Math.min(
+           IERC20(vars.creditToken).allowance(seller, address(this)),
+           vars.price.percentMul(DEFAULT_SUPPLY_RATIO)
+       );
+       if (supplyAmount == 0) {
+           return;
+       }

        DataTypes.ReserveData storage reserve = ps._reserves[vars.creditToken];
        DataTypes.UserConfigurationMap storage sellerConfig = ps._usersConfig[
            seller
        ];
        DataTypes.ReserveCache memory reserveCache = reserve.cache();
        uint16 reserveId = reserve.id; // cache to reduce one storage read

        reserve.updateState(reserveCache);

-       uint256 supplyAmount = Math.min(
-           IERC20(vars.creditToken).allowance(seller, address(this)),
-           vars.price.percentMul(DEFAULT_SUPPLY_RATIO)
-       );
-       if (supplyAmount == 0) {
-           return;
-       }
```

Client Response

We confirm this issue.

PSV-5: `executeBurnMultiple` should validate `user` is not address 0

Category	Severity	Code Reference	Status	Contributor
Logical	**Low**	• **code/contracts/protocol/tokenization/libraries/MintableERC721Logic.sol#L407**	**Declined**	**comcat**

Code

```
407:    function executeBurnMultiple(
```

Description

comcat : executeBurnMultiple normally burn user's Ntoken, inside its implementation, it first check the user is the owner of the tokenId. however, it doesn't check whether the tokenId exists first. by passing the user to address zero, and an non exists tokenId, it will pass the check

```
address owner = erc721Data.owners[tokenId];
require(owner == user, "not the owner of Ntoken");
```

and it will call the `_removeTokenFromAllTokensEnumeration` function, which will swap and pop the tokenId stored inside.

Recommendation

comcat : add another check:

```
function executeBurnMultiple(
        MintableERC721Data storage erc721Data,
        IPool POOL,
        bool ATOMIC_PRICING,
        address user,
        uint256[] calldata tokenIds
    ) external returns (uint64, uint64) {
        LocalVars memory vars = _cache(erc721Data, user);
        uint256 oldTotalSupply = erc721Data.allTokens.length;
        bool shouldUpdateUserAvgMultiplier = _shouldUpdateUserAvgMultiplier(
            erc721Data,
            ATOMIC_PRICING
        );

        for (uint256 index = 0; index < tokenIds.length; index++) {
            uint256 tokenId = tokenIds[index];
            address owner = erc721Data.owners[tokenId];
            require(owner == user, "not the owner of Ntoken");
+           require(owner != address(0), "not address zero");
```

Client Response

We don't need to validate for burn because mint & transfer will guarantee the receiver is not address 0.

PSV-6:safetransferfrom doesn't have the callback check

Category	Severity	Code Reference	Status	Contributor
Logical	Informational	• code/contracts/protocol/tokenization/base/MintableIncentivizedERC721.sol#L380	Acknowledged	comcat

Code

```
380:    function _safeTransfer(
```

Description

comcat : inside the MintableIncentivizedERC721 contract, the behavior of `safeTransferFrom` is the same as the `transferFrom`,

```
function _safeTransfer(
        address from,
        address to,
        uint256 tokenId,
        bytes memory
    ) internal virtual {
        _transfer(from, to, tokenId);
    }
```

which doesn't follow the EIP-721 standard, which requires the safetransferfrom function to check the `to` address has the corresponding callback function `onERC721Received`.

Recommendation

comcat : follow the standard, add the check to `to` address for the `safetransferfrom`

```
function _safeTransfer(
        address from,
        address to,
        uint256 tokenId,
        bytes memory
    ) internal virtual {
        _transfer(from, to, tokenId);
+      require(_checkOnERC721Received(from, to, tokenId, data), "ERC721: transfer to non
ERC721Receiver implementer");
    }
```

Client Response

It's intended behaviour at the moment for reducing re-entrancy probability.

Disclaimer