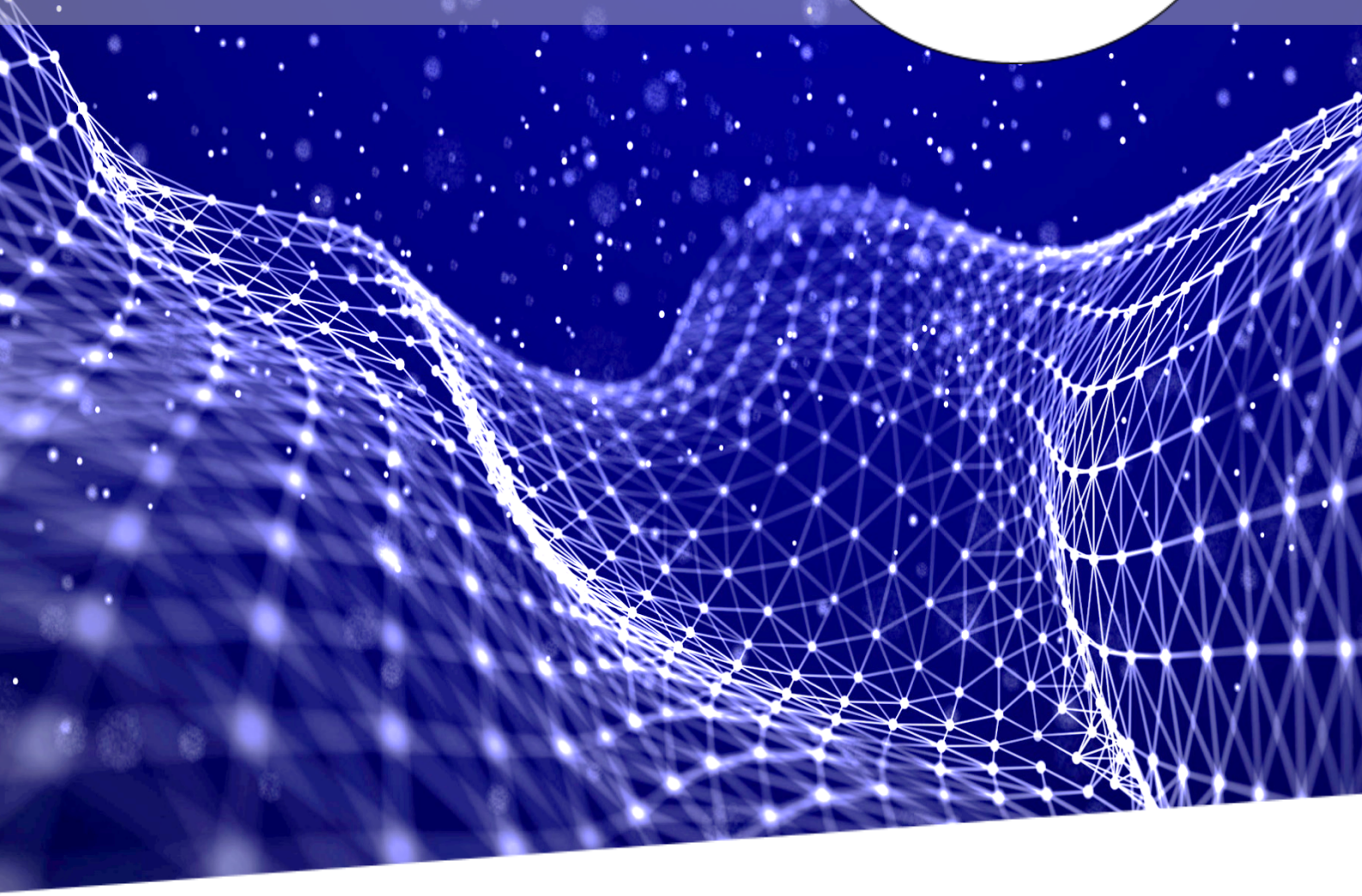
Xtatuz DMCC

XTATUZ asset tokenization

Smart Contract Audit Report



Date Issued: 17 Mar 2023

Version: Final v1.0

Public

ValiX Consulting

Table of Contents

Executive Summary

Overview

Valix conducted a smart contract audit to evaluate potential security issues of the **XTATUZ asset tokenization feature**. This audit report was published on *17 Mar 2023*. The audit scope is limited to the **XTATUZ asset tokenization feature.** Our security best practices strongly recommend that the **Xtatuz DMCC team** conduct a full security audit for both on-chain and off-chain components of its infrastructure and their interaction. A comprehensive examination has been performed during the audit process utilizing Valix's Formal Verification, Static Analysis, and Manual Review techniques.

About XTATUZ asset tokenization

The **XTATUZ asset tokenization** feature is a tool for the tokenization of assets with the objective to unlock the possibility of co-ownership or the fractionalization of ownership of traditional assets that already exist in this current world. Each individual token, which represents an ownership or a partial ownership of a particular asset, should possess the quality of transferability and can be traced and tracked on a decentralized ledger. This allows ownership to be traded with ease in a decentralized manner, enhancing transparency, security and accessibility.

Scope of Work

The security audit conducted does not replace the full security audit of the overall Xtatuz DMCC protocol. The scope is limited to the **XTATUZ asset tokenization feature** and their related smart contracts.

The security audit covered the components at this specific state:

Item	Description
Components	▪ *XTATUZ asset tokenization smart contract* ▪ *Imported associated smart contracts and libraries*
Git Repository	▪ *https://github.com/XTATUZ/xtatuz-contracts-v1*
Audit Commit	▪ *ffb2d2388cb5d9446aa3cb4105b4156dd1362677 (branch: main)*
Reassessment Commit	▪ *ebf430277b06e355a2199392e926d923235355f6 (branch: main)*

Audited Files	▪ ./contracts/Presaled.sol ▪ ./contracts/PresaledFactory.sol ▪ ./contracts/ProjectFactory.sol ▪ ./contracts/Property.sol ▪ ./contracts/PropertyFactory.sol ▪ ./contracts/XtatuzFactory.sol ▪ ./contracts/XtatuzProject.sol ▪ ./contracts/XtatuzReferral.sol ▪ ./contracts/XtatuzReroll.sol ▪ ./contracts/XtatuzRouter.sol ▪ Other imported associated Solidity files
Excluded Files/Contracts	▪ ./contract/Token.sol ▪ ./contract/XTAToken.sol

Remark: Our security best practices strongly recommend that the Xtatuz DMCC team conduct a full security audit for both on-chain and off-chain components of its infrastructure and the interaction between them.

Auditors

Role	Staff List
Auditors	**Anak Mirasing (Lead Auditor)** **Kritsada Dechawattana** **Parichaya Thanawuthikrai**
Authors	**Anak Mirasing** **Kritsada Dechawattana** **Parichaya Thanawuthikrai**
Reviewers	**Phuwanai Thummavet (Technical Advisor)** **Sumedt Jitpukdebodin**

Disclaimer

Our smart contract audit was conducted over a limited period and was performed on the smart contract at a single point in time. As such, the scope was limited to current known risks during the work period. The review does not indicate that the smart contract and blockchain software has no vulnerability exposure.

We reviewed the security of the smart contracts with our best effort, and we do not guarantee a hundred percent coverage of the underlying risk existing in the ecosystem. The audit was scoped only in the provided code repository. The on-chain code is not in the scope of auditing.

This audit report does not provide any warranty or guarantee, nor should it be considered an “approval” or “endorsement” of any particular project. This audit report should also not be used as investment advice nor provide any legal compliance.

Audit Result Summary

From the audit results and the remediation and response from the developer, Valix trusts that the **XTATUZ asset tokenization feature** has sufficient security protections to be safe for use.

Initially, Valix was able to identify **50 issues** that were categorized from the "Critical" to "Informational" risk level in the given timeframe of the assessment.

For the reassessment, the *Xtatuz* team acknowledged a total of 50 issues, including 8 critical issues, 11 high issues, 17 medium issues, 4 low issues, and 10 informational issues. Of these, the team was able to completely fix 34 issues, partially fix 1 issue, and acknowledge 15 issues.

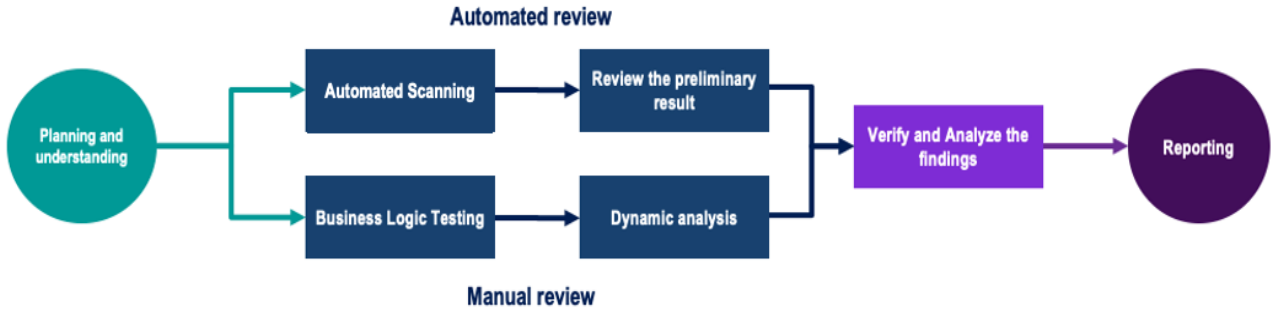
Below is the breakdown of the vulnerabilities found and their associated risk rating for each assessment conducted.

Target	Assessment Result					Reassessment Result				
	C	H	M	L	I	C	H	M	L	I
XTATUZ asset tokenization	8	11	17	4	10	1	5	7	1	2

Note: *Risk Rating* **C** *Critical,* **H** *High,* **M** *Medium,* **L** *Low,* **I** *Informational*

Methodology

The smart contract security audit methodology is based on Smart Contract Weakness Classification and Test Cases (SWC Registry), CWE, well-known best practices, and smart contract hacking case studies. Manual and automated review approaches can be mixed and matched, including business logic analysis in terms of the malicious doer's perspective. Using automated scanning tools to navigate or find offending software patterns in the codebase along with a purely manual or semi-automated approach, where the analyst primarily relies on one's knowledge, is performed to eliminate the false-positive results.



Planning and Understanding

- Determine the scope of testing and understanding of the application's purposes and workflows.
- Identify key risk areas, including technical and business risks.
- Determine which sections to review within the resource constraints and review method – automated, manual or mixed.

Automated Review

- Adjust automated source code review tools to inspect the code for known unsafe coding patterns.
- Verify the tool's output to eliminate false-positive results, and adjust and re-run the code review tool if necessary.

Manual Review

- Analyzing the business logic flaws requires thinking in unconventional methods.
- Identify unsafe coding behavior via static code analysis.

Reporting

- Analyze the root cause of the flaws.
- Recommend improvements for secure source code.

Audit Items

We perform the audit according to the following categories and test names.

Category	ID	Test Name
Security Issue	SEC01	*Authorization Through tx.origin*
	SEC02	*Business Logic Flaw*
	SEC03	*Delegatecall to Untrusted Callee*
	SEC04	*DoS With Block Gas Limit*
	SEC05	*DoS with Failed Call*
	SEC06	*Function Default Visibility*
	SEC07	*Hash Collisions With Multiple Variable Length Arguments*
	SEC08	*Incorrect Constructor Name*
	SEC09	*Improper Access Control or Authorization*
	SEC10	*Improper Emergency Response Mechanism*
	SEC11	*Insufficient Validation of Address Length*
	SEC12	*Integer Overflow and Underflow*
	SEC13	*Outdated Compiler Version*
	SEC14	*Outdated Library Version*
	SEC15	*Private Data On-Chain*
	SEC16	*Reentrancy*
	SEC17	*Transaction Order Dependence*
	SEC18	*Unchecked Call Return Value*
	SEC19	*Unexpected Token Balance*
	SEC20	*Unprotected Assignment of Ownership*
	SEC21	*Unprotected SELFDESTRUCT Instruction*
	SEC22	*Unprotected Token Withdrawal*
	SEC23	*Unsafe Type Inference*
	SEC24	*Use of Deprecated Solidity Functions*
	SEC25	*Use of Untrusted Code or Libraries*
	SEC26	*Weak Sources of Randomness from Chain Attributes*
	SEC27	*Write to Arbitrary Storage Location*

Category	ID	Test Name
Functional Issue	FNC01	*Arithmetic Precision*
	FNC02	*Permanently Locked Fund*
	FNC03	*Redundant Fallback Function*
	FNC04	*Timestamp Dependence*
Operational Issue	OPT01	*Code With No Effects*
	OPT02	*Message Call with Hardcoded Gas Amount*
	OPT03	*The Implementation Contract Flow or Value and the Document is Mismatched*
	OPT04	*The Usage of Excessive Byte Array*
	OPT05	*Unenforced Timelock on An Upgradeable Proxy Contract*
Developmental Issue	DEV01	*Assert Violation*
	DEV02	*Other Compilation Warnings*
	DEV03	*Presence of Unused Variables*
	DEV04	*Shadowing State Variables*
	DEV05	*State Variable Default Visibility*
	DEV06	*Typographical Error*
	DEV07	*Uninitialized Storage Pointer*
	DEV08	*Violation of Solidity Coding Convention*
	DEV09	*Violation of Token (ERC20) Standard API*

Risk Rating

To prioritize the vulnerabilities, we have adopted the scheme of five distinct levels of risk: **Critical**, **High**, **Medium**, **Low**, and **Informational**, based on OWASP Risk Rating Methodology. The risk level definitions are presented in the table.

Risk Level	Definition
Critical	The code implementation does not match the specification, and it could disrupt the platform.
High	The code implementation does not match the specification, or it could result in losing funds for contract owners or users.
Medium	The code implementation does not match the specification under certain conditions, or it could affect the security standard by losing access control.
Low	The code implementation does not follow best practices or use suboptimal design patterns, which may lead to security vulnerabilities further down the line.
Informational	Findings in this category are informational and may be further improved by following best practices and guidelines.

The **risk value** of each issue was calculated from the product of the **impact** and **likelihood values**, as illustrated in a two-dimensional matrix below.

- **Likelihood** represents how likely a particular vulnerability is exposed and exploited in the wild.
- **Impact** measures the technical loss and business damage of a successful attack.
- **Risk** demonstrates the overall criticality of the risk.

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

The shading of the matrix visualizes the different risk levels. Based on the acceptance criteria, the risk levels "Critical" and "High" are unacceptable. Any issue obtaining the above levels must be resolved to lower the risk to an acceptable level.

Findings

Review Findings Summary

The table below shows the summary of our assessments.

No.	Issue	Risk	Status	Functionality is in use
1	Inconsistent State After Presaled NFT Owners Transfer Tokens Directly	Critical	Fixed	In use
2	Unable To Pull Back Property NFTs	Critical	Fixed	In use
3	Lack Of Validating Master Token ID Leads To Unable To Minting Property NFTs	Critical	Fixed	In use
4	Lack Of Validating The Project Status	Critical	Fixed	In use
5	Function Calls Ordering Issues On Project Finalization Process	Critical	Fixed	In use
6	Out-Of-Sync Operator Transfer	Critical	Fixed	In use
7	Improper Project's Statuses	Critical	Fixed	In use
8	Potential Denial-Of-Service On Adding Project Member	Critical	Acknowledged	In use
9	Potential Denial-Of-Service On Retrieving Member NFTs	High	Acknowledged	In use
10	Potential Denial-Of-Service On Defragmentation	High	Acknowledged	In use
11	Potential Denial-Of-Service On Retrieving Token List	High	Acknowledged	In use
12	Use Of Incorrect Approval Function	High	Fixed	In use
13	Design Flaw In Directly Minting Presale NFTs	High	Fixed	In use
14	Design Flaw In Directly Burning Presale NFTs	High	Fixed	In use
15	Design Flaw In Directly Minting Property NFTs	High	Fixed	In use
16	State Inconsistency Upon Directly Claiming Property NFT Or Directly Refunding Presale NFT	High	Fixed	In use
17	Possibly Insufficient Referral Fee	High	Fixed	In use
18	Lack Of Updating State Variables Upon Claiming Property NFT Or Refunding Presale NFT	High	Acknowledged	In use
19	Possibly Unable To Finish Project	High	Acknowledged	In use

20	Arbitrarily Setting Presale Period	Medium	Fixed	In use
21	Possibly Setting Inconsistent Token URI	Medium	Acknowledged	In use
22	Lack Of Time Delay In Pullback Process	Medium	Fixed	In use
23	Potential Denial-Of-Service On Getting Member Projects	Medium	Acknowledged	In use
24	Potential Denial-Of-Service On Getting All Collections	Medium	Fixed	In use
25	Permanent Lock Of Ether Funds In Contracts	Medium	Fixed	In use
26	Double Registering Project Member	Medium	Fixed	In use
27	Usage Of Unsafe Token Transfer Functions	Medium	Fixed	In use
28	Double Spending On The finishProject Function	Medium	Fixed	In use
29	Unchecking Bounds Of Presale Token ID	Medium	Acknowledged	In use
30	Lack Of Resetting States On Operator And Trustee Transfers	Medium	Acknowledged	In use
31	Unsafe Privileged Role Transfer	Medium	Acknowledged	In use
32	Lack Of Proper Validation On The underwriteCount Parameter	Medium	Acknowledged	In use
33	Lack Of Upper Bound On Referral Percentage	Medium	Fixed	In use
34	Incorrectly Updating Referral Amount	Medium	Fixed	In use
35	Recommended Improving Transparency And Trustworthiness	Medium	Acknowledged	In use
36	Permanent Lock Of Referral Fee	Medium	Fixed	In use
37	Incorrect Time Verification On Extending Presale Period	Low	Fixed	In use
38	Potential Denial-Of-Service On Setting Reroll Data	Low	Acknowledged	In use
39	Compiler Is Not Locked To Specific Version	Low	Fixed	In use
40	Lack Of Validating Reroll Data	Low	Fixed	In use
41	Missing Validation Of Zero Address	Informational	Fixed	In use
42	Predictable Randomness	Informational	Acknowledged	In use
43	Recommended Improving Event Emissions	Informational	Fixed	In use
44	Recommended Improving Transparency And Traceability Of Crucial Variables	Informational	Partially Fixed	In use

45	Recommended Enforcing Checks-Effects-Interactions Pattern	Informational	Fixed	In use
46	Recommended Event Emissions For Transparency And Traceability	Informational	Fixed	In use
47	Recommended Removing Unused Code	Informational	Fixed	In use
48	Inconsistent Error Message With The Code	Informational	Fixed	In use
49	Misspelling State Variable	Informational	Fixed	In use
50	Inconsistent Interfaces With The Implementation	Informational	Fixed	In use

The statuses of the issues are defined as follows:

Fixed: The issue has been completely resolved and has no further complications.

Partially Fixed: The issue has been partially resolved.

Acknowledged: The issue's risk has been reported and acknowledged.

Detailed Result

This section provides all issues that we found in detail.

No. 1	Inconsistent State After Presaled NFT Owners Transfer Tokens Directly		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/XtatuzProject.sol*		
Locations	*XtatuzProject.sol L: 165 - 174*		

Detailed Issue

The *addProjectMember* function enables users to purchase presale tokens. Upon invocation, the function first checks the availability of tokens and subsequently mints the purchased tokens. The *getMemberedNFTList* state is then updated to reflect the changes. (L38, L385 in code snippet 1.2).

However, if a member transfers their token directly to another wallet, the *getMemberedNFTList* state will not be updated, which can cause inconsistencies in the *refund* function. The size of the *getMemberedNFTList* state will not decrease after the transfer, resulting in miscalculations in the payout fund for the refund (L172 in code snippet 1.3).

XtatuzProject.sol

```
 38  mapping(address => uint256[]) public getMemberedNFTList;

     // (...SNIPPED...)
120  function addProjectMember(address member_, uint256[] memory nftList_)
121      public
122      isAvailable
123      isLeftReserve
124      whenNotPaused
125      onlyOwner
126      returns (uint256)
127  {
128      uint256 amount = nftList_.length;
129      require(amount > 0, "PROJECT: ZERO_AMOUNT");
130      for(uint i = 0; i < amount; ++i){
131          require(nftList_[i] <= count, "PROJECT: ID_OVER_FRAGMENT");
```

```
132     }
133     _checkAvailableNFT(nftList_);
134 
135     uint256 price = minPrice * amount;
136     projectValue += price;
137 
138     _pickupAvailableNFT(nftList_, member_);
139 
140     countReserve -= amount;
141 
142     projectMember.push(member_);
143 
144     IPresaled(_presaledAddress).mint(member_, nftList_);
145     emit AddProjectMember(projectId, member_, price);
146 
147     return price;
148 }
```

Listing 1.1 The *addProjectMember* function

XtatuzProject.sol

```
382 function _pickupAvailableNFT(uint256[] memory nftList_, address member_)
    internal {
383     for (uint256 i = 0; i < nftList_.length; i++) {
384         unavailableNFT.push(nftList_[i]);
385         getMemberedNFTList[member_].push(nftList_[i]);
386     }
387 }
```

Listing 1.2 The *_pickupAvailableNFT* functions

XtatuzProject.sol

```
167 function refund(address member_) public onlyOperator whenNotPaused{
168         uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
169         require(projectStatus() == IXtatuzProject.Status.REFUND, "PROJECT:
    PROJECT_UNREFUNDED");
170 
171         IPresaled(_presaledAddress).burn(tokenList);
172 
173         uint256 totalToken = getMemberedNFTList[member_].length * minPrice;
174         IERC20(tokenAddress).transfer(member_, totalToken);
175 }
```

Listing 1.3 The *refund* function

Recommendations

We recommend implementing a hook function that disables the transfer feature for presale tokens. This would help ensure that presale tokens cannot be transferred, thereby preventing potential issues with state consistency.

Presaled.sol

```
100 function _beforeTokenTransfer(address from, address to, uint256 tokenId)
101     internal
102     override(ERC721Enumerable) {
103     super._beforeTokenTransfer(from, to, tokenId);
104     require(from == address(0) || to == address(0), "PRESALED:
    UNABLE_TO_TRANSFER");
105 }
```

Listing 1.4 Improved *Presaled* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 2	Unable To Pull Back Property NFTs		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/Property.sol* *contract/XtatuzRouter.sol*		
Locations	*Property.sol L: 65 - 76* *XtatuzRouter.sol L: 288 - 299*		

Detailed Issue

The *pullbackInactive* function enables SPV accounts to retrieve property tokens from inactive wallets (L295 in code snippet 2.2). This is made possible because the *mintFragment* function grants approval to the SPV account during the minting process (L74 in code snippet 2.1).

However, if a member transfers their tokens directly to another wallet, the approval for the recipient wallet will not be automatically granted to the SPV account (L144 in code snippet 2.3), which can cause issues when the SPV account attempts to invoke the *pullbackInactive* function.

As a result, the transaction will revert, and the SPV account will be unable to pull back tokens from the inactive wallet (L181 in code snippet 2.4).

Property.sol

```
65 function mintFragment(address to, uint256[] memory tokenIdList) public
   onlyOperator {
66     require(ownerOf(0) == address(this), "PROPERTY: NO_MASTER_NFT");
67     require(isMintedMaster == true, "PROPERTY: MASTER_NOT_MINTED");
68     uint256 amount = tokenIdList.length;
69     for (uint256 index = 0; index < amount; index++) {
70         uint256 tokenId = tokenIdList[index];
71         require(_exists(tokenId) == false, "PROJECT: ALREADY_EXITS");
72         _safeMint(to, tokenId);
73     }
74     _setApprovalForAll(to, _routerAddress, true);
75     emit MintFragment(to, tokenIdList);
76 }
```

Listing 2.1 The *mintFragment* function

XtatuzRouter.sol

```
288 function pullbackInactive(uint256 projectId_, address inactiveWallet_) public
    onlySpv {
289     require(_isNotice[projectId_][inactiveWallet_] == true, "ROUTER:
    NOTICE_BEFORE");
290     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
291     IProperty property = IProperty(propertyAddress);
292     uint256[] memory nftList = property.getTokenIdList(inactiveWallet_);
293
294     for(uint i = 0; i < nftList.length; i++){
295         IERC721(propertyAddress).safeTransferFrom(inactiveWallet_, msg.sender,
    nftList[i]);
296     }
297
298     emit PullbackInactive(projectId_, inactiveWallet_);
299 }
```

Listing 2.2 The *pullbackInactive* function

ERC721.sol

```
143 function isApprovedForAll(address owner, address operator) public view
    virtual override returns (bool) {
144     return _operatorApprovals[owner][operator];
145 }
```

Listing 2.3 The *isApprovedForAll* function

ERC721.sol

```
175 function safeTransferFrom(
176     address from,
177     address to,
178     uint256 tokenId,
179     bytes memory data
180 ) public virtual override {
181     require(_isApprovedOrOwner(_msgSender(), tokenId), "ERC721: caller is not
    token owner nor approved");
182     _safeTransfer(from, to, tokenId, data);
183 }

    // (...SNIPPED...)

232 function _isApprovedOrOwner(address spender, uint256 tokenId) internal view
    virtual returns (bool) {
233     address owner = ERC721.ownerOf(tokenId);
234     return (spender == owner || isApprovedForAll(owner, spender) ||
    getApproved(tokenId) == spender);
```

```
235 }
```

Listing 2.4 The *safeTransferFrom and _isApprovedOrOwner* function

Recommendations

We recommend including a hook function that allows *_routerAddress* to perform tokenId activities after the token has been transferred. This means that whenever a token owner transfers a token to another account, this hook function is invoked, the *_routerAddress* will always be granted approval to perform tokenId activities.

Property.sol

```
146 function _afterTokenTransfer(
147     address from,
148     address to,
149     uint256 tokenId
150 ) internal override(ERC721) {
151     super._afterTokenTransfer(from, to, tokenId);
152     _setApprovalForAll(to, _routerAddress, true);
153 }
```

Listing 2.5 Improved *Property* contract

Please note that although we recommend implementing the hook function as described, there may still be a scenario that cannot be resolved. Specifically, if a token receiver revokes their own approval, the router's approval to operate the token will also be revoked, which could potentially lead to a malfunction of the *pullbackInactive* function. Nevertheless, by implementing the recommended hook function, you can help ensure proper authorization for most scenarios involving token transfers.

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 3	Lack Of Validating Master Token ID Leads To Unable To Minting Property NFTs		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzProject.sol* *contracts/Presaled.sol* *contracts/Property.sol*		
Locations	*XtatuzProject.sol L: 108 - 138* *Presaled.sol L: 43 - 55* *Property.sol L: 43 - 55*		

Detailed Issue

The *Presaled* contract represents the presale tokens when the project status is available, and the *Property* contract represents the property fragment tokens when the project status is finished.

When the project is finished, the *claim* function of the *XtatuzProject* contract (L162 in code snippet 3.1) allows the operator to claim their fragments tokens by **burning their presale tokens and then minting the fragment tokens with the same token list** (L161 - 162 in code snippet 3.1).

We noticed that the *mintFragment* function of the *Property* contract (L65 - 76 in code snippet 3.2) allows the operator to mint any token IDs except the **token ID 0, which is reserved to represent the master token that is owned by the *Property* contract** (L66 in code snippet 21.2) and must exist before the claim process.

The issue occurs when invoking the *claim* function if the *tokenList* (L151 in code snippet 3.1) contains the token ID 0, the invocation of the *mintFragment* function (L162 in code snippet 3.1) would be reverted because the token ID 0 token already exists (L71 in code snippet 3.2).

The root cause of this issue is due to the *mint* function of the *Presaled* contract (L43 - 55 in code snippet 3.3) allows the operator to mint the token ID 0, which crashes the master token of the *Property* contract later.

XtatuzProject.sol

```
150 function claim(address member_) public onlyOperator whenNotPaused {
151     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
152     require(tokenList.length > 0, "PROJECT: TOKENLIST_ZERO");
153     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
154
155     IProperty property = IProperty(_propertyAddress);
156     IPresaled presaled = IPresaled(_presaledAddress);
157
158     bool isMintedMaster = property.isMintedMaster();
159     require(isMintedMaster == true, "PROJECT: MASTER_NOT_MINTED");
160
161     presaled.burn(tokenList);
162     property.mintFragment(member_, tokenList);
163 }
```

Listing 3.1 The *claim* function that burns the presale token and mint the property fragment

Property.sol

```
65  function mintFragment(address to, uint256[] memory tokenIdList) public
    onlyOperator {
66      require(ownerOf(0) == address(this), "PROPERTY: NO_MASTER_NFT");
67      require(isMintedMaster == true, "PROPERTY: MASTER_NOT_MINTED");
68      uint256 amount = tokenIdList.length;
69      for (uint256 index = 0; index < amount; index++) {
70          uint256 tokenId = tokenIdList[index];
71          require(_exists(tokenId) == false, "PROJECT: ALREADY_EXITS");
72          _safeMint(to, tokenId);
73      }
74      _setApprovalForAll(to, _routerAddress, true);
75      emit MintFragment(to, tokenIdList);
76  }
```

Listing 3.2 The *mintFragment* function that mint the property fragment

Presaled.sol

```
43  function mint(address to, uint256[] memory tokenIdList_) public onlyOperator {
44      require(to != address(0), "Presaled: Reciever address is address 0");
45      uint256 amount = tokenIdList_.length;
46      for (uint256 index = 0; index < amount; index++) {
47          uint256 tokenId = tokenIdList_[index];
48          require(_exists(tokenId) == false, "Presaled: TokenId already exits");
49          _safeMint(to, tokenId);
50          _mintedTimestamp[tokenId] = block.timestamp;
51          _tokenIdCount.increment();
52      }
53      setApprovalForAll(_operator, true);
54      emit Minted(to, tokenIdList_, tokenIdList_.length);
55  }
```

Listing 3.3 The *mint* function
that lack of validated token ID 0

Recommendations

We recommend adding the ***require*** statement (L48 in the code snippet below) to prevent the operator from mistakenly minting the token ID 0 on the *Presaled* contract.

Presaled.sol

```
43  function mint(address to, uint256[] memory tokenIdList_) public onlyOperator {
44      require(to != address(0), "Presaled: Reciever address is address 0");
45      uint256 amount = tokenIdList_.length;
46      for (uint256 index = 0; index < amount; index++) {
47          uint256 tokenId = tokenIdList_[index];
48          require(tokenId > 0, "PROJECT: NOT_ALLOWED_TO_MINT_MASTER");
49          require(_exists(tokenId) == false, "Presaled: TokenId already exits");
50          _safeMint(to, tokenId);
51          _mintedTimestamp[tokenId] = block.timestamp;
52          _tokenIdCount.increment();
53      }
54      setApprovalForAll(_operator, true);
55      emit Minted(to, tokenIdList_, tokenIdList_.length);
56  }
```

Listing 3.4 The improved *mint* function
that not allow to mint token ID 0

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 4	Lack Of Validating The Project Status		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzRouter.sol* *contracts/XtatuzProject.sol*		
Locations	*XtatuzRouter.sol L: 108 - 138* *XtatuzProject.sol L:120 - 148*		

Detailed Issue

A user can purchase the specified presale tokens using the *addProjectMember* function of the *XtatuzRouter* contract as shown in the code snippet 4.1.

However, we discovered that **although the specified *Xtatuz* project was already finished, a user could still buy the presale tokens of that *Xtatuz* project** since there is no validation of the project status, as shown in the code snippet 4.2, resulting in the inconsistency of the associated contract states.

XtatuzRouter.sol

```
108 function addProjectMember(
109     uint256 projectId_,
110     uint256[] memory nftList_,
111     string memory referral_
112 ) public {
113     uint256 amount = nftList_.length;
114     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
115     IXtatuzProject project = IXtatuzProject(projectAddress);
116     uint256 price = project.addProjectMember(msg.sender, nftList_);
117
118     uint256 minPrice = project.minPrice();
119     IXtatuzReferral referralContract = IXtatuzReferral(_referralAddress);
120     referralContract.increaseBuyerRef(projectId_, referral_, amount * minPrice);
121
122     address tokenAddress = project.tokenAddress();
123     IERC20(tokenAddress).transferFrom(msg.sender, projectAddress, price);
124
125     uint256[] memory memberedProject = _memberdProject[msg.sender];
126     bool foundedIndex;
127     for (uint256 index = 0; index < memberedProject.length; index++) {
128         if (memberedProject[index] == projectId_) {
```

```
            foundedIndex = true;
        }
    }
    if (!foundedIndex) {
        _memberdProject[msg.sender].push(projectId_);
    }

    _isMemberClaimed[msg.sender][projectId_] = false;
    emit AddProjectMember(projectId_, msg.sender, referral_, price);
}
```

Listing 4.1 The *addProjectMember* function of the *XtatuzRouter* contract

XtatuzProject.sol

```
function addProjectMember(address member_, uint256[] memory nftList_)
    public
    isAvailable
    isLeftReserve
    whenNotPaused
    onlyOwner
    returns (uint256)
{
    uint256 amount = nftList_.length;
    require(amount > 0, "PROJECT: ZERO_AMOUNT");
    for(uint i = 0; i < amount; ++i){
        require(nftList_[i] <= count, "PROJECT: ID_OVER_FRAGMENT");
    }
    _checkAvailableNFT(nftList_);

    uint256 price = minPrice * amount;
    projectValue += price;

    _pickupAvailableNFT(nftList_, member_);

    countReserve -= amount;

    projectMember.push(member_);

    IPresaled(_presaledAddress).mint(member_, nftList_);
    emit AddProjectMember(projectId, member_, price);

    return price;
}
```

Listing 4.2 The *addProjectMember* function of the *XtatuzProject* contract

Recommendations

We recommend **validating the project status** in the *addProjectMember* function of the *XtatuzProject* contract to prevent adding a project member after the project has finished as shown in the code snippet below.

XtatuzProject.sol

```
120 function addProjectMember(address member_, uint256[] memory nftList_)
121     public
122     isAvailable
123     isLeftReserve
124     whenNotPaused
125     onlyOwner
126     returns (uint256)
127 {
128     require(projectStatus() == IXtatuzProject.Status.AVAILABLE, "PROJECT:
    PROJECT_UNAVIALABLE");
129     uint256 amount = nftList_.length;
130     require(amount > 0, "PROJECT: ZERO_AMOUNT");
131     for(uint i = 0; i < amount; ++i){
132         require(nftList_[i] <= count, "PROJECT: ID_OVER_FRAGMENT");
133     }
134     _checkAvailableNFT(nftList_);
135 
136     uint256 price = minPrice * amount;
137     projectValue += price;
138 
139     _pickupAvailableNFT(nftList_, member_);
140 
141     countReserve -= amount;
142 
143     projectMember.push(member_);
144 
145     IPresaled(_presaledAddress).mint(member_, nftList_);
146     emit AddProjectMember(projectId, member_, price);
147 
148     return price;
149 }
```

Listing 4.3 The improved *addProjectMember* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team fixed this issue as per our suggestion.

No. 5	Function Calls Ordering Issues On Project Finalization Process		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/XtatuzProject.sol* *contract/Property.sol*		
Locations	*XtatuzProject.sol L: 175 - 188, L: 197 - 204* *Property.sol L: 46 - 51*		

Detailed Issue

The *finishProject* function enables the only operator to finalize the project by invoking the function, which triggers the distribution of funds to the *project owner*, *referral contract*, and *Xtatuz wallet*.

However, we identified two malfunctions related to the finalization of the project that could occur during the operation.

1. **Unable to finish the project.**
    - In this case, **if the *operator* and *trustee* call the *_multiSigMint* function before calling the *finishProject* function**, the *finishProject* function will be unable to be called, and the transaction will revert. The root cause of this issue is that the condition on ***_multiSigMint[_operatorAddress] && _multiSigMint[_trusteeAddress]*** will pass (L200 in code snippet 5.2), **causing the master token to be minted to the system**. As a result, when the *operator* attempts to invoke the *finishProject* function after that, the transaction will revert because the master token has already been minted (L47 in code snippet 5.3), and the funds will not be distributed to the beneficiaries.

2. **Fund transferred without minting master token**
    - In this case, **if the *operator* calls the *finishProject* function before calling the *multiSigMint* function**, the funds will be distributed to the beneficiaries immediately without minting the master token. However, this can lead to inconsistencies in the system in the event that the operator and trustee incorrectly call the *multiSigMint* function.

In both of the malfunctions we described, the issue was caused by a problem with the order of function calls.

XtatuzProject.sol

```
175 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
176     address referralAddress = IXtatuzRouter(owner()).referralAddress();
177
178     isFinished = true;
179     multiSigMint();
180
181     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
182     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
183     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
184     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
185     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
186
187     emit FinishProject(projectId, xtatuzWallet_);
189 }
```

Listing 5.1 The *finishProject* function

XtatuzProject.sol

```
197 function multiSigMint() public isFullReserve spvAndTrustee {
198     _multiSigMint[msg.sender] = true;
199
200     if (_multiSigMint[_operatorAddress] && _multiSigMint[_trusteeAddress]) {
201         IProperty(_propertyAddress).mintMaster();
202         checkCanClaim = true;
203     }
204 }
```

Listing 5.2 The *multiSigMint* function

Property.sol

```
46 function mintMaster() public onlyOwner {
47     require(isMintedMaster == false, "PROPERTY: MASTER_MINTED");
48     _mint(address(this), 0);
49     isMintedMaster = true;
50     emit MasterMinted(msg.sender);
51 }
```

Listing 5.3 The *mintMaster* function

Recommendations

To address the malfunctions related to the finalization of the project, we recommend enforcing the correct order of function calls. First, the trustee should call the *_multiSigMint* function to set their approval for minting the master token. Then, the operator should call the *finishProject* function to minting the master token and distribute funds to the beneficiaries. This order of function calls would ensure that the master token is minted before the funds are distributed, preventing inconsistencies in the system.

XtatuzProject.sol

```
177 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
178     _multiSigMint[msg.sender] = true;
179     require(_multiSigMint[_operatorAddress] && _multiSigMint[_trusteeAddress],
    "PROJECT: NOT_ALLOWS_BY_MULTISIGMINT");
180
181     isFinished = true;
182     checkCanClaim = true;
183     IProperty(_propertyAddress).mintMaster();
184
185     address referralAddress = IXtatuzRouter(owner()).referralAddress();
186     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
187     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
188     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
189     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
190     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
191
192     emit FinishProject(projectId, xtatuzWallet_);
193 }
```

Listing 5.4 The improved *finishProject* function

XtatuzProject.sol

```
202 function multiSigMint() public isFullReserve spvAndTrustee {
203     _multiSigMint[msg.sender] = true;
204 }
```

Listing 5.5 The improved *multiSigMint* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 6	Out-Of-Sync Operator Transfer		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzProject.sol*		
Locations	*XtatuzProject.sol L: 265 - 267 and L: 299 - 303*		

Detailed Issue

We noticed that the *Presaled*, *Property*, and *XtatuzProject* contracts introduce an operator role, which enables the management of crucial functionalities (as shown in code snippets 6.2 and 6.3).

However, we considered a potential issue whereby, in the event that the operator transfers their role to a new operator address on the *XtatuzProject* contract via the *transferOperator* function, the operator address in the *Presaled* and *Property* contracts will not be updated accordingly.

As a result, **the old operator will still have access to operate on the *Presaled* and *Property* contracts even after the *XtatuzProject* contract has transferred the operator role to the new operator**.

XtatuzProject.sol

```
265 function transferOperator(address newOperator_) public whenNotPaused
    onlyOperator {
266     _transferOperator(newOperator_);
267 }

    //(...SNIPPED...)

299 function _transferOperator(address newOperator_) internal
    ProhibitZeroAddress(newOperator_) {
300     address prevOperator = _operatorAddress;
301     _operatorAddress = newOperator_;
302     emit OperatorTransfered(prevOperator, newOperator_);
303 }
```

Listing 6.1 The *transferOperator* function of the *XtatuzProject* contract

Property.sol

```
14  constructor(
15      string memory name_,
16      string memory symbol_,
17      address operator_,
18      address routerAddress_,
19      uint256 count_
20  ) ERC721(name_, symbol_) {
21      _setOperator(operator_);
22      setPropertyStatus(IProperty.PropertyStatus.INCOMPLETE);
23      _routerAddress = routerAddress_;
24      count = count_;
25  }
26
27  address private _operator;

    //(...SNIPPED...)

65  function mintFragment(address to, uint256[] memory tokenIdList) public
    onlyOperator {
66      require(ownerOf(0) == address(this), "PROPERTY: NO_MASTER_NFT");
67      require(isMintedMaster == true, "PROPERTY: MASTER_NOT_MINTED");
68      uint256 amount = tokenIdList.length;
69      for (uint256 index = 0; index < amount; index++) {
70          uint256 tokenId = tokenIdList[index];
71          require(_exists(tokenId) == false, "PROJECT: ALREADY_EXITS");
72          _safeMint(to, tokenId);
73      }
74      _setApprovalForAll(to, _routerAddress, true);
75      emit MintFragment(to, tokenIdList);
76  }
```

Listing 6.2 The example operator privilege of the *Property* contract

Presaled.sol

```
12  address private _operator;

    //(...SNIPPED...)

19  constructor(
20      string memory _name,
21      string memory _symbol,
22      uint256 count_,
23      address operator_,
24      address routerAddress_
25  ) ERC721(_name, _symbol) {
26      _setOperator(operator_);
27      _tokenIdCount.increment();
```

```
28      _routerAddress = routerAddress_;
29      presaleIdList = new uint256[](count_);
30      for(uint256 index = 0; index < count_; index++){
31          presaleIdList[index] = index + 1;
32      }
33  }

    //(...SNIPPED...)

57  function burn(uint256[] memory tokenIdList_) public onlyOperator {
58      for (uint256 index = 0; index < tokenIdList_.length; index++) {
59          _burn(tokenIdList_[index]);
60      }
61      emit Burned(tokenIdList_);
62  }
```

Listing 6.3 The example operator privilege of the *Presaled* contract

Recommendations

Since there is the need for careful consideration and management of operator roles across the *Presaled*, *Property,* and *XtatuzProject* contracts.

We recommend **adding the logic to update the operator address** on the *Presaled* and *Property* contracts whenever the *transferOperator* function is invoked on the *XtatuzProject* contract.

XtatuzProject.sol

```
299 function _transferOperator(address newOperator_) internal
    ProhibitZeroAddress(newOperator_) {
300     address prevOperator = _operatorAddress;
301     _operatorAddress = newOperator_;
302     IPresaled(_presaledAddress).setOperator(newOperator_);
303     IProperty(_propertyAddress).setOperator(newOperator_);
304     emit OperatorTransfered(prevOperator, newOperator_);
305 }
```

Listing 6.4 The improved *_transferOperator* function of the *XtatuzProject* contract

Presaled.sol

```
104 function setOperator(address operator_) external onlyOwner{
105     _setOperator(operator_);
106 }
```

Listing 6.5 The new external *setOperator* function of the *Presaled* contract

Property.sol

```
104 function setOperator(address operator_) external onlyOwner{
105     _setOperator(operator_);
106 }
```

Listing 6.6 The new external *setOperator* function of the *Property* contract

Furthermore, **there are additional details that require updating in conjunction with this recommendation, specifically pertaining to the interface of the *Presaled* and *Property* contract**.

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 7	Improper Project's Statuses		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzProject.sol* *contracts/Property.sol*		
Locations	*XtatuzProject.sol* • *L: 226 - 236 (projectStatus function),* • *L: 150 - 163 (claim function),* • *L: 165 - 173 (refund function),* • *L: 175 - 189 (finishProject function),* *Property.sol L: 65 - 76*		

Detailed Issue

We found improper project status considerations that create three issues

1. The ***FINISH*** status was not considered in the case of the ***_underwriteCount >= countReserve***, resulting in the project status falling into the ***REFUND*** instead of the ***FINISH*** (L229 - 230 in code snippet 7.1).
2. When the project status is ***REFUND***, the user could call the *refund* function to withdraw their tokens, while the operator could still call the *finishProject* function to finish the project simultaneously.
3. Even if the project status is ***REFUND***, the operator could still call the *finishProject* function to finish the project well.

XtatuzProject.sol

```
226 function projectStatus() public view returns (IXtatuzProject.Status status) {
227     if (!isFinished && block.timestamp >= startPresale && block.timestamp <=
    endPresale && countReserve > 0) {
228         return IXtatuzProject.Status.AVAILABLE;
229     } else if (countReserve == 0 || isFinished) {
230         return IXtatuzProject.Status.FINISH;
231     } else if (block.timestamp > endPresale && countReserve > 0) {
232         return IXtatuzProject.Status.REFUND;
233     } else {
234         return IXtatuzProject.Status.UNAVAILABLE;
235     }
236 }
```

Listing 7.1 The improper project status consideration

To understand the first issue, please consider the scenario below.

1. The project status shows up as ***REFUND*** even if it should be marked as ***FINISH***, resulting in the user could refund their token instead of the operator could finish the project.

   The root cause of this issue is that the ***FINISH*** status condition (L229 in code snippet 7.1) was not considered in the case of the ***_underwriteCount >= countReserve*** (L350 in code snippet 7.2), resulting in the project status falling into the ***REFUND*** (L231 in code snippet 7.1) instead.

XtatuzProject.sol

```
70  modifier isFullReserve() {
71      _checkIsFullReserve();
72      _;
73  }

    // (...SNIPPED...)

175 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
176     address referralAddress = IXtatuzRouter(owner()).referralAddress();
177
178     isFinished = true;
179     multiSigMint();
180
181     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
182     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
183     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
184     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
185     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
186
187     emit FinishProject(projectId, xtatuzWallet_);
188 }

    // (...SNIPPED...)

349 function _checkIsFullReserve() internal view {
350     require(_underwriteCount >= countReserve, "PROJECT: NOT_FULL");
351 }
```

Listing 7.2 The *finishProject* project function that lack of checking project status

To understand the second issue, please consider the scenario below.

1. From the first scenario result, if the project status falls to ***REFUND***, it will cause two sub issues.

1.1 If a user calls the ***refund*** function first (code snippet 7.3) and the operator then tries to invoke the ***finishProject*** function afterwards (code snippet 7.4), there may be insufficient funds in the project to pay the beneficiaries. Therefore, the transaction to finish the project would be reverted (L181 - 185 in code snippet 7.4).

1.2 Although the project status falls to ***REFUND***, the operator can still call the ***finishProject*** function (code snippet 7.4). Consequently, the project status will be changed to ***FINISH***, which will cause users to be unable to refund their tokens later.

The root cause of this scenario is the *finishProject* function (code snippet 7.4) does not check the project status properly before performing function functionality.

XtatuzProject.sol

```
165 function refund(address member_) public onlyOperator whenNotPaused{
166     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
167     require(projectStatus() == IXtatuzProject.Status.REFUND, "PROJECT:
    PROJECT_UNREFUNDED");
168
169     IPresaled(_presaledAddress).burn(tokenList);
170
171     uint256 totalToken = getMemberedNFTList[member_].length * minPrice;
172     IERC20(tokenAddress).transfer(member_, totalToken);
173 }
```

Listing 7.3 The *refund* function that check the project status is *REFUND*

XtatuzProject.sol

```
175 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
176     address referralAddress = IXtatuzRouter(owner()).referralAddress();
177
178     isFinished = true;
179     multiSigMint();
180
181     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
182     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
183     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
184     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
185     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
186
```

```
187     emit FinishProject(projectId, xtatuzWallet_);
188 }
```

Listing 7.4 The *finishProject* project function that lack of checking project status

To understand the third issue, please consider the scenario below.

1. The project status falling to ***FINISH*** from the condition of ***countReserve == 0*** (L229 in code snippet 7.1) is met.

   After this step, the users could call the *claim* function immediately because the require statement ***require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT: PROJECT_UNFINISH")*** at line 153 in code snippet 7.5 is passed.

2. The user calls the ***claim*** function (code snippet 7.5) to claim their property tokens.

   In this step, if the operator did not invoke the ***finishProject*** function to mint the master token before, the transaction would be reverted due to the ***require*** statement (L159 in code snippet 7.5).

XtatuzProject.sol

```
150 function claim(address member_) public onlyOperator whenNotPaused {
151     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
152     require(tokenList.length > 0, "PROJECT: TOKENLIST_ZERO");
153     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
154
155     IProperty property = IProperty(_propertyAddress);
156     IPresaled presaled = IPresaled(_presaledAddress);
157
158     bool isMintedMaster = property.isMintedMaster();
159     require(isMintedMaster == true, "PROJECT: MASTER_NOT_MINTED");
160
161     presaled.burn(tokenList);
162     property.mintFragment(member_, tokenList);
163 }
```

Listing 7.5 The *claim* function that allows claiming token when the project is finished

Recommendations

We recommend adding the ***PREPARE_FINISH*** status (L12 in code snippet 7.6) to cover in case the project could finish and waiting for the operator to completely finish the project. Then, revise all the project status considerations in the *projectStatus* function (code snippet 7.7) to consider all possible project statuses.

Finally, adding the ***require*** statement (L176 in code snippet 7.8) to ensure that the operator could call the *finishProject* function only if the project status is ***PREPARE_FINISH***.

We provide recommended code to address this issue as a suggested remediation concept only. We recommend that the *Xtatuz* team should adjust the recommendation code according to the business design.

IXtatuzProject.sol

```
6  interface IXtatuzProject {
7      enum Status {
8          AVAILABLE,
9          FINISH,
10         REFUND,
11         UNAVAILABLE,
12         PREPARE_FINISH
13     }

       // (...SNIPPED...)
```

Listing 7.6 The improved *Status* enum

XtatuzProject.sol

```
226 function projectStatus() public view returns (IXtatuzProject.Status status) {
227     if (isFinished) {
228         return IXtatuzProject.Status.FINISH;
229     } else if (countReserve == 0 || _underwriteCount >= countReserve) {
230         return IXtatuzProject.Status.PREPARE_FINISH;
231     } else if (block.timestamp > endPresale && countReserve > 0 &&
    !(_underwriteCount >= countReserve)) {
232         return IXtatuzProject.Status.REFUND;
233     } else if (!isFinished && block.timestamp >= startPresale && block.timestamp
    <= endPresale) {
234         return IXtatuzProject.Status.AVAILABLE;
235     } else {
236         return IXtatuzProject.Status.UNAVAILABLE;
237     }
238 }
```

Listing 7.7 The improved *projectStatus* function that consider all possible project statuses

XtatuzProject.sol

```
175 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
176     require(projectStatus() == IXtatuzProject.Status.PREPARE_FINISH, "PROJECT:
    NOT_READY_TO_FINISH");
177     address referralAddress = IXtatuzRouter(owner()).referralAddress();
178
179     isFinished = true;
180     multiSigMint();
181
182     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
183     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
184     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
185     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
186     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
187
188     emit FinishProject(projectId, xtatuzWallet_);
189 }
```

Listing 7.8 The improved *finishProject* function that
checks the project status before performing the finalization process

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 8	Potential Denial-Of-Service On Adding Project Member		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/XtatuzProject.sol*		
Locations	*XtatuzProject.sol L: 120 - 148, 238 - 240, and 370 - 380*		

Detailed Issue

The *addProjectMember* function of the *XtatuzProject* (code snippet 8.1) allows the contract owner to add the new member along with minting the new unowned presale tokens to the given member address. This function contains the function calls to verify the given tokens list by calling the *_checkAvailableNFT* function and pushing the unowned token to the *unavailableNFT* array (L138 and 382 - 287 in code snippet 8.2).

The *_checkAvailableNFT* function has to iterate over all a token list and the *unavailableNFT* list, which tracks the owned token ID, to verify the given tokens (L370 - 380 in code snippet 8.2).

We noticed that the process of verifying the tokens list with the *unavailableNFT* array in the *_checkAvailableNFT* function can consume more gas than the block gas limit **if the number of owned tokens is too large, causing the transaction for adding project member to be reverted, leading to the denial-of-service issue.**

XtatuzProject.sol

```
120 function addProjectMember(address member_, uint256[] memory nftList_)
121     public
122     isAvailable
123     isLeftReserve
124     whenNotPaused
125     onlyOwner
126     returns (uint256)
127 {
128     uint256 amount = nftList_.length;
129     require(amount > 0, "PROJECT: ZERO_AMOUNT");
130     for(uint i = 0; i < amount; ++i){
131         require(nftList_[i] <= count, "PROJECT: ID_OVER_FRAGMENT");
132     }
133     _checkAvailableNFT(nftList_);
134
```

```
135     uint256 price = minPrice * amount;
136     projectValue += price;
137
138     _pickupAvailableNFT(nftList_, member_);
139
140     countReserve -= amount;
141
142     projectMember.push(member_);
143
144     IPresaled(_presaledAddress).mint(member_, nftList_);
145     emit AddProjectMember(projectId, member_, price);
146
147     return price;
148 }

    //(...SNIPPED...)

382 function _pickupAvailableNFT(uint256[] memory nftList_, address member_)
    internal {
383     for (uint256 i = 0; i < nftList_.length; i++) {
384         unavailableNFT.push(nftList_[i]);
385         getMemberedNFTList[member_].push(nftList_[i]);
386     }
387 }
```

Listing 8.1 The *addProjectMember* function of the *XtatuzProject* contract

XtatuzProject.sol

```
370 function _checkAvailableNFT(uint256[] memory nftList_) internal view {
371     for (uint256 i = 0; i < nftList_.length; i++) {
372         for (uint256 j = 0; j < unavailableNFT.length; j++) {
373             bool isAvailableNFT = false;
374             if (nftList_[i] != unavailableNFT[j]) {
375                 isAvailableNFT = true;
376             }
377             require(isAvailableNFT == true, "PROJECT: UNAVAILABLE_NFT_ID");
378         }
379     }
380 }
```

Listing 8.2 The *_checkAvailableNFT* function of the *XtatuzProject* contract

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the verification of unowned presale tokens mechanism.

Reassessment

The *Xtatuz* team acknowledged this issue. However, they tested the adding project member process with the maximum number of presale tokens (90 presale tokens) to ensure that their system can perform without creating a denial-of-service issue.

No. 9	Potential Denial-Of-Service On Retrieving Member NFTs		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/Presaled.sol* *contracts/XtatuzProjcet.sol* *contracts/XtatuzRouter.sol*		
Locations	*Presaled.sol L: 64 - 71* *XtatuzProjcet.sol L: 150 - 163, and 165 - 173* *XtatuzRouter.sol L: 215 - 247*		

Detailed Issue

The *getPresaledOwner* function is used in multiple functions across different contracts, listed below:

- The *claim* and *refund* functions of the *XtatuzProject* contract (code snippet 9.2).
  These functions are used to claim the property token when the presale project finishes successfully and refund if the presale project is decided to be a refunding phrase.

- The *getAllCollection* function of the *XtatuzRouter* contract (code snippet 9.3).
  This function is used to retrieve all the collections owned by the caller.

The *getPresaledOwner* function has to iterate over an owned token list of the given member in order to retrieve the corresponding token IDs (code snippet 9.1).

We noticed that this process of the *getPresaledOwner* function can consume more gas than the block gas limit, particularly if the given member has a large number of owned tokens. This excessive gas consumption may exceed the block gas limit and cause the transaction of the affected functions to be reverted, which can result in a denial-of-service issue.

XtatuzProject.sol

```
64  function getPresaledOwner(address owner) public view returns (uint256[] memory)
    {
65      uint256 balance = balanceOf(owner);
66      uint256[] memory tokenList = new uint256[](balance);
67      for (uint256 index = 0; index < balance; index++) {
68          tokenList[index] = tokenOfOwnerByIndex(owner, index);
69      }
70      return tokenList;
71  }
```

Listing 9.1 The *getPresaledOwner* function of the *Presaled* contract

XtatuzProject.sol

```
150 function claim(address member_) public onlyOperator whenNotPaused {
151     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
152     require(tokenList.length > 0, "PROJECT: TOKENLIST_ZERO");
153     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
154
155     IProperty property = IProperty(_propertyAddress);
156     IPresaled presaled = IPresaled(_presaledAddress);
157
158     bool isMintedMaster = property.isMintedMaster();
159     require(isMintedMaster == true, "PROJECT: MASTER_NOT_MINTED");
160
161     presaled.burn(tokenList);
162     property.mintFragment(member_, tokenList);
163 }
164
165 function refund(address member_) public onlyOperator whenNotPaused{
166     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
167     require(projectStatus() == IXtatuzProject.Status.REFUND, "PROJECT:
    PROJECT_UNREFUNDED");
168
169     IPresaled(_presaledAddress).burn(tokenList);
170
171     uint256 totalToken = getMemberedNFTList[member_].length * minPrice;
172     IERC20(tokenAddress).transfer(member_, totalToken);
173 }
```

Listing 9.2 The *claim* and *refund* functions of the *XtatuzProject* contract

XtatuzProject.sol

```
215 function getAllCollection() public view returns (Collection[] memory) {
216     uint256[] memory projectList = _memberdProject[msg.sender];
217     Collection[] memory collections = new Collection[](projectList.length);
218     for (uint256 index = 0; index < projectList.length; index++) {
219         if (projectList[index] > 0) {
220             uint256 projectId = projectList[index];
221             address projectAddress =
    _xtatuzFactory.getProjectAddress(projectId);
222             IXtatuzProject.Status status =
    IXtatuzProject(projectAddress).projectStatus();
223             if (status == IXtatuzProject.Status.FINISH &&
    _isMemberClaimed[msg.sender][projectId]) {
224                 address propertyAddress =
    _xtatuzFactory.getPropertyAddress(projectId);
225                 uint256[] memory tokenList =
    IProperty(propertyAddress).getTokenIdList(msg.sender);
226                 IProperty.PropertyStatus propStatus =
    IProperty(propertyAddress).propertyStatus();
227                 CollectionType collecType = CollectionType(uint256(propStatus) +
    1);
228                 Collection memory collect = Collection({
229                     contractAddress: propertyAddress,
230                     tokenIdList: tokenList,
231                     collectionType: collecType
232                 });
233                 collections[index] = collect;
234             } else {
235                 address presaledAddress =
    _xtatuzFactory.getPresaledAddress(projectId);
236                 uint256[] memory tokenList =
    IPresaled(presaledAddress).getPresaledOwner(msg.sender);
237                 Collection memory collect = Collection({
238                     contractAddress: presaledAddress,
239                     tokenIdList: tokenList,
240                     collectionType: CollectionType.PRESALE
241                 });
242                 collections[index] = collect;
243             }
244         }
245     }
246     return collections;
247 }
```

Listing 9.3 The *getAllCollection* function of the *XtatuzRouter* contract

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the getting owned presale tokens mechanism.

Reassessment

The *Xtatuz* team acknowledged this issue. However, they tested the retrieving member NFTs process with the maximum number of presale tokens (90 presale tokens) to ensure that their system can perform without creating a denial-of-service issue.

No. 10	Potential Denial-Of-Service On Defragmentation		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contract/Property.sol*		
Locations	*Property.sol L: 78 - 86*		

Detailed Issue

The *defragment* function is designed to allow the owner of all fragments (*Property* tokens) to claim the master token (*Property* token ID 0). During the defragmentation process, all fragments are burned, and the master token is transferred to the eligible sender.

However, **we noticed that if the number of fragments is too large (*count* state variable in code snippet 10.1), the gas required to burn them may exceed the block gas limit.** This can result in the defragment transaction being reverted, which would prevent the member from claiming the master token, leading to the denial-of-service issue.

Property.sol

```
78  function defragment() public {
79      require(balanceOf(msg.sender) == count, "PROPERTY:
    ONLY_ALL_FRAGMENT_OWNER");
80      for (uint256 i = 1; i <= count; i++) {
81          _burn(i);
82      }
83      _approve(msg.sender, 0);
84      safeTransferFrom(address(this), tx.origin, 0);
85      emit Defragment(msg.sender);
86  }
```

Listing 10.1 The *defragment* function

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the defragmentation process.

Reassessment

The *Xtatuz* team acknowledged this issue. However, they tested the defragmentation process with the maximum number of property tokens (90 property tokens) to ensure that their system can perform without creating a denial-of-service issue.

No. 11	Potential Denial-Of-Service On Retrieving Token List		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contract/Property.sol* *contract/XtatuzRouter.sol*		
Locations	*Property.sol L: 101 - 108* *XtatuzRouter.sol L: 215 - 247, L: 288 - 299*		

Detailed Issue

The *getTokenIdList* function is a convenient method used to retrieve a list of all *Property* tokens owned by a specific member.

This function is used in both the *getAllCollection* and *pullbackInactive* functions as part of their processes. Specifically, the *getAllCollection* function retrieves a list of tokens and sets it as part of the collection, while *pullbackInactive* retrieves a list of tokens and pulls them back to the SPV account.

However, **if the number of tokens retrieved by *getTokenIdList* is too large (L102 - 103 in code snippet 11.1), the gas required to use the *getAllCollection* and *pullbackInactive* functions may exceed the block gas limit.** This can cause the transaction to be reverted, leading to the denial-of-service issue.

Property.sol

```
101 function getTokenIdList(address member) public view returns (uint256[] memory) {
102     uint256 balance = balanceOf(member);
103     uint256[] memory tokenList = new uint256[](balance);
104     for (uint256 index = 0; index < balance; index++) {
105         tokenList[index] = tokenOfOwnerByIndex(member, index);
106     }
107     return tokenList;
108 }
```

Listing 11.1 The *getTokenIdList* function

XtatuzRouter.sol

```
215 function getAllCollection() public view returns (Collection[] memory) {
216     uint256[] memory projectList = _memberdProject[msg.sender];
217     Collection[] memory collections = new Collection[](projectList.length);
218     for (uint256 index = 0; index < projectList.length; index++) {
219         if (projectList[index] > 0) {
220             uint256 projectId = projectList[index];
221             address projectAddress =
    _xtatuzFactory.getProjectAddress(projectId);
222             IXtatuzProject.Status status =
    IXtatuzProject(projectAddress).projectStatus();
223             if (status == IXtatuzProject.Status.FINISH &&
    _isMemberClaimed[msg.sender][projectId]) {
224                 address propertyAddress =
    _xtatuzFactory.getPropertyAddress(projectId);
225                 uint256[] memory tokenList =
    IProperty(propertyAddress).getTokenIdList(msg.sender);
226                 IProperty.PropertyStatus propStatus =
    IProperty(propertyAddress).propertyStatus();
227                 CollectionType collecType = CollectionType(uint256(propStatus) +
    1);
228                 Collection memory collect = Collection({
229                     contractAddress: propertyAddress,
230                     tokenIdList: tokenList,
231                     collectionType: collecType
232                 });
233                 collections[index] = collect;
234             } else {
235                 address presaledAddress =
    _xtatuzFactory.getPresaledAddress(projectId);
236                 uint256[] memory tokenList =
    IPresaled(presaledAddress).getPresaledOwner(msg.sender);
237                 Collection memory collect = Collection({
238                     contractAddress: presaledAddress,
239                     tokenIdList: tokenList,
240                     collectionType: CollectionType.PRESALE
241                 });
242                 collections[index] = collect;
243             }
244         }
245     }
246     return collections;
247 }
```

Listing 11.2 The *getAllCollection* function

XtatuzRouter.sol

```
288 function pullbackInactive(uint256 projectId_, address inactiveWallet_) public
    onlySpv {
289     require(_isNotice[projectId_][inactiveWallet_] == true, "ROUTER:
    NOTICE_BEFORE");
290     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
291     IProperty property = IProperty(propertyAddress);
292     uint256[] memory nftList = property.getTokenIdList(inactiveWallet_);
293
294     for(uint i = 0; i < nftList.length; i++){
295         IERC721(propertyAddress).safeTransferFrom(inactiveWallet_, msg.sender,
    nftList[i]);
296     }
297
298     emit PullbackInactive(projectId_, inactiveWallet_);
299 }
```

Listing 11.3 The *pullbackInactive* function

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the mechanism for obtaining a list of token IDs from members, which will also impact the related mechanisms.

Reassessment

The *Xtatuz* team acknowledged this issue. However, they tested the mechanism for obtaining a list of token IDs from members process with the maximum number of property tokens (90 property tokens) to ensure that their system can perform without creating a denial-of-service issue.

No. 12	Use Of Incorrect Approval Function		
Risk	High	Likelihood	High
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/Presaled.sol*		
Locations	*Presaled.sol L: 43 - 55*		

Detailed Issue

The *mint* function enables the operator to mint *Presaled* tokens for the user using a list of valid token IDs. After minting, this function invokes the *setApprovalForAll* function (L53 in code snippet 12.1), which grants permission to the operator for managing all tokens owned by the user.

However, we found that the *mint* function uses the wrong function to grant approval to handle the *Presaled* tokens.

More specifically, when the *setApprovalForAll* function is invoked, the *_operator* is passed as a parameter to the function. **If the caller of the function is also the *_operator* account (e.g., if msg.sender is the *_operator* account), the condition *owner != operator* in the *_setApprovalForAll* function will not pass** (L372 in code snippet 12.3), causing the transaction to revert. This is because the *owner* and *operator* parameters are the same address (L368, 369 in code snippet 12.3), which means that the operator will not be granted approval to manage the presale tokens.

Presaled.sol

```
43  function mint(address to, uint256[] memory tokenIdList_) public onlyOperator {
44      require(to != address(0), "Presaled: Reciever address is address 0");
45      uint256 amount = tokenIdList_.length;
46      for (uint256 index = 0; index < amount; index++) {
47          uint256 tokenId = tokenIdList_[index];
48          require(_exists(tokenId) == false, "Presaled: TokenId already exits");
49          _safeMint(to, tokenId);
50          _mintedTimestamp[tokenId] = block.timestamp;
51          _tokenIdCount.increment();
52      }
53      setApprovalForAll(_operator, true);
54      emit Minted(to, tokenIdList_, tokenIdList_.length);
55  }
```

Listing 12.1 The *mint* function

Presaled.sol

```
35  modifier onlyOperator() {
36      _checkOperator();
37      _;
38  }

43  // (...SNIPPED...)
44
45  function _checkOperator() private view {
46      require(msg.sender == _operator || msg.sender == owner(), "PRESALED:
    PERMISSION_DENIED");
    }
```

Listing 12.2 The *_checkOperator* function

ERC721.sol

```
136 function setApprovalForAll(address operator, bool approved) public virtual
    override {
137     _setApprovalForAll(_msgSender(), operator, approved);
138 }

    // (...SNIPPED...)

367 function _setApprovalForAll(
368     address owner,
369     address operator,
370     bool approved
371 ) internal virtual {
372     require(owner != operator, "ERC721: approve to caller");
```

```
373         _operatorApprovals[owner][operator] = approved;
374         emit ApprovalForAll(owner, operator, approved);
375     }
```

Listing 12.3 The *setApprovalForAll and _setApprovalForAll* function

Recommendations

We recommend revising the *mint* function by calling the *_setApprovalForAll* function with the appropriate parameters. The *mint* function can pass the *member's address* as the "owner" parameter and *_operator* as the "operator" parameter, and set the "approved" parameter to *true*. This will grant approval to the operator to handle the member's tokens without causing the transaction to revert.

Presaled.sol

```
43  function mint(address to, uint256[] memory tokenIdList_) public onlyOperator {
44      require(to != address(0), "Presaled: Reciever address is address 0");
45      uint256 amount = tokenIdList_.length;
46      for (uint256 index = 0; index < amount; index++) {
47          uint256 tokenId = tokenIdList_[index];
48          require(_exists(tokenId) == false, "Presaled: TokenId already exits");
49          _safeMint(to, tokenId);
50          _mintedTimestamp[tokenId] = block.timestamp;
51          _tokenIdCount.increment();
52      }
53      _setApprovalForAll(to, _operator, true);
54      emit Minted(to, tokenIdList_, tokenIdList_.length);
55  }
```

Listing 12.4 The improved *mint* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 13	Design Flaw In Directly Minting Presale NFTs		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/Presaled.sol*		
Locations	*Presaled.sol L: 45 - 57 (*The *mint* function*)*		

Detailed Issue

The *mint* function in the *Presaled* Contract is only accessible by the operator and the contract owner, as demonstrated in lines 45 and 90 of the code snippet 13.1.

In order to mint presale tokens, the user must interact with the *XtatuzRouter* contract. The *XtatuzRouter* contract will then track the relevant state variables and make a call to the specified *XtatuzProject* contract, which will verify the conditions and handle the minting of the presale tokens. This includes updating the related state variables in the *XtatuzProject* contract before executing the mint function from the *Presaled* contract.

Therefore, It is imperative to adhere to the business conditions and accurately update the crucial state variables in the associated contracts when minting presale tokens.

Our investigation revealed that direct invocation of the *mint* function can result in several issues, such as

- **Mistracking of the state variables used** in the *XtatuzProject* and *XtatuzRouter* contracts.
- **Unbounded token ID** from the maximum supply and **incorrect token ID minting**.
- **Bypassing the business conditions** for minting presale tokens.

Presaled.sol

```
37  modifier onlyOperator() {
38      _checkOperator();
39      _;
40  }

    //(...SNIPPED...)

45  function mint(address to, uint256[] memory tokenIdList_) public onlyOperator {
46      require(to != address(0), "Presaled: Reciever address is address 0");
47      uint256 amount = tokenIdList_.length;
48      for (uint256 index = 0; index < amount; index++) {
49          uint256 tokenId = tokenIdList_[index];
50          require(_exists(tokenId) == false, "Presaled: TokenId already exits");
51          _safeMint(to, tokenId);
52          _mintedTimestamp[tokenId] = block.timestamp;
53          _tokenIdCount.increment();
54      }
55      setApprovalForAll(_operator, true);
56      emit Minted(to, tokenIdList_, tokenIdList_.length);
57  }

    //(...SNIPPED...)

89  function _checkOperator() private view {
90      require(msg.sender == _operator || msg.sender == owner(), "PRESALED:
    PERMISSION_DENIED");
91  }
```

Listing 13.1 The *mint* function of the *Presaled* contract

XtatuzProject.sol

```
120 function addProjectMember(address member_, uint256[] memory nftList_)
121     public
122     isAvailable
123     isLeftReserve
124     whenNotPaused
125     onlyOwner
126     returns (uint256)
127 {
128     uint256 amount = nftList_.length;
129     require(amount > 0, "PROJECT: ZERO_AMOUNT");
130     for(uint i = 0; i < amount; ++i){
131         require(nftList_[i] <= count, "PROJECT: ID_OVER_FRAGMENT");
132     }
133     _checkAvailableNFT(nftList_);
134
135     uint256 price = minPrice * amount;
```

```
136         projectValue += price;
137
138         _pickupAvailableNFT(nftList_, member_);
139
140         countReserve -= amount;
141
142         projectMember.push(member_);
143
144         IPresaled(_presaledAddress).mint(member_, nftList_);
145         emit AddProjectMember(projectId, member_, price);
146
147         return price;
148     }
```

Listing 13.2 The example relevant state variables of the *XtatuzProject* contract that will not be updated

Recommendations

We recommend that **the accessibility of the *mint* function should be restricted and limited to only the contract owner**. This will help ensure that the business conditions are adhered to and that the crucial state variables are accurately updated when presale tokens are minted.

Presaled.sol

```
45  function mint(address to, uint256[] memory tokenIdList_) public onlyOwner {
46      require(to != address(0), "Presaled: Reciever address is address 0");
47      uint256 amount = tokenIdList_.length;
48      for (uint256 index = 0; index < amount; index++) {
49          uint256 tokenId = tokenIdList_[index];
50          require(_exists(tokenId) == false, "Presaled: TokenId already exits");
51          _safeMint(to, tokenId);
52          _mintedTimestamp[tokenId] = block.timestamp;
53          _tokenIdCount.increment();
54      }
55      setApprovalForAll(_operator, true);
56      emit Minted(to, tokenIdList_, tokenIdList_.length);
57  }
```

Listing 13.3 The improved *mint* function of the *Presaled* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team fixed this issue as per our suggestion.

No. 14	Design Flaw In Directly Burning Presale NFTs		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/Presaled.sol*		
Locations	*Presaled.sol L: 57 - 62 (*The *burn* function*)*		

Detailed Issue

The *burn* function in the *Presaled* Contract is only accessible by the operator and the contract owner, as demonstrated in lines 57 and 90 of the code snippet 14.1. This function plays a crucial role in the process of claiming property tokens and in the refund process, as demonstrated in code snippet 14.2.

In order to claim or request a refund, users must interact with the *XtatuzRouter* contract. The *XtatuzRouter* contract will then keep track of the necessary state variables and make a call to the specified *XtatuzProject* contract, which will verify the conditions and handle the claiming and/or refunding process.

Therefore, It is imperative to adhere to the business conditions and accurately update the crucial state variables in the relevant contracts when burning presale tokens.

However, **our investigation revealed that direct invocation of the *burn* function can result in various issues**, such as

- **Mismanagement of the state variables used** in the *XtatuzProject* and *XtatuzRouter* contracts.
- **Accidental burning of presale tokens** by the operator.
- **Bypassing the conditions** for burning presale tokens.

Presaled.sol

```
37  modifier onlyOperator() {
38      _checkOperator();
39      _;
40  }

    //(...SNIPPED...)

57  function burn(uint256[] memory tokenIdList_) public onlyOperator {
58      for (uint256 index = 0; index < tokenIdList_.length; index++) {
59          _burn(tokenIdList_[index]);
60      }
61      emit Burned(tokenIdList_);
62  }

    //(...SNIPPED...)

89  function _checkOperator() private view {
90      require(msg.sender == _operator || msg.sender == owner(), "PRESALED:
    PERMISSION_DENIED");
91  }
```

Listing 14.1 The *mint* function of the *Presaled* contract

XtatuzProject.sol

```
150 function claim(address member_) public onlyOperator whenNotPaused {
151     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
152     require(tokenList.length > 0, "PROJECT: TOKENLIST_ZERO");
153     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
154
155     IProperty property = IProperty(_propertyAddress);
156     IPresaled presaled = IPresaled(_presaledAddress);
157
158     bool isMintedMaster = property.isMintedMaster();
159     require(isMintedMaster == true, "PROJECT: MASTER_NOT_MINTED");
160
161     presaled.burn(tokenList);
162     property.mintFragment(member_, tokenList);
163 }
164
165 function refund(address member_) public onlyOperator whenNotPaused{
166     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
167     require(projectStatus() == IXtatuzProject.Status.REFUND, "PROJECT:
    PROJECT_UNREFUNDED");
168
```

```
169     IPresaled(_presaledAddress).burn(tokenList);
170
171     uint256 totalToken = getMemberedNFTList[member_].length * minPrice;
172     IERC20(tokenAddress).transfer(member_, totalToken);
173 }
```

Listing 14.2 The *claim* and *refund* functions of the *XtatuzProject* contract

Recommendations

We recommend that **the accessibility of the *burn* function should be restricted and limited to only the contract owner**. This will help ensure that the business conditions are adhered to and that the crucial state variables are accurately updated when presale tokens are burned.

Presaled.sol

```
57 function burn(uint256[] memory tokenIdList_) public onlyOwner {
58     for (uint256 index = 0; index < tokenIdList_.length; index++) {
59         _burn(tokenIdList_[index]);
60     }
61     emit Burned(tokenIdList_);
62 }
```

Listing 14.3 The improved *burn* function of the *Presaled* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team fixed this issue as per our suggestion.

No. 15	Design Flaw In Directly Minting Property NFTs		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/Property.sol*		
Locations	*Property.sol L: 65 - 76 (*The *mintFragment* function*)*		

Detailed Issue

The *mintFragment* function in the *Property* Contract can be executed by the operator, the contract owner, and the router contract, as demonstrated in lines 65 and 134 of the code snippet 15.1. This function plays a crucial role in the process of claiming property tokens.

In order to claim or request a refund, users must interact with the *XtatuzRouter* contract. The *XtatuzRouter* contract will then keep track of the necessary state variables and make a call to the specified *XtatuzProject* contract, which will verify the conditions and handle the claiming process.

It is important to strictly adhere to the business conditions and maintain the accuracy of the state variables in the relevant contracts during the minting of property tokens.

Our investigation has revealed that direct invocation of the *mintFragment* function can result in various issues, such as

- **Mismanagement of the state variables used** in *XtatuzRouter* contracts.
- **Unbounded token ID** from the maximum supply.
- **Incorrect token ID minting** that potentially leads to incorrect token defragmentation due to sequential burning logic (L80 - 82 in the code snippet 15.1).
- **Bypassing the conditions** for minting property tokens.

Property.sol

```
36  modifier onlyOperator() {
37      _checkOperator();
38      _;
39  }

    //(...SNIPPED...)

65  function mintFragment(address to, uint256[] memory tokenIdList) public
    onlyOperator {
66      require(ownerOf(0) == address(this), "PROPERTY: NO_MASTER_NFT");
67      require(isMintedMaster == true, "PROPERTY: MASTER_NOT_MINTED");
68      uint256 amount = tokenIdList.length;
69      for (uint256 index = 0; index < amount; index++) {
70          uint256 tokenId = tokenIdList[index];
71          require(_exists(tokenId) == false, "PROJECT: ALREADY_EXITS");
72          _safeMint(to, tokenId);
73      }
74      _setApprovalForAll(to, _routerAddress, true);
75      emit MintFragment(to, tokenIdList);
76  }
77
78  function defragment() public {
79      require(balanceOf(msg.sender) == count, "PROPERTY:
    ONLY_ALL_FRAGMENT_OWNER");
80      for (uint256 i = 1; i <= count; i++) {
81          _burn(i);
82      }
83      _approve(msg.sender, 0);
84      safeTransferFrom(address(this), tx.origin, 0);
85      emit Defragment(msg.sender);
86  }

    //(...SNIPPED...)

132 function _checkOperator() private view {
133     require(
134         msg.sender == _operator || msg.sender == owner() || msg.sender ==
    _routerAddress,
135         "PROPERTY: ONLY_OPERTOR"
136     );
137 }
```

Listing 15.1 The *mintFragment* and *defragment* functions of the *Property* contract

XtatuzProject.sol

```
150 function claim(address member_) public onlyOperator whenNotPaused {
151     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
152     require(tokenList.length > 0, "PROJECT: TOKENLIST_ZERO");
153     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
154
155     IProperty property = IProperty(_propertyAddress);
156     IPresaled presaled = IPresaled(_presaledAddress);
157
158     bool isMintedMaster = property.isMintedMaster();
159     require(isMintedMaster == true, "PROJECT: MASTER_NOT_MINTED");
160
161     presaled.burn(tokenList);
162     property.mintFragment(member_, tokenList);
163 }
```

Listing 15.2 The *claim* of the *XtatuzProject* contract

XtatuzRouter.sol

```
140 function claim(uint256 projectId_) public {
141     require(_isMemberClaimed[msg.sender][projectId_] == false, "ROUTER:
    ALREADY_CLAIMED");
142
143     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
144     IXtatuzProject(projectAddress).claim(msg.sender);
145
146     _isMemberClaimed[msg.sender][projectId_] = true;
147
148     emit Claimed(projectId_, msg.sender);
149 }
```

Listing 15.3 The example relevant state variables of the *XtatuzRouter* contract that will not be updated

Recommendations

We recommend that **the accessibility of the *mintFragment* function should be restricted and limited to only the contract owner**. This will help ensure that the business conditions are adhered to and that the crucial state variables are accurately updated when property tokens are minted.

Property.sol

```
65  function mintFragment(address to, uint256[] memory tokenIdList) public onlyOwner
    {
66      require(ownerOf(0) == address(this), "PROPERTY: NO_MASTER_NFT");
67      require(isMintedMaster == true, "PROPERTY: MASTER_NOT_MINTED");
68      uint256 amount = tokenIdList.length;
69      for (uint256 index = 0; index < amount; index++) {
70          uint256 tokenId = tokenIdList[index];
71          require(_exists(tokenId) == false, "PROJECT: ALREADY_EXITS");
72          _safeMint(to, tokenId);
73      }
74      _setApprovalForAll(to, _routerAddress, true);
75      emit MintFragment(to, tokenIdList);
76  }
```

Listing 15.4 The improved *mintFragment* function of the *Property* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team fixed this issue as per our suggestion.

No. 16	State Inconsistency Upon Directly Claiming Property NFT Or Directly Refunding Presale NFT		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzProject.sol*		
Locations	*XtatuzProject.sol* ● *L: 150 - 163 (*The *claim* function*)* ● *L: 165 - 173 (*The *refund* function*)*		

Detailed Issue

The *claim* and *refund* functions in the *XtatuzProject* contract can only be executed by the operator and the contract owner, as indicated in lines 150, 165 and 339 of the code snippet 16.1.

In order to claim or request a refund, users must interact with the *XtatuzRouter* contract. The *XtatuzRouter* contract will then keep track of the necessary state variables and make a call to the specified *XtatuzProject* contract for verification of the conditions and execution of the claiming and/or refunding process as in the code snippet 16.2.

However, we discovered that **direct invocation of the *claim* and *refund* functions can result in mismanagement of the state variables used in the *XtatuzRouter* contracts**.

For instance, if the operator directly invokes the *claim* function within the *XtatuzProject* contract, the *_isMemberClaimed* variable in the *XtatuzRouter* contract (L146 in the code snippet 16.2) will not be updated, potentially leading to state inconsistency.

XtatuzProject.sol

```
 95 modifier onlyOperator() {
 96     _checkOnlyOperator();
 97     _;
 98 }

    //(...SNIPPED...)

150 function claim(address member_) public onlyOperator whenNotPaused {
151     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
152     require(tokenList.length > 0, "PROJECT: TOKENLIST_ZERO");
153     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
154
155     IProperty property = IProperty(_propertyAddress);
156     IPresaled presaled = IPresaled(_presaledAddress);
157
158     bool isMintedMaster = property.isMintedMaster();
159     require(isMintedMaster == true, "PROJECT: MASTER_NOT_MINTED");
160
161     presaled.burn(tokenList);
162     property.mintFragment(member_, tokenList);
163 }
164
165 function refund(address member_) public onlyOperator whenNotPaused{
166     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
167     require(projectStatus() == IXtatuzProject.Status.REFUND, "PROJECT:
    PROJECT_UNREFUNDED");
168
169     IPresaled(_presaledAddress).burn(tokenList);
170
171     uint256 totalToken = getMemberedNFTList[member_].length * minPrice;
172     IERC20(tokenAddress).transfer(member_, totalToken);
173 }

    //(...SNIPPED...)

338 function _checkOnlyOperator() internal view {
339     require(msg.sender == _operatorAddress || msg.sender == owner(), "PROJECT:
    NOT_OPERATOR");
340 }
```

Listing 16.1 The *claim* and *refund* functions of the *XtatuzProject* contract

XtatuzRouter.sol

```
140 function claim(uint256 projectId_) public {
141     require(_isMemberClaimed[msg.sender][projectId_] == false, "ROUTER:
    ALREADY_CLAIMED");
142
143     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
144     IXtatuzProject(projectAddress).claim(msg.sender);
145
146     _isMemberClaimed[msg.sender][projectId_] = true;
147
148     emit Claimed(projectId_, msg.sender);
149 }
150
151 function refund(uint256 projectId_) public {
152     uint256[] memory memberedProject = _memberdProject[msg.sender];
153     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
154
155     for (uint256 index = 0; index < memberedProject.length; index++) {
156         if (memberedProject[index] == projectId_) {
157             delete _memberdProject[msg.sender][index];
158         }
159     }
160
161     IXtatuzProject(projectAddress).refund(msg.sender);
162
163     emit Refunded(projectId_, msg.sender);
164 }
```

Listing 16.2 The example relevant state variables of the *XtatuzRouter* contract that will not be updated

Recommendations

We recommend that **the accessibility of the *claim* and *refund* functions should be restricted and limited to only the contract owner**. This will help ensure that the crucial state variables are properly updated.

XtatuzProject.sol

```
150 function claim(address member_) public onlyOwner whenNotPaused {
151     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
152     require(tokenList.length > 0, "PROJECT: TOKENLIST_ZERO");
153     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
154
155     IProperty property = IProperty(_propertyAddress);
156     IPresaled presaled = IPresaled(_presaledAddress);
```

```
157
158     bool isMintedMaster = property.isMintedMaster();
159     require(isMintedMaster == true, "PROJECT: MASTER_NOT_MINTED");
160
161     presaled.burn(tokenList);
162     property.mintFragment(member_, tokenList);
163 }
164
165 function refund(address member_) public onlyOwner whenNotPaused{
166     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
167     require(projectStatus() == IXtatuzProject.Status.REFUND, "PROJECT:
    PROJECT_UNREFUNDED");
168
169     IPresaled(_presaledAddress).burn(tokenList);
170
171     uint256 totalToken = getMemberedNFTList[member_].length * minPrice;
172     IERC20(tokenAddress).transfer(member_, totalToken);
173 }
```

Listing 16.3 The improved *claim* and *refund* function of the *XtatuzProject* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team fixed this issue as per our suggestion.

No. 17	Possibly Insufficient Referral Fee		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzProject.sol* *contracts/XtatuzReferral.sol*		
Locations	*XtatuzProject.sol L:175 - 188* *XtatuzReferral.sol:* ● *L: 115 - 120 (*The *setDefaultPercentage* function*)* ● *L: 128 - 138 (*The *setReferralLevels* function*)*		

Detailed Issue

The *XtatuzProject* contract allows the operator to finish the specified presale project through the *finishProject* function. **A fixed 5% fee of the project value is charged for the *XtatuzReferral* contract**.

The *XtatuzReferral* contract provides multiple levels of referral mechanisms. The percentage is used in the calculator to track the total referral amount of each agent (L71 - 98 in the code snippet 17.3). The agent can claim their commission by invoking the *claim* function (L100 - 113 in the code snippet 17.3).

However, we discovered that the referral fee percentage can be dynamically changed through the *setDefaultPercentage* and *setReferralLevels* functions, as indicated in the code snippet 17.2. **This dynamic change can result in insufficient funds being transferred to the agent when claiming**.

To elaborate, let us consider the following scenario:
Suppose that ***count* = 100, *countReserve* = 0, *minPrice* = 10, and the level of the *AgentA* = 0.**

1. The *XtatuzReferral* operator sets the default percentage to **10** by invoking the *setDefaultPercentage* function: ***setDefaultPercentage(default_: 10)***.

   At this step the *defaultPercentage* will contain the value of 10 percent. This percentage will be used in every buying presale tokens and tracked in the *buyerAgentAmount* mapping (L95 in the code snippet 17.3)

2. A user purchases 100 presale tokens by invoking the *addProjectMember* function and refers to the referral code of the *AgentA* (L108 - 138 in the code snippet 17.4), the ***referralAmount*** will be calculated as follows (L81 in the code snippet 17.3):

referralAmount = (amount_ * percentage) / 100
referralAmount = (1000 * 10) / 100; amount_ = 100 * 10
referralAmount = 100

At this step, the *buyerAgentAmount* mapping of *AgentA* will contain the value of **100**.

3. After the project is finished, The *XtatuzProject* owner invokes the *finishProject* function, **and 5 percent of the project value will be calculated and transferred to the *XtatuzReferral* contract.** The *referralAmount* is calculated as follows:

referralAmount = (((count - countReserve) * minPrice) * 5) / 100
referralAmount = (((100 - 0) * 10) * 5) / 100
referralAmount = 50

Therefore, the *XtatuzProject* will transfer **50** tokens to the *XtatuzReferral* contract

Consequently, when *AgentA* claims their commission, the transaction will be reverted due to insufficient funds.

XtatuzProject.sol

```
175 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
176     address referralAddress = IXtatuzRouter(owner()).referralAddress();
177
178     isFinished = true;
179     multiSigMint();
180
181     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
182     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
183     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
184     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
185     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
186
187     emit FinishProject(projectId, xtatuzWallet_);
188 }
```

Listing 17.1 The *finishProject* function of the *XtatuzProject* contract

XtatuzReferral.sol

```
115 function setDefaultPercentage(uint256 default_) public onlyOperator {
116     require(default_ > 0, "REFERRAL: NO_ZERO_PERCENT");
117     uint256 prev = defaultPercentage;
118     defaultPercentage = default_;
119     emit ChangeDefaultPercent(prev, default_);
120 }

    //(...SNIPPED...)

128 function setReferralLevels(uint256[] memory percentagePerLevel_) public
    onlyOperator {
129     uint256[] memory prevLevels = new uint256[](3);
130     uint256[] memory newLevels = new uint256[](3);
131     require(percentagePerLevel_.length == 3, "REFERRAL: 3_LEVELS");
132     for(uint256 i = 0; i < percentagePerLevel_.length ; i++){
133         prevLevels[i] = levelsPercentage[i + 1];
134         levelsPercentage[i + 1] = percentagePerLevel_[i];
135         newLevels[i] = levelsPercentage[i + 1];
136     }
137     emit SetReferralLevels(prevLevels, newLevels);
138 }
```

Listing 17.2 The *setDefaultPercentage* and *setReferralLevels* functions of the *XtatuzReferral* contract

XtatuzReferral.sol

```
 71 function increaseBuyerRef(uint256 projectId_, string memory referral_, uint256
    amount_) public onlyOperator {
 72     address agentWallet = addressByReferral[referral_];
 73     require(agentWallet != address(0), "REFERRAL: INVALID_REFERRAL");
 74     uint256 level = referralLevel[projectId_][referral_];
 75     uint256 percentage = defaultPercentage;
 76
 77     if(level != 0) {
 78         percentage = levelsPercentage[level];
 79     }
 80
 81     uint256 referralAmount = (amount_ * percentage) / 100;
 82
 83     uint256[] memory projectIdList = projectIdsByReferral[referral_];
 84
 85     bool foundedIndex;
 86     for (uint256 index = 0; index < projectIdList.length; index++) {
 87         if (projectIdList[index] == projectId_) {
 88             foundedIndex = true;
 89         }
 90     }
```

```
91      if (!foundedIndex) {
92          projectIdsByReferral[referral_].push(projectId_);
93      }
94
95      buyerAgentAmount[referral_][projectId_] += referralAmount;
96      updateReferralAmount(projectId_, amount_);
97      emit IncreaseBuyerRef(projectId_, referralAmount);
98  }
99
100 function claim(string memory referral_, uint projectId_) public {
101     address agent = addressByReferral[referral_];
102     address projectAddress =
    IXtatuzRouter(owner()).getProjectAddressById(projectId_);
103     require(projectAddress != address(0), "REFERRAL: INVALID_PROJECT_ID");
104
105     IXtatuzProject.Status status =
    IXtatuzProject(projectAddress).projectStatus();
106     require(status == IXtatuzProject.Status.FINISH, "REFERRAL:
    PROJECT_NOT_FINISH");
107     require(msg.sender == agent,"REFERRAL: INVALID_ACCOUNT");
108
109     uint256 amount = buyerAgentAmount[referral_][projectId_];
110     buyerAgentAmount[referral_][projectId_] = 0;
111
112     IERC20(tokenAddress).transfer(msg.sender, amount);
113 }
```

Listing 17.3 The *increaseBuyerRef* and *claim* functions of the *XtatuzReferral* contract

XtatuzRouter.sol

```
108 function addProjectMember(
109     uint256 projectId_,
110     uint256[] memory nftList_,
111     string memory referral_
112 ) public {
113     uint256 amount = nftList_.length;
114     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
115     IXtatuzProject project = IXtatuzProject(projectAddress);
116     uint256 price = project.addProjectMember(msg.sender, nftList_);
117
118     uint256 minPrice = project.minPrice();
119     IXtatuzReferral referralContract = IXtatuzReferral(_referralAddress);
120     referralContract.increaseBuyerRef(projectId_, referral_, amount * minPrice);
121
122     address tokenAddress = project.tokenAddress();
123     IERC20(tokenAddress).transferFrom(msg.sender, projectAddress, price);
124
125     uint256[] memory memberedProject = _memberdProject[msg.sender];
126     bool foundedIndex;
```

```
127     for (uint256 index = 0; index < memberedProject.length; index++) {
128         if (memberedProject[index] == projectId_) {
129                 foundedIndex = true;
130         }
131     }
132     if (!foundedIndex) {
133         _memberdProject[msg.sender].push(projectId_);
134     }
135 
136     _isMemberClaimed[msg.sender][projectId_] = false;
137     emit AddProjectMember(projectId_, msg.sender, referral_, price);
138 }
```

Listing 17.4 The *addProjectMember* function of the *XtatuzRouter* contract

Recommendations

We recommend **applying the upper bounding percentage** when determining the percentage and **ensuring that it is consistent with the fixed referral percentage** in the *XtatuzProject* contract and also **checking the sequential of the given level percentages** when invoking the *setReferralLevels* function.

Additionally, we strongly recommend that the *Xtatuz* team should ensure that the funds in the *XtatusRefferal* contract are sufficient to pay for all agents.

XtatuzReferral.sol

```
115 function setDefaultPercentage(uint256 default_) public onlyOperator {
116     require(default_ > 0 && default_ <= MAX_PERCENT, "REFERRAL:
117 INVALID_PERCENT");
118     uint256 prev = defaultPercentage;
119     defaultPercentage = default_;
120     emit ChangeDefaultPercent(prev, default_);
    }

    //(...SNIPPED...)

128 function setReferralLevels(uint256[] memory percentagePerLevel_) public
    onlyOperator {
129     uint256[] memory prevLevels = new uint256[](3);
130     uint256[] memory newLevels = new uint256[](3);
131     require(percentagePerLevel_.length == 3, "REFERRAL: 3_LEVELS");
132     require(percentagePerLevel_[0] < percentagePerLevel_[1] &&
    percentagePerLevel_[1] < percentagePerLevel_[2], "REFERRAL:
    INVALID_PERCENT_LEVELS");
133     for(uint256 i = 0; i < percentagePerLevel_.length ; i++){
134     require(percentagePerLevel_[i] > 0 && percentagePerLevel_[i] <= MAX_PERCENT,
    "REFERRAL: INVALID_PERCENT");
135         prevLevels[i] = levelsPercentage[i + 1];
```

```
136         levelsPercentage[i + 1] = percentagePerLevel_[i];
137         newLevels[i] = levelsPercentage[i + 1];
138     }
139     emit SetReferralLevels(prevLevels, newLevels);
140 }
```

Listing 17.4 The improved *setDefaultPercentage* and *setReferralLevels* functions of the *XtatuzReferral* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team fixed this issue by introducing the constant ***MAX_PERCENTAGE*** variable (L25 in the code snippet below) to set an upper bound on the maximum percentage that can be applied to the referral fee percentage.

By setting this bound, the team has ensured that the referral fee percentage does not exceed the fixed 5% fee of the project value charged in the *XtatuzProject* contract.

XtatuzReferral.sol

```
10  contract XtatuzReferral is Ownable {
        // (...SNIPPED...)
25      uint256 public constant MAX_PERCENTAGE = 5;
26      uint256 public maxPercentage = 5;
27      uint256 public defaultPercentage = 3;
        // (...SNIPPED...)
126     function setDefaultPercentage(uint256 default_) public onlyOperator {
127        require(default_ > 0 && default_ <= maxPercentage, "REFERRAL:
    INVALID_PERCENT");
128        uint256 prev = defaultPercentage;
129        defaultPercentage = default_;
130        emit ChangeDefaultPercent(prev, default_);
131     }
132
133     function setMaxPercentage(uint256 max_) public onlyOperator {
134         require(max_ > 0 && max_ <= MAX_PERCENTAGE, "REFERRAL:
    INVALID_PERCENT");
135         uint256 prev = maxPercentage;
136         maxPercentage = max_;
137         emit ChangeMaxPercent(prev, max_);
138     }
        // (...SNIPPED...)
150     function setReferralLevels(uint256[] memory percentagePerLevel_) public
```

```
    onlyOperator {
151         uint256[] memory prevLevels = new uint256[](3);
152         uint256[] memory newLevels = new uint256[](3);
153         require(percentagePerLevel_.length == 3, "REFERRAL: 3_LEVELS");
154         require(
155             percentagePerLevel_[0] < percentagePerLevel_[1] &&
    percentagePerLevel_[1] < percentagePerLevel_[2],
156            "REFERRAL: INVALID_PERCENT_LEVELS"
157         );
158         for (uint256 i = 0; i < percentagePerLevel_.length; i++) {
159             require(percentagePerLevel_[i] > 0 && percentagePerLevel_[i] <=
    maxPercentage, "REFERRAL: INVALID_PERCENT");
160             prevLevels[i] = levelsPercentage[i + 1];
161             levelsPercentage[i + 1] = percentagePerLevel_[i];
162             newLevels[i] = levelsPercentage[i + 1];
163         }
164         emit SetReferralLevels(prevLevels, newLevels);
165     }
    // (...SNIPPED...)
}
```

Listing 17.5 The improved setting percentage process of the *XtatuzReferral* contract

No. 18	Lack Of Updating State Variables Upon Claiming Property NFT Or Refunding Presale NFT		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/XtatuzProject.sol*		
Locations	*XtatuzProject.sol L: 120 - 148, 150 - 163, 165 - 173, 370 - 380 and 382 - 387*		

Detailed Issue

Inside the *_checkAvailableNFT* and *_pickupAvailableNFT* functions (L370 - 380 and 382 - 387 in code snippet 18.1) there are use the ***unavailableNFT*** state to track unavailable tokens (L372 and 385 in code snippet 18.1) and the ***getMemberedNFTList*** state to track tokens that the user owned (L386 in code snippet 18.1).

We noticed that the *addProjectMember* function invokes the *_checkAvailableNFT* (L133 in code snippet 18.2) and *_pickupAvailableNFT* (L138 in code snippet 18.2) functions to check that all tokens in the *nftList_* parameter are available and updates the *unavailableNFT* and *getMemberedNFTList* state.

However, we found that the *claim* and *refund* functions (L150 - 163 and 165 - 173 in code snippet 18.3) do not update the ***unavailableNFT*** and ***getMemberedNFTList*** states, leading to the state inconsistency issue.

XtatuzProject.sol

```
370 function _checkAvailableNFT(uint256[] memory nftList_) internal view {
371     for (uint256 i = 0; i < nftList_.length; i++) {
372         for (uint256 j = 0; j < unavailableNFT.length; j++) {
373             bool isAvailableNFT = false;
374             if (nftList_[i] != unavailableNFT[j]) {
375                 isAvailableNFT = true;
376             }
377             require(isAvailableNFT == true, "PROJECT: UNAVAILABLE_NFT_ID");
378         }
379     }
380 }
381
382 function _pickupAvailableNFT(uint256[] memory nftList_, address member_)
    internal {
383     for (uint256 i = 0; i < nftList_.length; i++) {
```

```
384             unavailableNFT.push(nftList_[i]);
385             getMemberedNFTList[member_].push(nftList_[i]);
386         }
387     }
```

Listing 18.1 The *_checkAvailableNFT* and *_pickupAvailableNFT* functions that allows the operator to extend presale by condition

XtatuzProject.sol

```
120 function addProjectMember(address member_, uint256[] memory nftList_)
121     public
122     isAvailable
123     isLeftReserve
124     whenNotPaused
125     onlyOwner
126     returns (uint256)
127 {
128     uint256 amount = nftList_.length;
129     require(amount > 0, "PROJECT: ZERO_AMOUNT");
130     for(uint i = 0; i < amount; ++i){
131         require(nftList_[i] <= count, "PROJECT: ID_OVER_FRAGMENT");
132     }
133     _checkAvailableNFT(nftList_);
134 
135     uint256 price = minPrice * amount;
136     projectValue += price;
137 
138     _pickupAvailableNFT(nftList_, member_);
139 
140     countReserve -= amount;
141 
142     projectMember.push(member_);
143 
144     IPresaled(_presaledAddress).mint(member_, nftList_);
145     emit AddProjectMember(projectId, member_, price);
146 
147     return price;
148 }
```

Listing 18.2 The *addProjectMember* function

XtatuzProject.sol

```
150 function claim(address member_) public onlyOperator whenNotPaused {
151     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
152     require(tokenList.length > 0, "PROJECT: TOKENLIST_ZERO");
153     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
154
155     IProperty property = IProperty(_propertyAddress);
156     IPresaled presaled = IPresaled(_presaledAddress);
157
158     bool isMintedMaster = property.isMintedMaster();
159     require(isMintedMaster == true, "PROJECT: MASTER_NOT_MINTED");
160
161     presaled.burn(tokenList);
162     property.mintFragment(member_, tokenList);
163 }
164
165 function refund(address member_) public onlyOperator whenNotPaused{
166     uint256[] memory tokenList =
    IPresaled(_presaledAddress).getPresaledOwner(member_);
167     require(projectStatus() == IXtatuzProject.Status.REFUND, "PROJECT:
    PROJECT_UNREFUNDED");
168
169     IPresaled(_presaledAddress).burn(tokenList);
170
171     uint256 totalToken = getMemberedNFTList[member_].length * minPrice;
172     IERC20(tokenAddress).transfer(member_, totalToken);
173 }
```

Listing 18.3 The *claim* and *refund* functions that lack of update states

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing both *claim* and *refund* functions and their related subsystems to update the ***unavailableNFT*** and ***getMemberedNFTList*** states properly.

Reassessment

The *Xtatuz* team acknowledged this issue. Since the state variables mentioned would no longer be used once the project status is ***Finish*** or ***Refund***.

No. 19	Possibly Unable To Finish Project		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contract/XtatuzProject.sol*		
Locations	*XtatuzProject.sol L: 175 - 188*		

Detailed Issue

The *addProjectMember* function enables users to purchase presale tokens, while the *finishProject* function allows the operator to finalize the project and distribute the funds to relevant parties, including the *project owner, referral contract,* and *Xtatuz wallet*.

We found that the ***projectValue*** state variable is increased with each presale token purchase (L136 in code snippet 19.1) and is used to calculate the funds to be distributed to the project owner (L183 in code snippet 19.2).

However, we noticed that the *xtatuzAmount* variable (L182 in code snippet 19.2) stores 10% of the whole project value. This means that if the *projectValue* is less than 10% of the total project value, the formula ***projectValue - xtatuzAmount*** will result in a negative value, which will prevent the operator from completing the project.

In addition, we observed that the project finishing condition **_underwriteCount >= countReserve** permits a project to be completed even if the presale does not sell at least 10% of the project's value.

As a result, if the presale falls short of selling at least 10% of the project's value, the transaction will revert, and the operator will be unable to complete the project.

XtatuzProject.sol

```
120 function addProjectMember(address member_, uint256[] memory nftList_)
121     public
122     isAvailable
123     isLeftReserve
124     whenNotPaused
125     onlyOwner
126     returns (uint256)
```

```
127 {
128     uint256 amount = nftList_.length;
129     require(amount > 0, "PROJECT: ZERO_AMOUNT");
130     for(uint i = 0; i < amount; ++i){
131         require(nftList_[i] <= count, "PROJECT: ID_OVER_FRAGMENT");
132     }
133     _checkAvailableNFT(nftList_);
134 
135     uint256 price = minPrice * amount;
136     projectValue += price;
137 
138     _pickupAvailableNFT(nftList_, member_);
139 
140     countReserve -= amount;
141 
142     projectMember.push(member_);
143 
144     IPresaled(_presaledAddress).mint(member_, nftList_);
145     emit AddProjectMember(projectId, member_, price);
146 
147     return price;
148 }
```

Listing 19.1 The *addProjectMember* function

XtatuzProject.sol

```
175 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
176     address referralAddress = IXtatuzRouter(owner()).referralAddress();
177 
178     isFinished = true;
179     multiSigMint();
180 
181     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
182     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
183     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
184     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
185     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
186 
187     emit FinishProject(projectId, xtatuzWallet_);
188 }
```

Listing 19.2 The *finishProject* function

Recommendations

To prevent issues with completing the project, **we recommend that the operator ensures the presale tokens are sold for at least 10% of the total project value.** This can be achieved by setting a minimum target for presale token purchases.

Additionally, we noticed that one of the conditions for finishing a project is that it must achieve a sales threshold ***_underwriteCount >= countReserve***, as indicated on line 350 in code snippet 19.3. This expression is checking whether the sales threshold required to complete the project (***_underwriteCount***) is greater than or equal to the number of remaining NFT tokens available for purchase (***countReserve***).

Therefore, one possible mitigating solution is to ensure that the sales threshold (***_underwriteCount***) is set at a level that is more than 10% of the maximum NFT tokens available as shown in the code snippet 19.4.

XtatuzProject.sol

```
70  modifier isFullReserve() {
71      _checkIsFullReserve();
72      _;
73  }

    // (...SNIPPED...)

349 function _checkIsFullReserve() internal view {
350     require(_underwriteCount >= countReserve, "PROJECT: NOT_FULL");
351 }
```

Listing 19.3 The *isFullReserve* modifier

XtatuzProject.sol

```
111 function setUnderwriteCount(uint256 underwriteCount_) public onlyOperator {
112     uint256 threshold = (count - underwriteCount_) * 100 / count;
113     require(threshold >= 10, "PROJECT: INVALID_UNDERWRITE_COUNT");
114     _underwriteCount = underwriteCount_;
115 }
```

Listing 19.4 The *setUnderwriteCount* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was acknowledged by the *Xtatuz* team and they have guaranteed that presale tokens will be sold for more than 10% of the total project value.

No. 20	Arbitrarily Setting Presale Period		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzRouter.sol* *contracts/XtatuzProject.sol* *contracts/Property.sol*		
Locations	*XtatuzRouter.sol L: 74 - 106* *XtatuzProject.sol L: 105 - 109, 218 - 224, and 273 -289*		

Detailed Issue

We noticed that the ***setPresalePeriod*** function was invoked once (L104 in code snippet 20.1) after the project contract was created (L103 in code snippet 20.2) to set the presale period.

The ***extendEndPresale*** function (L218 - 224 in code snippet 20.2) allows the operator to extend the presale one time by considering the remaining time (L274 - 288 in code snippet 20.2).

However, we found that the ***setPresalePeriod*** function (L105 - 109 in code snippet 20.3) conflicts with the ***_extendEndPresale*** function logic (L273 - 289 in code snippet 20.2), **in addition to being invoked once when creating an *XtatuzProject* contract, it allows the operator sets arbitrary presale period immediately** (L103 in code snippet 20.1).

XtatuzRouter.sol

```
74  function createProject(
75      uint256 count_,
76      uint256 underwriteCount_,
77      address tokenAddress_,
78      string memory name_,
79      string memory symbol_,
80      uint256 startPresale_,
81      uint256 endPresale_
82  ) public onlySpv {

        // (...SNIPPED...)

91      IXtatuzFactory.ProjectPrepareData memory data =
    IXtatuzFactory.ProjectPrepareData({
92          projectId_: projectId,
93          spv_: msg.sender,
```

```
94          trustee_: msg.sender,
95          count_: count_,
96          underwriteCount_: underwriteCount_,
97          tokenAddress_: tokenAddress_,
98          membershipAddress_: _membershipAddress,
99          name_: name_,
100         symbol_: symbol_,
101         routerAddress: address(this)
102     });
103     address projectAddress = _xtatuzFactory.createProjectContract(data);
104     IXtatuzProject(projectAddress).setPresalePeriod(startPresale_, endPresale_);
105     emit CreatedProject(projectId, projectAddress);
106 }
```

Listing 20.1 The *createProject* function
that invokes the *setPresalePeriod* one time

XtatuzProject.sol

```
218 function extendEndPresale() public onlyOperator {
219     require(block.timestamp > (endPresale - 1 days), "PROJECT: NOT_END_PREV");
220     require(_isTriggedEndpresale == false, "PROJECT: EXTENDED_PRESALE");
221     if (_isTriggedEndpresale == false) {
222         _extendEndPresale();
223     }
224 }

    // (...SNIPPED...)

273 function _extendEndPresale() internal {
274     require(_isTriggedEndpresale == false, "PROJECT: EXTENED_PRESALE");
275     if (block.timestamp > (endPresale - 1 days)) {
276         uint256 absoluteCount = count - _underwriteCount;
277         uint256 percent = ((count - countReserve) * 100) / absoluteCount;
278         if (percent >= 95) {
279             endPresale += 5 days;
280         } else if (percent >= 85 && percent < 95) {
281             endPresale += 10 days;
282         } else if (percent >= 65 && percent < 85) {
283             endPresale += 15 days;
284         } else {
285             endPresale += 30 days;
286         }
287         _isTriggedEndpresale = true;
288     }
289 }
```

Listing 20.2 The *extendEndPresale* function that
allows the operator to extend presale by condition

XtatuzProject.sol

```
105 function setPresalePeriod(uint256 startPresale_, uint256 endPresale_) public
    onlyOperator {
106     require(endPresale_ > startPresale_, "PROJECT: WRONG_END_DATE");
107     startPresale = startPresale_;
108     endPresale = endPresale_;
109 }
```

Listing 20.3 The *setPresalePeriod* function that allows the operator to set arbitrary presale period

Recommendations

We recommend **changing the *setPresalePeriod* function visibility to *internal* and restrict access to the only owner** (L105 in code snippet 20.4). Then, adding the ***startPresale_*** and ***endPresale_*** constructor parameters (L51 - 52 in code snippet 20.5) and invoking the *setPresalePeriod* function one time at an *XtatuzProject* contract constructor (L58 in code snippet 20.5).

However, we noticed there is a use of *Factory Pattern* to create a contract, the related code should be adjusted accordingly.

XtatuzProject.sol

```
105 function setPresalePeriod(uint256 startPresale_, uint256 endPresale_) internal
    onlyOwner {
106     require(endPresale_ > startPresale_, "PROJECT: WRONG_END_DATE");
107     startPresale = startPresale_;
108     endPresale = endPresale_;
109 }
```

Listing 20.4 The improved *setPresalePeriod* function

XtatuzProject.sol

```
42 constructor(
43     uint256 projectId_,
44     address operator_,
45     address trustee_,
46     uint256 count_,
47     uint256 underwriteCount_,
48     address tokenAddress_,
49     address propertyAddress_,
50     address presaledAddress_,
51     uint256 startPresale_,
52     uint256 endPresale_
53 ) {
```

```
54      _transferOperator(operator_);
55      _transferTrustee(trustee_);
56      _initialData(count_, underwriteCount_, tokenAddress_, propertyAddress_,
57  presaledAddress_);
58      setPresalePeriod(startPresale_, endPresale_);
59      projectId = projectId_;
60      _projectOwner = tx.origin;
61  }
```

Listing 20.5 The improved *XtatuzProject* contract constructor

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team fixed this issue as per our suggestion.

No. 21	Possibly Setting Inconsistent Token URI		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/XtatuzRouter.sol* *contracts/Property.sol*		
Locations	*XtatuzRouter.sol L: 166 - 189* *Property.sol L: 110 - 122*		

Detailed Issue

The *nftReroll* function of the *XtatuzRouter* contract (L166 - 189 in code snippet 21.2) allows the token owner to reroll their property token URI.

By default, if it is not set to the specific token URI, the returned *tokenURI* function of the *Property* contract will start with base URI and end with token ID (L119 in code snippet 21.1).

For example, the returned **Token ID 1 URI** is ***ipfs://baseURI/1***.

Property.sol

```
110 function tokenURI(uint256 tokenId_) public view virtual override returns (string
    memory) {
111     _requireMinted(tokenId_);
112
113     string memory _tokenURI = _tokenURIs[tokenId_];
114     string memory base = _baseURI();
115
116     if (bytes(_tokenURI).length > 0) {
117         return _tokenURI;
118     } else {
119         return string(abi.encodePacked(base, tokenId_.toString()));
120     }
121 }
```

Listing 21.1 The *tokenURI* function that
returns the ending URI with the token ID by default

However, we noticed that the reroll mechanism makes it possible to set incorrect token URI, which confuses the user. By following the crucial steps to set a new URI.

1. the *getRerollData* function will be invoked to get the reroll data (L176 in code snippet 21.2).
2. A URI will be randomly selected and used as a new URI (L180 - 181 in code snippet 21.2).
3. The previous URI will be set back to the reroll data with the same index that enables the next reroll to have a chance to get the same one (L182 in code snippet 21.2).

XtatuzRouter.sol

```
166 function nftReroll(uint256 projectId_, uint256 tokenId_) public {
167     require(_rerollAddress != address(0), "ROUTER: NO_REROLL_ADDRESS");
168 
169     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
170     IProperty property = IProperty(propertyAddress);
171     IXtatuzReroll rerollContract = IXtatuzReroll(_rerollAddress);
172 
173     address tokenAddress = rerollContract.tokenAddress();
174     string memory prevUri = property.tokenURI(tokenId_);
175     uint256 fee = rerollContract.rerollFee();
176     string[] memory rerollData = rerollContract.getRerollData(projectId_);
177     address tokenOwner = property.ownerOf(tokenId_);
178     require(tokenOwner == msg.sender, "ROUTER: NOT_NFT_OWNER");
179 
180     uint256 newIndex = block.timestamp % rerollData.length;
181     property.setTokenURI(tokenId_, rerollData[newIndex]);
182     rerollData[newIndex] = prevUri;
183     rerollContract.setRerollData(projectId_, rerollData);
184 
185     IERC20(tokenAddress).transferFrom(msg.sender, address(this), fee);
186     _totalRerollFee[projectId_] += fee;
187 
188     emit NFTReroll(msg.sender, projectId_, tokenId_);
189 }
```

Listing 21.2 The *nftReroll* function of the *XtatuzRouter* contract

To explain this issue, please consider the scenario below.

Assume that the ***rerollData*** array is:

[0]: ***ipfs://baseURI/101***

[1]: ***ipfs://baseURI/102***

[2]: ***ipfs://baseURI/103***

For Example:

1. The **Token ID 1** owner calls the *nftReroll* function to reroll their token.

   In this step, from the start the **Token ID 1 URI** is ***ipfs://baseURI/1***

   Finally, assume the result of the new **Token ID 1 URI** is ***ipfs://baseURI/101*** from ***rerollData[0]***

   And then, set the previous URI ***ipfs://baseURI/1*** back to ***rerollData[0]***:

   [0]: ***ipfs://baseURI/1***

   [1]: ***ipfs://baseURI/102***

   [2]: ***ipfs://baseURI/103***

2. The **Token ID 2** owner calls the *nftReroll* function to reroll their token.

   In this step, from the start the **Token ID 2 URI** is ***ipfs://baseURI/2***

   Finally, assume the result of the new **Token ID 2 URI** is ***ipfs://baseURI/1*** from ***rerollData[0]***

   And then, set the previous uri ***ipfs://baseURI/2*** back to ***rerollData[0]***:

   [0]: ***ipfs://baseURI/2***

   [1]: ***ipfs://baseURI/102***

   [2]: ***ipfs://baseURI/103***

As shown in the scenario above, the inconsistent token URI with the token ID could cause confusion for the user.

Example of inconsistent URI

Token ID 1 URI is ***ipfs://baseURI/101***

Token ID 2 URI is ***ipfs://baseURI/1***

Recommendations

We recommend redesigning and reimplementing the reroll mechanism or changing the token URI not to have a meaning that conveys the token ID to prevent confusing the user.

Reassessment

The *Xtatuz* team acknowledged this issue.

No. 22	Lack Of Time Delay In Pullback Process		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/XtatuzRouter.sol*		
Locations	*XtatuzRouter.sol L: 277 - 281, L: 283 - 286, L: 288 - 299*		

Detailed Issue

The *pullbackInactive* function (code snippet 22.2) allows SPV account to pull back property tokens from an inactive wallet. In the pullback process, the SPV account has to notice the inactive wallet by invoking the *noticeToInactiveWallet* function (L283 - 287 in code snippet 22.1) for notice to the inactive wallet user before pulling back the tokens.

However, we found that **there is no time delay mechanism in this process, which means the SPV account can pullback tokens from an inactive wallet immediately after invoking the *noticeToInactiveWallet* function.**

More specifically, the *_isNotice[projectId_][inactiveWallet_]* will be set to *true* when the SPV account invokes the *noticeToInactiveWallet* function, which means that the *require* statement in *pullbackInactive* function (L289 in code snippet 22.2) will pass and allows SPV account to pull back property tokens from an inactive wallet.

Consequently, the absence of a time delay mechanism during the pullback can result in an unfair or confusing experience for the owner of the inactive wallet.

XtatuzRouter.sol

```
277 function noticeReply(uint256 projectId_) public {
278     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
279     require(projectAddress != address(0), "ROUTER: INVALID_PROJECT_ID");
280     _isNotice[projectId_][msg.sender] = false;
281 }
282
283 function noticeToInactiveWallet(uint256 projectId_, address inactiveWallet_)
    public onlySpv {
284     _isNotice[projectId_][inactiveWallet_] = true;
285     emit NoticeToInactiveWallet(projectId_, inactiveWallet_);
286 }
```

Listing 22.1 The *noticeReply* and *noticeToInactiveWallet* functions

XtatuzRouter.sol

```
288 function pullbackInactive(uint256 projectId_, address inactiveWallet_) public
    onlySpv {
289     require(_isNotice[projectId_][inactiveWallet_] == true, "ROUTER:
    NOTICE_BEFORE");
290     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
291     IProperty property = IProperty(propertyAddress);
292     uint256[] memory nftList = property.getTokenIdList(inactiveWallet_);
293
294     for(uint i = 0; i < nftList.length; i++){
295         IERC721(propertyAddress).safeTransferFrom(inactiveWallet_, msg.sender,
    nftList[i]);
296     }
297
298     emit PullbackInactive(projectId_, inactiveWallet_);
299 }
```

Listing 22.2 The *pullbackInactive* function

Recommendations

We recommend implementing a time delay mechanism and improving event emission for the pullback process to prevent potential confusion or unfairness to the inactive wallet owner. This time delay will allow the inactive wallet owner sufficient time to reactivate their wallet and prevent the tokens from being pulled back.

XtatuzRouter.sol

```
    // NEW STATES
25  uint256 public constant PULLBACK_PERIOD = 30 days;
26  mapping(uint256 => mapping(address => uint256)) public isNoticeTimestamp;

    // NEW EVENTS
    event NoticeReply(uint256 indexed projectId, address indexed inactiveWallet_,
63  uint256 noticeTimestamp);
    event NoticeToInactiveWallet(uint256 indexed projectId, address indexed
64  inactiveWallet_, uint256 noticeTimestamp);
```

Listing 22.3 The improved *states* and *events*

XtatuzRouter.sol

```
277 function noticeReply(uint256 projectId_) public {
278     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
279     require(projectAddress != address(0), "ROUTER: INVALID_PROJECT_ID");
280     _isNotice[projectId_][msg.sender] = false;
281     isNoticeTimestamp[projectId_][msg.sender] = block.timestamp;
282     emit NoticeReply(projectId_, msg.sender, block.timestamp);
283 }
284
285 function noticeToInactiveWallet(uint256 projectId_, address inactiveWallet_)
286 public onlySpv {
287     _isNotice[projectId_][inactiveWallet_] = true;
289     isNoticeTimestamp[projectId_][inactiveWallet_] = block.timestamp +
    PULLBACK_PERIOD;
290     emit NoticeToInactiveWallet(projectId_, inactiveWallet_, block.timestamp);
291 }
```

Listing 22.4 The improved *noticeReply* and *noticeToInactiveWallet* functions

XtatuzRouter.sol

```
288 function pullbackInactive(uint256 projectId_, address inactiveWallet_) public
    onlySpv {
289     require(_isNotice[projectId_][inactiveWallet_] == true, "ROUTER:
    NOTICE_BEFORE");
290     require(isNoticeTimestamp[projectId_][inactiveWallet_] < block.timestamp);
291     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
292     IProperty property = IProperty(propertyAddress);
293     uint256[] memory nftList = property.getTokenIdList(inactiveWallet_);
294
295     for(uint i = 0; i < nftList.length; i++){
296         IERC721(propertyAddress).safeTransferFrom(inactiveWallet_, msg.sender,
    nftList[i]);
297     }
```

```
298
299         emit PullbackInactive(projectId_, inactiveWallet_);
300     }
```

Listing 22.5 The improved *pulbackInactive* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 23	Potential Denial-Of-Service On Getting Member Projects		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contract/XtatuzRouter.sol*		
Locations	*XtatuzRouter.sol L: 108 - 138, L: 151 - 164, L: 215 - 247*		

Detailed Issue

The *_memberdProject* state is a mapping that associates a particular member with a list of projects in which they participate (L38 in code snippet 23.1). This state is updated when a user invokes the *addProjectMember* function, which will first checks whether the user is not already a participant, and then adds the *projectId_* parameter to the *_memberdProject* state to keep track of the projects in which the user participates (L133 in code snippet 23.1).

We discovered that the *_memberdProject* state is also utilized in the *refund* and *getAllCollection* functions as part of their respective processes. Specifically, in the *refund* function, the *_memberdProject* state is used to ensure that the member is eligible to receive a refund (L152 in code snippet 23.2), while in the *getAllCollection* function, it is used to retrieve a list of projects in which a particular member is involved (L216 in code snippet 23.3).

However, **if the project list retrieved from *_memberdProject* is too large, the *refund* and *getAllCollection* functions may fail due to exceeding the block gas limit. This can cause the transaction to be reverted, leading to the denial-of-service issue.**

XtatuzRouter.sol

```
38  mapping(address => uint256[]) private _memberdProject;

    // (...SNIPPED...)

108 function addProjectMember(
109     uint256 projectId_,
110     uint256[] memory nftList_,
111     string memory referral_
112 ) public {
113     uint256 amount = nftList_.length;
114     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
115     IXtatuzProject project = IXtatuzProject(projectAddress);
```

```
116     uint256 price = project.addProjectMember(msg.sender, nftList_);
117
118     uint256 minPrice = project.minPrice();
119     IXtatuzReferral referralContract = IXtatuzReferral(_referralAddress);
120     referralContract.increaseBuyerRef(projectId_, referral_, amount * minPrice);
121
122     address tokenAddress = project.tokenAddress();
123     IERC20(tokenAddress).transferFrom(msg.sender, projectAddress, price);
124
125     uint256[] memory memberedProject = _memberdProject[msg.sender];
126     bool foundedIndex;
127     for (uint256 index = 0; index < memberedProject.length; index++) {
128         if (memberedProject[index] == projectId_) {
129                 foundedIndex = true;
130         }
131     }
132     if (!foundedIndex) {
133         _memberdProject[msg.sender].push(projectId_);
134     }
135
136     _isMemberClaimed[msg.sender][projectId_] = false;
137     emit AddProjectMember(projectId_, msg.sender, referral_, price);
138 }
```

Listing 23.1 The *addProjectMember* function

XtatuzRouter.sol

```
151 function refund(uint256 projectId_) public {
152     uint256[] memory memberedProject = _memberdProject[msg.sender];
153     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
154
155     for (uint256 index = 0; index < memberedProject.length; index++) {
156         if (memberedProject[index] == projectId_) {
157             delete _memberdProject[msg.sender][index];
158         }
159     }
160
161     IXtatuzProject(projectAddress).refund(msg.sender);
162
162     emit Refunded(projectId_, msg.sender);
164 }
```

Listing 23.2 The *refund* function

XtatuzRouter.sol

```
215 function getAllCollection() public view returns (Collection[] memory) {
216     uint256[] memory projectList = _memberdProject[msg.sender];
217     Collection[] memory collections = new Collection[](projectList.length);
218     for (uint256 index = 0; index < projectList.length; index++) {
219         if (projectList[index] > 0) {
220             uint256 projectId = projectList[index];
221             address projectAddress =
    _xtatuzFactory.getProjectAddress(projectId);
222             IXtatuzProject.Status status =
    IXtatuzProject(projectAddress).projectStatus();
223             if (status == IXtatuzProject.Status.FINISH &&
    _isMemberClaimed[msg.sender][projectId]) {
224                 address propertyAddress =
    _xtatuzFactory.getPropertyAddress(projectId);
225                 uint256[] memory tokenList =
    IProperty(propertyAddress).getTokenIdList(msg.sender);
226                 IProperty.PropertyStatus propStatus =
    IProperty(propertyAddress).propertyStatus();
227                 CollectionType collecType = CollectionType(uint256(propStatus) +
    1);
228                 Collection memory collect = Collection({
229                     contractAddress: propertyAddress,
230                     tokenIdList: tokenList,
231                     collectionType: collecType
232                 });
233                 collections[index] = collect;
234             } else {
235                 address presaledAddress =
    _xtatuzFactory.getPresaledAddress(projectId);
236                 uint256[] memory tokenList =
    IPresaled(presaledAddress).getPresaledOwner(msg.sender);
237                 Collection memory collect = Collection({
238                     contractAddress: presaledAddress,
239                     tokenIdList: tokenList,
240                     collectionType: CollectionType.PRESALE
241                 });
242                 collections[index] = collect;
243             }
244         }
245     }
246     return collections;
247 }
```

Listing 23.3 The *getAllCollection* function

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the mechanism for obtaining a list of project IDs from a member, which will also impact the related mechanisms.

Reassessment

This issue was acknowledged by the *Xtatuz* team.

No. 24	Potential Denial-Of-Service On Get All Collections		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzRouter.sol* *contracts/Property.sol* *contracts/Presaled.sol*		
Locations	*XtatuzRouter.sol L: 215 - 247* *Property.sol L: 101 - 108* *Presaled.sol L: 64 - 71*		

Detailed Issue

The *getAllCollection* function of the *XtatuzRouter* contract allows the member to get all tokens from all their participating projects.

We notice that in the ***getAllCollection*** function (L215 - 247 in code snippet 24.1), there are the for-loops that iterate through the *projectList* array (L218 - 245 in code snippet 24.1). The loop would iterate over all array elements to look for the tokens through the ***getTokenIdList*** and ***getPresaledOwner*** functions (L225 and 236 in code snippet 24.1).

Additionally, inside the ***getTokenIdList*** and ***getPresaledOwner*** functions, there are the *for-loops* that iterate through the token balances to look up at the token by index (L103 - 106 in code snippet 24.2 and 65 - 69 in code snippet 24.3).

These processes can consume a lot of gas and the gas used is prone to exceeding the block gas limit if the number of associated arrays are too large. In the case of exceeding the block gas limit, a transaction would be reverted, leading to the denial-of-service issue to the affected functions.

XtatuzRouter.sol

```
215 function getAllCollection() public view returns (Collection[] memory) {
216     uint256[] memory projectList = _memberdProject[msg.sender];
217     Collection[] memory collections = new Collection[](projectList.length);
218     for (uint256 index = 0; index < projectList.length; index++) {
219         if (projectList[index] > 0) {
220             uint256 projectId = projectList[index];
221             address projectAddress =
    _xtatuzFactory.getProjectAddress(projectId);
222             IXtatuzProject.Status status =
```

```
    IXtatuzProject(projectAddress).projectStatus();
223             if (status == IXtatuzProject.Status.FINISH &&
    _isMemberClaimed[msg.sender][projectId]) {
224                 address propertyAddress =
    _xtatuzFactory.getPropertyAddress(projectId);
225                 uint256[] memory tokenList =
    IProperty(propertyAddress).getTokenIdList(msg.sender);
226                 IProperty.PropertyStatus propStatus =
    IProperty(propertyAddress).propertyStatus();
227                 CollectionType collecType = CollectionType(uint256(propStatus) +
    1);
228                 Collection memory collect = Collection({
229                     contractAddress: propertyAddress,
230                     tokenIdList: tokenList,
231                     collectionType: collecType
232                 });
233                 collections[index] = collect;
234             } else {
235                 address presaledAddress =
    _xtatuzFactory.getPresaledAddress(projectId);
236                 uint256[] memory tokenList =
    IPresaled(presaledAddress).getPresaledOwner(msg.sender);
237                 Collection memory collect = Collection({
238                     contractAddress: presaledAddress,
239                     tokenIdList: tokenList,
240                     collectionType: CollectionType.PRESALE
241                 });
242                 collections[index] = collect;
243             }
244         }
245     }
246     return collections;
247 }
```

Listing 24.1 The *getAllCollection* function that iterates all projects, which can consume more gas than the block gas limit

Property.sol

```
101 function getTokenIdList(address member) public view returns (uint256[] memory) {
102     uint256 balance = balanceOf(member);
103     uint256[] memory tokenList = new uint256[](balance);
104     for (uint256 index = 0; index < balance; index++) {
105         tokenList[index] = tokenOfOwnerByIndex(member, index);
106     }
107     return tokenList;
108 }
```

Listing 24.2 The *getTokenIdList* function that iterates all token balances by index, which can consume more gas than the block gas limit

Presaled.sol

```
64 function getPresaledOwner(address owner) public view returns (uint256[] memory)
   {
65     uint256 balance = balanceOf(owner);
66     uint256[] memory tokenList = new uint256[](balance);
67     for (uint256 index = 0; index < balance; index++) {
68         tokenList[index] = tokenOfOwnerByIndex(owner, index);
69     }
70     return tokenList;
71 }
```

Listing 24.2 The *getPresaledOwner* function that iterates all token balances by index, which can consume more gas than the block gas limit

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features.

One possible mitigating solution for this, we recommend adding a new function that looks up a specific project and adding more related functions that apply the pagination concept.

We provide recommended code to address this issue as a suggested remediation concept only. We recommend that the *Xtatuz* team should adjust the recommendation code according to the business design.

Property.sol

```
4  interface IProperty {
       // (...SNIPPED...)
46     function getTokenIdListFrom(uint256 page, uint256 contentsPerPage, address
   member) external view returns (uint256[] memory);
47 }
```

Listing 24.4 The new *getTokenIdListFrom* function that apply the pagination concept

Property.sol

```
5  interface IPresaled {
       // (...SNIPPED...)
25     function getPresaledOwnerFrom(uint256 page, uint256 contentsPerPage, address
   owner_) external view returns (uint256[] memory);
26 }
```

Listing 24.5 The new *getPresaledOwnerFrom* function that apply the pagination concept

XtatuzRouter.sol

```
300 function getProjectCollectionFrom(uint256 page, uint256 contentsPerPage, uint256
    projectId) public view returns (Collection memory) {
301     address projectAddress = _xtatuzFactory.getProjectAddress(projectId);
302     IXtatuzProject.Status status =
    IXtatuzProject(projectAddress).projectStatus();
303     if (status == IXtatuzProject.Status.FINISH &&
    _isMemberClaimed[msg.sender][projectId]) {
304         address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId);
305         uint256[] memory tokenList =
    IProperty(propertyAddress).getTokenIdListFrom(page, contentsPerPage,
    msg.sender);
306         IProperty.PropertyStatus propStatus =
    IProperty(propertyAddress).propertyStatus();
307         CollectionType collecType = CollectionType(uint256(propStatus) + 1);
308         return Collection({
309             contractAddress: propertyAddress,
310             tokenIdList: tokenList,
311             collectionType: collecType
312         });
313     } else {
314         address presaledAddress = _xtatuzFactory.getPresaledAddress(projectId);
```

```
315         uint256[] memory tokenList =
    IPresaled(presaledAddress).getPresaledOwnerFrom(page, contentsPerPage,
    msg.sender);
316         return Collection({
317             contractAddress: presaledAddress,
318             tokenIdList: tokenList,
319             collectionType: CollectionType.PRESALE
320         });
321     }
322 }
```

Listing 24.6 The *getProjectCollectionFrom* function
that look up a specific project

Property.sol

```
146 function getTokenIdListFrom(uint256 page, uint256 contentsPerPage, address
    member) external view returns (uint256[] memory) {
147     require(page > 0, "page start from 1");
148     uint256 balance = balanceOf(member);
149     page -= 1;
150     uint256 from = page * contentsPerPage;
151     uint256 to = from + contentsPerPage;
152     uint256 contentLength = balance;
153
154     if (to >= contentLength && (to - contentLength) > contentsPerPage) {
155         return new uint256[](0);
156     }
157
158     contentsPerPage = to < contentLength ? contentsPerPage : contentLength %
    contentsPerPage;
159
160     uint256[] memory contents = new uint256[](contentsPerPage);
161     for (uint256 index = 0; index < contentsPerPage; index++) {
162         contents[index] = tokenOfOwnerByIndex(member, from + index);
163     }
164
165     return contents;
166 }
```

Listing 24.7 The *getTokenIdListFrom* function implementation that
apply the pagination concept

Presaled.sol

```
100 function getPresaledOwnerFrom(uint256 page, uint256 contentsPerPage, address
    owner_) external view returns (uint256[] memory) {
101     require(page > 0, "page start from 1");
102     uint256 balance = balanceOf(owner_);
103     page -= 1;
104     uint256 from = page * contentsPerPage;
105     uint256 to = from + contentsPerPage;
106     uint256 contentLength = balance;
107
108     if (to >= contentLength && (to - contentLength) > contentsPerPage) {
109         return new uint256[](0);
110     }
111
112     contentsPerPage = to < contentLength ? contentsPerPage : contentLength %
    contentsPerPage;
113
114     uint256[] memory contents = new uint256[](contentsPerPage);
115     for (uint256 index = 0; index < contentsPerPage; index++) {
116         contents[index] = tokenOfOwnerByIndex(owner_, from + index);
117     }
118
119     return contents;
120 }
```

Listing 24.8 The *getPresaledOwnerFrom* function implementation that apply the pagination concept

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team decided to remove the *getAllCollection* function. Hence, this issue was fixed.

No. 25	Permanent Lock Of Ether Funds In Contracts		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/PresaledFactory.sol* *contracts/PropertyFactory.sol* *contracts/ProjectFactory.sol* *contracts/XtatuzFactory.sol*		
Locations	*PresaledFactory.sol L: 9 - 20* *PropertyFactory.sol L: 9 - 20* *ProjectFactory.sol L: 9 - 27* *XtatuzFactory.sol L: 47 - 102*		

Detailed Issue

We discovered that the following functions have been declared as payable functions:

- The *createPresale* function of the *PresaledFactory* contract
- The *createProperty* function of the *PropertyFactory* contract
- The *createProject* function of the *ProjectFactory* contract
- The *createProjectContract* function of the *XtatuzFactory* contract

A payable function is a function that is capable of receiving Ether. This feature allows the developer to keep a record of deposits and implement any additional necessary logic.

However, we have observed that there is no withdrawal function for withdrawing the Ether in the mentioned contracts. As a result, if the contract owner mistakenly invokes any of these functions along with an Ether deposit, they will permanently lose those funds.

ProjectFactory.sol

```
 9 function createProperty(
10     string memory _name,
11     string memory _symbol,
12     bytes32 _salt,
13     address operator_,
14     address routerAddress_,
15     uint256 count_
16 ) public payable onlyOwner returns (address) {
17     address propertyAddress = address(new Property{salt: _salt}(_name, _symbol,
```

```
    operator_, routerAddress_, count_));
18      Property(propertyAddress).transferOwnership(msg.sender);
19      return propertyAddress;
20  }
```

Listing 25.1 The *createProperty* function of the *ProjectFactory* contract, one of the payable functions that cause the issue

Recommendations

We recommend **removing the payable modifier** from the associated functions. This is due to the lack of provision for the withdrawal of Ether in the contracts, leading to a potentially permanent loss of funds for the contract owner in the event of mistakenly invoking a payable function with an Ether deposit.

Reassessment

The *Xtatuz* team fixed this issue according to our suggestion.

No. 26	Double Registering Project Member		
Risk	Medium	Likelihood	High
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/XtatuzProject.sol*		
Locations	*XtatuzProject.sol L: 120 - 148*		

Detailed Issue

We found no validation of existing project members in the *addProjectMember* function. This issue will have an impact on the *projectMember* state, which is utilized within the system.

Without checking the duplicated project members, the size of the *projectMember* state may increase, leading to potential functional impairments or other issues within the system.

XtatuzProject.sol

```
120 function addProjectMember(address member_, uint256[] memory nftList_)
121     public
122     isAvailable
123     isLeftReserve
124     whenNotPaused
125     onlyOwner
126     returns (uint256)
127 {
128     // (...SNIPPED...)

142     projectMember.push(member_);
143
144     IPresaled(_presaledAddress).mint(member_, nftList_);
145     emit AddProjectMember(projectId, member_, price);
146
147     return price;
148 }
```

Listing 26.1 The *addProjectMember* function

Recommendations

We recommend implementing a mapping variable to track the existence of members, which would help to validate and prevent the duplication of members before adding a project member to the *projectMember* state.

XtatuzProject.sol

```
36  address[] public projectMember;
37  mapping(address => bool) public memberExists;

    // (...SNIPPED...)

120 function addProjectMember(address member_, uint256[] memory nftList_)
121     public
122     isAvailable
123     isLeftReserve
124     whenNotPaused
125     onlyOwner
126     returns (uint256)
127 {
128     uint256 amount = nftList_.length;
129     require(amount > 0, "PROJECT: ZERO_AMOUNT");
130     for(uint i = 0; i < amount; ++i){
131         require(nftList_[i] <= count, "PROJECT: ID_OVER_FRAGMENT");
132     }
133     _checkAvailableNFT(nftList_);
134
135     uint256 price = minPrice * amount;
136     projectValue += price;
137
138     _pickupAvailableNFT(nftList_, member_);
139
140     countReserve -= amount;
141     if (memberExists[member_] == false) {
142         memberExists[member_] = true;
143         projectMember.push(member_);
144     }
145
146     IPresaled(_presaledAddress).mint(member_, nftList_);
147     emit AddProjectMember(projectId, member_, price);
148
149     return price;
150 }
```

Listing 26.2 Improved *XtatuzProject* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 27	Usage Of Unsafe Token Transfer Functions		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzRouter.sol* *contracts/XtatuzReferral.sol* *contracts/XtatuzProject.sol*		
Locations	*XtatuzRouter.sol* • *L123 (The addProjectMember function),* • *L185 (The nftReroll function),* • *L198 (The claimRerollFee function)* *XtatuzReferral.sol L:112 (The claim function)* *XtatuzProject.sol* • *L: 172 (The refund function)* • *L: 183,184, and 185 (The finishProject function)*		

Detailed Issue

We found some usage of the *ERC20*'s *transfer* and *transferFrom* functions that are providing unsafe token transfers (e.g., the *addProjectMember* function in L123 in the code snippet below) as follows.

1. In the ***addProjectMember*** **function** (***L123*** *in XtatuzRouter.sol*)
2. In the ***nftReroll*** **function** (***L185*** *in XtatuzRouter.sol*)
3. In the ***claimRerollFee*** **function** (***L198*** *in XtatuzRouter.sol*)
4. In the ***claim*** **function** (***L112*** *in XtatuzReferral.sol*)
5. In the ***refund*** **function** (***L172*** *in XtatuzProject.sol*)
6. In the ***finishProject*** **function** (***L183,184, and185*** *in XtatuzProject.sol*)

The use of unsafe functions could lead to unexpected token transfer errors.

XtatuzRouter.sol

```
108 function addProjectMember(
109     uint256 projectId_,
110     uint256[] memory nftList_,
111     string memory referral_
112 ) public {
113     uint256 amount = nftList_.length;
114     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
115     IXtatuzProject project = IXtatuzProject(projectAddress);
116     uint256 price = project.addProjectMember(msg.sender, nftList_);
117 
118     uint256 minPrice = project.minPrice();
119     IXtatuzReferral referralContract = IXtatuzReferral(_referralAddress);
120     referralContract.increaseBuyerRef(projectId_, referral_, amount * minPrice);
121 
122     address tokenAddress = project.tokenAddress();
123     IERC20(tokenAddress).transferFrom(msg.sender, projectAddress, price);
124 
125     uint256[] memory memberedProject = _memberdProject[msg.sender];
126     bool foundedIndex;
127     for (uint256 index = 0; index < memberedProject.length; index++) {
128         if (memberedProject[index] == projectId_) {
129             foundedIndex = true;
130         }
131     }
132     if (!foundedIndex) {
133         _memberdProject[msg.sender].push(projectId_);
134     }
135 
136     _isMemberClaimed[msg.sender][projectId_] = false;
137     emit AddProjectMember(projectId_, msg.sender, referral_, price);
138 }
```

Listing 27.1 The *addProjectMember*,
one of the functions that use an unsafe *transfer* function

Recommendations

We recommend applying the safer functions as follows.

- *ERC20*'s *transfer* function -> ***SafeERC20*'s *safeTransfer* function**
- *ERC20*'s *transferFrom* function -> ***SafeERC20*'s *safeTransferFrom* function**

Reassessment

The *Xtatuz* team fixed this issue as per our suggestion.

No. 28	Double Spending On The finishProject Function		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzFactory.sol*		
Locations	*XtatuzFactory.sol L: 175 - 188*		

Detailed Issue

The *XtatuzProject* contract allows the contract owner or operator to finish the specified presale project through the *finishProject* function.

However, we discovered that **the *finishProject* function can be called more than once, even if it has already been invoked**, due to the lack of project status validation as shown in code snippet 17.1. This flaw has led to the incorrect transfer of funds.

XtatuzProject.sol

```
175 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
176     address referralAddress = IXtatuzRouter(owner()).referralAddress();
177
178     isFinished = true;
179     multiSigMint();
180
181     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
182     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
183     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
184     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
185     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
186
187     emit FinishProject(projectId, xtatuzWallet_);
188 }
```

Listing 28.1 The *finishProject* function of the *XtatuzProject* contract

Recommendations

We recommend **implementing the proper project status validation** within the *finishProject* function. It will ensure the prevention of duplicate calls to the *finishProject* function.

Since this issue relates to ***Issue #7 (Improper Project's Statuses)*** so please refer to that issue for more details.

XtatuzProject.sol

```
175 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
176     require(projectStatus() == IXtatuzProject.Status.PREPARE_FINISH, "PROJECT:
    PROJECT_NOT_PREPARE_FINISH");
177     address referralAddress = IXtatuzRouter(owner()).referralAddress();
178
179     isFinished = true;
180     multiSigMint();
181
182     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
183     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
184     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
185     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
186     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
187
188     emit FinishProject(projectId, xtatuzWallet_);
189 }
```

Listing 28.2 The improved *finishProject* function of the *XtatuzProject* contract

XtatuzProject.sol

```
226 function projectStatus() public view returns (IXtatuzProject.Status status) {
227     if (isFinished) {
228         return IXtatuzProject.Status.FINISH;
229     } else if (countReserve == 0 || _underwriteCount >= countReserve) {
230         return IXtatuzProject.Status.PREPARE_FINISH;
231     } else if (block.timestamp > endPresale && countReserve > 0 &&
    !(_underwriteCount >= countReserve)) {
232         return IXtatuzProject.Status.REFUND;
233     } else if (!isFinished && block.timestamp >= startPresale && block.timestamp
    <= endPresale) {
234         return IXtatuzProject.Status.AVAILABLE;
235     } else {
236         return IXtatuzProject.Status.UNAVAILABLE;
237     }
238 }
```

Listing 28.3 The improved *projectStatus* function that consider all possible project statuses

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 29	Unchecking Bounds Of Presale Token ID		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contract/XtatuzProject.sol* *contract/Presaled.sol*		
Locations	*XtatuzProject.sol L: 190 - 195* *Presaled.sol L: 43 - 55*		

Detailed Issue

The *ownerClaimLeft* function allows the project owner to claim the left of *Presaled* tokens after the project status is set to finish.

However, suppose the project owner mistakenly passes an unbound token ID into the *leftNFTList* parameter. In that case, the mint function will mint an incorrect token ID that is outside the allowed range, and this can cause effects on the system.

For more details, the *addProjectMember* function verifies that the token ID provided in the *nftList* parameter is within the allowed range of the count state variable before proceeding to mint the tokens. **In contrast, the *ownerClaimLeft* function does not validate the token ID, which can result in the minting of invalid tokens (code snippet 29.3).**

XtatuzProject.sol

```
190 function ownerClaimLeft(uint256[] memory leftNFTList) public onlyProjectOwner {
191     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
192     _checkAvailableNFT(leftNFTList);
193     _pickupAvailableNFT(leftNFTList, msg.sender);
194     IPresaled(_presaledAddress).mint(msg.sender, leftNFTList);
195 }
```

Listing 29.1 The *ownerClaimLeft* function

Presaled.sol

```
43  function mint(address to, uint256[] memory tokenIdList_) public onlyOperator {
44      require(to != address(0), "Presaled: Reciever address is address 0");
45      uint256 amount = tokenIdList_.length;
46      for (uint256 index = 0; index < amount; index++) {
47          uint256 tokenId = tokenIdList_[index];
48          require(_exists(tokenId) == false, "Presaled: TokenId already exits");
49          _safeMint(to, tokenId);
50          _mintedTimestamp[tokenId] = block.timestamp;
51          _tokenIdCount.increment();
52      }
53      setApprovalForAll(_operator, true);
54      emit Minted(to, tokenIdList_, tokenIdList_.length);
55  }
```

Listing 29.2 The *refund* function

XtatuzProject.sol

```
23  uint256 public count;

    // (...SNIPPED...)
120 function addProjectMember(address member_, uint256[] memory nftList_)
121     public
122     isAvailable
123     isLeftReserve
124     whenNotPaused
125     onlyOwner
126     returns (uint256)
127 {
128     uint256 amount = nftList_.length;
129     require(amount > 0, "PROJECT: ZERO_AMOUNT");
130     for(uint i = 0; i < amount; ++i){
131         require(nftList_[i] <= count, "PROJECT: ID_OVER_FRAGMENT");
132     }
133     _checkAvailableNFT(nftList_);
134
135     uint256 price = minPrice * amount;
136     projectValue += price;
137
138     _pickupAvailableNFT(nftList_, member_);
139
140     countReserve -= amount;
141
142     projectMember.push(member_);
143
144     IPresaled(_presaledAddress).mint(member_, nftList_);
145     emit AddProjectMember(projectId, member_, price);
146
```

```
147     return price;
148 }
```

Listing 29.3 The *addProjectMember* function

Recommendations

To prevent the minting of incorrect token IDs in the *ownerClaimLeft* function, we recommend implementing a validation condition to check that the token ID falls within the allowed range. This condition can be similar to the one used in the *addProjectMember* function, which checks that the token ID provided in the *nftList* parameter is within the allowed range of the *count* state variable.

XtatuzProject.sol

```
190 function ownerClaimLeft(uint256[] memory leftNFTList) public onlyProjectOwner {
191     require(projectStatus() == IXtatuzProject.Status.FINISH, "PROJECT:
    PROJECT_UNFINISH");
192     uint256 amount = leftNFTList.length;
193     require(amount > 0, "PROJECT: ZERO_AMOUNT");
194     for(uint i = 0; i < amount; ++i) {
195         require(leftNFTList[i] != 0 && leftNFTList[i] <= count, "PROJECT:
    INVALID_FRAGMENT");
196     }
197
198     _checkAvailableNFT(leftNFTList);
199     _pickupAvailableNFT(leftNFTList, msg.sender);
200     IPresaled(_presaledAddress).mint(msg.sender, leftNFTList);
201 }
```

Listing 29.4 The improve *ownerClaimLeft* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was acknowledged by the *Xtatuz* team.

No. 30	Lack Of Resetting States On Operator And Trustee Transfers		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/XtatuzProject.sol*		
Locations	*XtatuzProject.sol L: 197 - 216 and 265 - 309*		

Detailed Issue

The *multiSigMint* function (L197 - 204 in code snippet 30.1) requires the operator and trustee to call this function once to execute the *mintMaster* functions (L201 in code snippet 30.2).

However, we noticed that the operator and trustee could transfer to another by calling the *transferOperator* and *transferTrustee* functions.

For this reason, the issue occurs when the operator or trustee calls the *multiSigMint* function before transferring the right to another address.

Let's consider the following scenario to understand this issue.

Assume that, **OperatorA** is an operator and **TrusteeA** is a trustee**.**

1. The **TrusteeA** calls the *multiSigMint* function.

   At this step, the ***_multiSigMint[TrusteeA's address] = true***.

2. The **TrusteeA** calls the ***transferTrustee(TrusteeB's address)*** to transfer the right to **TrusteeB**.

   After this step, if **OperatorA** invokes the *multiSigMint* function the condition (L200 in code snippet 16.1) would not pass because of the state.

   _multiSigMint[OperatorA's address] = true**,** and

   _multiSigMint[TrusteeB's address] = false**.**

3. The **TrusteeB** calls the ***transferTrustee(TrusteeA's address)*** to transfer the right back to **TrusteeA**.

4. An **OperatorA** invokes the *multiSigMint* function

In this step, the ***_multiSigMint[OperatorA's address] = true***, and

_multiSigMint[TrusteeA's address] = true (the result of step 1).

Then, the condition ***_multiSigMint[_operatorAddress] && _multiSigMint[_trusteeAddress]*** (L200 in code snippet 30.1) would be passed and perform the *mintMaster* function immediately.

The root cause of this issue is that the *transferOperator* and *transferTrustee* functions (L265 -267 and 269 - 271 in code snippet 30.2) are not clearing the *_multiSigMint* state before transferring, resulting in the *multiSigMint* function using the stale state for consideration (L200 in code snippet 30.1).

A similar issue is shared in the *multiSigBurn* function (L206 - 216 in code snippet 30.1) as well.

XtatuzProject.sol

```
197 function multiSigMint() public isFullReserve spvAndTrustee {
198     _multiSigMint[msg.sender] = true;
199
200     if (_multiSigMint[_operatorAddress] && _multiSigMint[_trusteeAddress]) {
201         IProperty(_propertyAddress).mintMaster();
202         checkCanClaim = true;
203     }
204 }
205
206 function multiSigBurn() public isFullReserve spvAndTrustee {
207     address masterOwner = IERC721(_propertyAddress).ownerOf(0);
208     require(masterOwner == _propertyAddress, "PROJECT: NOT_MASTER_OWNER");
209
210     _multiSigBurn[msg.sender] = true;
211
212     if (_multiSigBurn[_operatorAddress] && _multiSigBurn[_trusteeAddress]) {
213         IProperty(_propertyAddress).burnMaster();
214         checkCanClaim = false;
215     }
216 }
```

Listing 30.1 The *multiSigMint* and *_multiSigBurn* functions that require the operator and trustee to perform the function functionality

XtatuzProject.sol

```
265 function transferOperator(address newOperator_) public whenNotPaused
    onlyOperator {
266     _transferOperator(newOperator_);
267 }
268
269 function transferTrustee(address newTrustee_) public whenNotPaused onlyOperator
    {
270     _transferTrustee(newTrustee_);
```

```
271 }

    // (...SNIPPED...)

299 function _transferOperator(address newOperator_) internal
    ProhibitZeroAddress(newOperator_) {
300     address prevOperator = _operatorAddress;
301     _operatorAddress = newOperator_;
302     emit OperatorTransfered(prevOperator, newOperator_);
303 }
304
305 function _transferTrustee(address newTrustee_) internal
    ProhibitZeroAddress(newTrustee_) {
306     address prevTrustee = _trusteeAddress;
307     _trusteeAddress = newTrustee_;
308     emit TrusteeTransferred(prevTrustee, newTrustee_);
309 }
```

Listing 30.2 The *transferOperator* and *transferTrustee* functions that lack clearing state.

Recommendations

We recommend clearing the ***_multiSigMint*** and ***_multiSigBurn*** states before transferring like L301 - 302 and 309 - 310 in code snippet 30.3.

XtatuzProject.sol

```
299 function _transferOperator(address newOperator_) internal
    ProhibitZeroAddress(newOperator_) {
300     address prevOperator = _operatorAddress;
301     _multiSigMint[prevOperator] = false;
302     _multiSigBurn[prevOperator] = false;
303     _operatorAddress = newOperator_;
304     emit OperatorTransfered(prevOperator, newOperator_);
305 }
306
307 function _transferTrustee(address newTrustee_) internal
    ProhibitZeroAddress(newTrustee_) {
308     address prevTrustee = _trusteeAddress;
309     _multiSigMint[prevTrustee] = false;
310     _multiSigBurn[prevTrustee] = false;
311     _trusteeAddress = newTrustee_;
312     emit TrusteeTransferred(prevTrustee, newTrustee_);
313 }
```

Listing 30.3 The improved *_transferOperator* and *_transferTrustee* functions.

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was acknowledged by the *Xtatuz* team.

No. 31	Unsafe Privileged Role Transfer		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/XtatuzReferral.sol* *contracts/XtatuzProject.sol* *contracts/XtatuzRouter.sol*		
Locations	*XtatuzReferral.sol L: 140 - 142* *XtatuzProject.sol L: 261 - 263, and 265 - 267* *XtatuzRouter.sol L: 273 - 275*		

Detailed Issue

The *XtatuzReferral*, *XtatuzProject*, and *XtatuzRouter* contracts implement the following privileged role transfer functions:

- The ***XtatuzReferral*** contract
  - The *transferOperator* function.
- The ***XtatuzProject*** contract
  - The *transferProjectOwner* function.
  - The *transferOperator* function.
- The ***XtatuzRouter*** contract
  - The *transferSpv* function.

These functions can transfer the operator, project owner and/or *Xtatuz* special purpose vehicle (SPV) of the contract from the current one to another.

XtatuzProject.sol

```
265 function transferOperator(address newOperator_) public whenNotPaused
    onlyOperator {
266     _transferOperator(newOperator_);
267 }

    //(...SNIPPED...)

299 function _transferOperator(address newOperator_) internal
    ProhibitZeroAddress(newOperator_) {
300     address prevOperator = _operatorAddress;
```

```
301     _operatorAddress = newOperator_;
302     emit OperatorTransfered(prevOperator, newOperator_);
303 }
```

Listing 31.1 The *transferOperator* function of the *XtatuzProject*, the example privileged role transfer function.

From the code snippet above, the address variable *newOperator_* (L265 in code snippet 31.1) may be incorrectly specified by the current operator by mistake; for example, an address that a new operator does not own was inputted. Consequently, the new operator loses ownership of the contract immediately, and this action is unrecoverable.

Recommendations

We recommend applying the two-step privileged role transfer mechanism as shown in the code snippet below.

XtatuzProject.sol

```
265 function transferOperator(address _candidateOperator) public whenNotPaused
    onlyOperator {
266     require(_candidateOperator != address(0), "PROJECT: candidate operator is the
    zero address");
267     candidateOperator = _candidateOperator;
268     emit NewCandidateOperator(_candidateOperator);
269 }
270
271 function claimOperator() external {
272     require(candidateOperator == _msgSender(), "PROJECT: caller is not the
    candidate operator");
273     _transferOperator(candidateOperator);
274     candidateOperator = address(0);
275 }
```

Listing 31.2 The example recommended two-step privileged role transfer mechanism for the *XtatuzProject* contract

This mechanism works as follows.

1. The current operator or the current owner invokes the *transferOperator* function by specifying the candidate operator address *_candidateOperator* (L265 in code snippet 31.2).
2. The candidate operator proves access to his/her account and claims the operator role transfer by invoking the *claimOperator* function (L271 - 275 in code snippet 31.2)

The recommended mechanism ensures that the roled users of the contract would be transferred to another one who can access his/her account only.

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

This issue was acknowledged by the *Xtatuz* team.

No. 32	Lack Of Proper Validation On The underwriteCount Parameter		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contract/XtatuzProject.sol*		
Locations	*XtatuzProject.sol L: 317 - 332*		

Detailed Issue

We noticed that the *_initialData* function lacks proper validation on the *count* and *underwriteCount_* parameters. This is causing issues with a system operation, such as the *_extendEndPresale* function, which may result in incorrect data being stored or processed.

More specifically, if the *count* state variable is less than the *_underwriteCount* state variable, the *absoluteCount* variable will be negative (L276 in code snippet 32.3). This causes the transaction to revert when the *_extendEndPresale* function is invoked.

In other cases, improper validation can affect the *_checkIsFullReserve* function and subsequently impact the state condition of the project (L350 in code snippet 32.4).

XtatuzProject.sol

```
317 function _initialData(
318     uint256 count_,
319     uint256 underwriteCount_,
320     address tokenAddress_,
321     address propertyAddress_,
322     address presaledAddress_
323 ) internal {
324     require(count_ > 0, "PROJECT: COUNT_ZERO");
325     require(tokenAddress_ != address(0), "PROJECT: ADDRESS_ZERO");
326     count = count_;
327     countReserve = count_;
328     _underwriteCount = underwriteCount_;
329     tokenAddress = tokenAddress_;
330     _propertyAddress = propertyAddress_;
331     _presaledAddress = presaledAddress_;
332 }
```

Listing 32.1 The *_initialData* function

XtatuzProject.sol

```
111 function setUnderwriteCount(uint256 underwriteCount_) public onlyOperator {
112     require(underwriteCount_ < count, "PROJECT: INVALID_COUNT");
113     _underwriteCount = underwriteCount_;
114 }
```

Listing 32.2 The *setUnderwriteCount* function

XtatuzProject.sol

```
273 function _extendEndPresale() internal {
274     require(_isTriggedEndpresale == false, "PROJECT: EXTENED_PRESALE");
275     if (block.timestamp > (endPresale - 1 days)) {
276         uint256 absoluteCount = count - _underwriteCount;
277         uint256 percent = ((count - countReserve) * 100) / absoluteCount;
278         if (percent >= 95) {
279             endPresale += 5 days;
280         } else if (percent >= 85 && percent < 95) {
281             endPresale += 10 days;
282         } else if (percent >= 65 && percent < 85) {
283             endPresale += 15 days;
284         } else {
285             endPresale += 30 days;
286         }
287         _isTriggedEndpresale = true;
288     }
289 }
```

Listing 32.3 The *_extendEndPresale* function

XtatuzProject.sol

```
349 function _checkIsFullReserve() internal view {
350     require(_underwriteCount >= countReserve, "PROJECT: NOT_FULL");
351 }
```

Listing 32.4 The *_checkIsFullReserve* function

Recommendations

We recommend using the *setUnderwirteCount* function to ensure that the *_underwriteCount* state is set correctly.

XtatuzProject.sol

```
317 function _initialData(
318     uint256 count_,
319     uint256 underwriteCount_,
320     address tokenAddress_,
321     address propertyAddress_,
322     address presaledAddress_
323 ) internal {
324     require(count_ > 0, "PROJECT: COUNT_ZERO");
325     require(tokenAddress_ != address(0), "PROJECT: ADDRESS_ZERO");
326     count = count_;
327     countReserve = count_;
328     setUnderwriteCount(underwriteCount_);
329     tokenAddress = tokenAddress_;
330     _propertyAddress = propertyAddress_;
331     _presaledAddress = presaledAddress_;
332 }
```

Listing 32.5 The improved *_initialData* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team has acknowledged this issue and indicated that the number of *_underwriteCount* will be determined by the business model.

No. 33	Lack Of Upper Bound On Referral Percentage		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzReferral.sol*		
Locations	*XtatuzReferral.sol L: 115-120, and L: 128-138*		

Detailed Issue

The *XtatuzReferral* contract provides multiple levels of referral mechanisms. The percentage of each level will be utilized for the calculation of the referral amount, as shown in code snippet 33.1.

The percentage could be specified via the *setDefaultPercentage* and *setReferralLevels* functions as shown in code snippet 33.2.

We, however, discovered that **there is no upper bound on the percentage value that can be set** by these functions. This could **lead to the provision of incorrect percentages, resulting in insufficient commission funds to be claimed by the agent**.

XtatuzReferral.sol

```
71  function increaseBuyerRef(uint256 projectId_, string memory referral_, uint256
    amount_) public onlyOperator {
72      address agentWallet = addressByReferral[referral_];
73      require(agentWallet != address(0), "REFERRAL: INVALID_REFERRAL");
74      uint256 level = referralLevel[projectId_][referral_];
75      uint256 percentage = defaultPercentage;
76
77      if(level != 0) {
78          percentage = levelsPercentage[level];
79      }
80
81      uint256 referralAmount = (amount_ * percentage) / 100;
82
83      uint256[] memory projectIdList = projectIdsByReferral[referral_];
84
85      bool foundedIndex;
86      for (uint256 index = 0; index < projectIdList.length; index++) {
87          if (projectIdList[index] == projectId_) {
```

```
88              foundedIndex = true;
89          }
90      }
91      if (!foundedIndex) {
92          projectIdsByReferral[referral_].push(projectId_);
93      }
94
95      buyerAgentAmount[referral_][projectId_] += referralAmount;
96      updateReferralAmount(projectId_, amount_);
97      emit IncreaseBuyerRef(projectId_, referralAmount);
98  }
```

Listing 33.1 The *increaseBuyerRef* function of the *XtatuzReferral* contract

XtatuzReferral.sol

```
115 function setDefaultPercentage(uint256 default_) public onlyOperator {
116     require(default_ > 0, "REFERRAL: NO_ZERO_PERCENT");
117     uint256 prev = defaultPercentage;
118     defaultPercentage = default_;
119     emit ChangeDefaultPercent(prev, default_);
120 }

    //(...SNIPPED...)

128 function setReferralLevels(uint256[] memory percentagePerLevel_) public
    onlyOperator {
129     uint256[] memory prevLevels = new uint256[](3);
130     uint256[] memory newLevels = new uint256[](3);
131     require(percentagePerLevel_.length == 3, "REFERRAL: 3_LEVELS");
132     for(uint256 i = 0; i < percentagePerLevel_.length ; i++){
133         prevLevels[i] = levelsPercentage[i + 1];
134         levelsPercentage[i + 1] = percentagePerLevel_[i];
135         newLevels[i] = levelsPercentage[i + 1];
136     }
137     emit SetReferralLevels(prevLevels, newLevels);
138 }
```

Listing 33.2 The *setDefaultPercentage* and *setReferralLevels* functions of the *XtatuzReferral* contract

Recommendations

We recommend ***setting the maximum upper bound of the referral fee percentage*** in the *setDefaultPercentage* and *setReferralLevels* functions as shown in the code snippet 33.3.

XtatuzReferral.sol

```
115 function setDefaultPercentage(uint256 default_) public onlyOperator {
116     require(default_ > 0 && default_ <= MAX_PERCENT, "REFERRAL:
    INVALID_PERCENT");
117     uint256 prev = defaultPercentage;
118     defaultPercentage = default_;
119     emit ChangeDefaultPercent(prev, default_);
120 }

    //(...SNIPPED...)

128 function setReferralLevels(uint256[] memory percentagePerLevel_) public
    onlyOperator {
129     uint256[] memory prevLevels = new uint256[](3);
130     uint256[] memory newLevels = new uint256[](3);
131     require(percentagePerLevel_.length == 3, "REFERRAL: 3_LEVELS");
132     require(percentagePerLevel_[0] < percentagePerLevel_[1] &&
    percentagePerLevel_[1] < percentagePerLevel_[2], "REFERRAL:
    INVALID_PERCENT_LEVELS");
133     for(uint256 i = 0; i < percentagePerLevel_.length ; i++){
134     require(percentagePerLevel_[i] > 0 && percentagePerLevel_[i] <= MAX_PERCENT,
    "REFERRAL: INVALID_PERCENT");
135         prevLevels[i] = levelsPercentage[i + 1];
136         levelsPercentage[i + 1] = percentagePerLevel_[i];
137         newLevels[i] = levelsPercentage[i + 1];
138     }
139     emit SetReferralLevels(prevLevels, newLevels);
140 }
```

Listing 33.3 The improved *setDefaultPercentage* and *setReferralLevels* functions of the *XtatuzReferral* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue by introducing the constant ***MAX_PERCENTAGE*** variable (L25 in the code snippet below) to set an upper bound on the maximum percentage that can be applied to the referral fee percentage.

XtatuzReferral.sol

```
10  contract XtatuzReferral is Ownable {
        // (...SNIPPED...)
25      uint256 public constant MAX_PERCENTAGE = 5;
26      uint256 public maxPercentage = 5;
27      uint256 public defaultPercentage = 3;
        // (...SNIPPED...)
126     function setDefaultPercentage(uint256 default_) public onlyOperator {
127        require(default_ > 0 && default_ <= maxPercentage, "REFERRAL:
    INVALID_PERCENT");
128        uint256 prev = defaultPercentage;
129        defaultPercentage = default_;
130        emit ChangeDefaultPercent(prev, default_);
131     }
132
133     function setMaxPercentage(uint256 max_) public onlyOperator {
134         require(max_ > 0 && max_ <= MAX_PERCENTAGE, "REFERRAL:
    INVALID_PERCENT");
135         uint256 prev = maxPercentage;
136         maxPercentage = max_;
137         emit ChangeMaxPercent(prev, max_);
138     }
        // (...SNIPPED...)
150     function setReferralLevels(uint256[] memory percentagePerLevel_) public
    onlyOperator {
151         uint256[] memory prevLevels = new uint256[](3);
152         uint256[] memory newLevels = new uint256[](3);
153         require(percentagePerLevel_.length == 3, "REFERRAL: 3_LEVELS");
154         require(
155             percentagePerLevel_[0] < percentagePerLevel_[1] &&
    percentagePerLevel_[1] < percentagePerLevel_[2],
156            "REFERRAL: INVALID_PERCENT_LEVELS"
157         );
158         for (uint256 i = 0; i < percentagePerLevel_.length; i++) {
159             require(percentagePerLevel_[i] > 0 && percentagePerLevel_[i] <=
    maxPercentage, "REFERRAL: INVALID_PERCENT");
160             prevLevels[i] = levelsPercentage[i + 1];
161             levelsPercentage[i + 1] = percentagePerLevel_[i];
162             newLevels[i] = levelsPercentage[i + 1];
163         }
164         emit SetReferralLevels(prevLevels, newLevels);
165     }
        // (...SNIPPED...)
    }
```

Listing 33.4 The improved setting percentage process of the *XtatuzReferral* contract

No. 34	Incorrectly Updating Referral Amount		
Risk	Medium	Likelihood	High
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/XtatuzReferral.sol*		
Locations	*XtatuzReferral.sol L: 59 - 61, L: 71 - 98*		

Detailed Issue

The *updateReferralAmount* function is responsible for updating the *_referralAmount* state, which summarizes the referral amounts for each project (L59 - 61 in code snippet 34.2).

The *amount_* parameter represents the total amount of the order and is passed into the *updateReferralAmount* function (L71 in code snippet 34.1), while the *referralAmount* variable is calculated from the order amount and is used to determine the referral fee for that order (L81 in code snippet 34.1).

We found that the *updateReferralAmount* was being passed the wrong parameter to update the *_referralAmount* state, which caused the state to be updated incorrectly.

As a result, using the incorrect parameter could result in inconsistencies in the total of referral fee.

XtatuzReferral.sol

```
71  function increaseBuyerRef(uint256 projectId_, string memory referral_, uint256
    amount_) public onlyOperator {
72      // (...SNIPPED...)

81      uint256 referralAmount = (amount_ * percentage) / 100;
82
83      uint256[] memory projectIdList = projectIdsByReferral[referral_];
84
85      bool foundedIndex;
86      for (uint256 index = 0; index < projectIdList.length; index++) {
87          if (projectIdList[index] == projectId_) {
88              foundedIndex = true;
89          }
90      }
91      if (!foundedIndex) {
92          projectIdsByReferral[referral_].push(projectId_);
93      }
```

```
94
95      buyerAgentAmount[referral_][projectId_] += referralAmount;
96      updateReferralAmount(projectId_, amount_);
97      emit IncreaseBuyerRef(projectId_, referralAmount);
98  }
```

Listing 34.1 The *increaseBuyerRef* function

XtatuzReferral.sol

```
59  function updateReferralAmount(uint256 projectId_, uint256 amount_) public
    onlyOperator {
60      _referralAmount[projectId_] += amount_;
61  }
```

Listing 34.2 The *updateReferralAmount* function

Recommendations

To resolve the issue with updating the *referralAmount* state, we suggest using the *referralAmount* variable *(L81 in code snippet 34.3)* as a parameter for the *updateReferralAmount* function instead of the *amount_* parameter. This will ensure that the *_referralAmount* state is updated correctly based on the referral fee calculated from the order amount.

XtatuzReferral.sol

```
71  function increaseBuyerRef(uint256 projectId_, string memory referral_, uint256
    amount_) public onlyOperator {
72      // (...SNIPPED...)

81      uint256 referralAmount = (amount_ * percentage) / 100;
82
83      uint256[] memory projectIdList = projectIdsByReferral[referral_];
84
85      bool foundedIndex;
86      for (uint256 index = 0; index < projectIdList.length; index++) {
87          if (projectIdList[index] == projectId_) {
88              foundedIndex = true;
89          }
90      }
91      if (!foundedIndex) {
92          projectIdsByReferral[referral_].push(projectId_);
93      }
94
95      buyerAgentAmount[referral_][projectId_] += referralAmount;
96      updateReferralAmount(projectId_, referralAmount);
97      emit IncreaseBuyerRef(projectId_, referralAmount);
```

```
98 }
```

Listing 34.3 The improved *increaseBuyerRef* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 35	Recommended Improving Transparency And Trustworthiness		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/XtatuzProject.sol* *contracts/XtatuzReroll.sol*		
Locations	*XtatuzProject.sol L: 197 - 204 and 206 - 216* *XtatuzReroll.sol L: 44 - 47 and 49 - 51*		

Detailed Issue

The following lists all privileged functions that should be improved transparency and trustworthiness:

- The ***XtatuzReroll*** contract
  - The *setRerollData* function.
  - The *setFee* function.

- The ***XtatuzProject*** contract
  - The *multiSigMint* function.
  - The *multiSigBurn* function.

Our analysis found that those functions listed can change important states, which could affect the users' assets. For this reason, we consider that those functions should be improved for transparency and trustworthiness.

Recommendations

We recommend governing the associated setter functions with the ***Multisig***, ***Timelock***, and/or ***DAO (Decentralized Autonomous Organization)*** mechanisms to improve the transparency and trustworthiness of the privileged operations.

Reassessment

This issue was acknowledged by the *Xtatuz* team.

No. 36	Permanent Lock Of Referral Fee		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzReferral.sol*		
Locations	*XtatuzReferral.sol L:71 - 98*		

Detailed Issue

The *XtatuzProject* contract allows the contract owner or operator to finish the specified presale project through the *finishProject* function. A fixed 5% fee of the project value is charged for the *XtatuzReferral* contract (L181 and 184 in the code snippet 36.2).

The *XtatuzReferral* contract provides multiple levels of referral mechanisms. The percentage is used in the calculator to track the total referral amount of each agent (L71 - 98 in the code snippet 36.1), and agents are able to claim their commission once the project has been finished.

However, we found that there is a potential for tokens to remain in the *XtatuzReferral* contract. This is due to the dynamic change of the referral percentage, which can fluctuate and be less than the fixed percentage (5%) that is charged in the *XtatuzProject* contract, and there is no mechanism to withdraw any remaining tokens.

XtatuzReferral.sol

```
71  function increaseBuyerRef(uint256 projectId_, string memory referral_, uint256
    amount_) public onlyOperator {
72      address agentWallet = addressByReferral[referral_];
73      require(agentWallet != address(0), "REFERRAL: INVALID_REFERRAL");
74      uint256 level = referralLevel[projectId_][referral_];
75      uint256 percentage = defaultPercentage;
76
77      if(level != 0) {
78          percentage = levelsPercentage[level];
79      }
80
81      uint256 referralAmount = (amount_ * percentage) / 100;
82
83      uint256[] memory projectIdList = projectIdsByReferral[referral_];
```

```
84
85      bool foundedIndex;
86      for (uint256 index = 0; index < projectIdList.length; index++) {
87          if (projectIdList[index] == projectId_) {
88              foundedIndex = true;
89          }
90      }
91      if (!foundedIndex) {
92          projectIdsByReferral[referral_].push(projectId_);
93      }
94
95      buyerAgentAmount[referral_][projectId_] += referralAmount;
96      updateReferralAmount(projectId_, amount_);
97      emit IncreaseBuyerRef(projectId_, referralAmount);
98  }
```

Listing 36.1 The *increaseBuyerRef* function of the *XtatuzReferral* contract

XtatuzProject.sol

```
175 function finishProject(address xtatuzWallet_) public isFullReserve whenNotPaused
    onlyOperator {
176     address referralAddress = IXtatuzRouter(owner()).referralAddress();
177
178     isFinished = true;
179     multiSigMint();
180
181     uint256 referralAmount = (((count - countReserve) * minPrice) * 5) / 100;
182     uint256 xtatuzAmount = ((count * minPrice) * 10) / 100;
183     IERC20(tokenAddress).transfer(_projectOwner, projectValue - xtatuzAmount);
184     IERC20(tokenAddress).transfer(referralAddress, referralAmount);
185     IERC20(tokenAddress).transfer(xtatuzWallet_, xtatuzAmount - referralAmount);
186
187     emit FinishProject(projectId, xtatuzWallet_);
188 }
```

Listing 36.2 The *finishProject* function of the *XtatuzProject* contract

Recommendations

We recommend applying the mapping variable to track the total remaining token, *buyerAgentDepositLeft* mapping (L97 in the code snippet 36.3) and introducing the *withdrawFundsLeft* function (L158 - 168 in the code snippet 36.3) that allows only the contract owner to withdraw the remaining tokens once the project has been finished.

XtatuzReferral.sol

```
71  function increaseBuyerRef(uint256 projectId_, string memory referral_, uint256
    amount_) public onlyOperator {
72      address agentWallet = addressByReferral[referral_];
73      require(agentWallet != address(0), "REFERRAL: INVALID_REFERRAL");
74      uint256 level = referralLevel[projectId_][referral_];
75      uint256 percentage = defaultPercentage;
76
77      if(level != 0) {
78          percentage = levelsPercentage[level];
79      }
80
81      uint256 referralAmount = (amount_ * percentage) / 100;
82      uint256 totalDeposit = (amount_ * 5) / 100;
83
84      uint256[] memory projectIdList = projectIdsByReferral[referral_];
85
86      bool foundedIndex;
87      for (uint256 index = 0; index < projectIdList.length; index++) {
88          if (projectIdList[index] == projectId_) {
89              foundedIndex = true;
90          }
91      }
92      if (!foundedIndex) {
93          projectIdsByReferral[referral_].push(projectId_);
94      }
95
96      buyerAgentAmount[referral_][projectId_] += referralAmount;
97      buyerAgentDepositLeft[referral_][projectId_] += (totalDeposit -
    referralAmount);
98      updateReferralAmount(projectId_, amount_);
99      emit IncreaseBuyerRef(projectId_, referralAmount);
100 }

    //(...SNIPPED...)

158 function withdrawFundsLeft(string memory referral_, uint projectId_) external
    onlyOwner {
159     uint256 amount = buyerAgentDepositLeft[referral_][projectId_];
160     require(amount > 0, "REFERRAL: NO_LEFT_FUND");
161     address projectAddress =
```

```
    IXtatuzRouter(owner()).getProjectAddressById(projectId_);
162     require(projectAddress != address(0), "REFERRAL: INVALID_PROJECT_ID");
163
164     IXtatuzProject.Status status =
    IXtatuzProject(projectAddress).projectStatus();
165     require(status == IXtatuzProject.Status.FINISH, "REFERRAL:
    PROJECT_NOT_FINISH");
166     buyerAgentDepositLeft[referral_][projectId_] = 0;
167     IERC20(tokenAddress).safeTransfer(msg.sender, amount);
168 }
```

Listing 36.3 The improved *increaseBuyerRef* function and the implementation of the *withdrawFundsLeft* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 37	Incorrect Time Verification On Extending Presale Period		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/XtatuzProject.sol*		
Locations	*XtatuzProject.sol L: 218 - 224, L: 273 - 289*		

Detailed Issue

We discovered the improper logic in the statement, which allows the operator to incorrectly extend the end presale date.

The root cause of this issue is the ***block.timestamp > (endPresale - 1 days)*** statement (L219 in code snippet 37.1 and L275 in code snippet 37.2). This statement allows access not only to the day preceding the end of the presale date but also to the day following it because the ***block.timestamp*** keeps increasing, which makes this state will always pass after the presale end date is reached.

As a result, if the *extendEndPresale* function is called by an operator after the end presale date, the calculation of the *endPresale* state variable will produce an incorrect result.

XtatuzProject.sol

```
218 function extendEndPresale() public onlyOperator {
219     require(block.timestamp > (endPresale - 1 days), "PROJECT: NOT_END_PREV");
220     require(_isTriggedEndpresale == false, "PROJECT: EXTENDED_PRESALE");
221     if (_isTriggedEndpresale == false) {
222         _extendEndPresale();
223     }
224 }
```

Listing 37.1 The *extendEndPresale* function

XtatuzProject.sol

```
273 function _extendEndPresale() internal {
274     require(_isTriggedEndpresale == false, "PROJECT: EXTENED_PRESALE");
275     if (block.timestamp > (endPresale - 1 days)) {
276         uint256 absoluteCount = count - _underwriteCount;
277         uint256 percent = ((count - countReserve) * 100) / absoluteCount;
278         if (percent >= 95) {
279             endPresale += 5 days;
280         } else if (percent >= 85 && percent < 95) {
281             endPresale += 10 days;
282         } else if (percent >= 65 && percent < 85) {
283             endPresale += 15 days;
284         } else {
285             endPresale += 30 days;
286         }
287         _isTriggedEndpresale = true;
288     }
289 }
```

Listing 37.2 The *_extendEndPresale* function

Recommendations

We recommend improving the *require* statement to ensure that the operator can only call the ***extendEndPresale*** function the day preceding the end presale date following the business requirement.

XtatuzProject.sol

```
218 function extendEndPresale() public onlyOperator {
219     _extendEndPresale();
220 }

    // (...SNIPPED...)

269 function _extendEndPresale() internal {
270     require(block.timestamp > (endPresale - 1 days) && block.timestamp <
    endPresale, "PROJECT: ONLY_THE_EXTENDING_PERIOD");
271     require(_isTriggedEndpresale == false, "PROJECT: EXTENDED_PRESALE");
272
273     uint256 absoluteCount = count - _underwriteCount;
274     uint256 percent = ((count - countReserve) * 100) / absoluteCount;
275     if (percent >= 95) {
276         endPresale += 5 days;
277     } else if (percent >= 85 && percent < 95) {
278         endPresale += 10 days;
279     } else if (percent >= 65 && percent < 85) {
280         endPresale += 15 days;
281     } else {
```

```
282             endPresale += 30 days;
283         }
284         _isTriggedEndpresale = true;
285     }
```

Listing 37.3 Improved *extendEndPresale and _extendEndPresale* functions

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 38	Potential Denial-Of-Service On Setting Reroll Data		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/XtatuzReroll.sol*		
Locations	*XtatuzReroll.sol L: 44 - 47*		

Detailed Issue

The *setRerollData* function of the *XtatuzReroll* contract allows the operator to set all the reroll data.

However, we noticed that in the *setRerollData* function (L44 - 49 in the code snippet below), there are assign all ***rerollData_*** arrays to the ***rerollData[projectId_]*** directly (L48 in the code snippet below). Behind the *ETHEREUM VIRTUAL MACHINE (EVM)*, the statement would iterate over all array elements to store.

This process can consume a lot of gas and the gas used is prone to exceeding the block gas limit if the operator needs to provide a large number of *rerollData_* (L44 in the code snippet below). In the case of exceeding the block gas limit, a transaction would be reverted, leading to the denial-of-service issue to the affected function.

XtatuzReroll.sol

```
44  function setRerollData(uint256 projectId_, string[] memory rerollData_) public
    onlyOperator {
45      require(rerollData_.length > 0, "REROLL: OUT_OF_DATA");
46      rerollData[projectId_] = rerollData_;
47  }
```

Listing 38.1 The *setRerollData* function that set all reroll data, which can consume more gas than the block gas limit

Recommendations

Since no recommended code or solution can fully fix this issue without breaking the contract's features, we recommend redesigning and reimplementing the operation to manage the *rerollData* state mechanism.

Reassessment

The *Xtatuz* team acknowledged this issue. However, the *Xtatuz* team would control the number of *rerollData_* array parameters to avoid the denial-of-service issue.

No. 39	Compiler Is Not Locked To Specific Version		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/Presaled.sol* *contract/Property.sol* *contract/PresaledFactory.sol* *contract/PropertyFactory.sol* *contract/ProjectFactory.sol* *contract/XtatuzFactory.sol* *contract/XtatuzProject.sol* *contract/XtatusReroll.sol* *contract/XtatuzReferral.sol* *contract/XtatuzRouter.sol*		
Locations	*Presaled.sol L: 2* *Property.sol L: 2* *PresaledFactory.sol L: 2* *PropertyFactory.sol L: 2* *ProjectFactory.sol L: 2* *XtatuzFactory.sol L: 2* *XtatuzProject.sol L: 2* *XtatusReroll.sol L: 2* *XtatuzReferral.sol L: 2* *XtatuzRouter.sol L: 2*		

Detailed Issue

We found that the smart contracts in this project should be deployed with the compiler version used in the development and testing process.

The compiler version that is not strictly locked via the *pragma* statement may make the contract incompatible against unforeseen circumstances.

List of smart contracts that should lock to the specific version.

- **Presaled.sol**
- **Property.sol**
- **PresaledFactory.sol**
- **PropertyFactory.sol**
- **ProjectFactory.sol**
- **XtatuzFactory.sol**
- **XtatuzProject.sol**
- **XtatuzReroll.sol**

- **XtatuzReferral.sol**
- **XtatuzRouter.sol**

An example code that is not locked to specific version (e.g., using => or ^ directive) is shown below

Presaled.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.17;
```

Listing 39.1 The *Presaled* contract

Recommendations

We recommend locking the pragma version like the example code snippet below.

```
pragma solidity 0.8.17;
// or
pragma solidity =0.8.17;

contract SemVerFloatingPragmaFixed {
}
```

Reference: *https://swcregistry.io/docs/SWC-103*

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 40	Lack Of Validating Of Reroll Data		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzProject.sol*		
Locations	*XtatuzRouter.sol L:* 166 - 189		

Detailed Issue

The *nftReroll* function of the *XtatuzRouter* contract (L166 - 189 in the code snippet below) allows the token owner to reroll their property token URI.

We discovered, if the *rerollData* is empty (L176 in the code snippet below), the ***block.timestamp % rerollData.length*** statement (L180 in the code snippet below) would be reverted due to the ***block.timestamp % 0***.

XtatuzRouter.sol

```
166 function nftReroll(uint256 projectId_, uint256 tokenId_) public {
167     require(_rerollAddress != address(0), "ROUTER: NO_REROLL_ADDRESS");
168
169     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
170     IProperty property = IProperty(propertyAddress);
171     IXtatuzReroll rerollContract = IXtatuzReroll(_rerollAddress);
172
173     address tokenAddress = rerollContract.tokenAddress();
174     string memory prevUri = property.tokenURI(tokenId_);
175     uint256 fee = rerollContract.rerollFee();
176     string[] memory rerollData = rerollContract.getRerollData(projectId_);
177     address tokenOwner = property.ownerOf(tokenId_);
178     require(tokenOwner == msg.sender, "ROUTER: NOT_NFT_OWNER");
179
180     uint256 newIndex = block.timestamp % rerollData.length;
181     property.setTokenURI(tokenId_, rerollData[newIndex]);
182     rerollData[newIndex] = prevUri;
183     rerollContract.setRerollData(projectId_, rerollData);
184
185     IERC20(tokenAddress).transferFrom(msg.sender, address(this), fee);
186     _totalRerollFee[projectId_] += fee;
187
188     emit NFTReroll(msg.sender, projectId_, tokenId_);
```

```
189 }
```

Listing 40.1 The *nftReroll* function of the XtatuzProject contract

Recommendations

We recommend **improving the validation by checking that the *rerollData* is not empty before continuing the reroll process** as shown at L177 in the code snippet below.

XtatuzRouter.sol

```
166 function nftReroll(uint256 projectId_, uint256 tokenId_) public {
167     require(_rerollAddress != address(0), "ROUTER: NO_REROLL_ADDRESS");
168 
169     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
170     IProperty property = IProperty(propertyAddress);
171     IXtatuzReroll rerollContract = IXtatuzReroll(_rerollAddress);
172 
173     address tokenAddress = rerollContract.tokenAddress();
174     string memory prevUri = property.tokenURI(tokenId_);
175     uint256 fee = rerollContract.rerollFee();
176     string[] memory rerollData = rerollContract.getRerollData(projectId_);
177     require(rerollData.length > 0, "ROUTER: NO_REROLL_DATA");
178     address tokenOwner = property.ownerOf(tokenId_);
179     require(tokenOwner == msg.sender, "ROUTER: NOT_NFT_OWNER");
180 
181     uint256 newIndex = block.timestamp % rerollData.length;
182     property.setTokenURI(tokenId_, rerollData[newIndex]);
183     rerollData[newIndex] = prevUri;
184     rerollContract.setRerollData(projectId_, rerollData);
185 
186     IERC20(tokenAddress).transferFrom(msg.sender, address(this), fee);
187     _totalRerollFee[projectId_] += fee;
188 
189     emit NFTReroll(msg.sender, projectId_, tokenId_);
190 }
```

Listing 40.2 The improved *nftReroll* function of the *XtatuzProject* contract

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 41	Missing Validation Of Zero Address		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzRouter.sol*		
Locations	*XtatuzRouter.sol L: 43 - 50*		

Detailed Issue

We found that there is no validation process to prevent the assignment of address zero within the constructor of the *XtatuzRouter* contract. The usage of address zero could have unforeseen implications, which presents a potential security issue.

XtatuzRouter.sol

```
43  constructor(
44      address spv_,
45      address factoryAddress_
46  ) {
47      _transferSpv(spv_);
48      _xtatuzFactory = IXtatuzFactory(factoryAddress_);
49      _projectIdCounter.increment();
50  }
```

Listing 41.1 The *constructor* of the *XtatuzRouter* contract

Recommendations

We recommend adding the ***require*** statement to avoid assigning the zero address as shown in the code snippet below.

XtatuzRouter.sol

```
43  constructor(
44      address spv_,
45      address factoryAddress_
46  ) {
47      require(spv_ != address(0) && factoryAddress_ != address(0), "ROUTER: INVALID
    ADDRESS");
```

```
48      _transferSpv(spv_);
49      _xtatuzFactory = IXtatuzFactory(factoryAddress_);
50      _projectIdCounter.increment();
51  }
```

Listing 41.2 The improved *constructor* of the *XtatuzRouter* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team fixed this issue by adopting our recommended code.

No. 42	Predictable Randomness		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contract/XtatuzRouter.sol*		
Locations	*XtatuzRouter.sol L: 166 - 189*		

Detailed Issue

We found that the ***nftReroll*** function contains weak randomness code (L180 in code snippet 42.1), which is generated using a predictable pseudo-random number generator ***newIndex*** value that could potentially be manipulated by an attacker, compromising the fairness of the contract.

To elaborate, the attacker can create a malicious contract using the same logic that checks the outcome of the ***newIndex*** that manipulates from ***block.timestamp*** and ***rerollData.length*** variables, which allows the attacker to predict the ***newIndex*** result accurately.

We report this information for ***acknowledgement*** only since the *Property NFT* without rarity, the randomness that generates the ***newIndex*** is not harmful to the system.

XtatuzProject.sol

```
166 function nftReroll(uint256 projectId_, uint256 tokenId_) public {
167     require(_rerollAddress != address(0), "ROUTER: NO_REROLL_ADDRESS");
168
169     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
170     IProperty property = IProperty(propertyAddress);
171     IXtatuzReroll rerollContract = IXtatuzReroll(_rerollAddress);
172
173     address tokenAddress = rerollContract.tokenAddress();
174     string memory prevUri = property.tokenURI(tokenId_);
175     uint256 fee = rerollContract.rerollFee();
176     string[] memory rerollData = rerollContract.getRerollData(projectId_);
177     address tokenOwner = property.ownerOf(tokenId_);
178     require(tokenOwner == msg.sender, "ROUTER: NOT_NFT_OWNER");
179
180     uint256 newIndex = block.timestamp % rerollData.length;
181     property.setTokenURI(tokenId_, rerollData[newIndex]);
182     rerollData[newIndex] = prevUri;
183     rerollContract.setRerollData(projectId_, rerollData);
```

```
184
185     IERC20(tokenAddress).transferFrom(msg.sender, address(this), fee);
186     _totalRerollFee[projectId_] += fee;
187
188     emit NFTReroll(msg.sender, projectId_, tokenId_);
189 }
```

Listing 42.1 The *nftReroll* function

Recommendations

Since the *Property NFT* without rarity, the randomness that generates the ***newIndex*** is not harmful to the system.

We report this information for acknowledgement only, as we care about the fairness of the contract.

Reassessment

This issue was acknowledged by the *Xtatuz* team.

No. 43	Recommended Improving Event Emissions		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzRouter.sol*		
Locations	*XtatuzRouter.sol L: 58, 264*		

Detailed Issue

We noticed that the *setPropertyStatus* function could be **improved by emitting the previous status for improving traceability.**

XtatuzRouter.sol

```
58  event ChangePropertyStatus(uint256 indexed projectId, IProperty.PropertyStatus
    status);

    // (...SNIPPED...)

261 function setPropertyStatus(uint256 projectId_, IProperty.PropertyStatus status)
    public onlySpv {
262     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
263     IProperty(propertyAddress).setPropertyStatus(status);
264     emit ChangePropertyStatus(projectId_, status);
265 }
```

Listing 43.1 The *ChangePropertyStatus* event that could improve

Recommendations

We recommend **emitting the *previous status*** on the *setPropertyStatus* function to improve transparency and traceability.

XtatuzRouter.sol

```
58  event ChangePropertyStatus(uint256 indexed projectId, IProperty.PropertyStatus
    prevStatus, IProperty.PropertyStatus newStatus);

    // (...SNIPPED...)

261 function setPropertyStatus(uint256 projectId_, IProperty.PropertyStatus status)
    public onlySpv {
262     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
263     IProperty.PropertyStatus prevStatus =
264 IProperty(propertyAddress).propertyStatus();
        IProperty(propertyAddress).setPropertyStatus(status);
265     emit ChangePropertyStatus(projectId_, prevStatus, newStatus);
266 }
```

Listing 43.2 The improved *ChangePropertyStatus* event

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 44	Recommended Improving Transparency And Traceability Of Crucial Variables		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Partially Fixed
Associated Files	*contracts/Presaled.sol* *contracts/Property.sol* *contracts/XtatuzFactory.sol* *contracts/XtatuzProject.sol* *contracts/XtatuzReroll.sol* *contracts/XtatuzReferral.sol* *contracts/XtatuzRouter.sol*		
Locations	*Presaled.sol: L: 12, 13, and 14* *Property.sol: L: 27, 28, and 29* *XtatuzFactory.sol: L: 22, 23, and 24* *XtatuzProject.sol: L: 14, 15, 16, 19, 20, 25, 34, 39, and 40* *XtatuzReroll.sol: L: 16, and 17* *XtatuzReferral.sol: L: 25* *XtatuzRouter.sol: L: 20, 21, 22, 23, 24, 38, and 41*		

Detailed Issue

We found certain variables from various contracts are declared as ***private*** that necessitate enhanced transparency and traceability by changing their declaration to ***public***. The mentioned variables are listed below:

- The ***Presaled*** contract
  - The *_operator* address variable.
  - The *_routerAddress* address variable.
  - The *baseURI* string variable.

- The ***Property*** contract
  - The *_operator* address variable.
  - The *_routerAddress* address variable.
  - The *baseURI* string variable.

- The ***XtatuzFactory*** contract
    - The *_propertyFactory* address variable.
    - The *_presaledFactory* address variable.
    - The *_projectFactory* address variable.

- The ***XtatuzProject*** contract
    - The *_operatorAddress* address variable.
    - The *_trusteeAddress* address variable.
    - The *_projectOwner* address variable.
    - The *_propertyAddress* address variable.
    - The *_presaledAddress* address variable.
    - The *_underwriteCount* unsigned-integer variable.
    - The *_isTriggedEndpresale* boolean variable.
    - The *_multiSigMint* mapping.
    - The *_multiSigBurn* mapping.

- The ***XtatuzReroll*** contract
    - The *_operator* address variable.
    - The *_routerAddress* address variable.

- The ***XtatuzReferral*** contract
    - The *_operatorAddress* address variable.

- The ***XtatuzRouter*** contract
    - The *_spvAddress* address variable.
    - The *_xtatuzFactoryAddress* address variable.
    - The *_membershipAddress* address variable.
    - The *_rerollAddress* address variable.
    - The *_referralAddress* address variable.
    - The *_memberdProject* mapping.
    - The *_isNotice* mapping.

Recommendations

We recommend changing the declaration of the associated variables to ***public***, in order to enhance transparency and traceability.

Reassessment

The *Xtatuz* team partially fixed this issue by enhancing the transparency and traceability of certain associated variables. However, some variables have been kept private.

No. 45	Recommended Enforcing Checks-Effects-Interactions Pattern		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/Presaled.sol* *contract/XtatuzRouter.sol*		
Locations	*Presaled.sol L: 43 - 55* *Router.sol L: 108 - 138, 140 - 149, 166 - 189 and 191 - 201*		

Detailed Issue

We noticed that the functions below ***do not follow the checks-effects-interactions pattern***, which is the best practice coding style to prevent potential reentrancy attacks.

List of functions that do not follow the checks-effects-interactions pattern.

- The ***mint*** function in the *Presaled* contract
- The ***addProjectMember*** function in the *XtatuzRouter* contract
- The ***claim*** function in the *XtatuzRouter* contract
- The ***nftReroll*** function in the *XtatuzRouter* contract
- The ***claimRerollFee*** function in the *XtatuzRouter* contract

Even if there are no reentrancy issues, we recommend that the list of functions above should enforce the *checks-effects-interactions* pattern.

For example, in the code snippet 45.1 below, the *mint* function transfers the presale token (*interactions part*) in L49 before updating the state variables (*effects part*) in L50 - 51.

Presaled.sol

```
43  function mint(address to, uint256[] memory tokenIdList_) public onlyOperator {
44      require(to != address(0), "Presaled: Reciever address is address 0");
45      uint256 amount = tokenIdList_.length;
46      for (uint256 index = 0; index < amount; index++) {
47          uint256 tokenId = tokenIdList_[index];
48          require(_exists(tokenId) == false, "Presaled: TokenId already exits");
49          _safeMint(to, tokenId);
50          _mintedTimestamp[tokenId] = block.timestamp;
51          _tokenIdCount.increment();
```

```
52      }
53      setApprovalForAll(_operator, true);
54      emit Minted(to, tokenIdList_, tokenIdList_.length);
55  }
```

Listing 45.1 The *mint* function

XtatuzRouter.sol

```
108 function addProjectMember(
109     uint256 projectId_,
110     uint256[] memory nftList_,
111     string memory referral_
112 ) public {
113     uint256 amount = nftList_.length;
114     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
115     IXtatuzProject project = IXtatuzProject(projectAddress);
116     uint256 price = project.addProjectMember(msg.sender, nftList_);
117 
118     uint256 minPrice = project.minPrice();
119     IXtatuzReferral referralContract = IXtatuzReferral(_referralAddress);
120     referralContract.increaseBuyerRef(projectId_, referral_, amount * minPrice);
121 
122     address tokenAddress = project.tokenAddress();
123     IERC20(tokenAddress).transferFrom(msg.sender, projectAddress, price);
124 
125     uint256[] memory memberedProject = _memberdProject[msg.sender];
126     bool foundedIndex;
127     for (uint256 index = 0; index < memberedProject.length; index++) {
128         if (memberedProject[index] == projectId_) {
129             foundedIndex = true;
130         }
131     }
132     if (!foundedIndex) {
133         _memberdProject[msg.sender].push(projectId_);
134     }
135 
136     _isMemberClaimed[msg.sender][projectId_] = false;
137     emit AddProjectMember(projectId_, msg.sender, referral_, price);
138 }
```

Listing 45.2 The *addProjectMember* function

XtatuzRouter.sol

```
140 function claim(uint256 projectId_) public {
141     require(_isMemberClaimed[msg.sender][projectId_] == false, "ROUTER:
    ALREADY_CLAIMED");
142 
```

```
143     address projectAddress = _xtatuzFactory.getProjectAddress(projectId_);
144     IXtatuzProject(projectAddress).claim(msg.sender);
145
146     _isMemberClaimed[msg.sender][projectId_] = true;
147
148     emit Claimed(projectId_, msg.sender);
149 }
```

Listing 45.3 The *claim* function

XtatuzRouter.sol

```
166 function nftReroll(uint256 projectId_, uint256 tokenId_) public {
167     require(_rerollAddress != address(0), "ROUTER: NO_REROLL_ADDRESS");
168
169     address propertyAddress = _xtatuzFactory.getPropertyAddress(projectId_);
170     IProperty property = IProperty(propertyAddress);
171     IXtatuzReroll rerollContract = IXtatuzReroll(_rerollAddress);
172
173     address tokenAddress = rerollContract.tokenAddress();
174     string memory prevUri = property.tokenURI(tokenId_);
175     uint256 fee = rerollContract.rerollFee();
176     string[] memory rerollData = rerollContract.getRerollData(projectId_);
177     address tokenOwner = property.ownerOf(tokenId_);
178     require(tokenOwner == msg.sender, "ROUTER: NOT_NFT_OWNER");
179
180     uint256 newIndex = block.timestamp % rerollData.length;
181     property.setTokenURI(tokenId_, rerollData[newIndex]);
182     rerollData[newIndex] = prevUri;
183     rerollContract.setRerollData(projectId_, rerollData);
184
185     IERC20(tokenAddress).transferFrom(msg.sender, address(this), fee);
186     _totalRerollFee[projectId_] += fee;
187
188     emit NFTReroll(msg.sender, projectId_, tokenId_);
189 }
```

Listing 45.4 The *nftReroll* function

XtatuzRouter.sol

```
191 function claimRerollFee(uint256 projectId_) public onlySpv {
192     require(_totalRerollFee[projectId_] > 0, "ROUTER: OUT_OF_FEE");
193
194     IXtatuzReroll rerollContract = IXtatuzReroll(_rerollAddress);
195     address tokenAddress = rerollContract.tokenAddress();
196
197     uint256 totalFee = _totalRerollFee[projectId_];
198     IERC20(tokenAddress).transfer(msg.sender, totalFee);
```

```
199         _totalRerollFee[projectId_] = 0;
200         emit ClaimedRerollFee(msg.sender, projectId_, totalFee);
201     }
```

Listing 45.5 The *claimRerollFee* contract

Recommendations

We recommend enforcing the checks-effects-interactions pattern to all of the functions below.

- The ***mint*** function in the *Presaled* contract
- The ***addProjectMember*** function in the *XtatuzRouter* contract
- The ***claim*** function in the *XtatuzRouter* contract
- The ***nftReroll*** function in the *XtatuzRouter* contract
- The ***claimRerollFee*** function in the *XtatuzRouter* contract

The example below is how to fix this issue, we moved the *interactions* part (the _safeMint function L51 in code snippet below) to get executed after the *effects* part (L49 - 50 in code snippet below).

Presaled.sol

```
43  function mint(address to, uint256[] memory tokenIdList_) public onlyOperator {
44      require(to != address(0), "Presaled: Reciever address is address 0");
45      uint256 amount = tokenIdList_.length;
46      for (uint256 index = 0; index < amount; index++) {
47          uint256 tokenId = tokenIdList_[index];
48          require(_exists(tokenId) == false, "Presaled: TokenId already exits");
49          _mintedTimestamp[tokenId] = block.timestamp;
50          _tokenIdCount.increment();
51          _safeMint(to, tokenId);
52      }
53      setApprovalForAll(_operator, true);
54      emit Minted(to, tokenIdList_, tokenIdList_.length);
55  }
```

Listing 45.6 The improved *mint* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 46	Recommended Event Emissions For Transparency And Traceability		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/Presaled.sol* *contracts/PresaledFactory.sol* *contracts/Property.sol* *contracts/PropertyFactory.sol* *contracts/ProjectFactory.sol* *contracts/XtatuzFactory.sol* *contracts/XtatuzReroll.sol* *contracts/XtatuzReferral.sol* *contracts/XtatuzProject.sol* *contracts/XtatuzRouter.sol*		
Locations	*Several functions throughout multiple contracts*		

Detailed Issue

We consider operations of the following state-changing functions important and require proper event emissions for improving transparency and traceability:

- The ***Presaled*** contract
  - The *setBaseURI* function
  - The *_setOperator* function
- The ***PresaledFactory*** contract
  - The *createPresale* function
- The ***Property*** contract
  - The *setPropertyStatus* function
  - The *_setOperator* function
  - The *setBaseURI* function
  - The *setTokenURI* function
- The ***PropertyFactory*** contract
  - The *createProperty* function

- The ***ProjectFactory*** contract
  - The *createProject* function

- The ***XtatuzFactory*** contract
  - The *createProjectContract* function

- The ***XtatuzReroll*** contract
  - The *setRerollData* function
  - The *setFee* function

- The ***XtatuzReferral*** contract
  - The *updateReferralAmount* function
  - The *claim* function

- The ***XtatuzProject*** contract
  - The *setPresalePeriod* function
  - The *setUnderwriteCount* function
  - The *claim* function
  - The *refund* function
  - The *extendEndPresale* function
  - The *ownerClaimLeft* function
  - The *multiSigMint* function
  - The *multiSigBurn* function
  - The *_initialData* function

- The ***XtatuzRouter*** contract
  - The *setRerollAddress* function
  - The *setMembershipAddress* function
  - The *setReferrallAddress* function

Recommendations

We recommend **emitting relevant events** on the associated functions to improve transparency and traceability.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 47	Recommended Removing Unused Code		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contract/Presaled.sol* *contract/Property.sol* *contract/PresaledFactory.sol* *contract/PropertyFactory.sol* *contract/XtatuzFactory.sol* *contract/XtatuzReroll.sol* *contract/XtatuzReferral.sol* *contract/XtatuzRouter.sol*		
Locations	*Presaled.sol L: 10* *Property.sol L: 8, 9* *PresaledFactory.sol L: 5* *PropertyFactory.sol L: 5* *XtatuzFactory.sol L: 4* *XtatuzReroll L: 4, 6, 8, 11, 15* *XtatuzReferral.sol L: 5, 6, 8, 10, 11 and 12* *XtatuzRouter.sol L38, 57, 98*		

Detailed Issue

We found that there were sections of unused code present in the smart contracts. This unused code could potentially cause confusion or misunderstandings among users or developers when attempting to maintain or modify the source code. Unused code can also increase the complexity of the codebase and lead to unnecessary computational overhead.

List of unused codes.

- Unused Imported Interface
  - ***IXtatuzProject***, ***IPresaled*** interfaces (*L8 - 9 in Property.sol*)
  - ***IProperty*** interface (*L5 in PresaledFactory.sol*)
  - ***IProperty*** interface (*L5 in PropertyFactory.sol*)
  - ***IERC20***, ***IProperty***, ***IXtatuzRouter*** interfaces (*L4, 6, 8 in XtatuzReroll.sol*)
  - ***IERC721***, ***IXtatuzFactory***, ***IPresaled***, ***IProperty***, ***IXtatuzReroll*** interfaces (*L5, 8, 10, 11, 12 in XtatuzReferral.sol*)

- Unused Imported Source/Library
  - ***XtatuzProject*** contract (*L4 in XtatuzFactory.sol*)
  - ***Counters*** library (*L6 in XtatuzReferral.sol*)

- Unused Event
  - ***Buyback*** event (*L57 in XtatuzRouter.sol*)

- Unused State
  - ***_tokenIdCount*** state variable (L10 in Presaled.sol)
  - ***_xtatusFactory***, ***_totalFee*** state variables (L11, 15 in XtatuzReroll.sol)

- Unused Modifier and Function
  - ***endedPresale*** modifier (*L85 - 88 in XtatuzProject.sol*)
  - ***_checkEndedPresale*** function (*L362 - 364 in XtatuzProject.sol*)

Example of unused imported interfaces in the *Property* contract (L8 - 9 in code snippet below).

Property.sol

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.17;
3
4   import "@openzeppelin/contracts/access/Ownable.sol";
5   import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
6   import "@openzeppelin/contracts/utils/Strings.sol";
7   import "../interfaces/IProperty.sol";
8   import "../interfaces/IXtatuzProject.sol";
9   import "../interfaces/IPresaled.sol";
10
11  contract Property is ERC721Enumerable, Ownable {
    // (...SNIPPED...)
145 }
```

Listing 47.1 The *Property* contract

Recommendations

We recommend removing unused code from the smart contracts as it can reduce the contract's complexity and also help to reduce confusion among users or developers when maintaining the source code.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 48	Inconsistent Error Message With The Code		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XtatuzProject.sol* *contracts/Property.sol*		
Locations	*XtatuzProject.sol L: 359* *Property.sol L:71*		

Detailed Issue

We found an error message inconsistent with the code in the functions *_checkIsAvailable* (L359 in code snippet 48.1) and *mintFragment* (L71 in code snippet 48.2). These inconsistencies can lead to misunderstanding among users or developers when maintaining the source code.

XtatuzProject.sol

```
357 function _checkIsAvailable() internal view {
358     uint256 timestamp = block.timestamp;
359     require(timestamp >= startPresale && timestamp <= endPresale, "PROJECT:
    ENDED_PRESALE");
360 }
```

Listing 48.1 The *_checkIsAvailable* function with an inconsistent error message

Property.sol

```
65  function mintFragment(address to, uint256[] memory tokenIdList) public
    onlyOperator {
66      require(ownerOf(0) == address(this), "PROPERTY: NO_MASTER_NFT");
67      require(isMintedMaster == true, "PROPERTY: MASTER_NOT_MINTED");
68      uint256 amount = tokenIdList.length;
69      for (uint256 index = 0; index < amount; index++) {
70          uint256 tokenId = tokenIdList[index];
71          require(_exists(tokenId) == false, "PROJECT: ALREADY_EXITS");
72          _safeMint(to, tokenId);
73      }
74      _setApprovalForAll(to, _routerAddress, true);
75      emit MintFragment(to, tokenIdList);
76  }
```

Listing 48.2 The *mintFragment* function with an inconsistent error message

Recommendations

We recommend revising the associated error message to reflect the actual code.

XtatuzProject.sol

```
357 function _checkIsAvailable() internal view {
358     uint256 timestamp = block.timestamp;
359     require(timestamp >= startPresale && timestamp <= endPresale, "PROJECT:
    UNAVAILABLE");
360 }
```

Listing 48.3 The improved *_checkIsAvailable* function

Property.sol

```
65  function mintFragment(address to, uint256[] memory tokenIdList) public
    onlyOperator {
66      require(ownerOf(0) == address(this), "PROPERTY: NO_MASTER_NFT");
67      require(isMintedMaster == true, "PROPERTY: MASTER_NOT_MINTED");
68      uint256 amount = tokenIdList.length;
69      for (uint256 index = 0; index < amount; index++) {
70          uint256 tokenId = tokenIdList[index];
71          require(_exists(tokenId) == false, "PROPERTY: TOKEN_ALREADY_EXITS");
72          _safeMint(to, tokenId);
73      }
74      _setApprovalForAll(to, _routerAddress, true);
75      emit MintFragment(to, tokenIdList);
76  }
```

Listing 48.4 The improved *mintFragment* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team adopted our recommended code to fix this issue.

No. 49	Misspelling State Variable		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/XatuzRouter.sol*		
Locations	*XatuzRouter.sol L: 38*		

Detailed Issue

In the code snippet below, we found the misspelling crucial state variable at line 38. This misspelling can lead to misunderstanding among users or developers when maintaining the source code.

XatuzRouter.sol

```
14  contract XtatuzRouter {
        // (...SNIPPED...)
38      mapping(address => uint256[]) private _memberdProject;
```

Listing 49.1 The misspell state variable

Recommendations

We recommend revising the misspelingl state and associated function.

XatuzRouter.sol

```
14  contract XtatuzRouter {
        // (...SNIPPED...)
38      mapping(address => uint256[]) private _memberedProject;
```

Listing 49.2 The improved state variable

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *Xtatuz* team fixed this issue as per our suggestion.

No. 50	Inconsistent Interface With The Implementation		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*interfaces/IPresaled.sol* *interfaces/IProperty.sol* *interfaces/IXtatuzRouter.sol*		
Locations	*IPresaled.sol L: 15* *IProperty.sol L: 36* *IXtatuzRouter.sol L: 47 and 51*		

Detailed Issue

We noticed the significant inconsistency between the functions listed in the interfaces of the following contracts and their contract implementation. This could result in compatibility issues with other contracts.

- The ***IPresaled*** interface
  - The *getPresaledPackage* function
- The ***IProperty*** interface
  - The *rerollData* function
- The ***IXtatuzRouter*** interface
  - The *refferalAmount* function.
  - The *getMembershipAddress* function

Recommendations

We recommend reviewing and updating the associated interfaces to accurately reflect the implemented functions.

Reassessment

The *Xtatuz* team fixed this issue as per our suggestion.

Appendix

About Us

Founded in 2020, Valix Consulting is a blockchain and smart contract security firm offering a wide range of cybersecurity consulting services such as blockchain and smart contract security consulting, smart contract security review, and smart contract security audit.

Our team members are passionate cybersecurity professionals and researchers in the areas of private and public blockchain technology, smart contract, and decentralized application (DApp).

We provide a service for assessing and certifying the security of smart contracts. Our service also includes recommendations on smart contracts' security and gas optimization to bring the most benefit to users and platform creators.

Contact Information


info@valix.io


https://www.facebook.com/ValixConsulting


https://twitter.com/ValixConsulting


https://medium.com/valixconsulting

References

Title	Link
OWASP Risk Rating Methodology	https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
Smart Contract Weakness Classification and Test Cases	https://swcregistry.io/