NFT Game

Smart Contract Audit Report
Prepared for Gold Fever

Date Issued:	Feb 22, 2023
Project ID:	AUDIT2022055
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022055
Version	v1.0
Client	Gold Fever
Project	NFT Game
Auditor(s)	Peeraphut Punsuwan Sorawish Laovakul Phitchakorn Apiratisakul Kongkit Chatchawanhirun
Author(s)	Peeraphut Punsuwan Phitchakorn Apiratisakul Sorawish Laovakul
Reviewer	Patipon Suwanbol
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Feb 22, 2023	Full report	Peeraphut Punsuwan Phitchakorn Apiratisakul Sorawish Laovakul

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Gold Fever, Inspex team conducted an audit to verify the security posture of the NFT Game smart contracts between Nov 23, 2022 and Dec 7, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of NFT Game smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 7 critical, 4 high, 4 medium, 5 low, 3 very low, and 2 info-severity issues. With the project team's prompt response, 7 critical, 4 high, 4 medium, 3 low, 3 very low, and 2 info-severity issues were resolved or mitigated in the reassessment, while 2 low-severity issues were acknowledged by the team. Therefore, Inspex trusts that NFT Game smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Gold Fever is a play-to-earn game that is working with blockchain technology allowing the acquisition and trading of NFTs, in-game. The players can create unique characters, craft tools for survival, and compete with other players for dominance in the search for gold.

The game mechanism is the smart contract working with the game server. Moreover, the project is using the gasless mechanism that allows players to interact with smart contracts without users needing native token for transaction fees.

Scope Information:

Project Name	NFT Game
Website	https://goldfever.io/
Smart Contract Type	Ethereum Smart Contract
Chain	Polygon
Programming Language	Solidity
Category	Game, NFT

Audit Information:

Audit Method	Whitebox
Audit Date	Nov 23, 2022 - Dec 7, 2022
Reassessment Date	Feb 15, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The smart contracts with the following bytecodes were audited and reassessed by Inspex in detail:

Initial Audit:

Contract	Bytecode SHA256 Hash
GoldFeverAuction	f7f53bb2405252ba30f809d1036e083728d715fe5bf1feeb1cf0a90ea22afecd
GoldFeverBuildingManager	fcaa7e885e79118c454ea2e3021134d73076f162c95f64830ed377d7f0d177c0
GoldFeverCharacter	9fb5878be26f3adc3da3c0e8d4ae4f1f2001d2b44d0ba95801b690b7f0752011
GoldFeverCollateral	8c685111e81cd4adb7d94323192449c33012aca27f8ed4665c32550b62778783
GoldFeverForwarder	1a66fd78aebc96b0822e0b180e5f1bcf60b3f9abda7afd189fd3dab6b7276ac8
GoldFeverGovernor	4c4c25a951532eda692d835a311bcdc200ac7751de728b175503a67f9e19d35c
GoldFeverGuild	72002459795f22e4a704663399f75c8f9045800195cd42a47557d48fac4f863b
GoldFeverGuildRight	3723d088394af71790052f6413dc47eb2c86a88e6169a8026b554add26a3050c
GoldFeverItem	dd38d34d4148df9e2f2a834a3e00f8b342bde5cbd115153b7e21679760debecd
GoldFeverItemTier	8d9518a3062d708f87db0b3df50310bf8e4fb7e8ba7ebfc94931f872857030d5
GoldFeverItemType	812ee0576e206c0338d2c9046d3daa9670d5cb5b841a47529339847efce0d581
GoldFeverLeasing	32124e39849795c32e61ad9580e93ab44b4cf22766e1ded7eba68dcb9a89645f
GoldFeverMarket	868d21944a5d33443d4cc80419590ae2ea3b117078ecaff487e670c6c84e1e7a
GoldFeverMask	45ac5cf7e8b0b548d1a542f6f16757825c3fa2b78ad8482a764f1d502bbe4f4a
GoldFeverMaskBox	44bd13ad8a955ac8963e65226f04386150f1b0ac2e606aed22b1cf2377c966fd
GoldFeverMerchantRight	8734000d678e55a0c4a4ef4ac3bb290e23c08d87ace2519a0d75ea435729ef4e
GoldFeverMiningClaim	b0a0aba49238e790446be81662132a8c9d9ac86becf64993e15bd2b3095d7fe8
GoldFeverNativeGold	858041bedbad2a602d09afd45f3a2f6838e2823869e8a97d5d7e87938a74345d
GoldFeverNpc	ea4e4fdcffe3c07f3be720780b32eee05c0d6721ff26d8bf00b680979960bcc1
GoldFeverPaymaster	8a514309de223c274206b1e992e48dec90958d4575a43438952febe355ab8bb5
GoldFeverRight	4facb721158f71734b8155f8427705d4c46d944f20736f09980f5cc3e6fc8f59

GoldFeverSwap	688f08280460fced71bd067f9fdee4c5b048c350c569bc6ab241d09b8e6a5795
GoldFeverTaxation	94a4f35b5192a619718ad9bd0493686358602fcaf752fab28c1ba61787cbb3cb
AccessControlMixin	f518d75ab5cbc7cf99195746855b649f2c3214ea318f91aebf6fd938d05339f2
EIP712Base	2a8fef7a02f500a5fcabc73c31c93428370a97b63e39f17dc84def1325b40605
IChildToken	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
Initializable	598e69a65223678621a3917a64d7ac33e647b021877e0068c37479058b7cad33
NativeMetaTransaction	a34d3e7008de6966ba5e23d03ad2791585d897e69ee8b805a828d01969e67128

Reassessment:

Contract	Bytecode SHA256 Hash
GoldFeverAuction	b9ff5d7a3b894d46c0d8bd4cbbdb48cf5093afc520d6c954dc47a41cc5b255a1
GoldFeverBuildingManager	fd581b3eec2a147a02d398898ca15daa830d46822d46304db7d25e2373556ec4
GoldFeverCharacter	743863f9e5fbdc3ea628d6820f496bb59b0f3c24f8ab15791eae0568c349b384
GoldFeverCollateral	130ed01d10e6486a2b6d2267d2e54ec934a846482dfe79bbff784c1aa3029f44
GoldFeverForwarder	385a351a4635a9550d317bae7524fe8e49aec1d35789ee5df1aa9e209dc5b896
GoldFeverGovernor	5ae6d497722d609628c8b380cf4701d07b09affef53f94a477f54d7fe51c6b13
GoldFeverGuild	1a0515274b4986d528a720130bf65659771c56ff455b4f921af0bc5510ed8fbb
GoldFeverGuildRight	7f95eaf52f37a8e2f2476375b6bad223f938b0e8cc39ac9384f5c29cecc29da2
GoldFeverItem	0225eaf9c975679710b1d0786326eba41e67f5975cafdd86ff88db8c6e1b41b2
GoldFeverItemTier	77dd40a485cefe95b0c01165e3ffe18bea3436bec24e3e10fb1414b478836ae7
GoldFeverItemType	de283230d3203016c5a751e6ff30ae2215fa781c6e58075e62aa0b8a8c9ddbf1
GoldFeverLeasing	0e6f4205f66437453d77b13e47120c1eac83507d5d537bfc7c1d54993f6490af
GoldFeverMarket	233304a439c59ce72fb4b61cb5a25c139ecbdfa836efb20e0ea1ebd2e740b299
GoldFeverMask	623a52a375ce415eb83d0e56645815661d65b338d2bad9d0cfcc2187960a4d1c
GoldFeverMaskBox	9237bb00b5b0203c4c4d7cae5d0f4421a30c9f6e642a0ccf89774b2a343d97fb
GoldFeverMerchantRig	8b5826420e69777d0d43948e02859400e769848042170d5fc900e1dc39f73560

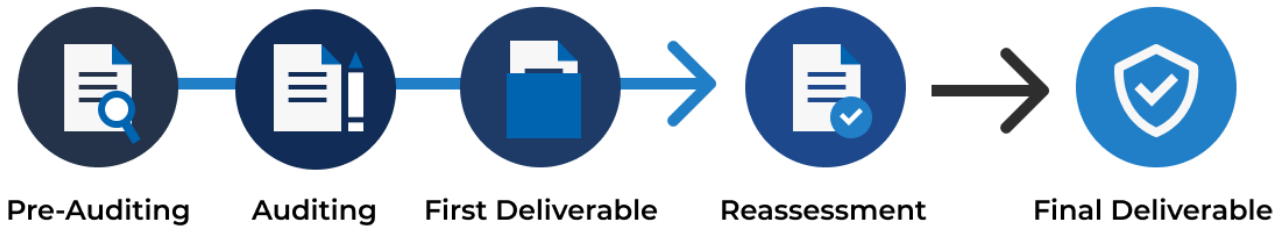
ht	
GoldFeverMiningClaim	c3dfc54c8d93ac530b7450e517519b4ee3495c00c6f11d8defb98d9efd18e14f
GoldFeverNativeGold	9f15c42c0a30ad4cc025ba78d6abbf3a9675c3648b2510213a2784f2d535e05a
GoldFeverNpc	8cbb0168f89df3c0d99d58a63872155aa15dbcffd2f2c1a817d6ae40289934f4
GoldFeverPaymaster	84cb030c67cc1e48ab7b819a3ca4d3312275a45507bbd8a37c28b458471017f4
GoldFeverRight	b6ff951de6c43df5fe7de90ca6165c420840ec01be951a5207f894ec9f565b68
GoldFeverSwap	fcaf9c00369ecda12e0e8d25ee4ef58e7068f5b6c12504b68b457594f5775ef2
GoldFeverTaxation	8f5b790031e091e7e53273bd2ab9a389f22e83315af265cfa5086aa802b8a031
AccessControlMixin	390cb53999ac23a6e75acac4c859262ae7876ae902b7068d0a68d7a6a30096ca
EIP712Base	dda9f9b43e28defdb55e294eb00765ef741b247df8d6dca8ebf965184da71c66
IChildToken	e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
Initializable	625b1a0970d4b8e6f21db6029b329ea426c71c6cf368d3c85db48a5bbfce170e
NativeMetaTransaction	15101bea416e784aef4ead1f04e915d467460897fb8cd9f1de90af378af3f9be

As the Gold Fever team has decided not to publish the source code until the full version is completed, the users should compare the bytecode hashes compiled by Solidity compiler version 0.8.17 with the smart contracts deployed before interacting with them to make sure that they are the same as the contracts audited.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

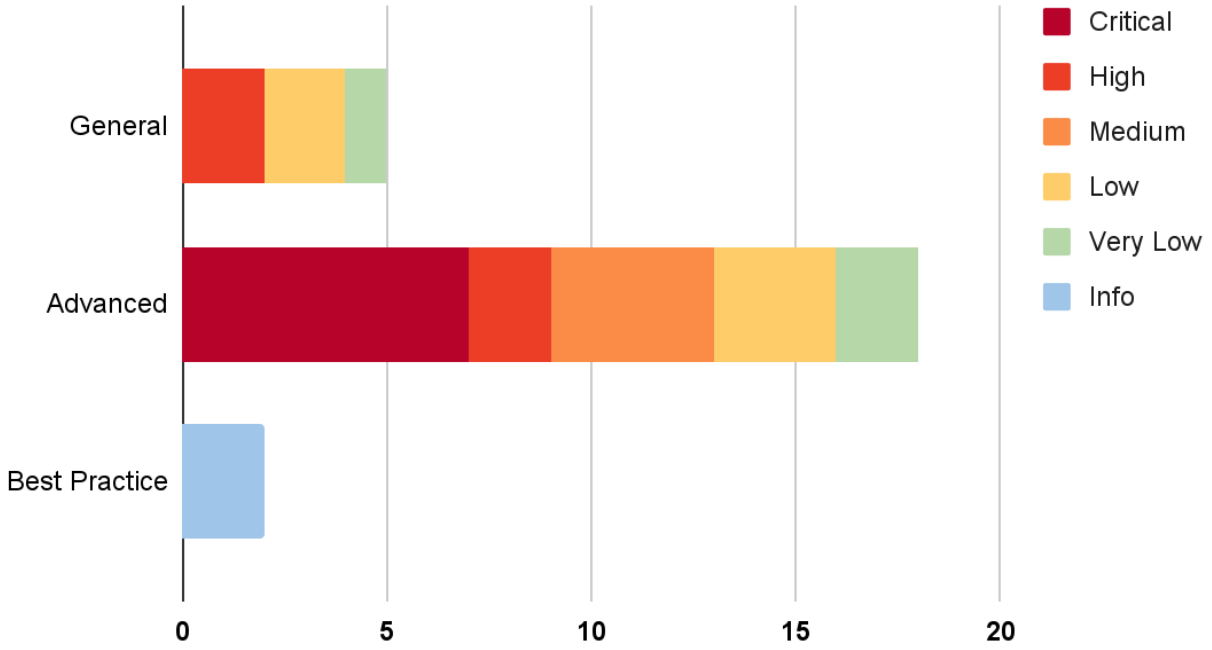
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

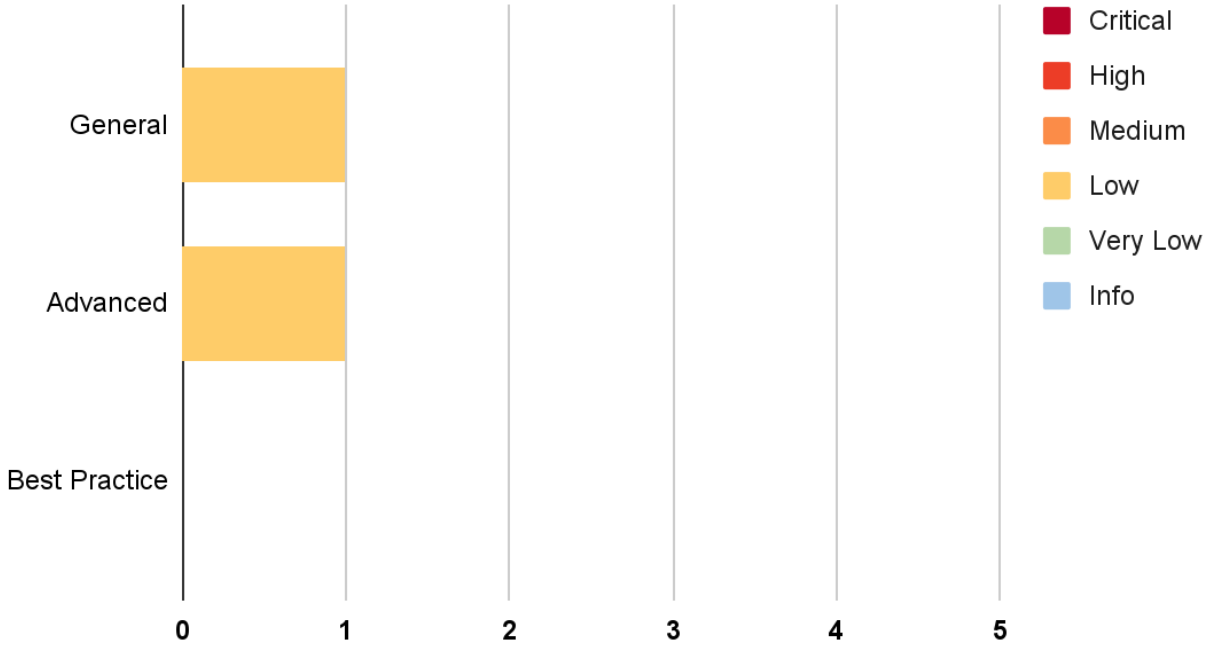
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Lack of NFT Contract Validation	Advanced	Critical	Resolved *
IDX-002	Lack of Collateral Offer Status Validation	Advanced	Critical	Resolved
IDX-003	Lack of Investment Status Validation	Advanced	Critical	Resolved
IDX-004	Insecure Randomness in openBox() Function	Advanced	Critical	Resolved
IDX-005	Lack of Closed Arena Validation	Advanced	Critical	Resolved
IDX-006	Incorrect Change State Operator in finalizeAuction() Function	Advanced	Critical	Resolved
IDX-007	Improper cancelCounterOffer() Function Implementation	Advanced	Critical	Resolved
IDX-008	Use of Upgradable Contract Design	General	High	Resolved *
IDX-009	Centralized Control of State Variable	General	High	Resolved *
IDX-010	Improper Design for Operator Privilege	Advanced	High	Resolved *
IDX-011	Improper Free Character Validation in createCharacter() Function	Advanced	High	Resolved
IDX-012	Missing Item Transfer of Disbanded Guild	Advanced	Medium	Resolved
IDX-013	Improper requestDonateItem() Function Implementation	Advanced	Medium	Resolved
IDX-014	Missing Boundary State Variable in upgradeItem() Function	Advanced	Medium	Resolved

IDX-015	Incorrect Expiry Time Calculation in renewHiring() Function	Advanced	Medium	Resolved
IDX-016	Division Before Multiplication	Advanced	Low	Resolved
IDX-017	Lack of Guild Disband Validation	Advanced	Low	Resolved
IDX-018	Denial of Service in Validating the Expiration of Right	Advanced	Low	Acknowledged
IDX-019	Lack of Hiring Status Validation in renewHiring() Function	Advanced	Low	Resolved
IDX-020	Smart Contract with Unpublished Source Code	General	Low	Acknowledged
IDX-021	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-022	Unbound Configuration Parameter (commissionRate)	Advanced	Very Low	Resolved
IDX-023	Improper Access Control in finalizeRightPurchase() Function	Advanced	Very Low	Resolved
IDX-024	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved
IDX-025	Improper Function Visibility	Best Practice	Info	No security impact

* The mitigations or clarifications by Gold Fever can be found in Chapter 5.

5. Detailed Findings Information

5.1. Lack of NFT Contract Validation

ID	IDX-001
Target	GoldFeverAuction GoldFeverBuildingManager GoldFeverCharacter GoldFeverCollateral GoldFeverGovernor GoldFeverGuild GoldFeverLeasing GoldFeverMarket GoldFeverNpc GoldFeverSwap
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity: Critical** **Impact: High** The attacker can supply the malicious NFT contract as a parameter in various functions to abuse the business logic. This results in the loss of users' platform assets. **Likelihood: High** The attacker can create a malicious NFT contract and use it to exploit the platform without any restriction.
Status	Resolved * The Gold Fever team has mitigated this issue by verifying the NFT contract by the `updateNftContract()` function in the `GoldFeverAdmin` contract. However, the manipulated NFT contract can be whitelisted by the admin, the users should check that the contract is trusted.

5.1.1. Description

The attacker can deploy a malicious NFT contract that does not relate to the Gold Fever platform and then use it to abuse the functions to gain profit, for example:

The `GoldFeverGuild` contract allows the guild owner to create the investment offer for buying the item with the specific `itemTypeId` by executing the `createInvestment()` function.

GoldFeverGuild.sol

```
749 function createInvestment(
```

```
750         uint256 itemTypeId,
751         uint256 nglAmount,
752         uint256 durationRaising,
753         uint256 durationListing,
754         uint256 characterId,
755         string memory name
756     ) public nonReentrant {
757         require(
758             nglAmount > 0,
759             "GoldFeverGuild: nglAmount should be greater than 0"
760         );
761 
762         require(
763             durationListing > durationRaising,
764             "GoldFeverGuild: listing time should be greater than raising time"
765         );
766 
767         uint256 guildId = characterIdToMember[characterId].guildId;
768         require(
769             IERC721(characterContract).ownerOf(characterId) == _msgSender(),
770             "GoldFeverGuild: not owner character"
771         );
772         require(
773             idToGuild[guildId].status == STATUS_CREATED,
774             "GoldFeverGuild: guild is not available"
775         );
776         require(
777             _msgSender() == idToGuild[guildId].owner,
778             "GoldFeverGuild: not owner guild"
779         );
780         require(
781             checkOwnerRight(idToGuild[guildId].owner, idToGuild[guildId].limit),
782             "GoldFeverGuild: owner don't have right to owned guild"
783         );
784         (bool isKickstarter, , ) = IGuildRight(guildRightContract)
785             .checkGuildType(idToGuild[guildId].owner);
786         require(
787             isKickstarter,
788             "GoldFeverGuild: has not enabled feature investment"
789         );
790 
791         _investmentIds.increment();
792         uint256 counter = _investmentIds.current();
793         guildIdToInvestmentIdToInvestmentItem[guildId][counter] = Investment(
794             counter,
795             guildId,
796             name,
```

```
797             itemTypeId,
798             nglAmount,
799             0,
800             block.timestamp + durationRaising,
801             block.timestamp + durationListing,
802             STATUS_CREATED
803         );
804         emit InvestmentCreated(
805             counter,
806             guildId,
807             name,
808             itemTypeId,
809             nglAmount,
810             0,
811             block.timestamp + durationRaising,
812             block.timestamp + durationListing
813         );
814     }
```

The guild member also contributes to the investment by executing the `contributeInvestment()` function.

GoldFeverGuild.sol

```
816 function contributeInvestment(
817     uint256 investmentId,
818     uint256 characterId,
819     uint256 nglAmount
820 ) public nonReentrant {
821     require(
822         nglAmount > 0,
823         "GoldFeverGuild: nglAmount should be greater than 0"
824     );
825     require(
826         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
827         "GoldFeverGuild: not owner character"
828     );
829     uint256 guildId = characterIdToMember[characterId].guildId;
830     require(
831         checkOwnerRight(idToGuild[guildId].owner, idToGuild[guildId].limit),
832         "GoldFeverGuild: owner don't have right to owned guild"
833     );
834     (bool isKickstarter, , ) = IGuildRight(guildRightContract)
835         .checkGuildType(idToGuild[guildId].owner);
836     require(
837         isKickstarter,
838         "GoldFeverGuild: has not enabled feature donation"
839     );
840     require(
```

```
841             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
842                 .expiredTimeRaising >
843                 block.timestamp &&
844                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
845                     .nglFund <
846                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
847                     .nglAmount,
848             "GoldFeverGuild: investment has ended"
849         );
850 
851         uint256 depositAmount = guildIdToInvestmentIdToInvestmentItem[guildId][
852             investmentId
853         ].nglFund +
854             nglAmount >
855             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
856                 .nglAmount
857             ? guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
858                 .nglAmount -
859                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
860                     .nglFund
861             : nglAmount;
862         ngl.transferFrom(_msgSender(), address(this), depositAmount);
863         investmentIdToCharacterIdToAmount[investmentId][
864             characterId
865         ] += depositAmount;
866         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
867             .nglFund += depositAmount;
868         emit InvestmentContributed(
869             investmentId,
870             characterId,
871             investmentIdToCharacterIdToAmount[investmentId][characterId],
872             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId].nglFund
873         );
874 }
```

When the target amount is reached, the contract allows any user to sell an item by executing the `sellItemToInvestment()` function. However, the attacker can sell a malicious item by supplying the malicious NFT contract as the `nftContract` parameter. The malicious NFT should be minted with the specific token ID to match the required item type.

GoldFeverGuild.sol

```
924 function sellItemToInvestment(
925     uint256 guildId,
926     uint256 investmentId,
927     uint256 itemId,
928     address nftContract
```

```
929 ) public nonReentrant {
930     Investment memory investment = guildIdToInvestmentIdToInvestmentItem[
931         guildId
932     ][investmentId];
933     require(
934         (IGoldFeverItemType(itemTypeContract).getItemType(itemId)) ==
935             investment.itemTypeId,
936         "GoldFeverGuild: item type is not match"
937     );
938     require(
939         investment.expiredTimeListing > block.timestamp &&
940             investment.nglAmount == investment.nglFund &&
941             investment.status == STATUS_CREATED,
942         "GoldFeverGuild: Investment is not avaible"
943     );
944
945     IERC721(nftContract).safeTransferFrom(
946         _msgSender(),
947         address(this),
948         itemId
949     );
950     ngl.transfer(_msgSender(), investment.nglAmount);
951     investment.status = STATUS_SUCCESS;
952     guildIdToInvestmentIdToInvestmentItem[guildId][
953         investmentId
954     ] = investment;
955
956     investmentIdToItem[investmentId] = InvestmentItem(
957         investmentId,
958         itemId,
959         nftContract
960     );
961     emit ItemToInvestmentSold(guildId, investmentId, itemId, nftContract);
962 }
```

The following table contains a list of the functions that have the `nftContract` parameter without the input validation.

Target	Function
GoldFeverAuction.sol (L:110)	createAuction()
GoldFeverBuildingManager.sol (L:78)	assignManager()
GoldFeverCharacter.sol (L:421)	attachItem()
GoldFeverCharacter.sol (L:502)	depositAttachedItem()

GoldFeverCharacter.sol (L:536)	donateAttachItemToGuild()
GoldFeverCollateral.sol (L:132)	createCollateralOffer()
GoldFeverGovernor.sol (L:298)	updateBuildingGovernor()
GoldFeverGovernor.sol (L:562)	payGovernorTax()
GoldFeverGovernor.sol (L:668)	calculateTax()
GoldFeverGuild.sol (L:552)	memberDonateItem()
GoldFeverGuild.sol (L:924)	sellItemToInvestment()
GoldFeverLeasing.sol (L:119)	createItem()
GoldFeverMarket.sol (L:109)	createListing()
GoldFeverNpc.sol (L:244)	createHiring()
GoldFeverNpc.sol (L:372)	depositItem()
GoldFeverNpc.sol (L:410)	approveWithdrawal()
GoldFeverNpc.sol (L:466)	approveWithdrawals()
GoldFeverNpc.sol (L:666)	cancelHiring()
GoldFeverNpc.sol (L:792)	depositItemFromContract()
GoldFeverSwap.sol (L:94)	createOffer()

5.1.2. Remediation

Inspex suggests whitelisting the NFT contract addresses in the constructor when deploying the contract. For example, adding the `nftContracts` state for storing the whitelisted NFT contract addresses:

GoldFeverGuild.sol

```
33  mapping(address => bool) private nftContracts;
34
35  constructor(
36      address admin,
37      address nglContract_,
38      address itemTypeContract_,
39      address[] memory _nftContracts
40  ) {
41      _setupContractId("GoldFeverGuild");
42      _setupRole(DEFAULT_ADMIN_ROLE, admin);
43      ngl = GoldFeverNativeGold(nglContract_);
44      itemTypeContract = itemTypeContract_;
```

```
45
46      for (uint256 i = 0; i < _nftContracts.length; i++) {
47          nftContracts[_nftContracts[i]] = true;
48      }
49  }
```

Validating the `nftContract` parameter to check whether it is in the whitelisted contract or not, as shown in line 933.

GoldFeverGuild.sol

```
    function sellItemToInvestment(
924     uint256 guildId,
925     uint256 investmentId,
926     uint256 itemId,
927     address nftContract
928 ) public nonReentrant {
929     Investment memory investment = guildIdToInvestmentIdToInvestmentItem[
930         guildId
931     ][investmentId];
932     require(nftContracts[nftContract] == true,"The nftContract is not in the
933 contracts whitelist");
        require(
934         (IGoldFeverItemType(itemTypeContract).getItemType(itemId)) ==
935             investment.itemTypeId,
936         "GoldFeverGuild: item type is not match"
937     );
938     require(
939         investment.expiredTimeListing > block.timestamp &&
940             investment.nglAmount == investment.nglFund &&
941             investment.status == STATUS_CREATED,
942         "GoldFeverGuild: Investment is not avaible"
943     );
944
945     IERC721(nftContract).safeTransferFrom(
946         _msgSender(),
947         address(this),
948         itemId
949     );
950     ngl.transfer(_msgSender(), investment.nglAmount);
951     investment.status = STATUS_SUCCESS;
952     guildIdToInvestmentIdToInvestmentItem[guildId][
953         investmentId
954     ] = investment;
955
956     investmentIdToItem[investmentId] = InvestmentItem(
957         investmentId,
958         itemId,
```

```
959             nftContract
960         );
961         emit ItemToInvestmentSold(guildId, investmentId, itemId, nftContract);
962     }
963
```

5.2. Lack of Collateral Offer Status Validation

ID	IDX-002
Target	GoldFeverCollateral
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** The attacker can execute the `takeCounterOfferIncludeTax()` function to set the `loaner` state to his address and then execute the `claimCollateral()` function to drain any NFTs in the contract. Additionally, unchecked collateral offer status validation allows the loanee to claim the collateral asset without paying back the loan. **Likelihood: High** It is very likely that the attacker will perform this action to drain all assets in the contract without any restrictions.
Status	Resolved The Gold fever team has resolved this issue by validating the collateral offer status The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.2.1. Description

Missing Authorization Check

The `GoldFeverCollateral` contract allows users to create a loan offer with NFT as collateral. By calling the `createCollateralOffer()` function, the NFT will be transferred to the contract and the loan offer will be created.

GoldFeverCollateral.sol

```
132 function createCollateralOffer(
133     address nftContract,
134     uint256 nftId,
135     uint256 loanValue,
136     uint256 fee,
137     uint256 duration
138 ) public nonReentrant {
139     require(
140         loanValue > 0,
141         "GoldFeverCollateral: Loan value must be greater than 0"
142     );
143     require(
144         fee < loanValue,
```

```
145             "GoldFeverCollateral: Fee must be less than loan value"
146         );
147
148         uint256 tax = (fee > 0)
149             ? ((fee * IGoldFeverTaxation(taxation).companyTax()) /
150                 (10**IGoldFeverTaxation(taxation).taxDecimals()) /
151                 100)
152             : 0;
153         _collateralOfferIds.increment();
154         uint256 collateralOfferId = _collateralOfferIds.current();
155
156         idToCollateralOffer[collateralOfferId] = CollateralOffer(
157             collateralOfferId,
158             _msgSender(),
159             nftContract,
160             nftId,
161             loanValue,
162             fee,
163             duration,
164             CREATED,
165             address(0),
166             tax
167         );
168
169         IERC721(nftContract).safeTransferFrom(
170             _msgSender(),
171             address(this),
172             nftId
173         );
174
175         emit CollateralOfferCreated(
176             collateralOfferId,
177             _msgSender(),
178             nftContract,
179             nftId,
180             loanValue,
181             fee,
182             duration,
183             tax
184         );
185     }
```

Normally, any user can use the `counterOffer()` function to counter the collateral offer with a different fee to the loanee. Then the `takeCounterOfferIncludeTax()` function is used by the loanee to take an offer from the loaner, who counter offers the loanee's NFT collateral price.

But there is no conditional checking to verify that the user who executes this function is the owner of their

NFT assets. Therefore, the attacker can execute this function to arbitrary set the `loaner` address at line 389.

GoldFeverCollateral.sol

```
350 function takeCounterOfferIncludeTax(
351     uint256 collateralOfferId,
352     address loaner
353 ) public nonReentrant {
354     address loanee = idToCollateralOffer[collateralOfferId].loanee;
355     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
356 
357     for (
358         uint256 i = 0;
359         i < collateralOfferIdToLoaner[collateralOfferId].length;
360         i++
361     ) {
362         address offerer = collateralOfferIdToLoaner[collateralOfferId][i]
363             .loaner;
364         uint256 fee = collateralOfferIdToLoaner[collateralOfferId][i].fee;
365         if (offerer == loaner) {
366             uint256 tax = collateralOfferIdToLoaner[collateralOfferId][i]
367                 .tax;
368             uint256 duration = idToCollateralOffer[collateralOfferId]
369                 .duration;
370             uint256 expiry = block.timestamp + duration;
371             idToExpiry[collateralOfferId] = expiry;
372 
373             ngl.transfer(IGoldFeverTaxation(taxation).taxCollector(), tax);
374             ngl.transfer(loanee, loanValue - fee - tax);
375             emit CollateralOfferAccepted(
376                 collateralOfferId,
377                 loaner,
378                 loanValue,
379                 fee,
380                 expiry,
381                 tax,
382                 true
383             );
384         } else {
385             ngl.transfer(offerer, loanValue - fee);
386         }
387     }
388     idToCollateralOffer[collateralOfferId].status = LOANED;
389     idToCollateralOffer[collateralOfferId].loaner = loaner;
390 }
```

If the collateral offer has not been accepted and its counter offer has not been taken yet, the attacker can perform the action above to bypass all conditional checking in the `claimCollateral()` function since the

`idToExpiry` state is `0` by default and steal the collateral asset at line 292.

GoldFeverCollateral.sol

```
272 function claimCollateral(uint256 collateralOfferId) public nonReentrant {
273     address loaner = idToCollateralOffer[collateralOfferId].loaner;
274     require(
275         idToCollateralOffer[collateralOfferId].status == LOANED,
276         "GoldFeverCollateral: Item must be in LOANED state"
277     );
278
279     require(
280         idToExpiry[collateralOfferId] <= block.timestamp,
281         "GoldFeverCollateral: Loan has not expired"
282     );
283     require(
284         loaner == _msgSender(),
285         "GoldFeverCollateral: Only loaner can claim"
286     );
287
288     uint256 nftId = idToCollateralOffer[collateralOfferId].nftId;
289     address nftContract = idToCollateralOffer[collateralOfferId]
290         .nftContract;
291
292     IERC721(nftContract).safeTransferFrom(address(this), loaner, nftId);
293     idToCollateralOffer[collateralOfferId].status = CLAIMED;
294     emit CollateralClaimed(collateralOfferId, loaner);
295 }
```

Missing Collateral Offer Status Check

There are functions that do not validate the collateral offer status. Hence, any user can still modify the collateral offer status even though they are on loan, for example:

The `counterOffer()` function is used to counteroffer the collateral price. However, when the collateral asset is on loan, NFT's owner can use another wallet address to call this function to arbitrary change the collateral asset status from `LOANED` to `OFFERED` at line 259.

GoldFeverCollateral.sol

```
224 function counterOffer(uint256 collateralOfferId, uint256 fee)
225     public
226     nonReentrant
227 {
228     require(
229         idToCollateralOffer[collateralOfferId].loanee != _msgSender(),
230         "GoldFeverCollateral: You cannot counter your own offer"
231     );
```

```
232     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
233
234     require(
235         fee >= 0,
236         "GoldFeverCollateral: Fee must be equal or greater than 0"
237     );
238     require(
239         fee < loanValue,
240         "GoldFeverCollateral: Fee must be less than loan value"
241     );
242     for (
243         uint256 i = 0;
244         i < collateralOfferIdToLoaner[collateralOfferId].length;
245         i++
246     ) {
247         address offerer = collateralOfferIdToLoaner[collateralOfferId][i]
248             .loaner;
249         if (offerer == _msgSender()) {
250             revert("GoldFeverCollateral: You can only counter offer once");
251         }
252     }
253     uint256 tax = (fee > 0)
254         ? ((fee * IGoldFeverTaxation(taxation).companyTax()) /
255             (10**IGoldFeverTaxation(taxation).taxDecimals()) /
256             100)
257         : 0;
258     ngl.transferFrom(_msgSender(), address(this), loanValue - fee);
259     idToCollateralOffer[collateralOfferId].status = OFFERED;
260     collateralOfferIdToLoaner[collateralOfferId].push(
261         CounterOffer(_msgSender(), loanValue, fee, tax)
262     );
263     emit CollateralOfferCountered(
264         collateralOfferId,
265         _msgSender(),
266         loanValue,
267         fee,
268         tax
269     );
270 }
```

Subsequently, the `cancelCollateralOffer()` function is used by the NFT's owner to cancel their collateral offer and claim the collateral NFT at line 205.

GoldFeverCollateral.sol

```
187 function cancelCollateralOffer(uint256 collateralOfferId)
188     public
189     nonReentrant
```

```
190 {
191     require(
192         idToCollateralOffer[collateralOfferId].loanee == _msgSender(),
193         "GoldFeverCollateral: Only loanee can cancel offer"
194     );
195     require(
196         idToCollateralOffer[collateralOfferId].status == CREATED ||
197             idToCollateralOffer[collateralOfferId].status == OFFERED,
198         "GoldFeverCollateral: Item must be in CREATED state"
199     );
200     address loanee = idToCollateralOffer[collateralOfferId].loanee;
201     uint256 nftId = idToCollateralOffer[collateralOfferId].nftId;
202     address nftContract = idToCollateralOffer[collateralOfferId]
203         .nftContract;
204     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
205     IERC721(nftContract).safeTransferFrom(address(this), loanee, nftId);
206     if (idToCollateralOffer[collateralOfferId].status == OFFERED) {
207         for (
208             uint256 i = 0;
209             i < collateralOfferIdToLoaner[collateralOfferId].length;
210             i++
211         ) {
212             address offerer = collateralOfferIdToLoaner[collateralOfferId][
213                 i
214             ].loaner;
215             uint256 fee = collateralOfferIdToLoaner[collateralOfferId][i]
216                 .fee;
217             ngl.transfer(offerer, loanValue - fee);
218         }
219     }
220     idToCollateralOffer[collateralOfferId].status = CANCELED;
221     emit CollateralOfferCanceled(collateralOfferId);
222 }
```

Since the collateral asset status is `OFFERED`, loanee can bypass all conditional checks in this function and claim the collateral asset without paying back the collateral. The `$NGL` will be transferred back to another wallet address of the loanee at line 217. Thus, the loaner will lose the loaned `$NGL` to this collateral offer.

The following table contains all functions that need to validate the collateral offer status.

File	Contract	Function
GoldFeverCollateral.sol (L:224)	GoldFeverCollateral	counterOffer()
GoldFeverCollateral.sol (L:323)	GoldFeverCollateral	cancelCounterOffer()
GoldFeverCollateral.sol (L:350)	GoldFeverCollateral	takeCounterOfferIncludeTax()

GoldFeverCollateral.sol (L:393)	GoldFeverCollateral	acceptCollateralOfferIncludeTax()

5.2.2. Remediation

Inspex suggests implementing an access control measure to allow only the `loanee` of their collateral offer to call the `takeCounterOfferIncludeTax()` function and validating the collateral offer status properly to handle the function that can change the status, as in the following examples:

For the `counterOffer()` function, add conditional checks that users can call this function only if collateral offer status is `CREATED` or `OFFERED` state in lines 232-236.

GoldFeverCollateral.sol

```
224 function counterOffer(uint256 collateralOfferId, uint256 fee)
225     public
226     nonReentrant
227 {
228     require(
229         idToCollateralOffer[collateralOfferId].loanee != _msgSender(),
230         "GoldFeverCollateral: You cannot counter your own offer"
231     );
232     require(
233         idToCollateralOffer[collateralOfferId].status == CREATED ||
234             idToCollateralOffer[collateralOfferId].status == OFFERED,
235         "GoldFeverCollateral: Item must be in CREATED or OFFERED state"
236     );
237     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
238
239     require(
240         fee >= 0,
241         "GoldFeverCollateral: Fee must be equal or greater than 0"
242     );
243     require(
244         fee < loanValue,
245         "GoldFeverCollateral: Fee must be less than loan value"
246     );
247     for (
248         uint256 i = 0;
249         i < collateralOfferIdToLoaner[collateralOfferId].length;
250         i++
251     ) {
252         address offerer = collateralOfferIdToLoaner[collateralOfferId][i]
253             .loaner;
254         if (offerer == _msgSender()) {
255             revert("GoldFeverCollateral: You can only counter offer once");
256         }
257     }
258     uint256 tax = (fee > 0)
```

```
259         ? ((fee * IGoldFeverTaxation(taxation).companyTax()) /
260             (10**IGoldFeverTaxation(taxation).taxDecimals()) /
261             100)
262         : 0;
263     ngl.transferFrom(_msgSender(), address(this), loanValue - fee);
264     idToCollateralOffer[collateralOfferId].status = OFFERED;
265     collateralOfferIdToLoaner[collateralOfferId].push(
266         CounterOffer(_msgSender(), loanValue, fee, tax)
267     );
268     emit CollateralOfferCountered(
269         collateralOfferId,
270         _msgSender(),
271         loanValue,
272         fee,
273         tax
274     );
275 }
```

For the `cancelCounterOffer()` function, add conditional checks that users can call this function only if the collateral offer status is `OFFERED` state in lines 328-331.

GoldFeverCollateral.sol

```
323 function cancelCounterOffer(uint256 collateralOfferId) public nonReentrant {
324     require(
325         idToCollateralOffer[collateralOfferId].loanee != _msgSender(),
326         "GoldFeverCollateral: Only offerer can cancel counter offer"
327     );
328     require(
329         idToCollateralOffer[collateralOfferId].status == OFFERED,
330         "GoldFeverCollateral: Item must be in OFFERED state"
331     );
332     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
333
334     for (
335         uint256 i = 0;
336         i < collateralOfferIdToLoaner[collateralOfferId].length;
337         i++
338     ) {
339         address offerer = collateralOfferIdToLoaner[collateralOfferId][i]
340             .loaner;
341         uint256 fee = collateralOfferIdToLoaner[collateralOfferId][i].fee;
342         if (offerer == _msgSender()) {
343             ngl.transfer(_msgSender(), loanValue - fee);
344             delete collateralOfferIdToLoaner[collateralOfferId][i];
345             emit CollateralCounterOfferCanceled(
346                 collateralOfferId,
347                 _msgSender()
```

```
348             );
349         }
350     }
351 }
```

For the `takeCounterOfferIncludeTax()` function, add conditional checks that allow only loanee to call this function and only collateral offer status is `OFFERED` state in lines 357-364.

GoldFeverCollateral.sol

```
350 function takeCounterOfferIncludeTax(
351     uint256 collateralOfferId,
352     address loaner
353 ) public nonReentrant {
354     address loanee = idToCollateralOffer[collateralOfferId].loanee;
355     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
356 
357     require(
358         loanee == _msgSender(),
359         "GoldFeverCollateral: Only loanee can take counter offer"
360     );
361     require(
362         idToCollateralOffer[collateralOfferId].status == OFFERED,
363         "GoldFeverCollateral: Item must be in OFFERED state"
364     );
365 
366     for (
367         uint256 i = 0;
368         i < collateralOfferIdToLoaner[collateralOfferId].length;
369         i++
370     ) {
371         address offerer = collateralOfferIdToLoaner[collateralOfferId][i]
372             .loaner;
373         uint256 fee = collateralOfferIdToLoaner[collateralOfferId][i].fee;
374         if (offerer == loaner) {
375             uint256 tax = collateralOfferIdToLoaner[collateralOfferId][i]
376                 .tax;
377             uint256 duration = idToCollateralOffer[collateralOfferId]
378                 .duration;
379             uint256 expiry = block.timestamp + duration;
380             idToExpiry[collateralOfferId] = expiry;
381 
382             ngl.transfer(IGoldFeverTaxation(taxation).taxCollector(), tax);
383             ngl.transfer(loanee, loanValue - fee - tax);
384             emit CollateralOfferAccepted(
385                 collateralOfferId,
386                 loaner,
387                 loanValue,
```

```
388                 fee,
389                 expiry,
390                 tax,
391                 true
392             );
393         } else {
394             ngl.transfer(offerer, loanValue - fee);
395         }
396     }
397     idToCollateralOffer[collateralOfferId].status = LOANED;
398     idToCollateralOffer[collateralOfferId].loaner = loaner;
399 }
```

For the `acceptCollateralOfferIncludeTax()` function, add conditional checks that allow users to call this function only if collateral status is `CREATED` or `OFFERED` state in lines 405-409.

GoldFeverCollateral.sol

```
393 function acceptCollateralOfferIncludeTax(uint256 collateralOfferId)
394     public
395     nonReentrant
396 {
397     address loanee = idToCollateralOffer[collateralOfferId].loanee;
398     uint256 fee = idToCollateralOffer[collateralOfferId].fee;
399     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
400     uint256 tax = idToCollateralOffer[collateralOfferId].tax;
401     require(
402         loanee != _msgSender(),
403         "GoldFeverCollateral: You cannot accept your own offer"
404     );
405     require(
406         idToCollateralOffer[collateralOfferId].status == CREATED ||
407             idToCollateralOffer[collateralOfferId].status == OFFERED,
408         "GoldFeverCollateral: Item must be in CREATED or OFFERED state"
409     );
410     for (
411         uint256 i = 0;
412         i < collateralOfferIdToLoaner[collateralOfferId].length;
413         i++
414     ) {
415         address offerer = collateralOfferIdToLoaner[collateralOfferId][i]
416             .loaner;
417         if (offerer == _msgSender()) {
418             revert("GoldFeverCollateral: You can only counter offer once");
419         }
420     }
421     if (idToCollateralOffer[collateralOfferId].status == OFFERED) {
422         for (
```

```
423             uint256 i = 0;
424             i < collateralOfferIdToLoaner[collateralOfferId].length;
425             i++
426         ) {
427             address offerer = collateralOfferIdToLoaner[collateralOfferId][
428                 i
429             ].loaner;
430             uint256 _fee = collateralOfferIdToLoaner[collateralOfferId][i]
431                 .fee;
432             ngl.transfer(offerer, loanValue - _fee);
433         }
434     }
435     uint256 duration = idToCollateralOffer[collateralOfferId].duration;
436     uint256 expiry = block.timestamp + duration;
437     idToExpiry[collateralOfferId] = expiry;
438 
439     ngl.transferFrom(
440         _msgSender(),
441         IGoldFeverTaxation(taxation).taxCollector(),
442         tax
443     );
444     ngl.transferFrom(_msgSender(), loanee, loanValue - fee);
445     idToCollateralOffer[collateralOfferId].status = LOANED;
446     idToCollateralOffer[collateralOfferId].loaner = _msgSender();
447     emit CollateralOfferAccepted(
448         collateralOfferId,
449         _msgSender(),
450         loanValue,
451         fee,
452         expiry,
453         tax,
454         false
455     );
456 }
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.3. Lack of Investment Status Validation

ID	IDX-003
Target	GoldFeverGuild
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** The users can still use the `withdrawFailedInvest()` function to withdraw their token from the contract even after their investment has already been successful. **Likelihood: High** This attack can be done by anyone who is in the guild and has ongoing investment.
Status	Resolved The Gold fever team has resolved this issue by validating the investment status The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.3.1. Description

The `GoldFeverGuild` contract allows the guild owner to create an investment. The guild members can contribute the created investment by executing the `contributeInvestment()` function.

GoldFeverGuild.sol

```
816 function contributeInvestment(
817     uint256 investmentId,
818     uint256 characterId,
819     uint256 nglAmount
820 ) public nonReentrant {
821     require(
822         nglAmount > 0,
823         "GoldFeverGuild: nglAmount should be greater than 0"
824     );
825     require(
826         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
827         "GoldFeverGuild: not owner character"
828     );
829     uint256 guildId = characterIdToMember[characterId].guildId;
830     require(
831         checkOwnerRight(idToGuild[guildId].owner, idToGuild[guildId].limit),
832         "GoldFeverGuild: owner don't have right to owned guild"
833     );
834     (bool isKickstarter, , ) = IGuildRight(guildRightContract)
```

```
835             .checkGuildType(idToGuild[guildId].owner);
836         require(
837             isKickstarter,
838             "GoldFeverGuild: has not enabled feature donation"
839         );
840         require(
841             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
842                 .expiredTimeRaising >
843                 block.timestamp &&
844                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
845                     .nglFund <
846                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
847                     .nglAmount,
848             "GoldFeverGuild: investment has ended"
849         );
850 
851         uint256 depositAmount = guildIdToInvestmentIdToInvestmentItem[guildId][
852             investmentId
853         ].nglFund +
854             nglAmount >
855             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
856                 .nglAmount
857             ? guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
858                 .nglAmount -
859                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
860                     .nglFund
861             : nglAmount;
862         ngl.transferFrom(_msgSender(), address(this), depositAmount);
863         investmentIdToCharacterIdToAmount[investmentId][
864             characterId
865         ] += depositAmount;
866         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
867             .nglFund += depositAmount;
868         emit InvestmentContributed(
869             investmentId,
870             characterId,
871             investmentIdToCharacterIdToAmount[investmentId][characterId],
872             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId].nglFund
873         );
874 }
```

When the investment has reached its target, any user can use the `sellItemToInvestment()` function to sell the item to the specified investment.

GoldFeverGuild.sol

```
924 function sellItemToInvestment(
925     uint256 guildId,
```

```
926     uint256 investmentId,
927     uint256 itemId,
928     address nftContract
929 ) public nonReentrant {
930     Investment memory investment = guildIdToInvestmentIdToInvestmentItem[
931         guildId
932     ][investmentId];
933     require(
934         (IGoldFeverItemType(itemTypeContract).getItemType(itemId)) ==
935             investment.itemTypeId,
936         "GoldFeverGuild: item type is not match"
937     );
938     require(
939         investment.expiredTimeListing > block.timestamp &&
940             investment.nglAmount == investment.nglFund &&
941             investment.status == STATUS_CREATED,
942         "GoldFeverGuild: Investment is not avaible"
943     );
944
945     IERC721(nftContract).safeTransferFrom(
946         _msgSender(),
947         address(this),
948         itemId
949     );
950     ngl.transfer(_msgSender(), investment.nglAmount);
951     investment.status = STATUS_SUCCESS;
952     guildIdToInvestmentIdToInvestmentItem[guildId][
953         investmentId
954     ] = investment;
955
956     investmentIdToItem[investmentId] = InvestmentItem(
957         investmentId,
958         itemId,
959         nftContract
960     );
961     emit ItemToInvestmentSold(guildId, investmentId, itemId, nftContract);
962 }
```

However, even after the item for this investment has been sold and the token has been transferred to the seller, the contributor can still withdraw their contribution by executing the `withdrawFailedInvest()` function to withdraw their contribution back because there is no validation for the success investment in this function.

GoldFeverGuild.sol

```
876 function withdrawFailedInvest(uint256 investmentId, uint256 characterId)
877     public
```

```
878     nonReentrant
879 {
880     require(
881         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
882         "GoldFeverGuild: not owner character"
883     );
884     uint256 guildId = characterIdToMember[characterId].guildId;
885     if (
886         idToGuild[guildId].joinFee > 0 &&
887         !characterIdToMember[characterId].isOwner
888     ) {
889         require(
890             characterIdToMember[characterId].expired > block.timestamp,
891             "GoldFeverGuild: member should pay guild fee first"
892         );
893     }
894     require(
895         investmentIdToCharacterIdToAmount[investmentId][characterId] > 0,
896         "GoldFeverGuild: has not invested"
897     );
898     require(
899         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
900             .expiredTimeRaising < block.timestamp,
901         "GoldFeverGuild: investment is running"
902     );
903
904     if (
905         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
906             .expiredTimeListing > block.timestamp
907     ) {
908         require(
909             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
910                 .nglFund <
911                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
912                     .nglAmount,
913             "GoldFeverGuild: investment is running"
914         );
915     }
916     ngl.transfer(
917         _msgSender(),
918         investmentIdToCharacterIdToAmount[investmentId][characterId]
919     );
920     delete investmentIdToCharacterIdToAmount[investmentId][characterId];
921     emit FailedInvestWithdrew(investmentId, characterId);
922 }
```

5.3.2. Remediation

Inspex suggests implementing the investment status validation to validate if the investment is already successful. For example, in lines 885–891:

GoldFeverGuild.sol

```
876 function withdrawFailedInvest(uint256 investmentId, uint256 characterId)
877     public
878     nonReentrant
879 {
880     require(
881         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
882         "GoldFeverGuild: not owner character"
883     );
884     uint256 guildId = characterIdToMember[characterId].guildId;
885     Investment memory investment = guildIdToInvestmentIdToInvestmentItem[
886             guildId
887         ][investmentId];
888     require(
889         investment.status != STATUS_SUCCESS,
890         "GoldFeverGuild: investment has already success"
891     );
892
893     if (
894         idToGuild[guildId].joinFee > 0 &&
895         !characterIdToMember[characterId].isOwner
896     ) {
897         require(
898             characterIdToMember[characterId].expired > block.timestamp,
899             "GoldFeverGuild: member should pay guild fee first"
900         );
901     }
902     require(
903         investmentIdToCharacterIdToAmount[investmentId][characterId] > 0,
904         "GoldFeverGuild: has not invested"
905     );
906     require(
907         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
908             .expiredTimeRaising < block.timestamp,
909         "GoldFeverGuild: investment is running"
910     );
911
912     if (
913         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
914             .expiredTimeListing > block.timestamp
915     ) {
916         require(
```

```
            guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
                .nglFund <
                guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
                    .nglAmount,
            "GoldFeverGuild: investment is running"
        );
    }
    ngl.transfer(
        _msgSender(),
        investmentIdToCharacterIdToAmount[investmentId][characterId]
    );
    delete investmentIdToCharacterIdToAmount[investmentId][characterId];
    emit FailedInvestWithdrew(investmentId, characterId);
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.4. Insecure Randomness in openBox() Function

ID	IDX-004
Target	GoldFeverMaskBox
Category	Advanced Smart Contract Vulnerability
CWE	CWE-330: Use of Insufficiently Random Values
Risk	**Severity: Critical** **Impact: High** The attacker can control the random result to select a specific NFT when opening the box. This gives an unfair advantage to the platform's users. **Likelihood: High** Since the NFTs were minted prior to transfer to this contract, the attacker is very likely to exploit this issue because the item ID of each material with a different rarity is publicly known.
Status	**Resolved** The Gold fever team has resolved this issue by implementing secure randomness with Chainlink VRF The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.4.1. Description

In the `GoldFeverMaskBox` contract, the `openBox()` function uses a value from the `_random()` function at line 145 in order to indicate which NFTs should be transferred to the user.

GoldFeverMaskBox.sol

```
138 function openBox(uint256 boxId) public {
139     address msgSender = _msgSender();
140     require(
141         msgSender == ownerOf(boxId),
142         "GoldFeverMaskBox: Only owner can open box"
143     );
144     uint256[] memory allMaskPartIds = new uint256[](6);
145     uint256 rnd_shape = _random(maskShapesPool.length);
146 
147     gfi.safeTransferFrom(
148         address(this),
149         msgSender,
150         maskShapesPool[rnd_shape]
151     );
152     allMaskPartIds[0] = maskShapesPool[rnd_shape];
```

```
153     maskShapesPool[rnd_shape] = maskShapesPool[maskShapesPool.length - 1];
154     maskShapesPool.pop();
155
156     uint256 rnd_material = _random(maskMaterialsPool.length);
157     gfi.safeTransferFrom(
158         address(this),
159         msgSender,
160         maskMaterialsPool[rnd_material]
161     );
162     allMaskPartIds[1] = maskMaterialsPool[rnd_material];
163     maskMaterialsPool[rnd_material] = maskMaterialsPool[
164         maskMaterialsPool.length - 1
165     ];
166     maskMaterialsPool.pop();
167
168     for (uint256 i = 0; i < 4; i++) {
169         uint256 rnd_element = _random(otherMaskElementsPool.length);
170         gfi.safeTransferFrom(
171             address(this),
172             msgSender,
173             otherMaskElementsPool[rnd_element]
174         );
175         allMaskPartIds[i + 2] = otherMaskElementsPool[rnd_element];
176         otherMaskElementsPool[rnd_element] = otherMaskElementsPool[
177             otherMaskElementsPool.length - 1
178         ];
179         otherMaskElementsPool.pop();
180     }
181
182     _burn(boxId);
183     emit MaskBoxOpened(boxId, allMaskPartIds);
184 }
```

The random number is generated from the Keccak256 value of the `block.difficulty` and `block.timestamp`.

GoldFeverMaskBox.sol

```
186 function _random(uint256 number) private view returns (uint256) {
187     return
188         uint256(
189             keccak256(abi.encodePacked(block.difficulty, block.timestamp))
190         ) % number;
191 }
```

The result of the `_random()` function can easily be predicted since the `block.difficulty` and `block.timestamp` values are publicly known and can also be used by anyone.

Therefore, the attacker can pre-calculate the result in each block before calling the `openBox()` function to get the expected NFTs. This results in an unfair advantage for the platform's users.

5.4.2. Remediation

Inspex suggests using a provably-fair and verifiable source of randomness to calculate in the `_random()` function.

In this case, Inspex recommends using Chainlink VRF as the randomness source in the `_random()` function (https://docs.chain.link/docs/vrf/v2/introduction/).

5.5. Lack of Closed Arena Validation

ID	IDX-005
Target	GoldFeverMiningClaim
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** Due to lack of arena status validation in the `cancelArenaRegistration()` function, the attacker can exploit this issue to drain $NGL from the contract. **Likelihood: High** This attack can be done by anyone that has a mining claim item.
Status	Resolved The Gold Fever team has resolved this issue by implementing arena status validation in the `cancelArenaRegistration()` function. The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.5.1. Description

In the `GoldFeverMiningClaim` contract, the mining claim owner can execute the `startArena()` function to start the arena with a specific duration, which allows other users to register and pay the upfront fee to the mining claim owner to join the arena.

GoldFeverMiningClaim.sol

```
289 function startArena(
290     uint256 miningClaimId,
291     uint256 duration,
292     uint256 upfrontFee,
293     uint256 commissionRate
294 ) public nonReentrant {
295     address msgSender = _msgSender();
296     require(
297         gfi.ownerOf(miningClaimId) == msgSender,
298         "GoldFeverMiningClaim: Only owner of mining claim can start arena"
299     );
300     require(
301         duration <= idToMiningClaim[miningClaimId].arenaHour,
302         "GoldFeverMiningClaim: Arena open duration must be less than or equal
    to arena total hour"
303     );
```

```
304     require(
305         duration > 0,
306         "GoldFeverMiningClaim: Arena open duration must be greater than 0"
307     );
308 
309     _arenaIds.increment();
310     uint256 arenaId = _arenaIds.current();
311     uint256 expiry = (duration * 3600) + block.timestamp;
312     arenaIdToExpiry[arenaId] = expiry;
313 
314     idToArena[arenaId] = Arena(
315         arenaId,
316         msgSender,
317         miningClaimId,
318         0,
319         ARENA_STARTED,
320         duration,
321         upfrontFee,
322         commissionRate,
323         0
324     );
325     idToMiningClaim[miningClaimId].arenaHour -= duration;
326     idToMiningClaim[miningClaimId].currentArenaId = arenaId;
327     gfi.safeTransferFrom(msgSender, address(this), miningClaimId);
328 
329     emit ArenaStarted(
330         arenaId,
331         msgSender,
332         miningClaimId,
333         0,
334         ARENA_STARTED,
335         duration,
336         expiry,
337         upfrontFee,
338         commissionRate
339     );
340 }
```

The user can register to enter the arena by executing the `registerAtArena()` function. At this state, the status is set to `MINER_REGISTERED` and the upfront fee will transfer from the user to the contract at line 383.

GoldFeverMiningClaim.sol

```
366 function registerAtArena(uint256 arenaId) public nonReentrant {
367     require(
368         idToArena[arenaId].status == ARENA_STARTED,
369         "GoldFeverMiningClaim: Arena is not started"
370     );
```

```
371     uint256 upfrontFee = idToArena[arenaId].upfrontFee;
372     address msgSender = _msgSender();
373     require(
374         isMinerRegistered(arenaId, msgSender) == false,
375         "GoldFeverMiningClaim: Miner already registered"
376     );
377     idToMinerByArena[arenaId][msgSender] = Miner(
378         arenaId,
379         msgSender,
380         MINER_REGISTERED
381     );
382
383     ngl.transferFrom(msgSender, address(this), upfrontFee);
384     idToArena[arenaId].holdingFee += upfrontFee;
385     emit MinerRegistered(arenaId, msgSender, MINER_REGISTERED);
386 }
```

When the arena time is up, the admin will execute the `closeArena()` function to close the arena, and the collected upfront fee will be sent to the mining claim owner in line 362.

GoldFeverMiningClaim.sol

```
342 function closeArena(uint256 arenaId)
343     public
344     nonReentrant
345     only(DEFAULT_ADMIN_ROLE)
346 {
347     require(
348         arenaIdToExpiry[arenaId] <= block.timestamp,
349         "GoldFeverMiningClaim: Arena is not finished"
350     );
351     uint256 miningClaimId = idToArena[arenaId].miningClaimId;
352     address miningClaimOwner = idToArena[arenaId].owner;
353     uint256 holdingFee = idToArena[arenaId].holdingFee;
354     idToArena[arenaId].status = ARENA_CLOSED;
355     gfi.safeTransferFrom(
356         address(this),
357         idToArena[arenaId].owner,
358         miningClaimId
359     );
360     idToMiningClaim[miningClaimId].currentArenaId = 0;
361     idToArena[arenaId].numMinersInArena = 0;
362     ngl.transfer(miningClaimOwner, holdingFee);
363     emit ArenaClosed(arenaId);
364 }
```

However, after the arena is closed and the user does not enter the arena, the user status will still be `MINER_REGISTERED`, which allows the user to cancel the registration and get the token back by executing the

`cancelArenaRegistration()` function.

GoldFeverMiningClaim.sol

```
388 function cancelArenaRegistration(uint256 arenaId) public nonReentrant {
389     address msgSender = _msgSender();
390     require(
391         idToMinerByArena[arenaId][msgSender].status == MINER_REGISTERED ||
392             idToMinerByArena[arenaId][msgSender].status == MINER_LEFT,
393         "GoldFeverMiningClaim: Miner not registered"
394     );
395     uint256 upfrontFee = idToArena[arenaId].upfrontFee;
396
397     if (idToMinerByArena[arenaId][msgSender].status == MINER_REGISTERED) {
398         ngl.transfer(msgSender, upfrontFee);
399         idToArena[arenaId].holdingFee -= upfrontFee;
400     }
401     delete idToMinerByArena[arenaId][msgSender];
402     emit MinerCanceledRegistration(arenaId, msgSender, MINER_UNREGISTERED);
403 }
```

As a result, the attacker can start the arena and register to enter the arena using different addresses and wait until the arena expires. The admin will close the arena by executing the `closeArena()` function then the attacker who owns the arena will receive the upfront payment. Finally, the attacker executes the `cancelArenaRegistration()` function by the registered addresses to take back the upfront fee, which will drain the `$NGL` from the contract.

5.5.2. Remediation

Inspex suggests implementing the arena status `ARENA_CLOSED` validation in the `cancelArenaRegistration()` function. For example, in lines 390-393;

GoldFeverMiningClaim.sol

```
388 function cancelArenaRegistration(uint256 arenaId) public nonReentrant {
389     address msgSender = _msgSender();
390     require(
391         idToArena[arenaId].status != ARENA_CLOSED,
392         "GoldFeverMiningClaim: Arena is closed"
393     );
394     require(
395         idToMinerByArena[arenaId][msgSender].status == MINER_REGISTERED ||
396             idToMinerByArena[arenaId][msgSender].status == MINER_LEFT,
397         "GoldFeverMiningClaim: Miner not registered"
398     );
399     uint256 upfrontFee = idToArena[arenaId].upfrontFee;
400
401     if (idToMinerByArena[arenaId][msgSender].status == MINER_REGISTERED) {
```

```
402             ngl.transfer(msgSender, upfrontFee);
403             idToArena[arenaId].holdingFee -= upfrontFee;
404         }
405         delete idToMinerByArena[arenaId][msgSender];
406         emit MinerCanceledRegistration(arenaId, msgSender, MINER_UNREGISTERED);
407     }
```

5.6. Incorrect Change State Operator in finalizeAuction() Function

ID	IDX-006
Target	GoldFeverAuction
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** The auction status is not changed after it is finalized. The attacker can take advantage of this flaw to drain all `$NGL` from the contract. **Likelihood: High** Anyone can perform this attack by setting up an auction with an unreasonable high starting bid and using a different wallet address to place a winning offer.
Status	Resolved The Gold Fever team has resolved this issue by implementing the correct auction state validation in the `finalizeAuction()` function. The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.6.1. Description

The `GoldFeverAuction` contract provides the auction system for the Gold Fever platform. The `createAuction()` function in this contract allows the user to create the auction for their item at line 141 - 155.

GoldFeverAuction.sol

```
110 function createAuction(
111     address nftContract,
112     uint256 nftId,
113     uint256 startingPrice,
114     uint256 startTime,
115     uint256 duration,
116     uint256 biddingStep
117 ) public nonReentrant returns (uint256) {
118     require(
119         biddingStep > 0,
120         "GoldFeverAuction: Bidding step must be at least 1 wei"
121     );
122     _auctionIds.increment();
123     uint256 auctionId = _auctionIds.current();
124     address msgSender = _msgSender();
```

```
125
126     (, uint256 limitType, uint256 generalLimitType) = IMerchantRight(
127         merchantContract
128     ).getItemTypeId(nftId);
129     if (limitType != 0) {
130         IMerchantRight(merchantContract).isItemMaxOut(
131             msgSender,
132             limitType,
133             generalLimitType,
134             addressToItemCount[msgSender][limitType],
135             addressToItemCount[msgSender][generalLimitType]
136         );
137         addressToItemCount[msgSender][limitType]++;
138         addressToItemCount[msgSender][generalLimitType]++;
139     }
140
141     idToAuction[auctionId] = Auction(
142         auctionId,
143         nftContract,
144         nftId,
145         msgSender,
146         startTime,
147         startingPrice,
148         biddingStep,
149         duration,
150         startingPrice,
151         address(0),
152         CREATED,
153         limitType,
154         generalLimitType
155     );
156
157     IERC721(nftContract).safeTransferFrom(msgSender, address(this), nftId);
158
159     emit AuctionCreated(
160         auctionId,
161         nftContract,
162         nftId,
163         msgSender,
164         startingPrice,
165         startTime,
166         duration,
167         biddingStep
168     );
169     return auctionId;
170 }
```

If the users want to bid for the auction, they can execute the `bid()` function to bid for that auction.

GoldFeverAuction.sol

```
172 function bid(uint256 auctionId, uint256 price)
173     public
174     nonReentrant
175     returns (bool)
176 {
177     uint256 startDate = idToAuction[auctionId].startTime;
178     uint256 endDate = idToAuction[auctionId].startTime +
179         idToAuction[auctionId].duration;
180     require(
181         block.timestamp >= startDate && block.timestamp < endDate,
182         "GoldFeverAuction: Auction is finished or not started yet"
183     );
184     address msgSender = _msgSender();
185     if (idToAuction[auctionId].status == CREATED) {
186         require(
187             price >= idToAuction[auctionId].startingPrice,
188             "GoldFeverAuction: Must bid equal or higher than current starting
    price"
189         );
190         require(
191             (price - idToAuction[auctionId].startingPrice) %
192                 idToAuction[auctionId].biddingStep ==
193                 0,
194             "GoldFeverAuction: Bid price must be divisible by bidding step"
195         );
196 
197         ngl.transferFrom(msgSender, address(this), price);
198         idToAuction[auctionId].highestBidAmount = price;
199         idToAuction[auctionId].highestBidder = msgSender;
200         idToAuction[auctionId].status = BID;
201         emit AuctionBid(auctionId, msgSender, price);
202         return true;
203     }
204     if (idToAuction[auctionId].status == BID) {
205         require(
206             price >=
207                 idToAuction[auctionId].highestBidAmount +
208                     idToAuction[auctionId].biddingStep,
209             "GoldFeverAuction: Must bid higher than current highest bid"
210         );
211         require(
212             (price - idToAuction[auctionId].highestBidAmount) %
213                 idToAuction[auctionId].biddingStep ==
214                 0,
215             "GoldFeverAuction: Bid price must be divisible by bidding step"
216         );
```

```
217
218         ngl.transferFrom(msgSender, address(this), price);
219         if (idToAuction[auctionId].highestBidder != address(0)) {
220             // return ngl to the previuos bidder
221             ngl.transfer(
222                 idToAuction[auctionId].highestBidder,
223                 idToAuction[auctionId].highestBidAmount
224             );
225         }
226
227         // register new bidder
228         idToAuction[auctionId].highestBidder = msgSender;
229         idToAuction[auctionId].highestBidAmount = price;
230
231         emit AuctionBid(auctionId, msgSender, price);
232         return true;
233     }
234     return false;
235 }
```

The `finalizeAuction()` function is used to finalize the auction after the time is up. If no one bids on this auction, the item will be returned to the owner at line 301; on the other hand, if there is a winner, the highest bidder's tokens will be sent to the item's owner, and the contract will send the item to the winner by calling the `claimItem()` function in line 309 to transfer the item to the winner.

GoldFeverAuction.sol

```
288 function finalizeAuction(uint256 auctionId) public nonReentrant {
289     require(
290         isFinished(auctionId),
291         "GoldFeverAuction: Auction is not finished"
292     );
293     require(
294         idToAuction[auctionId].status != FINISHED,
295         "GoldFeverAuction: Auction is already finalized"
296     );
297
298     address owner = idToAuction[auctionId].owner;
299
300     if (idToAuction[auctionId].highestBidder == address(0)) {
301         IERC721(idToAuction[auctionId].nftContract).safeTransferFrom(
302             address(this),
303             owner,
304             idToAuction[auctionId].nftId
305         );
306         idToAuction[auctionId].status == FINISHED;
307     } else {
```

```
308         ngl.transfer(owner, idToAuction[auctionId].highestBidAmount);
309         claimItem(auctionId);
310         idToAuction[auctionId].status == FINISHED;
311     }
312
313     uint256 limitType = idToAuction[auctionId].limitType;
314     if (limitType != 0) {
315         addressToItemCount[owner][limitType]--;
316         addressToItemCount[owner][
317             idToAuction[auctionId].generalLimitType
318         ]--;
319     }
320 }
```

However, either the case that there are winners in the auction or not the auction status is not set to `FINISHED`. As a result, when the attacker is the winner of any auction; they can continuously call the `finalizeAuction()` function to drain the `$NGL` from the contract until the contract runs out of `$NGL`.

The NFT must be transferred to the contract before each `finalizeAuction()` call in order to avoid an error in the `safeTransferFrom()` function called in the `claimItem()` function in line 280.

GoldFeverAuction.sol

```
275 function claimItem(uint256 auctionId) private {
276     address winner = getWinner(auctionId);
277     require(winner != address(0), "GoldFeverAuction: There is no winner");
278     address nftContract = idToAuction[auctionId].nftContract;
279
280     IERC721(nftContract).safeTransferFrom(
281         address(this),
282         winner,
283         idToAuction[auctionId].nftId
284     );
285     emit Claim(auctionId, winner);
286 }
```

5.6.2. Remediation

Inspex suggests implementing the proper auction status validation in the `finalizeAuction()` function. For example, in line 311:

GoldFeverAuction.sol

```
288 function finalizeAuction(uint256 auctionId) public nonReentrant {
289     require(
290         isFinished(auctionId),
291         "GoldFeverAuction: Auction is not finished"
292     );
```

```
293         require(
294             idToAuction[auctionId].status != FINISHED,
295             "GoldFeverAuction: Auction is already finalized"
296         );
297 
298         address owner = idToAuction[auctionId].owner;
299 
300         if (idToAuction[auctionId].highestBidder == address(0)) {
301             IERC721(idToAuction[auctionId].nftContract).safeTransferFrom(
302                 address(this),
303                 owner,
304                 idToAuction[auctionId].nftId
305             );
306         } else {
307             ngl.transfer(owner, idToAuction[auctionId].highestBidAmount);
308             claimItem(auctionId);
309         }
310 
311         idToAuction[auctionId].status = FINISHED;
312         uint256 limitType = idToAuction[auctionId].limitType;
313         if (limitType != 0) {
314             addressToItemCount[owner][limitType]--;
315             addressToItemCount[owner][
316                 idToAuction[auctionId].generalLimitType
317             ]--;
318         }
319 }
```

5.7. Improper cancelCounterOffer() Function Implementation

ID	IDX-007
Target	GoldFeverCollateral
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** After the `cancelCounterOffer()` function is called, the counter offer is not completely removed, resulting in the collateral NFT being unable to be withdrawn from the contract. **Likelihood: High** Anyone can execute the `counterOffer()` and `cancelCounterOffer()` functions freely.
Status	Resolved The Gold Fever team has resolved this issue by changing the `cancelCounterOffer()` function to use the `pop()` function instead of `delete` for canceling counter offers, as we suggested in the commit `2c0e553a332d267df513888bbf7a6af2190f0af9`.

5.7.1. Description

In the `GoldFeverCollateral` contract, the `counterOffer()` function is used for offerers to counteroffer collateral assets. When users execute this function, the `CounterOffer` struct is pushed into the `collateralOfferIdToLoaner` array at line 260 - 262.

GoldFeverCollateral.sol

```
224 function counterOffer(uint256 collateralOfferId, uint256 fee)
225     public
226     nonReentrant
227 {
228     require(
229         idToCollateralOffer[collateralOfferId].loanee != _msgSender(),
230         "GoldFeverCollateral: You cannot counter your own offer"
231     );
232     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
233
234     require(
235         fee >= 0,
236         "GoldFeverCollateral: Fee must be equal or greater than 0"
237     );
238     require(
239         fee < loanValue,
```

```
240             "GoldFeverCollateral: Fee must be less than loan value"
241         );
242         for (
243             uint256 i = 0;
244             i < collateralOfferIdToLoaner[collateralOfferId].length;
245             i++
246         ) {
247             address offerer = collateralOfferIdToLoaner[collateralOfferId][i]
248                 .loaner;
249             if (offerer == _msgSender()) {
250                 revert("GoldFeverCollateral: You can only counter offer once");
251             }
252         }
253         uint256 tax = (fee > 0)
254             ? ((fee * IGoldFeverTaxation(taxation).companyTax()) /
255                 (10**IGoldFeverTaxation(taxation).taxDecimals()) /
256                 100)
257             : 0;
258         ngl.transferFrom(_msgSender(), address(this), loanValue - fee);
259         idToCollateralOffer[collateralOfferId].status = OFFERED;
260         collateralOfferIdToLoaner[collateralOfferId].push(
261             CounterOffer(_msgSender(), loanValue, fee, tax)
262         );
263         emit CollateralOfferCountered(
264             collateralOfferId,
265             _msgSender(),
266             loanValue,
267             fee,
268             tax
269         );
270     }
```

The `cancelCounterOffer()` function is used for offerers to cancel their counter offer NFTs and return their $NGL back to the offerers then delete the `collateralOfferIdToLoaner` with offerers index at line 340.

GoldFeverCollateral.sol

```
323 function cancelCounterOffer(uint256 collateralOfferId) public nonReentrant {
324     require(
325         idToCollateralOffer[collateralOfferId].loanee != _msgSender(),
326         "GoldFeverCollateral: Only offerer can cancel counter offer"
327     );
328     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
329
330     for (
331         uint256 i = 0;
332         i < collateralOfferIdToLoaner[collateralOfferId].length;
333         i++
```

```
334     ) {
335         address offerer = collateralOfferIdToLoaner[collateralOfferId][i]
336             .loaner;
337         uint256 fee = collateralOfferIdToLoaner[collateralOfferId][i].fee;
338         if (offerer == _msgSender()) {
339             ngl.transfer(_msgSender(), loanValue - fee);
340             delete collateralOfferIdToLoaner[collateralOfferId][i];
341             emit CollateralCounterOfferCanceled(
342                 collateralOfferId,
343                 _msgSender()
344             );
345         }
346     }
347 }
```

If a user wants to cancel their counter offer, there will be `zero` address contained in the `collateralOfferIdToLoaner` array.

As mentioned above, the NFT owner cannot cancel their collateral asset to claim their NFT since the `cancelCollateralOffer()` function call will revert due to the transfer of `$NGL` to `zero` address at line 217.

GoldFeverCollateral.sol

```
187 function cancelCollateralOffer(uint256 collateralOfferId)
188     public
189     nonReentrant
190 {
191     require(
192         idToCollateralOffer[collateralOfferId].loanee == _msgSender(),
193         "GoldFeverCollateral: Only loanee can cancel offer"
194     );
195     require(
196         idToCollateralOffer[collateralOfferId].status == CREATED ||
197             idToCollateralOffer[collateralOfferId].status == OFFERED,
198         "GoldFeverCollateral: Item must be in CREATED state"
199     );
200     address loanee = idToCollateralOffer[collateralOfferId].loanee;
201     uint256 nftId = idToCollateralOffer[collateralOfferId].nftId;
202     address nftContract = idToCollateralOffer[collateralOfferId]
203         .nftContract;
204     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
205     IERC721(nftContract).safeTransferFrom(address(this), loanee, nftId);
206     if (idToCollateralOffer[collateralOfferId].status == OFFERED) {
207         for (
208             uint256 i = 0;
209             i < collateralOfferIdToLoaner[collateralOfferId].length;
210             i++
```

```
211         ) {
212             address offerer = collateralOfferIdToLoaner[collateralOfferId][
213                 i
214             ].loaner;
215             uint256 fee = collateralOfferIdToLoaner[collateralOfferId][i]
216                 .fee;
217             ngl.transfer(offerer, loanValue - fee);
218         }
219     }
220     idToCollateralOffer[collateralOfferId].status = CANCELED;
221     emit CollateralOfferCanceled(collateralOfferId);
222 }
```

Additionally, this problem still exists even if the `takeCounterOfferIncludeTax()` or `acceptCollateralOffer()` functions are called, since they both error while transferring $NGL to the `zero` address.

5.7.2. Remediation

Inspex suggests using the `pop()` function instead of using `delete` to cancel counter offers in lines 344-347. After users call this function, if there is no one offering the NFT, change the collateral offer status into `CREATED` state in lines 348–350.

GoldFeverCollateral.sol

```
function cancelCounterOffer(uint256 collateralOfferId) public nonReentrant {
323     require(
324         idToCollateralOffer[collateralOfferId].loanee != _msgSender(),
325         "GoldFeverCollateral: Only offerer can cancel counter offer"
326     );
327     require(
328         idToCollateralOffer[collateralOfferId].status == OFFERED,
329         "GoldFeverCollateral: Item must be in OFFERED state"
330     );
331     uint256 loanValue = idToCollateralOffer[collateralOfferId].loanValue;
332
333     for (
334         uint256 i = 0;
335         i < collateralOfferIdToLoaner[collateralOfferId].length;
336         i++
337     ) {
338         address offerer = collateralOfferIdToLoaner[collateralOfferId][i]
339             .loaner;
340         uint256 fee = collateralOfferIdToLoaner[collateralOfferId][i].fee;
341         if (offerer == _msgSender()) {
342             ngl.transfer(_msgSender(), loanValue - fee);
343             if (i != collateralOfferIdToLoaner[collateralOfferId].length - 1) {
```

```
344                 collateralOfferIdToLoaner[collateralOfferId][i] =
345 collateralOfferIdToLoaner[collateralOfferId][collateralOfferIdToLoaner[collater
    alOfferId].length - 1];
                }
346             collateralOfferIdToLoaner[collateralOfferId].pop();
347             if (collateralOfferIdToLoaner[collateralOfferId].length == 0) {
348                 collateralOfferIdToLoaner[collateralOfferId].status = CREATED;
349             }
350             emit CollateralCounterOfferCanceled(
351                 collateralOfferId,
352                 _msgSender()
353             );
354         }
355     }
356 }
357
```

5.8. Use of Upgradable Contract Design

ID	IDX-008
Target	GoldFeverAuction GoldFeverBuildingManager GoldFeverCharacter GoldFeverCollateral GoldFeverGovernor GoldFeverGuild GoldFeverItem GoldFeverItemTier GoldFeverLeasing GoldFeverMarket GoldFeverMask GoldFeverMaskBox GoldFeverMerchantRight GoldFeverMiningClaim GoldFeverNativeGold GoldFeverNpc GoldFeverRight
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The logic of affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions e.g., stealing the users' funds anytime they want. **Likelihood: Medium** This action can be performed by the proxy owner without any restriction.
Status	Resolved * The Gold Fever team has mitigated this issue by implementing a TimeLock contract as the owner of all contracts to prevent immediate changes or upgrades to the contract. However, the timelock mechanism was not in use at the time of the reassessment. Therefore, Inspex suggests that platform users confirm the use of the timelock mechanism and privilege role address before using the platform.

5.8.1. Description

Smart contracts are designed to be used as agreements that cannot be changed forever. When a smart contract is upgraded, the agreement can be changed from what was previously agreed upon.

As these smart contracts are upgradable, the logic of them can be modified by the owner anytime, making the smart contracts untrustworthy.

5.8.2. Remediation

Inspex suggests deploying the contracts without the proxy pattern or any solution that can make the smart contracts upgradeable.

However, if upgradability is needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the changes e.g., 24 hours on the proxy admin role. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.9. Centralized Control of State Variable

ID	IDX-009
Target	GoldFeverAuction GoldFeverBuildingManager GoldFeverCharacter GoldFeverCollateral GoldFeverGovernor GoldFeverGuild GoldFeverGuildRight GoldFeverItem GoldFeverItemTier GoldFeverLeasing GoldFeverMarket GoldFeverMask GoldFeverMaskBox GoldFeverMerchantRight GoldFeverMiningClaim GoldFeverNpc GoldFeverPaymaster GoldFeverRight GoldFeverTaxation
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood: Medium** There is nothing to restrict the changes from being done; however, this action can only be done by the privileged roles.
Status	**Resolved *** The Gold Fever team has mitigated this issue by implementing a TimeLock contract as the owner of all contracts for the `DEFAULT_ADMIN_ROLE` role to prevent immediate changes to the contract. However, the timelock mechanism was not in use at the time of the reassessment. Therefore, Inspex suggests that platform users confirm the use of the timelock mechanism and privilege role address before using the platform.

5.9.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Role
GoldFeverAuction.sol (L:77)	GoldFeverAuction	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverAuction.sol (L:84)	GoldFeverAuction	setMerchantContract()	DEFAULT_ADMIN_ROLE
GoldFeverBuildingManager.sol (L:45)	GoldFeverBuildingManager	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverBuildingManager.sol (L:52)	GoldFeverBuildingManager	setItemTypeContract()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:103)	GoldFeverCharacter	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:110)	GoldFeverCharacter	setNpcContract()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:117)	GoldFeverCharacter	setGuildContract()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:197)	GoldFeverCharacter	setAttachSlotLimit()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:205)	GoldFeverCharacter	setCommissionRate()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:577)	GoldFeverCharacter	setCreateCharacterFee()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:592)	GoldFeverCharacter	updateCharacterSetIds()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:644)	GoldFeverCharacter	setCharacterSetFee()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol	GoldFeverCharacter	setLimitPerType()	DEFAULT_ADMIN_ROLE

(L:656)			
GoldFeverCharacter.sol (L:680)	GoldFeverCharacter	updateTribalMaleIds()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:688)	GoldFeverCharacter	updateTribalFemaleIds()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:696)	GoldFeverCharacter	updateAdventurerMaleIds()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:704)	GoldFeverCharacter	updateAdventurerFemaleIds()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:748)	GoldFeverCharacter	addFreeCharacterForAddress()	DEFAULT_ADMIN_ROLE
GoldFeverCollateral.sol (L:98)	GoldFeverCollateral	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverCollateral.sol (L:106)	GoldFeverCollateral	setTaxationContract()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:201)	GoldFeverGovernor	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:208)	GoldFeverGovernor	setNpcContract()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:215)	GoldFeverGovernor	setTaxCollector()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:222)	GoldFeverGovernor	setCompanyDefault()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:241)	GoldFeverGovernor	setGovernorDefault()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:260)	GoldFeverGovernor	createGovernorTax()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:381)	GoldFeverGovernor	updateMaintenancePeriod()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:484)	GoldFeverGovernor	maintainGovernorZone()	DEFAULT_ADMIN_ROLE
GoldFeverGuild.sol (L:44)	GoldFeverGuild	setForwarderContract()	DEFAULT_ADMIN_ROLE

GoldFeverGuild.sol (L:964)	GoldFeverGuild	setCharacterContract()	DEFAULT_ADMIN_ROLE
GoldFeverGuild.sol (L:971)	GoldFeverGuild	setGuildRightContract()	DEFAULT_ADMIN_ROLE
GoldFeverGuildRight.sol (L:67)	GoldFeverGuildRight	setGuildLimitation()	DEFAULT_ADMIN_ROLE
GoldFeverItem.sol (L:53)	GoldFeverItem	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverItem.sol (L:252)	GoldFeverItem	mint()	DEFAULT_ADMIN_ROLE
GoldFeverItemTier.sol (L:56)	GoldFeverItemTier	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverItemTier.sol (L:82)	GoldFeverItemTier	addTier()	DEFAULT_ADMIN_ROLE
GoldFeverItemTier.sol (L:96)	GoldFeverItemTier	setItemTypeContract()	DEFAULT_ADMIN_ROLE
GoldFeverItemTier.sol (L:103)	GoldFeverItemTier	setItemTier()	DEFAULT_ADMIN_ROLE
GoldFeverItemTier.sol (L:139)	GoldFeverItemTier	setTierAttribute()	DEFAULT_ADMIN_ROLE
GoldFeverLeasing.sol (L:79)	GoldFeverLeasing	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverLeasing.sol (L:86)	GoldFeverLeasing	setMerchantContract()	DEFAULT_ADMIN_ROLE
GoldFeverLeasing.sol (L:93)	GoldFeverLeasing	setTaxationContract()	DEFAULT_ADMIN_ROLE
GoldFeverMarket.sol (L:69)	GoldFeverMarket	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverMarket.sol (L:76)	GoldFeverMarket	setMerchantContract()	DEFAULT_ADMIN_ROLE
GoldFeverMarket.sol (L:83)	GoldFeverMarket	setTaxationContract()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:61)	GoldFeverMask	setForwarderContract()	DEFAULT_ADMIN_ROLE

GoldFeverMask.sol (L:398)	GoldFeverMask	updateForgeMaskFee()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:407)	GoldFeverMask	updateUnforgeMaskFee()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:416)	GoldFeverMask	updatePurchaseMaskCost()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:428)	GoldFeverMask	withdrawCollectedFee()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:469)	GoldFeverMask	setCommissionRate()	DEFAULT_ADMIN_ROLE
GoldFeverMaskBox.sol (L:38)	GoldFeverMaskBox	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverMerchantRight.sol (L:67)	GoldFeverMerchantRight	setItemTypeContract()	DEFAULT_ADMIN_ROLE
GoldFeverMerchantRight.sol (L:265)	GoldFeverMerchantRight	updateCompanionLimit()	DEFAULT_ADMIN_ROLE
GoldFeverMerchantRight.sol (L:294)	GoldFeverMerchantRight	updateBoatAndPlaneLimit()	DEFAULT_ADMIN_ROLE
GoldFeverMerchantRight.sol (L:322)	GoldFeverMerchantRight	updateBuildingLimit()	DEFAULT_ADMIN_ROLE
GoldFeverMerchantRight.sol (L:353)	GoldFeverMerchantRight	updateMerchantLimit()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:58)	GoldFeverMiningClaim	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:161)	GoldFeverMiningClaim	createMiningClaim()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:208)	GoldFeverMiningClaim	addArenaHour()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:216)	GoldFeverMiningClaim	setArenaHour()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:228)	GoldFeverMiningClaim	setArenaHourPrice()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:265)	GoldFeverMiningClaim	setMiningSpeed()	DEFAULT_ADMIN_ROLE

GoldFeverMiningClaim.sol (L:281)	GoldFeverMiningClaim	setMaxMiners()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:150)	GoldFeverNpc	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:157)	GoldFeverNpc	setMerchantContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:164)	GoldFeverNpc	setItemTierContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:171)	GoldFeverNpc	setBuildingManagerContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:178)	GoldFeverNpc	setGovernorContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:185)	GoldFeverNpc	setNpcHiringFee()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:193)	GoldFeverNpc	setTierStatus()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:755)	GoldFeverNpc	setRentPercentageLimit()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:762)	GoldFeverNpc	setTicketPercentageLimit()	DEFAULT_ADMIN_ROLE
GoldFeverPaymaster.sol (L:29)	GoldFeverPaymaster	addWhitelist()	onlyOwner
GoldFeverRight.sol (L:86)	GoldFeverRight	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverRight.sol (L:112)	GoldFeverRight	createRightOption()	DEFAULT_ADMIN_ROLE
GoldFeverRight.sol (L:151)	GoldFeverRight	purchaseRight()	DEFAULT_ADMIN_ROLE
GoldFeverRight.sol (L:112)	GoldFeverSwap	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverTaxation.sol (L:21)	GoldFeverTaxation	setTaxCollector()	DEFAULT_ADMIN_ROLE
GoldFeverTaxation.sol (L:28)	GoldFeverTaxation	setCompanyTax()	DEFAULT_ADMIN_ROLE

5.9.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests implementing a community-run smart contract governance to control the use of these functions.

If removing the functions or implementing the smart contract governance is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time at least 24 hours.

5.10. Improper Design for Operator Privilege

ID	IDX-010
Target	GoldFeverCharacter GoldFeverGovernor GoldFeverGuild GoldFeverItemTier GoldFeverMask GoldFeverMaskBox GoldFeverMiningClaim GoldFeverNpc GoldFeverRight
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The permission to change the critical state variables is controlled by the `DEFAULT_ADMIN_ROLE` role, which is executed by the off-chain server. The private key holder of the owner address can arbitrarily execute any function that the role has right, which can be unfair to the users. This causes monetary damage to the users' account holders and reputational damage to the platform. **Likelihood: Medium** Only the `DEFAULT_ADMIN_ROLE` role can change the critical state variables, but there is no restriction to prevent this action from being done excessively.
Status	**Resolved *** The Gold Fever team has clarified that the private key of privileged roles will be stored in the AWS Systems Manager (SSM) Parameter and encrypted with the AWS Key Management Service (KMS). Only the backend has permission to decrypt and obtain the private key. Additionally, new roles are being added to manage each function.

5.10.1. Description

In the `DEFAULT_ADMIN_ROLE` role, can change the critical state variables, to gain profit and monetary damage to the users.The privileged function has a very high impact, for example:

The `DEFAULT_ADMIN_ROLE`, which can transfer supply tokens to any address, has control over the entire arena's token supply.

GoldFeverMiningClaim.sol.sol

```
455 function bankWithdraw(
```

```
456     uint256 arenaId,
457     address minerAddress,
458     uint256 nglWithdrawAmount,
459     address govAddress,
460     uint256 govTax,
461     address guildOwnerAddress,
462     uint256 guildFee
463 ) public nonReentrant only(DEFAULT_ADMIN_ROLE) {
464     uint256 miningClaimId = idToArena[arenaId].miningClaimId;
465     uint256 maxNglAmountCanWithdraw = idToArena[arenaId].duration *
466         idToMiningClaim[miningClaimId].maxMiners *
467         miningSpeed;
468     require(
469         nglWithdrawAmount > 0,
470         "GoldFeverMiningClaim: Amount must be greater than 0"
471     );
472     require(
473         nglWithdrawAmount <= maxNglAmountCanWithdraw,
474         "GoldFeverMiningClaim: Amount must be less than or equal to max amount"
475     );
476     require(
477         nglWithdrawAmount <= idToMiningClaim[miningClaimId].nglAmount,
478         "GoldFeverMiningClaim: Amount must be less than or equal to total
    amount"
479     );
480
481     uint256 nglAmountAfterTax = nglWithdrawAmount -
482         (nglWithdrawAmount * govTax) /
483         10**uint256(feeDecimals()) /
484         100;
485     uint256 ownerEarn = (nglAmountAfterTax *
486         idToArena[arenaId].commissionRate) /
487         10**uint256(feeDecimals()) /
488         100;
489     ngl.transfer(
490         govAddress,
491         (nglWithdrawAmount * govTax) / 10**uint256(feeDecimals()) / 100
492     );
493     ngl.transfer(idToArena[arenaId].owner, ownerEarn);
494     uint256 minerEarnBeforeFee = nglAmountAfterTax - ownerEarn;
495     uint256 guildEarn = (minerEarnBeforeFee * guildFee) /
496         10**uint256(feeDecimals()) /
497         100;
498     ngl.transfer(guildOwnerAddress, guildEarn);
499     ngl.transfer(minerAddress, minerEarnBeforeFee - guildEarn);
500
501     idToMiningClaim[miningClaimId].nglAmount -= nglWithdrawAmount;
```

```
502         emit MinerWithdrawn(
503             arenaId,
504             miningClaimId,
505             minerAddress,
506             nglWithdrawAmount
507         );
508     }
```

From the current design, the `DEFAULT_ADMIN_ROLE` is a wallet address with the private key controlled by the Gold Fever off-chain server. Therefore, the holder of the owner's private key can freely mint the funds from the contracts. If the mechanism to mint tokens from the off-chain server is working incorrectly or can be controlled by a malicious actor, the minting in the contracts will be incorrect too.

The controllable privileged state update functions are as follows:

File	Contract	Function	ROLE
GoldFeverCharacter.sol (L:477)	GoldFeverCharacter	approveItemDetachment()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:260)	GoldFeverGovernor	createGovernorTax()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:298)	GoldFeverGovernor	updateBuildingGovernor()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:308)	GoldFeverGovernor	createZoneUpgrade()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:343)	GoldFeverGovernor	swapZoneUpgradeTier()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:372)	GoldFeverGovernor	updateZoneUpgrade()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:381)	GoldFeverGovernor	updateMaintenancePeriod()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:390)	GoldFeverGovernor	updateZoneMaintenance()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:484)	GoldFeverGovernor	maintainGovernorZone()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:617)	GoldFeverGovernor	withdrawCompanyEarning()	DEFAULT_ADMIN_ROLE
GoldFeverGuild.sol	GoldFeverGuild	allocateDonationItemTo	DEFAULT_ADMIN_ROLE

(L:608)		User()	
GoldFeverGuild.sol (L:617)	GoldFeverGuild	takeBackDonationItemFromUser()	DEFAULT_ADMIN_ROLE
GoldFeverGuild.sol (L:725)	GoldFeverGuild	approveRequestDonationItem()	DEFAULT_ADMIN_ROLE
GoldFeverItemTier.sol (L:177)	GoldFeverItemTier	collectFee()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:428)	GoldFeverMask	withdrawCollectedFee()	DEFAULT_ADMIN_ROLE
GoldFeverMaskBox.sol (L:89)	GoldFeverMaskBox	createBoxes()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:342)	GoldFeverMiningClaim	closeArena()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:405)	GoldFeverMiningClaim	enterArena()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:441)	GoldFeverMiningClaim	leaveArena()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:455)	GoldFeverMiningClaim	bankWithdraw()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:514)	GoldFeverMiningClaim	withdrawNglFromSellingHour()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:323)	GoldFeverNpc	renewHiring()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:410)	GoldFeverNpc	approveWithdrawal()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:466)	GoldFeverNpc	approveWithdrawals()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:878)	GoldFeverNpc	withdrawCompanyEarning()	DEFAULT_ADMIN_ROLE
GoldFeverRight.sol (L:151)	GoldFeverRight	purchaseRight()	DEFAULT_ADMIN_ROLE

5.10.2. Remediation

Inspex suggests implementing a consensus mechanism to verify that the transaction is approved by the

majority of trusted validators before change the critical state variable to prevent unverified actions from being done.

To prevent the delay in implementing the time lock mechanism with the `DEFAULT_ADMIN_ROLE` role we suggest changing the `DEFAULT_ADMIN_ROLE` role to another role such as `OPERATOR_ROLE`.

For example, this can be done by storing a list of trusted validators on the smart contract. The trusted validators then sign each transaction with a signature of their own, and submit them to the smart contract via the operator. Those signatures can then be verified on the smart contract using `ecrecover()` function, comparing them with the list of trusted validators. Or setting a multisig wallet for the `OPERATOR_ROLE` address and requiring that at least half of the trusted validators be validated.

5.11. Improper Free Character Validation in createCharacter() Function

ID	IDX-011
Target	GoldFeverCharacter
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: High** **Impact: Medium** The attacker can mint the free character as an NFT, which conflicts with the business design. **Likelihood: High** It is very likely that this issue could be used by the attacker since there is no restriction in the `createCharacter()` function.
Status	**Resolved** The Gold Fever team has resolved this issue by implementing the validation for `idToCreateCharacterFee` and `isFreeCharacter` state in `createCharacter()` function as we suggested in commit `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.11.1. Description

The `GoldFeverCharacter` contract allows users to create a character by executing the `createCharacter()` function. The users can choose what type of character they want to create: a free character or a non-free character.

The difference between the free character and the non-free character is that for the non-free character, the contract will mint the NFT for the owner, but for the free character, the contract will create the state instead of minting the NFT, and the free character cannot join the guild.

In lines 319–320, the different fees between the free character and the non–free character are specified by the `typeId` value and the `characterSetId` value. The attacker can arbitrarily supply the `isFreeCharacter` value in line 277, which determines whether a character is free or non-free type in line 348. Then provide a free `typeId` value and a free `characterSetId` value while having a false `isFreeCharacter` value for a non-free character. As a result, attackers could therefore use the free character's fee to mint the NFT. The minted NFT from this method can be used as a non-free character, such as to join a guild.

GoldFeverCharacter.sol

```
275 function createCharacter(
276     string memory name,
277     bool isFreeCharacter,
```

```
278         uint256 typeId,
279         uint256 characterSetId
280     ) public nonReentrant {
281         require(
282             usedNames[keccak256(abi.encodePacked(name))] == false,
283             "GoldFeverCharacter: Character name has been used"
284         );
285         require(
286             checkOneOf(typeId, tribalFemaleIds) ||
287                 checkOneOf(typeId, tribalMaleIds) ||
288                 checkOneOf(typeId, adventurerFemaleIds) ||
289                 checkOneOf(typeId, adventurerMaleIds),
290             "GoldFeverCharacter: typeId is not correct"
291         );
292         require(
293             checkOneOf(characterSetId, characterSetIds),
294             "GoldFeverCharacter: setId is not correct"
295         );
296 
297         if (
298             addressToTypeIdToSetIdToCharacterId[_msgSender()][typeId][
299                 characterSetId
300             ] != 0
301         ) {
302             require(
303                 !idToCharacter[
304                     addressToTypeIdToSetIdToCharacterId[_msgSender()][typeId][
305                         characterSetId
306                     ]
307                 ].isAlive,
308                 "GoldFeverCharacter: character body and class has been used"
309             );
310         }
311 
312         usedNames[keccak256(abi.encodePacked(name))] = true;
313         _characterIds.increment();
314         uint256 counter = _characterIds.current();
315         uint256 characterId = IGoldFeverItemType(itemTypeContract).createItemId(
316             counter,
317             typeId
318         );
319         uint256 burnSetFee = characterSetFee[characterSetId];
320         uint256 burnCharacterFee = createCharacterFee[typeId];
321         if (burnSetFee != 0) {
322             uint256 characterSetFeeAdminEarn = (burnSetFee * commissionRate) /
323                 100;
324             ngl.transferFrom(
```

```
325             _msgSender(),
326             address(this),
327             characterSetFeeAdminEarn
328         );
329         ngl.burnFrom(_msgSender(), burnSetFee - characterSetFeeAdminEarn);
330         nglFromCollectedFee += characterSetFeeAdminEarn;
331     }
332
333     if (burnCharacterFee != 0) {
334         uint256 createCharacterAdminEarn = (burnCharacterFee *
335             commissionRate) / 100;
336         ngl.transferFrom(
337             _msgSender(),
338             address(this),
339             createCharacterAdminEarn
340         );
341         ngl.burnFrom(
342             _msgSender(),
343             burnCharacterFee - createCharacterAdminEarn
344         );
345         nglFromCollectedFee += createCharacterAdminEarn;
346     }
347
348     if (!isFreeCharacter) {
349         _mint(_msgSender(), characterId);
350     } else {
351         require(
352             addressToFreeCharacterTypeIds[_msgSender()].length <
353                 addressToLimitFreeCharacter[_msgSender()] + 1,
354             "GoldFeverCharacter: Out of free characters"
355         );
356
357         (
358             bool isTribalMale,
359             bool isTribalFemale,
360             bool isAdventurerMale,
361             bool isAdventurerFemale
362         ) = checkTypeCharacter(typeId);
363         if (isTribalMale) {
364             require(
365                 checkTypeIdCanCreateCharacter(_msgSender(), tribalMaleIds),
366                 "GoldFeverCharacter: Out of tribal male free characters"
367             );
368         }
369         if (isTribalFemale) {
370             require(
371                 checkTypeIdCanCreateCharacter(
```

```
372                     _msgSender(),
373                     tribalFemaleIds
374                 ),
375                 "GoldFeverCharacter: Out of tribal female free characters"
376             );
377         }
378         if (isAdventurerMale) {
379             require(
380                 checkTypeIdCanCreateCharacter(
381                     _msgSender(),
382                     adventurerMaleIds
383                 ),
384                 "GoldFeverCharacter: Out of adventurer male free characters"
385             );
386         }
387         if (isAdventurerFemale) {
388             require(
389                 checkTypeIdCanCreateCharacter(
390                     _msgSender(),
391                     adventurerFemaleIds
392                 ),
393                 "GoldFeverCharacter: Out of adventurer female free characters"
394             );
395         }
396 
397         addressToFreeCharacterTypeIds[_msgSender()].push(typeId);
398     }
399     addressToTypeIdToSetIdToCharacterId[_msgSender()][typeId][
400         characterSetId
401     ] = characterId;
402 
403     idToCharacter[characterId] = Character(
404         characterId,
405         name,
406         isFreeCharacter,
407         0,
408         _msgSender(),
409         characterSetId,
410         true
411     );
412     emit CharacterCreated(
413         characterId,
414         name,
415         _msgSender(),
416         isFreeCharacter,
417         characterSetId
418     );
```

```
419 }
```

5.11.2. Remediation

Inspex suggests implementing the `idToCreateCharacterFee` and `idTocharacterSetFee` mappings to store the boolean states that indicate the free and non-free character for each `typeId` and `setId`, then using it to validate in the `createCharacter()` function. For example:

First, add the `idToCreateCharacterFee` mapping in line 577 and store the `isFreeCharacter_` value in the `idToCreateCharacterFee[typeId_]` state in line 590.

GoldFeverCharacter.sol

```
577 mapping(uint256 => bool) public idToCreateCharacterFee;
578
579 function setCreateCharacterFee(uint256 typeId_, uint256 fee_, bool
    isFreeCharacter_)
580     public
581     only(DEFAULT_ADMIN_ROLE)
582 {
583     require(
584         checkOneOf(typeId_, tribalFemaleIds) ||
585             checkOneOf(typeId_, tribalMaleIds) ||
586             checkOneOf(typeId_, adventurerFemaleIds) ||
587             checkOneOf(typeId_, adventurerMaleIds),
588         "GoldFeverCharacter: typeId is not correct"
589     );
590     idToCreateCharacterFee[typeId_] = isFreeCharacter_;
591     createCharacterFee[typeId_] = fee_ * (10**decimals);
592     emit CharacterPriceUpdated(typeId_, fee_ * (10**decimals));
593 }
```

Second, add the `idTocharacterSetFee` mapping in line 644 and store the `isFreeCharacter_` value in the `idTocharacterSetFee[setId_]` state in line 654.

GoldFeverCharacter.sol

```
644 mapping(uint256 => bool) public idTocharacterSetFee;
645
646 function setCharacterSetFee(uint256 setId_, uint256 fee_, bool
    isFreeCharacter_)
647     public
648     only(DEFAULT_ADMIN_ROLE)
649 {
650     require(
651         checkOneOf(setId_, characterSetIds),
652         "GoldFeverCharacter: setId is not correct"
653     );
```

```
654     idTocharacterSetFee[setId_] = isFreeCharacter_;
655     characterSetFee[setId_] = fee_ * (10**decimals);
656     emit SetPriceUpdated(setId_, fee_ * (10**decimals));
657 }
```

Lastly, add the validation for the `isFreeCharacter` argument to validate the free or non-free type character in lines 281-285.

GoldFeverCharacter.sol

```
275 function createCharacter(
276     string memory name,
277     bool isFreeCharacter,
278     uint256 typeId,
279     uint256 characterSetId
280 ) public nonReentrant {
281     require(
282         idToCreateCharacterFee[typeId] == isFreeCharacter &&
283         idTocharacterSetFee[characterSetId] == isFreeCharacter,
284         "GoldFeverCharacter: isFreeCharacter is not correct"
285     );
286     require(
287         usedNames[keccak256(abi.encodePacked(name))] == false,
288         "GoldFeverCharacter: Character name has been used"
289     );
290     require(
291         checkOneOf(typeId, tribalFemaleIds) ||
292             checkOneOf(typeId, tribalMaleIds) ||
293             checkOneOf(typeId, adventurerFemaleIds) ||
294             checkOneOf(typeId, adventurerMaleIds),
295         "GoldFeverCharacter: typeId is not correct"
296     );
297     require(
298         checkOneOf(characterSetId, characterSetIds),
299         "GoldFeverCharacter: setId is not correct"
300     );
301
302     if (
303         addressToTypeIdToSetIdToCharacterId[_msgSender()][typeId][
304             characterSetId
305         ] != 0
306     ) {
307         require(
308             !idToCharacter[
309                 addressToTypeIdToSetIdToCharacterId[_msgSender()][typeId][
310                     characterSetId
311                 ]
312             ].isAlive,
```

```
313             "GoldFeverCharacter: character body and class has been used"
314         );
315     }
316
317     usedNames[keccak256(abi.encodePacked(name))] = true;
318     _characterIds.increment();
319     uint256 counter = _characterIds.current();
320     uint256 characterId = IGoldFeverItemType(itemTypeContract).createItemId(
321         counter,
322         typeId
323     );
324     uint256 burnSetFee = characterSetFee[characterSetId];
325     uint256 burnCharacterFee = createCharacterFee[typeId];
326     if (burnSetFee != 0) {
327         uint256 characterSetFeeAdminEarn = (burnSetFee * commissionRate) /
328             100;
329         ngl.transferFrom(
330             _msgSender(),
331             address(this),
332             characterSetFeeAdminEarn
333         );
334         ngl.burnFrom(_msgSender(), burnSetFee - characterSetFeeAdminEarn);
335         nglFromCollectedFee += characterSetFeeAdminEarn;
336     }
337
338     if (burnCharacterFee != 0) {
339         uint256 createCharacterAdminEarn = (burnCharacterFee *
340             commissionRate) / 100;
341         ngl.transferFrom(
342             _msgSender(),
343             address(this),
344             createCharacterAdminEarn
345         );
346         ngl.burnFrom(
347             _msgSender(),
348             burnCharacterFee - createCharacterAdminEarn
349         );
350         nglFromCollectedFee += createCharacterAdminEarn;
351     }
352
353     if (!isFreeCharacter) {
354         _mint(_msgSender(), characterId);
355     } else {
356         require(
357             addressToFreeCharacterTypeIds[_msgSender()].length <
358                 addressToLimitFreeCharacter[_msgSender()] + 1,
359             "GoldFeverCharacter: Out of free characters"
```

```
360         );
361
362         (
363             bool isTribalMale,
364             bool isTribalFemale,
365             bool isAdventurerMale,
366             bool isAdventurerFemale
367         ) = checkTypeCharacter(typeId);
368         if (isTribalMale) {
369             require(
370                 checkTypeIdCanCreateCharacter(_msgSender(), tribalMaleIds),
371                 "GoldFeverCharacter: Out of tribal male free characters"
372             );
373         }
374         if (isTribalFemale) {
375             require(
376                 checkTypeIdCanCreateCharacter(
377                     _msgSender(),
378                     tribalFemaleIds
379                 ),
380                 "GoldFeverCharacter: Out of tribal female free characters"
381             );
382         }
383         if (isAdventurerMale) {
384             require(
385                 checkTypeIdCanCreateCharacter(
386                     _msgSender(),
387                     adventurerMaleIds
388                 ),
389                 "GoldFeverCharacter: Out of adventurer male free characters"
390             );
391         }
392         if (isAdventurerFemale) {
393             require(
394                 checkTypeIdCanCreateCharacter(
395                     _msgSender(),
396                     adventurerFemaleIds
397                 ),
398                 "GoldFeverCharacter: Out of adventurer female free characters"
399             );
400         }
401
402         addressToFreeCharacterTypeIds[_msgSender()].push(typeId);
403     }
404     addressToTypeIdToSetIdToCharacterId[_msgSender()][typeId][
405         characterSetId
406     ] = characterId;
```

```
407
408     idToCharacter[characterId] = Character(
409         characterId,
410         name,
411         isFreeCharacter,
412         0,
413         _msgSender(),
414         characterSetId,
415         true
416     );
417     emit CharacterCreated(
418         characterId,
419         name,
420         _msgSender(),
421         isFreeCharacter,
422         characterSetId
423     );
424 }
425
```

5.12. Missing Item Transfer of Disbanded Guild

ID	IDX-012
Target	GoldFeverGuild
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High When the guild is disbanded, the item from the investment process will be stuck in the contract. **Likelihood:** Low There is no function that can withdraw the item from the contract, only the guild owner can execute the `disbandGuild()` function, and the owner will not retrieve any item for this action.
Status	Resolved The Gold Fever team has resolved this issue by implementing the `withdrawAfterDisbandGuild()` function to withdraw the stuck item when the guild is disbanded, as we suggested in the commit `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.12.1. Description

When the seller executes the `sellItemToInvestment()` function, the investment is successful, and the item will belong to the guild as long as the guild is not disbanded.

GoldFeverGuild.sol

```
924 function sellItemToInvestment(
925     uint256 guildId,
926     uint256 investmentId,
927     uint256 itemId,
928     address nftContract
929 ) public nonReentrant {
930     Investment memory investment = guildIdToInvestmentIdToInvestmentItem[
931         guildId
932     ][investmentId];
933     require(
934         (IGoldFeverItemType(itemTypeContract).getItemType(itemId)) ==
935             investment.itemTypeId,
936         "GoldFeverGuild: item type is not match"
937     );
```

```
938     require(
939         investment.expiredTimeListing > block.timestamp &&
940             investment.nglAmount == investment.nglFund &&
941             investment.status == STATUS_CREATED,
942         "GoldFeverGuild: Investment is not avaible"
943     );
944 
945     IERC721(nftContract).safeTransferFrom(
946         _msgSender(),
947         address(this),
948         itemId
949     );
950     ngl.transfer(_msgSender(), investment.nglAmount);
951     investment.status = STATUS_SUCCESS;
952     guildIdToInvestmentIdToInvestmentItem[guildId][
953         investmentId
954     ] = investment;
955 
956     investmentIdToItem[investmentId] = InvestmentItem(
957         investmentId,
958         itemId,
959         nftContract
960     );
961     emit ItemToInvestmentSold(guildId, investmentId, itemId, nftContract);
962 }
```

But when the guild is disbanded by executing the `disbandGuild()` function, the item will be stuck in the contract, which means the owner cannot retrieve the guild item since there is no function that has the ability to transfer the guild item.

GoldFeverGuild.sol

```
335 function disbandGuild(uint256 guildId) public nonReentrant {
336     require(
337         _msgSender() == idToGuild[guildId].owner,
338         "GoldFeverGuild: not owner"
339     );
340     require(
341         idToGuild[guildId].status == STATUS_CREATED,
342         "GoldFeverGuild: guild has been disband"
343     );
344     idToGuild[guildId].status = STATUS_DISBAND;
345     if (guildIdToCollectFee[guildId] > 0) {
346         ngl.transfer(_msgSender(), guildIdToCollectFee[guildId]);
347     }
348     guildIdToCollectFee[guildId] = 0;
349     delete ownerToGuildId[_msgSender()];
```

```
350         emit GuildDisbanded(guildId);
351     }
```

5.12.2. Remediation

Inspex suggests implementing an item transfer function to transfer the item that belongs to the guild out of the contract after the guild was disbanded, for example:

GoldFeverGuild.sol

```
1   function withdrawAfterDisbandGuild(uint256 guildId, uint256 investmentId)
    external nonReentrant {
2       require(idToGuild[guildId].status == STATUS_DISBAND,
3           "GoldFeverGuild: guild is not disbanded"
4       );
5       require(
6           _msgSender() == idToGuild[guildId].owner,
7           "GoldFeverGuild: not owner guild"
8       );
9       require(guildIdToInvestmentIdToInvestmentItem[guildId][investmentId].status
    == STATUS_SUCCESS, "GoldFeverGuild: invalid investment");
10
11      IERC721(investmentIdToItem[investmentId].nftContract).safeTransferFrom(
12          address(this),
13          _msgSender(),
14          investmentIdToItem[investmentId].itemId
15      );
16  }
```

5.13. Improper requestDonateItem() Function Implementation

ID	IDX-013
Target	GoldFeverGuild
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High The users cannot withdraw their donation items after they leave the guild. **Likelihood:** Low This issue may occur when a user has donated an item to a guild but is no longer a member of that guild. However, the users can execute the `requestDonationItem()` function before they leave the guild.
Status	Resolved The Gold Fever team has resolved this issue by modifying the `requestDonationItem()` function to allow the users that are no longer members of the guild to withdraw their donation item. The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.13.1. Description

In the `GoldFeverGuild` contract, the `memberDonateItem()` function is used by guild members to donate an item to the guild. The item will be transferred to the contract at line 582.

GoldFeverGuild.sol

```
552 function memberDonateItem(
553     uint256 characterId,
554     uint256 itemId,
555     address nftContract,
556     uint256 duration
557 ) public override nonReentrant {
558     uint256 guildId = characterIdToMember[characterId].guildId;
559
560     require(
561         guildId != 0 &&
562             characterIdToMember[characterId].status == STATUS_JOINING,
563         "GoldFeverGuild: character not in guild"
564     );
565
566     require(
```

```
567             idToGuild[guildId].status == STATUS_CREATED,
568             "GoldFeverGuild: guild is not available"
569         );
570 
571         require(
572             checkOwnerRight(idToGuild[guildId].owner, idToGuild[guildId].limit),
573             "GoldFeverGuild: owner don't have right to owned guild"
574         );
575         (bool isKickstarter, , ) = IGuildRight(guildRightContract)
576             .checkGuildType(idToGuild[guildId].owner);
577         require(
578             isKickstarter,
579             "GoldFeverGuild: owner has not enabled feature donation"
580         );
581 
582         IERC721(nftContract).safeTransferFrom(
583             _msgSender(),
584             address(this),
585             itemId
586         );
587 
588         guildIdToItemIdToDonationItem[guildId][itemId] = DonationItem(
589             guildId,
590             characterId,
591             itemId,
592             nftContract,
593             0,
594             block.timestamp + duration,
595             STATUS_CREATED
596         );
597 
598         emit MemberItemDonated(
599             guildId,
600             characterId,
601             itemId,
602             nftContract,
603             0,
604             block.timestamp + duration
605         );
606     }
```

When the users leave the guild by executing `quitGuild()` function, the `characterIdToMember[characterId].status` state will be set to `STATUS_QUIT` at line 436.

GoldFeverGuild.sol

```
420 function quitGuild(uint256 characterId) public nonReentrant {
421     require(
```

```
422         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
423         "GoldFeverGuild: not owner character"
424     );
425     require(
426         characterIdToMember[characterId].guildId != 0 &&
427             characterIdToMember[characterId].status == STATUS_JOINING,
428         "GoldFeverGuild: character not in guild"
429     );
430
431     require(
432         !characterIdToMember[characterId].isOwner,
433         "GoldFeverGuild: owner can't quit"
434     );
435
436     characterIdToMember[characterId].status = STATUS_QUIT;
437     idToGuild[characterIdToMember[characterId].guildId].members -= 1;
438     emit GuildQuit(
439         characterIdToMember[characterId].guildId,
440         characterId,
441         idToGuild[characterIdToMember[characterId].guildId].members
442     );
443 }
```

If the users leave the guild before requesting to withdraw their donation item from the guild, calling the `requestDonationItem()` function later will be reverted due to conditional checking in lines 677-681. Thus, the users cannot withdraw their donation item after they leave the guild.

GoldFeverGuild.sol

```
667 function requestDonationItem(uint256 characterId, uint256 itemId)
668     public
669     nonReentrant
670 {
671     uint256 guildId = characterIdToMember[characterId].guildId;
672     require(
673         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
674         "GoldFeverGuild: not owner character"
675     );
676
677     require(
678         guildId != 0 &&
679             characterIdToMember[characterId].status == STATUS_JOINING,
680         "GoldFeverGuild: character not in guild"
681     );
682     if (
683         idToGuild[guildId].joinFee > 0 &&
684         !characterIdToMember[characterId].isOwner
```

```
685     ) {
686         require(
687             characterIdToMember[characterId].expired > block.timestamp,
688             "GoldFeverGuild: member should pay guild fee first"
689         );
690     }
691 
692     require(
693         guildIdToItemIdToDonationItem[guildId][itemId].status !=
694             STATUS_REQUESTED,
695         "GoldFeverGuild: item is not avaible"
696     );
697     require(
698         characterId ==
699             guildIdToItemIdToDonationItem[guildId][itemId].characterId,
700         "GoldFeverGuild: Only item owner can request withdraw"
701     );
702 
703     require(
704         guildIdToItemIdToDonationItem[guildId][itemId].expiredTime <
705             block.timestamp,
706         "GoldFeverGuild: Donation time is not over"
707     );
708     if (
709         guildIdToItemIdToDonationItem[guildId][itemId].status ==
710         STATUS_INUSED
711     ) {
712         guildIdToItemIdToDonationItem[guildId][itemId].expiredTime =
713             block.timestamp +
714             1 hours;
715     }
716 
717     guildIdToItemIdToDonationItem[guildId][itemId]
718         .status = STATUS_REQUESTED;
719     emit DonationItemRequested(
720         itemId,
721         guildIdToItemIdToDonationItem[guildId][itemId].expiredTime
722     );
723 }
```

Additionally, if the guild has been disbanded and the users have left the guild, their donation item will be permanently locked in the contract.

5.13.2. Remediation

Inspex suggests modifying the `requestDonationItem()` function to allow users that are no longer members of the guild to withdraw their donation item by adding the `guildId_` state as an input parameter and adding

the conditional check to validate whether users are still in the previous guild in lines 676-693. For example:

GoldFeverGuild.sol

```
667 function requestDonationItem(uint256 characterId, uint256 itemId, uint256
    guildId_)
        public
        nonReentrant
668 {
669     uint256 guildId = characterIdToMember[characterId].guildId;
670     require(
671         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
672         "GoldFeverGuild: not owner character"
673     );
674     if (guildId == guildId_ &&
675         characterIdToMember[characterId].status == STATUS_JOINING &&
676         idToGuild[guildId_].status == STATUS_CREATED){
677         require(
678             guildId != 0,
679             "GoldFeverGuild: character not in guild"
680         );
681         if (
682             idToGuild[guildId].joinFee > 0 &&
683             !characterIdToMember[characterId].isOwner
684         ) {
685             require(
686                 characterIdToMember[characterId].expired > block.timestamp,
687                 "GoldFeverGuild: member should pay guild fee first"
688             );
689         }
690     }
691     guildId = guildId_;
692     require(
693         guildIdToItemIdToDonationItem[guildId][itemId].status !=
694             STATUS_REQUESTED,
695         "GoldFeverGuild: item is not avaible"
696     );
697     require(
698         characterId ==
699             guildIdToItemIdToDonationItem[guildId][itemId].characterId,
700         "GoldFeverGuild: Only item owner can request withdraw"
701     );
702
703     require(
704         guildIdToItemIdToDonationItem[guildId][itemId].expiredTime <
705             block.timestamp,
706         "GoldFeverGuild: Donation time is not over"
707     );
```

```
708     if (
709         guildIdToItemIdToDonationItem[guildId][itemId].status ==
710         STATUS_INUSED
711     ) {
712         guildIdToItemIdToDonationItem[guildId][itemId].expiredTime =
713             block.timestamp +
714             1 hours;
715     }
716 
717     guildIdToItemIdToDonationItem[guildId][itemId]
718         .status = STATUS_REQUESTED;
719     emit DonationItemRequested(
720         itemId,
721         guildIdToItemIdToDonationItem[guildId][itemId].expiredTime
722     );
723 }
724 
725 
```

5.14. Missing Boundary State Variable in upgradeItem() Function

ID	IDX-014
Target	GoldFeverItemTier
Category	Advanced Smart Contract Vulnerability
CWE	CWE-755: Improper Handling of Exceptional Conditions
Risk	**Severity:** Medium **Impact:** Medium The users can upgrade items beyond the added tier with 0 cost. However, if the users upgrade the item that is beyond the item property will be as item tier `0` until the admin creates a new tier. **Likelihood:** Medium Only the users with the maximum tier of items could upgrade further the limit.
Status	Resolved The Gold Fever team has resolved this issue by validating the maximum item tier for the `addTier()` and `upgradeItem()` functions. The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.14.1. Description

In the `GoldFeverItemTier` contract, the `upgradeItem()` function is used for users to upgrade their current item tier to the next tier.

At line 133, the `itemTypeIdToTierUpgradePrice[itemTypeId][currentTier + 1]` is the amount of `$NGL` that users have to pay for upgrading their item tier.

GoldFeverItemTier.sol

```
120 function upgradeItem(uint256 itemId) public nonReentrant {
121     address msgSender = _msgSender();
122     require(
123         IERC721(nftContract).ownerOf(itemId) == msgSender,
124         "GoldFeverItemTier: Only item owner can upgrade"
125     );
126     uint256 currentTier = getItemTier(itemId);
127     uint256 itemTypeId = IGoldFeverItemType(itemTypeContract).getItemType(
128         itemId
129     );
130     ngl.transferFrom(
131         msgSender,
132         address(this),
133         itemTypeIdToTierUpgradePrice[itemTypeId][currentTier + 1]
```

```
134     );
135     _tier[itemId] = currentTier + 1
136     emit ItemTierUpgraded(itemId, _tier[itemId]);
137 }
```

The `itemTypeIdToTierUpgradePrice` state can only be set in the `addTier()` function at line 91. So, this state value is `0` by default.

GoldFeverItemTier.sol

```
82  function addTier(
83      uint256 itemTypeId,
84      uint256 tierId,
85      uint256 upgradePrice
86  ) public only(DEFAULT_ADMIN_ROLE) {
87      require(
88          tierId >= 2,
89          "GoldFeverItemTier: Tier must be greater than or equal to 2"
90      );
91      itemTypeIdToTierUpgradePrice[itemTypeId][tierId] = upgradePrice;
92
93      emit TierAdded(itemTypeId, tierId, upgradePrice);
94  }
```

Without any boundaries, the upgrade price of the item will be `0`. The item that has the maximum tier added can be upgraded to the next tier with `0` `$NGL` upgrade price.

However, if any users upgrade an item beyond the added tier, the item's property will be considered to be at item tier `0` until an administrator creates a new tier.

5.14.2. Remediation

Inspex suggests modifying the `addTier()` function and adding the conditional checking to the `upgradeItem()` function to guarantee the `itemTypeIdToTierUpgradePrice` state will always be within the maximum tier added. For example:

GoldFeverItemTier.sol

```
80  mapping(uint256 => mapping(uint256 => bool)) public enabledTier;
81  function addTier(
82      uint256 itemTypeId,
83      uint256 tierId,
84      uint256 upgradePrice
85  ) public only(DEFAULT_ADMIN_ROLE) {
86      require(
87          tierId >= 2,
88          "GoldFeverItemTier: Tier must be greater than or equal to 2"
89      );
```

```
90      itemTypeIdToTierUpgradePrice[itemTypeId][tierId] = upgradePrice;
91      enabledTier[itemTypeId][tierId] = true;
92
93      emit TierAdded(itemTypeId, tierId, upgradePrice);
94  }
```

GoldFeverItemTier.sol

```
120 function upgradeItem(uint256 itemId) public nonReentrant {
121     address msgSender = _msgSender();
122     require(
123         IERC721(nftContract).ownerOf(itemId) == msgSender,
124         "GoldFeverItemTier: Only item owner can upgrade"
125     );
126     uint256 currentTier = getItemTier(itemId);
127     uint256 itemTypeId = IGoldFeverItemType(itemTypeContract).getItemType(
128         itemId
129     );
130     require(enabledTier[itemTypeId][currentTier + 1], "GoldFeverNpc: maximum
    tier");
131     ngl.transferFrom(
132         msgSender,
133         address(this),
134         itemTypeIdToTierUpgradePrice[itemTypeId][currentTier + 1]
135     );
136     _tier[itemId] = currentTier + 1;
137     emit ItemTierUpgraded(itemId, _tier[itemId]);
138 }
```

5.15. Incorrect Expiry Time Calculation in renewHiring() Function

ID	IDX-015
Target	GoldFeverNpc
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The extended expiry time could still be less than the current time after the `renewHiring()` function has been executed, causing the hiring to still expire. **Likelihood:** Medium This issue only occurs if the admin executes the `renewHiring()` function too late after the hiring has expired, causing the extended expiry time to be less than it should be.
Status	Resolved The Gold Fever team has resolved this issue by correcting the expired time calculation in the `renewHiring()` function. The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.15.1. Description

The `GoldFeverNpc` contract allows users to create hiring for the game's building, which will expire after 30 days. After the hiring expires, the player should allow the `$NGL` for the admin to call the `renewHiring()` function to extend the hiring's expiry time.

The `renewHiring()` function will extend the hiring's expiry time from the expired time by 30 days, so if this function is executed after the hiring's expiry time has already passed for a month, the hiring's expiry time will still expire as shown in lines 337-339.

GoldFeverNpc.sol

```
323 function renewHiring(uint256 hiringId) public only(DEFAULT_ADMIN_ROLE) {
324     require(
325         tierToFee[idToHiring[hiringId].tier] > 0,
326         "GoldFeverNpc: don't need to renewal free tier"
327     );
328     require(
329         idToHiring[hiringId].expired < block.timestamp,
330         "GoldFeverNpc: has not expired"
331     );
332     uint256 hiringFee = tierToFee[idToHiring[hiringId].tier];
333     address owner = idToHiring[hiringId].buildingOwner;
```

```
334     uint256 allowance = ngl.allowance(owner, address(this));
335     if (allowance >= hiringFee && ngl.balanceOf(owner) >= hiringFee) {
336         payNpcHiringFee(owner, hiringFee);
337         idToHiring[hiringId].expired =
338             idToHiring[hiringId].expired +
339             30 days;
340         emit HiringRenewed(
341             hiringId,
342             idToHiring[hiringId].expired + 30 days
343         );
344     } else {
345         idToHiring[hiringId].tier = 1;
346         emit HiringDowngraded(hiringId, 1);
347     }
348 }
```

5.15.2. Remediation

Inspex suggests extending the hiring's expiry time from the `block.timestamp` rather than the previous expired time. For example in lines 337-339:

GoldFeverNpc.sol

```
323 function renewHiring(uint256 hiringId) public only(DEFAULT_ADMIN_ROLE) {
324     require(
325         tierToFee[idToHiring[hiringId].tier] > 0,
326         "GoldFeverNpc: don't need to renewal free tier"
327     );
328     require(
329         idToHiring[hiringId].expired < block.timestamp,
330         "GoldFeverNpc: has not expired"
331     );
332     uint256 hiringFee = tierToFee[idToHiring[hiringId].tier];
333     address owner = idToHiring[hiringId].buildingOwner;
334     uint256 allowance = ngl.allowance(owner, address(this));
335     if (allowance >= hiringFee && ngl.balanceOf(owner) >= hiringFee) {
336         payNpcHiringFee(owner, hiringFee);
337         idToHiring[hiringId].expired =
338             block.timestamp +
339             30 days;
340         emit HiringRenewed(
341             hiringId,
342             block.timestamp + 30 days
343         );
344     } else {
345         idToHiring[hiringId].tier = 1;
346         emit HiringDowngraded(hiringId, 1);
347     }
```

```
348 }
```

5.16. Division Before Multiplication

ID	IDX-016
Target	GoldFeverGovernor
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low The rounding error from the division before multiplication can cause the company's earnings to be slightly miscalculated. **Likelihood:** Medium It is likely that the company's earnings will result in a floating number that is rounded off from the division before applying the multiplication.
Status	Resolved The Gold Fever team has resolved this issue by reordering the operation order in the `calculateTaxById()` function. The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.16.1. Description

Solidity supports only integer values but not floating point values. The division of integers can result in a value with decimal points, which will be rounded off. This rounding error can cause the calculation to be different from what it should be, especially when that value is later multiplied with another value.

In the `calculateTaxById()` function, the `companyEarn` value in line 655 is calculated by multiplying the `totalEarn` and the value from `getCompanyTaxOnGovernor(governorTax)` function and dividing the result by `(103) / 100`. However, the `totalEarn` value is calculated by multiplying the amountExcludeTax with the governorTax and dividing the result by `(103) / 100`. It's resulting in the `totalEarn` value getting a rounding error caused by the division being amplified in the multiplication and can increase the amount of miscalculation.

GoldFeverGovernor.sol

```
630 function calculateTaxById(uint256 governorId, uint256 amountExcludeTax)
631     public
632     view
633     returns (
634         uint256 governorTax,
635         uint256 totalEarn,
636         uint256 companyEarn,
```

```
637             uint256 governorEarn,
638             bool isHiringCreated,
639             address governor
640         )
641     {
642         (, governor, isHiringCreated) = IGoldFeverNpc(npcContract)
643             .getHiringStatusByBuildingId(
644                 idToGovernor[governorId].governorBuildingId,
645                 idToGovernor[governorId].governorNftContract
646             );
647         if (isHiringCreated) {
648             uint256 maxGovernorTaxById = getMaxGovernorTaxById(governorId);
649             if (idToGovernor[governorId].governorTax < maxGovernorTaxById) {
650                 governorTax = idToGovernor[governorId].governorTax;
651             } else {
652                 governorTax = maxGovernorTaxById;
653             }
654             totalEarn = (amountExcludeTax * governorTax) / (10**3) / 100;
655             companyEarn =
656                 (totalEarn * getCompanyTaxOnGovernor(governorTax)) /
657                 (10**3) /
658                 100;
659             governorEarn = totalEarn - companyEarn;
660         } else {
661             governorTax = defaultTax;
662             companyEarn = (amountExcludeTax * defaultTax) / (10**3) / 100;
663             governorEarn = 0;
664             totalEarn = companyEarn;
665         }
666     }
```

5.16.2. Remediation

Inspex suggests modifying the affected lines of code to perform multiplication before division, for example:

GoldFeverGovernor.sol

```
630 function calculateTaxById(uint256 governorId, uint256 amountExcludeTax)
631     public
632     view
633     returns (
634         uint256 governorTax,
635         uint256 totalEarn,
636         uint256 companyEarn,
637         uint256 governorEarn,
638         bool isHiringCreated,
639         address governor
640     )
```

```
641 {
642     (, governor, isHiringCreated) = IGoldFeverNpc(npcContract)
643         .getHiringStatusByBuildingId(
644             idToGovernor[governorId].governorBuildingId,
645             idToGovernor[governorId].governorNftContract
646         );
647     if (isHiringCreated) {
648         uint256 maxGovernorTaxById = getMaxGovernorTaxById(governorId);
649         if (idToGovernor[governorId].governorTax < maxGovernorTaxById) {
650             governorTax = idToGovernor[governorId].governorTax;
651         } else {
652             governorTax = maxGovernorTaxById;
653         }
654         totalEarn = (amountExcludeTax * governorTax) / (10**3) / 100;
655         companyEarn =
656             ((amountExcludeTax * governorTax) *
    getCompanyTaxOnGovernor(governorTax)) /
657             (10**3) / 100 / (10**3) / 100;
658         governorEarn = totalEarn - companyEarn;
659     } else {
660         governorTax = defaultTax;
661         companyEarn = (amountExcludeTax * defaultTax) / (10**3) / 100;
662         governorEarn = 0;
663         totalEarn = companyEarn;
664     }
665 }
```

5.17. Lack of Guild Disband Validation

ID	IDX-017
Target	GoldFeverGuild
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The guild member still can be contributing to the investment while the guild has been disbanded, it results when the investment is successful the investor will lost the fund and the guild item can not use. By the way, when the investment is unsuccess the investor can withdraw the fund back. **Likelihood:** Low It's unlikely that the users will contribute to the guild that has been disbanded
Status	Resolved The Gold Fever team has resolved this issue by adding the validation for the disbanded guild in the affected function as we suggested in commit `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.17.1. Description

Users are able to contribute to the created investment through the `contributeInvestment()` function of the `GoldFeverGuild` contract. However, there is no validation to verify that the guild is not disbanded. As a result, users may misinterpret and contribute to the investment of the disbanded guild.

GoldFeverGuild.sol

```
816 function contributeInvestment(
817     uint256 investmentId,
818     uint256 characterId,
819     uint256 nglAmount
820 ) public nonReentrant {
821     require(
822         nglAmount > 0,
823         "GoldFeverGuild: nglAmount should be greater than 0"
824     );
825     require(
826         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
827         "GoldFeverGuild: not owner character"
828     );
829     uint256 guildId = characterIdToMember[characterId].guildId;
```

```
830         require(
831             checkOwnerRight(idToGuild[guildId].owner, idToGuild[guildId].limit),
832             "GoldFeverGuild: owner don't have right to owned guild"
833         );
834         (bool isKickstarter, , ) = IGuildRight(guildRightContract)
835             .checkGuildType(idToGuild[guildId].owner);
836         require(
837             isKickstarter,
838             "GoldFeverGuild: has not enabled feature donation"
839         );
840         require(
841             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
842                 .expiredTimeRaising >
843                 block.timestamp &&
844                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
845                     .nglFund <
846                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
847                     .nglAmount,
848             "GoldFeverGuild: investment has ended"
849         );
850 
851         uint256 depositAmount = guildIdToInvestmentIdToInvestmentItem[guildId][
852             investmentId
853         ].nglFund +
854             nglAmount >
855             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
856                 .nglAmount
857             ? guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
858                 .nglAmount -
859                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
860                     .nglFund
861             : nglAmount;
862         ngl.transferFrom(_msgSender(), address(this), depositAmount);
863         investmentIdToCharacterIdToAmount[investmentId][
864             characterId
865         ] += depositAmount;
866         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
867             .nglFund += depositAmount;
868         emit InvestmentContributed(
869             investmentId,
870             characterId,
871             investmentIdToCharacterIdToAmount[investmentId][characterId],
872             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId].nglFund
873         );
874 }
```

After that, if the investment goal is not reached in time, the contributors can execute the

`withdrawFailedInvest()` function to get their tokens back. Nevertheless, if the guild is already disbanded, contributors still have to wait until the duration ends. If the investment has a long duration, the tokens will lock in the contract for a long time.

GoldFeverGuild.sol

```
876 function withdrawFailedInvest(uint256 investmentId, uint256 characterId)
877     public
878     nonReentrant
879 {
880     require(
881         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
882         "GoldFeverGuild: not owner character"
883     );
884     uint256 guildId = characterIdToMember[characterId].guildId;
885     if (
886         idToGuild[guildId].joinFee > 0 &&
887         !characterIdToMember[characterId].isOwner
888     ) {
889         require(
890             characterIdToMember[characterId].expired > block.timestamp,
891             "GoldFeverGuild: member should pay guild fee first"
892         );
893     }
894     require(
895         investmentIdToCharacterIdToAmount[investmentId][characterId] > 0,
896         "GoldFeverGuild: has not invested"
897     );
898     require(
899         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
900             .expiredTimeRaising < block.timestamp,
901         "GoldFeverGuild: investment is running"
902     );
903
904     if (
905         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
906             .expiredTimeListing > block.timestamp
907     ) {
908         require(
909             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
910                 .nglFund <
911                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
912                     .nglAmount,
913             "GoldFeverGuild: investment is running"
914         );
915     }
916     ngl.transfer(
917         _msgSender(),
```

```
            investmentIdToCharacterIdToAmount[investmentId][characterId]
        );
        delete investmentIdToCharacterIdToAmount[investmentId][characterId];
        emit FailedInvestWithdrew(investmentId, characterId);
    }
```

When the investment goal is reached, the user can sell the item to the investment by using the `sellItemToInvestment()` function. However, just like the `contributeInvestment()` function, there is no validation for the disbanded guild, so if the guild is disbanded, the user can still sell the item to the disbanded guild's investment.

GoldFeverGuild.sol

```
function sellItemToInvestment(
    uint256 guildId,
    uint256 investmentId,
    uint256 itemId,
    address nftContract
) public nonReentrant {
    Investment memory investment = guildIdToInvestmentIdToInvestmentItem[
        guildId
    ][investmentId];
    require(
        (IGoldFeverItemType(itemTypeContract).getItemType(itemId)) ==
            investment.itemTypeId,
        "GoldFeverGuild: item type is not match"
    );
    require(
        investment.expiredTimeListing > block.timestamp &&
            investment.nglAmount == investment.nglFund &&
            investment.status == STATUS_CREATED,
        "GoldFeverGuild: Investment is not avaible"
    );

    IERC721(nftContract).safeTransferFrom(
        _msgSender(),
        address(this),
        itemId
    );
    ngl.transfer(_msgSender(), investment.nglAmount);
    investment.status = STATUS_SUCCESS;
    guildIdToInvestmentIdToInvestmentItem[guildId][
        investmentId
    ] = investment;

    investmentIdToItem[investmentId] = InvestmentItem(
        investmentId,
```

```
958             itemId,
959             nftContract
960         );
961         emit ItemToInvestmentSold(guildId, investmentId, itemId, nftContract);
962     }
```

The `allocateDonationItemToUser()` function, which can only be used by admins, does not check for disbanded guilds, so admins can allocate the donated item to a player after the guild has already disbanded. Even so, the item owner can execute the `requestDonationItem()` function to request the item back, and the expiration time of the donation item will be reduced to an hour.

GoldFeverGuild.sol

```
608 function allocateDonationItemToUser(uint256 characterId, uint256 itemId)
609     public
610     only(DEFAULT_ADMIN_ROLE)
611     nonReentrant
612 {
613     uint256 guildId = characterIdToMember[characterId].guildId;
614 
615     require(
616         guildId != 0 &&
617             characterIdToMember[characterId].status == STATUS_JOINING,
618         "GoldFeverGuild: character not in guild"
619     );
620 
621     if (
622         idToGuild[guildId].joinFee > 0 &&
623         !characterIdToMember[characterId].isOwner
624     ) {
625         require(
626             characterIdToMember[characterId].expired > block.timestamp,
627             "GoldFeverGuild: member should pay guild fee first"
628         );
629     }
630     require(
631         checkOwnerRight(idToGuild[guildId].owner, idToGuild[guildId].limit),
632         "GoldFeverGuild: owner don't have right to owned guild"
633     );
634     (bool isKickstarter, , ) = IGuildRight(guildRightContract)
635         .checkGuildType(idToGuild[guildId].owner);
636     require(
637         isKickstarter,
638         "GoldFeverGuild: owner has not enabled feature donation"
639     );
640     require(
641         guildIdToItemIdToDonationItem[guildId][itemId].status ==
```

```
642             STATUS_CREATED,
643         "GoldFeverGuild: item is not avaible"
644     );
645     guildIdToItemIdToDonationItem[guildId][itemId].status = STATUS_INUSED;
646     guildIdToItemIdToDonationItem[guildId][itemId]
647         .userCharacterId = characterId;
648     emit DonationItemAllocated(itemId, characterId);
649 }
```

5.17.2. Remediation

Inspex suggests implementing the validation for the disbanded guild in the `contributeInvestment()`, `sellItemToInvestment()`, and `allocateDonationItemToUser()` functions and allowing contributors to withdraw their investment immediately after the guild disbanded, for example:

Implementing the disbanded guild status validation in lines 830–833 of the `contributeInvestment()` function.

GoldFeverGuild.sol

```
816 function contributeInvestment(
817     uint256 investmentId,
818     uint256 characterId,
819     uint256 nglAmount
820 ) public nonReentrant {
821     require(
822         nglAmount > 0,
823         "GoldFeverGuild: nglAmount should be greater than 0"
824     );
825     require(
826         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
827         "GoldFeverGuild: not owner character"
828     );
829     uint256 guildId = characterIdToMember[characterId].guildId;
830     require(
831         idToGuild[guildId].status == STATUS_CREATED,
832         "GoldFeverGuild: guild is not available"
833     );
834     require(
835         checkOwnerRight(idToGuild[guildId].owner, idToGuild[guildId].limit),
836         "GoldFeverGuild: owner don't have right to owned guild"
837     );
838     (bool isKickstarter, , ) = IGuildRight(guildRightContract)
839         .checkGuildType(idToGuild[guildId].owner);
840     require(
841         isKickstarter,
842         "GoldFeverGuild: has not enabled feature donation"
843     );
```

```
844         require(
845             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
846                 .expiredTimeRaising >
847                 block.timestamp &&
848                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
849                     .nglFund <
850                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
851                     .nglAmount,
852             "GoldFeverGuild: investment has ended"
853         );
854 
855         uint256 depositAmount = guildIdToInvestmentIdToInvestmentItem[guildId][
856             investmentId
857         ].nglFund +
858             nglAmount >
859             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
860                 .nglAmount
861             ? guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
862                 .nglAmount -
863                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
864                     .nglFund
865             : nglAmount;
866         ngl.transferFrom(_msgSender(), address(this), depositAmount);
867         investmentIdToCharacterIdToAmount[investmentId][
868             characterId
869         ] += depositAmount;
870         guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
871             .nglFund += depositAmount;
872         emit InvestmentContributed(
873             investmentId,
874             characterId,
875             investmentIdToCharacterIdToAmount[investmentId][characterId],
876             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId].nglFund
877         );
878     }
```

Implementing the disbanded guild status validation in lines 930–933 of the `sellItemToInvestment()` function.

GoldFeverGuild.sol

```
924 function sellItemToInvestment(
925     uint256 guildId,
926     uint256 investmentId,
927     uint256 itemId,
928     address nftContract
929 ) public nonReentrant {
930     require(
```

```
931             idToGuild[guildId].status == STATUS_CREATED,
932             "GoldFeverGuild: guild is not available"
933         );
934         Investment memory investment = guildIdToInvestmentIdToInvestmentItem[
935             guildId
936         ][investmentId];
937         require(
938             (IGoldFeverItemType(itemTypeContract).getItemType(itemId)) ==
939                 investment.itemTypeId,
940             "GoldFeverGuild: item type is not match"
941         );
942         require(
943             investment.expiredTimeListing > block.timestamp &&
944                 investment.nglAmount == investment.nglFund &&
945                 investment.status == STATUS_CREATED,
946             "GoldFeverGuild: Investment is not avaible"
947         );
948 
949         IERC721(nftContract).safeTransferFrom(
950             _msgSender(),
951             address(this),
952             itemId
953         );
954         ngl.transfer(_msgSender(), investment.nglAmount);
955         investment.status = STATUS_SUCCESS;
956         guildIdToInvestmentIdToInvestmentItem[guildId][
957             investmentId
958         ] = investment;
959 
960         investmentIdToItem[investmentId] = InvestmentItem(
961             investmentId,
962             itemId,
963             nftContract
964         );
965         emit ItemToInvestmentSold(guildId, investmentId, itemId, nftContract);
966     }
```

Implementing the disbanded guild status validation in lines 614-617 of the `allocateDonationItemToUser()` function.

GoldFeverGuild.sol

```
608 function allocateDonationItemToUser(uint256 characterId, uint256 itemId)
609     public
610     only(DEFAULT_ADMIN_ROLE)
611     nonReentrant
612 {
613     uint256 guildId = characterIdToMember[characterId].guildId;
```

```
614     require(
615         idToGuild[guildId].status == STATUS_CREATED,
616         "GoldFeverGuild: guild is not available"
617     );
618     require(
619         guildId != 0 &&
620             characterIdToMember[characterId].status == STATUS_JOINING,
621         "GoldFeverGuild: character not in guild"
622     );
623
624     if (
625         idToGuild[guildId].joinFee > 0 &&
626         !characterIdToMember[characterId].isOwner
627     ) {
628         require(
629             characterIdToMember[characterId].expired > block.timestamp,
630             "GoldFeverGuild: member should pay guild fee first"
631         );
632     }
633     require(
634         checkOwnerRight(idToGuild[guildId].owner, idToGuild[guildId].limit),
635         "GoldFeverGuild: owner don't have right to owned guild"
636     );
637     (bool isKickstarter, , ) = IGuildRight(guildRightContract)
638         .checkGuildType(idToGuild[guildId].owner);
639     require(
640         isKickstarter,
641         "GoldFeverGuild: owner has not enabled feature donation"
642     );
643     require(
644         guildIdToItemIdToDonationItem[guildId][itemId].status ==
645             STATUS_CREATED,
646         "GoldFeverGuild: item is not avaible"
647     );
648     guildIdToItemIdToDonationItem[guildId][itemId].status = STATUS_INUSED;
649     guildIdToItemIdToDonationItem[guildId][itemId]
650         .userCharacterId = characterId;
651     emit DonationItemAllocated(itemId, characterId);
652 }
```

Adding the check condition in line 881, allowing the contributors to withdraw their tokens right away after the guild disbanded.

GoldFeverGuild.sol

```
876 function withdrawFailedInvest(uint256 investmentId, uint256 characterId)
877     public
878     nonReentrant
```

```
879 {
880     uint256 guildId = characterIdToMember[characterId].guildId;
881     if (idToGuild[guildId].status == STATUS_CREATED) {
882         require(
883             IERC721(characterContract).ownerOf(characterId) == _msgSender(),
884             "GoldFeverGuild: not owner character"
885         );
886         if (
887             idToGuild[guildId].joinFee > 0 &&
888             !characterIdToMember[characterId].isOwner
889         ) {
890             require(
891                 characterIdToMember[characterId].expired > block.timestamp,
892                 "GoldFeverGuild: member should pay guild fee first"
893             );
894         }
895         require(
896             investmentIdToCharacterIdToAmount[investmentId][characterId] > 0,
897             "GoldFeverGuild: has not invested"
898         );
899         require(
900             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
901                 .expiredTimeRaising < block.timestamp,
902             "GoldFeverGuild: investment is running"
903         );
904 
905         if (
906             guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
907                 .expiredTimeListing > block.timestamp
908         ) {
909             require(
910                 guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
911                     .nglFund <
912 
    guildIdToInvestmentIdToInvestmentItem[guildId][investmentId]
913                         .nglAmount,
914                 "GoldFeverGuild: investment is running"
915             );
916         }
917     }
918     ngl.transfer(
919         _msgSender(),
920         investmentIdToCharacterIdToAmount[investmentId][characterId]
921     );
922     delete investmentIdToCharacterIdToAmount[investmentId][characterId];
923     emit FailedInvestWithdrew(investmentId, characterId);
924 }
```

5.18. Denial of Service in Validating the Expiration of Right

ID	IDX-018
Target	GoldFeverGuildRight GoldFeverMerchantRight
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium If the previous right is expired, the currently available right will be unusable in the function that validates the expiry of each right. cause the function that uses the result from these functions before taking an action to be unavailable. **Likelihood:** Low This issue occurs when the user has available higher right privilege and lower right privilege that has expired.
Status	**Acknowledged** The Gold Fever team has acknowledged this issue. As the team responds, the issue is expected behavior.

5.18.1. Description

In the `GoldFeverGuild` contract, the `createGuid()` function allows users to create guild by validating the caller's right with `checkOwnerRight()` to check the limit, at line 271.

GoldFeverGuild.sol

```
244 function createGuild(
245     string memory name,
246     uint256 limit,
247     uint256 characterId,
248     uint256 joinFee
249 ) public nonReentrant {
250     require(
251         ICharacter(characterContract).checkIsCharacterAlive(characterId),
252         "GoldFeverGuild: character is dead"
253     );
254
255     require(
256         usedNames[keccak256(abi.encodePacked(name))] == false,
257         "GoldFeverCharacter: guild name has been used"
258     );
259
```

```
260     require(
261         isAdventureCharacter(characterId),
262         "GoldFeverGuild: tribal character don't have right to create guild"
263     );
264 
265     require(
266         IERC721(characterContract).ownerOf(characterId) == _msgSender(),
267         "GoldFeverGuild: not owner character"
268     );
269 
270     require(
271         checkOwnerRight(_msgSender(), limit),
272         "GoldFeverGuild: don't have right to create guild"
273     );
274     require(
275         !characterIdToMember[characterId].isOwner,
276         "GoldFeverGuild: character created guild already"
277     );
278     if (characterIdToMember[characterId].guildId != 0) {
279         require(
280             idToGuild[characterIdToMember[characterId].guildId].status ==
281                 STATUS_DISBAND ||
282                 characterIdToMember[characterId].status == STATUS_QUIT,
283             "GoldFeverGuild: user in guild already"
284         );
285     }
286 
287     require(
288         ownerToGuildId[_msgSender()] == 0,
289         "GoldFeverGuild: user created guild already"
290     );
291 
292     require(limit > 0, "GoldFeverGuild: limit should be greater than 0");
293 
294     if (joinFee != 0) {
295         (, bool isExclusivity, ) = IGuildRight(guildRightContract)
296             .checkGuildType(_msgSender());
297         require(
298             isExclusivity,
299             "GoldFeverGuild: don't have right to ask your members to pay for a
    membership"
300         );
301     }
302     usedNames[keccak256(abi.encodePacked(name))] = true;
303     _guildIds.increment();
304     uint256 counter = _guildIds.current();
305     idToGuild[counter] = Guild(
```

```
306             counter,
307             name,
308             characterId,
309             _msgSender(),
310             limit < 501 ? limit : 501,
311             joinFee,
312             1,
313             STATUS_CREATED
314         );
315         ownerToGuildId[_msgSender()] = counter;
316 
317         characterIdToMember[characterId] = Member(
318             counter,
319             characterId,
320             STATUS_JOINING,
321             block.timestamp,
322             true
323         );
324         emit GuildCreated(
325             counter,
326             name,
327             characterId,
328             _msgSender(),
329             limit,
330             joinFee,
331             1
332         );
333     }
```

The `checkOwnerRight()` function is used to send the owner address to function `checkGuildLimitation()` for checking the limit of current right that owner has.

GoldFeverGuild.sol

```
217 function checkOwnerRight(address owner, uint256 limit)
218     internal
219     view
220     returns (bool result)
221 {
222     uint256 limitGuildRight = IGuildRight(guildRightContract)
223         .checkGuildLimitation(owner);
224     if (limitGuildRight < 501) {
225         result = limitGuildRight >= limit;
226     } else {
227         result = true;
228     }
229 }
```

In the `checkGuildLimitation()` function, this function loops the right id between 601 and 603 to check expiration of each right that owner has.

But if the owner has right id 603 that is not yet expired, and right id 601 that expired, the `rightId` state will be zero, resulting in the loop will break and the state `rightIdToguildLimitation[rightId]` result will equal to zero at line 35.

GoldFeverGuildRight.sol

```
22 function checkGuildLimitation(address wallet)
23     external
24     view
25     override
26     returns (uint256 limit)
27 {
28     uint256 rightId = 0;
29     for (uint256 i = 601; i <= 603; i++) {
30         if (gfr.addressToRightExpiry(wallet, i) < block.timestamp) {
31             break;
32         }
33         rightId = i;
34     }
35     limit = rightIdToguildLimitation[rightId];
36 }
```

The following table contains all functions that need to validate the expiration of right.

Target	Contract	Function
GoldFeverGuildRight.sol (L: 22)	GoldFeverGuildRight	checkGuildLimitation()
GoldFeverMerchantRight.sol (L: 74)	GoldFeverMerchantRight	getWalletLimit()
GoldFeverMerchantRight.sol (L: 118)	GoldFeverMerchantRight	getWalletItemLimit()
GoldFeverMerchantRight.sol (L: 187)	GoldFeverMerchantRight	isItemMaxOut()

5.18.2. Remediation

Inspex suggests changing the condition while looping from breaking the loop when the right is expired to pass the condition when the right is not expired and set the `rightId` state inside the condition instead in lines 30-32, for example:

GoldFeverGuildRight.sol

```
22 function checkGuildLimitation(address wallet)
23     external
24     view
25     override
26     returns (uint256 limit)
27 {
28     uint256 rightId = 0;
29     for (uint256 i = 601; i <= 603; i++) {
30         if (gfr.addressToRightExpiry(wallet, i) >= block.timestamp) {
31             rightId = i;
32         }
33     }
34     limit = rightIdToguildLimitation[rightId];
35 }
```

5.19. Lack of Hiring Status Validation in renewHiring() Function

ID	IDX-019
Target	GoldFeverNpc
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium After the `buildingOwner` executed the `cancelHiring()` function, the `buildingOwner` still lost the token when the admin executed the `renewHiring()` function. However, the token that the `buildingOwner` lost will be limited to the token that the `buildingOwner` approved for admin only. **Likelihood:** Low Only the admin role has the authority to execute the `renewHiring()` function to drain the owner's approved token.
Status	Resolved The Gold Fever team has resolved this issue by implementing the `onlyHiringCreated` modifier to the `renewHiring()` function. The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.19.1. Description

The `GoldFeverNpc` contract allows users to create hiring for the game's building, which will expire after 30 days. After that, if the users want to cancel the hiring, they can execute the `cancelHiring()` function to cancel the hiring.

GoldFeverNpc.sol

```
666 function cancelHiring(uint256 hiringId, address nftContract)
667     public
668     nonReentrant
669     onlyHiringCreated(hiringId)
670 {
671     address buildingOwner = idToHiring[hiringId].buildingOwner;
672     require(
673         buildingOwner == _msgSender(),
674         "GoldFeverNpc: Only building owner can cancel hiring"
675     );
676     if (addressToPendingEarning[buildingOwner] > 0) {
677         ngl.transfer(buildingOwner, addressToPendingEarning[buildingOwner]);
678         addressToPendingEarning[buildingOwner] = 0;
```

```
679         }
680         IERC721(nftContract).safeTransferFrom(
681             address(this),
682             buildingOwner,
683             idToHiring[hiringId].buildingItem
684         );
685         idToHiring[hiringId].status = STATUS_CANCELED;
686 
687         addressToItemCount[buildingOwner][18]--;
688         addressToItemCount[buildingOwner][idToHiring[hiringId].itemType]--;
689 
690         uint256[] memory itemIds = new uint256[](
691             hiringIdToItems[hiringId].length
692         );
693 
694         // emit all items include already Requested items
695         for (uint256 i = 0; i < hiringIdToItems[hiringId].length; i++) {
696             itemIds[i] = hiringIdToItems[hiringId][i];
697             hiringIdToRentableItem[hiringId][itemIds[i]]
698                 .status = STATUS_REQUESTED;
699         }
700 
701         emit WithdrawalsRequested(hiringId, itemIds);
702 
703         emit HiringCanceled(hiringId);
704     }
```

However, if the admin called the `renewHiring()` function after the hiring had already been canceled to extend the expiry of the canceled hiring, the `renewHiring()` function still works as if the hiring had never been canceled. As a result, the token that the owner approved for the admin will be drained.

GoldFeverNpc.sol

```
323 function renewHiring(uint256 hiringId) public only(DEFAULT_ADMIN_ROLE) {
324     require(
325         tierToFee[idToHiring[hiringId].tier] > 0,
326         "GoldFeverNpc: don't need to renewal free tier"
327     );
328     require(
329         idToHiring[hiringId].expired < block.timestamp,
330         "GoldFeverNpc: has not expired"
331     );
332     uint256 hiringFee = tierToFee[idToHiring[hiringId].tier];
333     address owner = idToHiring[hiringId].buildingOwner;
334     uint256 allowance = ngl.allowance(owner, address(this));
335     if (allowance >= hiringFee && ngl.balanceOf(owner) >= hiringFee) {
336         payNpcHiringFee(owner, hiringFee);
```

```
337         idToHiring[hiringId].expired =
338             idToHiring[hiringId].expired +
339             30 days;
340         emit HiringRenewed(
341             hiringId,
342             idToHiring[hiringId].expired + 30 days
343         );
344     } else {
345         idToHiring[hiringId].tier = 1;
346         emit HiringDowngraded(hiringId, 1);
347     }
348 }
```

5.19.2. Remediation

Inspex suggests implementing the hiring status validation for the `renewHiring()` function in line 328 to prevent the admin from executing the `renewHiring()` function on the canceled hiring.

GoldFeverNpc.sol

```
323 function renewHiring(uint256 hiringId) public only(DEFAULT_ADMIN_ROLE) {
324     require(
325         tierToFee[idToHiring[hiringId].tier] > 0,
326         "GoldFeverNpc: don't need to renewal free tier"
327     );
328     require(
329         idToHiring[hiringId].status != STATUS_CANCELED,
330         "GoldFeverNpc: hiring has already been canceled."
331     );
332     require(
333         idToHiring[hiringId].expired < block.timestamp,
334         "GoldFeverNpc: has not expired"
335     );
336     uint256 hiringFee = tierToFee[idToHiring[hiringId].tier];
337     address owner = idToHiring[hiringId].buildingOwner;
338     uint256 allowance = ngl.allowance(owner, address(this));
339     if (allowance >= hiringFee && ngl.balanceOf(owner) >= hiringFee) {
340         payNpcHiringFee(owner, hiringFee);
341         idToHiring[hiringId].expired =
342             idToHiring[hiringId].expired +
343             30 days;
344         emit HiringRenewed(
345             hiringId,
346             idToHiring[hiringId].expired + 30 days
347         );
348     } else {
349         idToHiring[hiringId].tier = 1;
350         emit HiringDowngraded(hiringId, 1);
```

```
351         }
352 }
```

5.20. Smart Contract with Unpublished Source Code

ID	IDX-020
Target	GoldFeverAuction GoldFeverBuildingManager GoldFeverCharacter GoldFeverCollateral GoldFeverForwarder GoldFeverGovernor GoldFeverGuild GoldFeverGuildRight GoldFeverItem GoldFeverItemTier GoldFeverItemType GoldFeverLeasing GoldFeverMarket GoldFeverMask GoldFeverMaskBox GoldFeverMerchantRight GoldFeverMiningClaim GoldFeverNativeGold GoldFeverNpc GoldFeverPaymaster GoldFeverRight GoldFeverSwap GoldFeverTaxation AccessControlMixin EIP712Base IChildToken Initializable NativeMetaTransaction
Category	General Smart Contract Vulnerability
CWE	CWE-1006: Bad Coding Practices
Risk	**Severity:** Low **Impact:** Medium The logic of the smart contract may not align with the user's understanding, causing undesired actions to be taken when the user interacts with the smart contract. **Likelihood:** Low The possibility for the users to misunderstand the functionalities of the contract is not very high with the help of the documentation and user interface.

Status	**Acknowledged** The Gold Fever team has acknowledged this issue and decided not to publish the source code because the project is under development until the third and fourth quarters. After the full version is completed, they will publish the source code.

5.20.1. Description

The smart contract source code is not publicly published, so the users will not be able to easily verify the correctness of the functionalities and the logic of the smart contract by themselves. Therefore, it is possible that the user's understanding of the smart contract does not align with the actual implementation, leading to undesired actions on interacting with the smart contract.

5.20.2. Remediation

Inspex suggests publishing the contract source code through a public code repository or verifying the smart contract source code on the blockchain explorer so that the users can easily read and understand the logic of the smart contract by themselves.

5.21. Insufficient Logging for Privileged Functions

ID	IDX-021
Target	GoldFeverAuction GoldFeverBuildingManager GoldFeverCharacter GoldFeverCollateral GoldFeverGovernor GoldFeverGuild GoldFeverGuildRight GoldFeverItem GoldFeverItemTier GoldFeverLeasing GoldFeverMarket GoldFeverMask GoldFeverMerchantRight GoldFeverMiningClaim GoldFeverNativeGold GoldFeverNpc GoldFeverRight GoldFeverSwap GoldFeverTaxation
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved The Gold Fever team has resolved this issue by emitting events for the execution of privileged functions in commit `2c0e553a332d267df513888bbf7a6af2190f0af9`.

5.21.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can set the fee recipient by executing the `setCommissionRate()` function in the `GoldFeverMask` contract, and no events are emitted.

GoldFeverMask.sol

```
469 function setCommissionRate(uint256 _rate)
470     public
471     nonReentrant
472     only(DEFAULT_ADMIN_ROLE)
473 {
474     require(
475         _rate > 0,
476         "GoldFeverMask: Commission rate must be greater than 0"
477     );
478     commissionRate = _rate;
479 }
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function	Role
GoldFeverAuction.sol (L:77)	GoldFeverAuction	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverAuction.sol (L:84)	GoldFeverAuction	setMerchantContract()	DEFAULT_ADMIN_ROLE
GoldFeverBuildingManager.sol (L:45)	GoldFeverBuildingManager	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverBuildingManager.sol (L:52)	GoldFeverBuildingManager	setItemTypeContract()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:103)	GoldFeverCharacter	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:110)	GoldFeverCharacter	setNpcContract()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:117)	GoldFeverCharacter	setGuildContract()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:197)	GoldFeverCharacter	setAttachSlotLimit()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:205)	GoldFeverCharacter	setCommissionRate()	DEFAULT_ADMIN_ROLE
GoldFeverCharacter.sol (L:656)	GoldFeverCharacter	setLimitPerType()	DEFAULT_ADMIN_ROLE
GoldFeverCollateral.sol	GoldFeverCollateral	setForwarderContract()	DEFAULT_ADMIN_ROLE

(L:99)			
GoldFeverCollateral.sol (L:106)	GoldFeverCollateral	setTaxationContract()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:201)	GoldFeverGovernor	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:208)	GoldFeverGovernor	setNpcContract()	DEFAULT_ADMIN_ROLE
GoldFeverGovernor.sol (L:215)	GoldFeverGovernor	setTaxCollector()	DEFAULT_ADMIN_ROLE
GoldFeverGuild.sol (L:44)	GoldFeverGuild	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverGuild.sol (L:964)	GoldFeverGuild	setCharacterContract()	DEFAULT_ADMIN_ROLE
GoldFeverGuild.sol (L:971)	GoldFeverGuild	setGuildRightContract()	DEFAULT_ADMIN_ROLE
GoldFeverGuildRight.sol (L:67)	GoldFeverGuildRight	setGuildLimitation()	DEFAULT_ADMIN_ROLE
GoldFeverItem.sol (L:53)	GoldFeverItem	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverItemTier.sol (L:56)	GoldFeverItemTier	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverItemTier.sol (L:96)	GoldFeverItemTier	setItemTypeContract()	DEFAULT_ADMIN_ROLE
GoldFeverLeasing.sol (L:79)	GoldFeverLeasing	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverLeasing.sol (L:86)	GoldFeverLeasing	setMerchantContract()	DEFAULT_ADMIN_ROLE
GoldFeverLeasing.sol (L:93)	GoldFeverLeasing	setTaxationContract()	DEFAULT_ADMIN_ROLE
GoldFeverMarket.sol (L:69)	GoldFeverMarket	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverMarket.sol (L:76)	GoldFeverMarket	setMerchantContract()	DEFAULT_ADMIN_ROLE
GoldFeverMarket.sol	GoldFeverMarket	setTaxationContract()	DEFAULT_ADMIN_ROLE

(L:83)			
GoldFeverMask.sol (L:61)	GoldFeverMask	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:398)	GoldFeverMask	updateForgeMaskFee()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:407)	GoldFeverMask	updateUnforgeMaskFee()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:416)	GoldFeverMask	updatePurchaseMaskCost()	DEFAULT_ADMIN_ROLE
GoldFeverMask.sol (L:469)	GoldFeverMask	setCommissionRate()	DEFAULT_ADMIN_ROLE
GoldFeverMaskBox.sol (L:38)	GoldFeverMaskBox	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverMerchantRight.sol (L:67)	GoldFeverMerchantRight	setItemTypeContract()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:58)	GoldFeverMiningClaim	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:228)	GoldFeverMiningClaim	setArenaHourPrice()	DEFAULT_ADMIN_ROLE
GoldFeverMiningClaim.sol (L:265)	GoldFeverMiningClaim	setMiningSpeed()	DEFAULT_ADMIN_ROLE
GoldFeverNativeGold.sol (L:31)	GoldFeverNativeGold	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:150)	GoldFeverNpc	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:157)	GoldFeverNpc	setMerchantContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:164)	GoldFeverNpc	setItemTierContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:171)	GoldFeverNpc	setBuildingManagerContract()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:178)	GoldFeverNpc	setGovernorContract()	DEFAULT_ADMIN_ROLE

GoldFeverNpc.sol (L:185)	GoldFeverNpc	setNpcHiringFee()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:193)	GoldFeverNpc	setTierStatus()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:706)	GoldFeverNpc	assignManager()	onlyHiringCreated(hiringId)
GoldFeverNpc.sol (L:726)	GoldFeverNpc	unassignManager()	onlyHiringCreated(hiringId)
GoldFeverNpc.sol (L:755)	GoldFeverNpc	setRentPercentageLimit()	DEFAULT_ADMIN_ROLE
GoldFeverNpc.sol (L:762)	GoldFeverNpc	setTicketPercentageLimit()	DEFAULT_ADMIN_ROLE
GoldFeverRight.sol (L:86)	GoldFeverRight	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverSwap.sol (L:68)	GoldFeverSwap	setForwarderContract()	DEFAULT_ADMIN_ROLE
GoldFeverTaxation.sol (L:21)	GoldFeverTaxation	setTaxCollector()	DEFAULT_ADMIN_ROLE
GoldFeverTaxation.sol (L:28)	GoldFeverTaxation	setCompanyTax()	DEFAULT_ADMIN_ROLE

5.21.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

GoldFeverMask.sol

```
469 event CommissionRateSetted(uint256 _oldRate, uint256 _newRate);
470 function setCommissionRate(uint256 _rate)
471     public
472     nonReentrant
473     only(DEFAULT_ADMIN_ROLE)
474 {
475     require(
476         _rate > 0,
477         "GoldFeverMask: Commission rate must be greater than 0"
478     );
479     emit CommissionRateSetted(commissionRate, _rate);
480     commissionRate = _rate;
481 }
```

5.22. Unbound Configuration Parameter (commissionRate)

ID	IDX-022
Target	GoldFeverMask
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low When the `commissionRate` state is over 100 the users can not use the `forgeMask()`, `unforgeMask()` and `purchaseMask()` functions. However, there is the `setCommissionRate()` function that allows the admin to set the `commissionRate` state to the number that functions are usable. **Likelihood:** Low The default value of the `commissionRate` state is 1 and only admin can set the `commissionRate` state.
Status	Resolved The Gold Fever team has resolved this issue by implementing an upper bound for the commission rate in the `setCommissionRate()` function. The issue has been resolved in commit: `2c0e553a332d267df513888bbf7a6af2190f0af9`

5.22.1. Description

The`forgeMask()`, `unforgeMask()`, and `purchaseMask()` functions will be reverted at the `burnFrom()` function call because the Solidity version 0.8.0 does not allow the math operations of type `uint` to be the negative value to prevent overflow and underflow. For example,

At line 291, the `adminEarn` is calculated from the `unforgeMaskFee` multiplied by the `commissionRate` and divided by 100. When the `commissionRate` is over 100, it results in the `adminEarn` being more than the `unforgeMaskFee` state and it will be reverted while calling the `burnFrom()` function at line 295.

GoldFeverMask.sol

```
133 function forgeMask(
134     uint256 maskShape,
135     uint256 maskMaterial,
136     uint256 topElement,
137     uint256 frontElement,
138     uint256 scratches,
139     uint256 paintOver
140 ) public nonReentrant {
141     require(maskShape > 0, "GoldFeverMask: Need at least one shape");
```

```
142     require(maskMaterial > 0, "GoldFeverMask: Need at least one material");
143
144     require(
145         IGoldFeverItemType(itemTypeContract).getItemType(maskShape) >=
146             231010 &&
147             IGoldFeverItemType(itemTypeContract).getItemType(maskShape) <=
148             231024,
149         "GoldFeverMask: Invalid mask shape"
150     );
151
152     require(
153         IGoldFeverItemType(itemTypeContract).getItemType(maskMaterial) >=
154             231110 &&
155             IGoldFeverItemType(itemTypeContract).getItemType(
156                 maskMaterial
157             ) <=
158             231124,
159         "GoldFeverMask: Invalid mask material"
160     );
161
162     require(
163         (IGoldFeverItemType(itemTypeContract).getItemType(topElement) >=
164             231210 &&
165             IGoldFeverItemType(itemTypeContract).getItemType(topElement) <=
166             231224) ||
167             IGoldFeverItemType(itemTypeContract).getItemType(topElement) ==
168             0,
169         "GoldFeverMask: Invalid top element"
170     );
171
172     require(
173         (IGoldFeverItemType(itemTypeContract).getItemType(frontElement) >=
174             231310 &&
175             IGoldFeverItemType(itemTypeContract).getItemType(
176                 frontElement
177             ) <=
178             231325) ||
179             IGoldFeverItemType(itemTypeContract).getItemType(
180                 frontElement
181             ) ==
182             0,
183         "GoldFeverMask: Invalid front element"
184     );
185
186     require(
187         (IGoldFeverItemType(itemTypeContract).getItemType(scratches) >=
188             231410 &&
```

```
            IGoldFeverItemType(itemTypeContract).getItemType(scratches) <=
            231424) ||
            IGoldFeverItemType(itemTypeContract).getItemType(scratches) ==
            0,
        "GoldFeverMask: Invalid scratches"
    );

    require(
        (IGoldFeverItemType(itemTypeContract).getItemType(paintOver) >=
            231510 &&
            IGoldFeverItemType(itemTypeContract).getItemType(paintOver) <=
            231526) ||
            IGoldFeverItemType(itemTypeContract).getItemType(paintOver) ==
            0,
        "GoldFeverMask: Invalid paintOver"
    );
    uint256 maskId = parseInt(
        string(
            abi.encodePacked(
                zeroPadNumber(
                    (
                        IGoldFeverItemType(itemTypeContract).getItemType(
                            maskShape
                        )
                    ) % 10000
                ),
                zeroPadNumber(
                    (
                        IGoldFeverItemType(itemTypeContract).getItemType(
                            maskMaterial
                        )
                    ) % 10000
                ),
                zeroPadNumber(
                    (
                        IGoldFeverItemType(itemTypeContract).getItemType(
                            topElement
                        )
                    ) % 10000
                ),
                zeroPadNumber(
                    (
                        IGoldFeverItemType(itemTypeContract).getItemType(
                            frontElement
                        )
                    ) % 10000
                ),
```

```
                zeroPadNumber(
                    (
                        IGoldFeverItemType(itemTypeContract).getItemType(
                            scratches
                        )
                    ) % 10000
                ),
                zeroPadNumber(
                    (
                        IGoldFeverItemType(itemTypeContract).getItemType(
                            paintOver
                        )
                    ) % 10000
                )
            )
        )
    );

    require(
        idToMask[maskId].status != FORGED,
        "GoldFeverMask: This mask type is already forged by other user"
    );

    address msgSender = _msgSender();
    gfi.safeTransferFrom(msgSender, address(this), maskShape);
    gfi.safeTransferFrom(msgSender, address(this), maskMaterial);
    if (IGoldFeverItemType(itemTypeContract).getItemType(topElement) != 0) {
        gfi.safeTransferFrom(msgSender, address(this), topElement);
    }
    if (
        IGoldFeverItemType(itemTypeContract).getItemType(frontElement) != 0
    ) {
        gfi.safeTransferFrom(msgSender, address(this), frontElement);
    }
    if (IGoldFeverItemType(itemTypeContract).getItemType(scratches) != 0) {
        gfi.safeTransferFrom(msgSender, address(this), scratches);
    }
    if (IGoldFeverItemType(itemTypeContract).getItemType(paintOver) != 0) {
        gfi.safeTransferFrom(msgSender, address(this), paintOver);
    }

    _mint(msgSender, maskId);

    idToMask[maskId] = Mask(
        maskId,
        msgSender,
        FORGED,
```

```
283             maskShape,
284             maskMaterial,
285             topElement,
286             frontElement,
287             scratches,
288             paintOver
289         );
290
291         uint256 adminEarn = (forgeMaskFee * commissionRate) / 100;
292
293         if (addressToFirstFreeForge[msgSender]) {
294             ngl.transferFrom(msgSender, address(this), adminEarn);
295             ngl.burnFrom(msgSender, forgeMaskFee - adminEarn);
296             nglFromCollectedFee += adminEarn;
297         } else {
298             addressToFirstFreeForge[msgSender] = true;
299         }
300
301         emit MaskForged(
302             maskId,
303             msgSender,
304             IGoldFeverItemType(itemTypeContract).getItemType(maskShape),
305             IGoldFeverItemType(itemTypeContract).getItemType(maskMaterial),
306             IGoldFeverItemType(itemTypeContract).getItemType(topElement),
307             IGoldFeverItemType(itemTypeContract).getItemType(frontElement),
308             IGoldFeverItemType(itemTypeContract).getItemType(scratches),
309             IGoldFeverItemType(itemTypeContract).getItemType(paintOver)
310         );
311     }
```

5.22.2. Remediation

Inspex suggests implementing an upper bound for the commission rate to prevent the setting of the `commissionRate` above 100.

GoldFeverMask.sol

```
469 function setCommissionRate(uint256 _rate)
470     public
471     nonReentrant
472     only(DEFAULT_ADMIN_ROLE)
473 {
474     require(
475         _rate > 0 && _rate <= 100 ,
476         "GoldFeverMask: Commission rate must be between 0 and 100"
477     );
478     commissionRate = _rate;
479 }
```

5.23. Improper Access Control in finalizeRightPurchase() Function

ID	IDX-023
Target	GoldFeverRight
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** Very Low **Impact:** Low There is no access control in the `finalizeRightPurchase()` function anyone can use this function to finalize any existing right. However, there is no profit from this action. **Likelihood:** Low It is very unlikely that the attacker will call `finalizeRightPurchase()` to change the status of the target right from `STAKED` to `FINALIZED`.
Status	Resolved The Gold Fever team has resolved this issue by validating for the right owner in `finalizeRightPurchase()` function as we suggested in commit `6184061d6745fb03b010ae2725b1709a05189be5`

5.23.1. Description

In the `GoldFeverRight` contract, the function `finalizeRightPurchase()` can change the status state of `rightPurchaseId`, which requires the status to be `STAKED` before changing in lines 252-256.

GoldFeverRight.sol

```
248 function finalizeRightPurchase(
249     uint256 rightOptionId,
250     uint256 rightPurchaseId
251 ) public nonReentrant {
252     require(
253         rightIdToRightPurchase[rightOptionId][rightPurchaseId].status ==
254             STAKED,
255         "GoldFeverRight: You can only finalize Stake option"
256     );
257     require(
258         rightIdToRightPurchase[rightOptionId][rightPurchaseId]
259             .stakeExpiry <= block.timestamp,
260         "GoldFeverRight: Staking duration has not expired"
261     );
262
263     ngl.transfer(
264         rightIdToRightPurchase[rightOptionId][rightPurchaseId].buyer,
```

```
265             rightIdToRightPurchase[rightOptionId][rightPurchaseId].price
266         );
267
268         rightIdToRightPurchase[rightOptionId][rightPurchaseId]
269             .status = FINALIZED;
270
271         emit RightOptionPurchaseFinished(rightPurchaseId);
272     }
```

Anyone can execute this function with any `rightOptionId` of the target to set the purchase right status to `FINALIZED` without any restriction.

5.23.2. Remediation

Inspex suggests implementing the authorization check whether the caller is the buyer as shown in lines 252-256.

GoldFeverRight.sol

```
248 function finalizeRightPurchase(
249     uint256 rightOptionId,
250     uint256 rightPurchaseId
251 ) public nonReentrant {
252     address msgSender = _msgSender();
253     require(
254         rightIdToRightPurchase[rightOptionId][rightPurchaseId].buyer ==
    msgSender,
255             "GoldFeverRight: Please use your address to finalize"
256     );
257     require(
258         rightIdToRightPurchase[rightOptionId][rightPurchaseId].status ==
259             STAKED,
260         "GoldFeverRight: You can only finalize Stake option"
261     );
262     require(
263         rightIdToRightPurchase[rightOptionId][rightPurchaseId]
264             .stakeExpiry <= block.timestamp,
265         "GoldFeverRight: Staking duration has not expired"
266     );
267
268     ngl.transfer(
269         rightIdToRightPurchase[rightOptionId][rightPurchaseId].buyer,
270         rightIdToRightPurchase[rightOptionId][rightPurchaseId].price
271     );
272
273     rightIdToRightPurchase[rightOptionId][rightPurchaseId]
274         .status = FINALIZED;
275
```

```
276         emit RightOptionPurchaseFinished(rightPurchaseId);
277     }
```

5.24. Inexplicit Solidity Compiler Version

ID	IDX-024
Target	GoldFeverAuction GoldFeverBuildingManager GoldFeverCharacter GoldFeverCollateral GoldFeverForwarder GoldFeverGovernor GoldFeverGuild GoldFeverGuildRight GoldFeverItem GoldFeverItemTier GoldFeverItemType GoldFeverLeasing GoldFeverMarket GoldFeverMask GoldFeverMaskBox GoldFeverMerchantRight GoldFeverMiningClaim GoldFeverNativeGold GoldFeverNpc GoldFeverPaymaster GoldFeverRight GoldFeverSwap GoldFeverTaxation AccessControlMixin EIP712Base IChildToken Initializable NativeMetaTransaction
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The Gold Fever team has resolved this issue as suggested by fixing the Solidity compiler to the latest stable version in commit `2c0e553a332d267df513888bbf7a6af2190f0af9`.

5.24.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

GoldFeverAuction.sol

```
2 //SPDX-License-Identifier: UNLICENSED
3 pragma solidity ^0.8.0;
```

File	Version
GoldFeverAuction.sol (L:2)	^0.8.0
GoldFeverBuildingManager.sol (L:2)	^0.8.0
GoldFeverCharacter.sol (L:2)	^0.8.0
GoldFeverCollateral.sol (L:2)	^0.8.0
GoldFeverForwarder.sol (L:2)	^0.8.0
GoldFeverGovernor.sol (L:2)	^0.8.0
GoldFeverGuild.sol (L:2)	^0.8.0
GoldFeverGuildRight.sol (L:2)	^0.8.0
GoldFeverItem.sol (L:2)	^0.8.0
GoldFeverItemTier.sol (L:2)	^0.8.0
GoldFeverItemType.sol (L:2)	^0.8.0
GoldFeverLeasing.sol (L:2)	^0.8.0
GoldFeverMarket.sol (L:2)	^0.8.0
GoldFeverMask.sol (L:2)	^0.8.0
GoldFeverMaskBox.sol (L:2)	^0.8.0
GoldFeverMerchantRight.sol (L:2)	^0.8.0
GoldFeverMiningClaim.sol (L:2)	^0.8.0
GoldFeverNativeGold.sol (L:2)	^0.8.0
GoldFeverNpc.sol (L:2)	^0.8.0
GoldFeverPaymaster.sol (L:4)	^0.8.0

GoldFeverRight.sol (L:2)	^0.8.0
GoldFeverSwap.sol (L:2)	^0.8.0
GoldFeverTaxation.sol (L:2)	^0.8.0
AccessControlMixin.sol (L:2)	^0.8.0
EIP712Base.sol (L:2)	^0.8.0
IChildToken.sol (L:2)	^0.8.0
Initializable.sol (L:2)	^0.8.0
NativeMetaTransaction.sol (L:2)	^0.8.0

5.24.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is 0.8.17, for example:

GoldFeverAuction.sol

```
2 //SPDX-License-Identifier: UNLICENSED
3 pragma solidity 0.8.17;
```

5.25. Improper Function Visibility

ID	IDX-025
Target	GoldFeverAuction GoldFeverBuildingManager GoldFeverCharacter GoldFeverCollateral GoldFeverGovernor GoldFeverGuild GoldFeverGuildRight GoldFeverItem GoldFeverItemTier GoldFeverLeasing GoldFeverMarket GoldFeverMask GoldFeverMaskBox GoldFeverMerchantRight GoldFeverMiningClaim GoldFeverNativeGold GoldFeverNpc GoldFeverRight GoldFeverSwap GoldFeverTaxation NativeMetaTransaction
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No security impact The Gold Fever team has partially fixed this issue. There are some functions left with public visibility. However, there is no security impact for the function's visibility in the function that we are listed in the table.

5.25.1. Description

Public functions that are never called internally by the contract itself should have external visibility. This improves the readability of the contract, allowing clear distinction between functions that are externally used and functions that are also called internally.

For example, the following source code shows that the `setForwarderContract()` function of the

Public

`GoldFeverAuction` contract is set to public and it is never called from any internal function.

GoldFeverAuction.sol

```
110 function setForwarderContract(address forwarder_)
111     public
112     only(DEFAULT_ADMIN_ROLE)
113 {
114     _setTrustedForwarder(forwarder_);
115 }
```

The following table contains all functions that have public visibility and are never called from any internal function.

File	Contract	Function
GoldFeverAuction.sol (L:77)	GoldFeverAuction	setForwarderContract()
GoldFeverAuction.sol (L:84)	GoldFeverAuction	setMerchantContract()
GoldFeverAuction.sol (L:110)	GoldFeverAuction	createAuction()
GoldFeverAuction.sol (L:172)	GoldFeverAuction	bid()
GoldFeverAuction.sol (L:237)	GoldFeverAuction	getCurrentBidOwner()
GoldFeverAuction.sol (L:245)	GoldFeverAuction	getCurrentBidAmount()
GoldFeverAuction.sol (L:288)	GoldFeverAuction	finalizeAuction()
GoldFeverBuildingManager.sol (L:45)	GoldFeverBuildingManager	setForwarderContract()
GoldFeverBuildingManager.sol (L:52)	GoldFeverBuildingManager	setItemTypeContract()
GoldFeverCharacter.sol (L:103)	GoldFeverCharacter	setForwarderContract()
GoldFeverCharacter.sol (L:110)	GoldFeverCharacter	setNpcContract()
GoldFeverCharacter.sol (L:117)	GoldFeverCharacter	setGuildContract()
GoldFeverCharacter.sol (L:197)	GoldFeverCharacter	setAttachSlotLimit()
GoldFeverCharacter.sol (L:205)	GoldFeverCharacter	setCommissionRate()
GoldFeverCharacter.sol (L:217)	GoldFeverCharacter	withdrawCollectedFee()
GoldFeverCharacter.sol (L:275)	GoldFeverCharacter	createCharacter()
GoldFeverCharacter.sol (L:421)	GoldFeverCharacter	attachItem()

GoldFeverCharacter.sol (L:456)	GoldFeverCharacter	requestItemDetachment()
GoldFeverCharacter.sol (L:477)	GoldFeverCharacter	approveItemDetachment()
GoldFeverCharacter.sol (L:533)	GoldFeverCharacter	donateAttachItemToGuild()
GoldFeverCharacter.sol (L:569)	GoldFeverCharacter	getCreateCharacterFee()
GoldFeverCharacter.sol (L:577)	GoldFeverCharacter	setCreateCharacterFee()
GoldFeverCharacter.sol (L:592)	GoldFeverCharacter	updateCharacterSetIds()
GoldFeverCharacter.sol (L:600)	GoldFeverCharacter	getTribalFemaleIds()
GoldFeverCharacter.sol (L:608)	GoldFeverCharacter	getTribalMaleIds()
GoldFeverCharacter.sol (L:616)	GoldFeverCharacter	getAdventurerMaleIds()
GoldFeverCharacter.sol (L:624)	GoldFeverCharacter	getAdventurerFemaleIds()
GoldFeverCharacter.sol (L:632)	GoldFeverCharacter	getCharacterSetIds()
GoldFeverCharacter.sol (L:640)	GoldFeverCharacter	getCharacterSetFee()
GoldFeverCharacter.sol (L:644)	GoldFeverCharacter	setCharacterSetFee()
GoldFeverCharacter.sol (L:656)	GoldFeverCharacter	setLimitPerType()
GoldFeverCharacter.sol (L:660)	GoldFeverCharacter	checkIsAdventureCharacter()
GoldFeverCharacter.sol (L:671)	GoldFeverCharacter	checkIsCharacterAlive()
GoldFeverCharacter.sol (L:680)	GoldFeverCharacter	updateTribalMaleIds()
GoldFeverCharacter.sol (L:688)	GoldFeverCharacter	updateTribalFemaleIds()
GoldFeverCharacter.sol (L:696)	GoldFeverCharacter	updateAdventurerMaleIds()
GoldFeverCharacter.sol (L:704)	GoldFeverCharacter	updateAdventurerFemaleIds()
GoldFeverCharacter.sol (L:712)	GoldFeverCharacter	characterDie()
GoldFeverCharacter.sol (L:748)	GoldFeverCharacter	addFreeCharacterForAddress()
GoldFeverCollateral.sol (L:99)	GoldFeverCollateral	setForwarderContract()
GoldFeverCollateral.sol (L:106)	GoldFeverCollateral	setTaxationContract()
GoldFeverCollateral.sol (L:132)	GoldFeverCollateral	createCollateralOffer()
GoldFeverCollateral.sol (L:187)	GoldFeverCollateral	cancelCollateralOffer()

GoldFeverCollateral.sol (L:224)	GoldFeverCollateral	counterOffer()
GoldFeverCollateral.sol (L:272)	GoldFeverCollateral	claimCollateral()
GoldFeverCollateral.sol (L:297)	GoldFeverCollateral	payBackCollateral()
GoldFeverCollateral.sol (L:323)	GoldFeverCollateral	cancelCounterOffer()
GoldFeverCollateral.sol (L:350)	GoldFeverCollateral	takeCounterOfferIncludeTax()
GoldFeverCollateral.sol (L:393)	GoldFeverCollateral	acceptCollateralOfferIncludeTax()
GoldFeverGovernor.sol (L:201)	GoldFeverGovernor	setForwarderContract()
GoldFeverGovernor.sol (L:208)	GoldFeverGovernor	setNpcContract()
GoldFeverGovernor.sol (L:215)	GoldFeverGovernor	setTaxCollector()
GoldFeverGovernor.sol (L:260)	GoldFeverGovernor	createGovernorTax()
GoldFeverGovernor.sol (L:298)	GoldFeverGovernor	updateBuildingGovernor()
GoldFeverGovernor.sol (L:308)	GoldFeverGovernor	createZoneUpgrade()
GoldFeverGovernor.sol (L:343)	GoldFeverGovernor	swapZoneUpgradeTier()
GoldFeverGovernor.sol (L:367)	GoldFeverGovernor	updateZoneUpgrade()
GoldFeverGovernor.sol (L:381)	GoldFeverGovernor	updateMaintenancePeriod()
GoldFeverGovernor.sol (L:390)	GoldFeverGovernor	updateZoneMaintenance()
GoldFeverGovernor.sol (L:405)	GoldFeverGovernor	upgradeGovernorZone()
GoldFeverGovernor.sol (L:447)	GoldFeverGovernor	updateGovernorTax()
GoldFeverGovernor.sol (L:484)	GoldFeverGovernor	maintainGovernorZone()
GoldFeverGovernor.sol (L:525)	GoldFeverGovernor	payGovernorTaxById()
GoldFeverGovernor.sol (L:600)	GoldFeverGovernor	withdrawGovernorEarning()
GoldFeverGovernor.sol (L:617)	GoldFeverGovernor	withdrawCompanyEarning()
GoldFeverGovernor.sol (L:707)	GoldFeverGovernor	getGovernorEarning()
GoldFeverGuild.sol (L:44)	GoldFeverGuild	setForwarderContract()
GoldFeverGuild.sol (L:244)	GoldFeverGuild	createGuild()
GoldFeverGuild.sol (L:335)	GoldFeverGuild	disbandGuild()

GoldFeverGuild.sol (L:353)	GoldFeverGuild	joinGuild()
GoldFeverGuild.sol (L:420)	GoldFeverGuild	quitGuild()
GoldFeverGuild.sol (L:445)	GoldFeverGuild	withdrawGuildFee()
GoldFeverGuild.sol (L:458)	GoldFeverGuild	updateMemberFee()
GoldFeverGuild.sol (L:490)	GoldFeverGuild	updateMemberLimit()
GoldFeverGuild.sol (L:516)	GoldFeverGuild	payMemberFee()
GoldFeverGuild.sol (L:552)	GoldFeverGuild	memberDonateItem()
GoldFeverGuild.sol (L:608)	GoldFeverGuild	allocateDonationItemToUser()
GoldFeverGuild.sol (L:651)	GoldFeverGuild	takeBackDonationItemFromUser()
GoldFeverGuild.sol (L:667)	GoldFeverGuild	requestDonationItem()
GoldFeverGuild.sol (L:725)	GoldFeverGuild	approveRequestDonationItem()
GoldFeverGuild.sol (L:749)	GoldFeverGuild	createInvestment()
GoldFeverGuild.sol (L:816)	GoldFeverGuild	contributeInvestment()
GoldFeverGuild.sol (L:876)	GoldFeverGuild	withdrawFailedInvest()
GoldFeverGuild.sol (L:924)	GoldFeverGuild	sellItemToInvestment()
GoldFeverGuild.sol (L:964)	GoldFeverGuild	setCharacterContract()
GoldFeverGuild.sol (L:971)	GoldFeverGuild	setGuildRightContract()
GoldFeverGuildRight.sol (L:68)	GoldFeverGuildRight	setGuildLimitation()
GoldFeverItem.sol (L:53)	GoldFeverItem	setForwarderContract()
GoldFeverItem.sol (L:252)	GoldFeverItem	mint()
GoldFeverItemTier.sol (L:56)	GoldFeverItemTier	setForwarderContract()
GoldFeverItemTier.sol (L:82)	GoldFeverItemTier	addTier()
GoldFeverItemTier.sol (L:96)	GoldFeverItemTier	setItemTypeContract()
GoldFeverItemTier.sol (L:103)	GoldFeverItemTier	setItemTier()
GoldFeverItemTier.sol (L:120)	GoldFeverItemTier	upgradeItem()
GoldFeverItemTier.sol (L:139)	GoldFeverItemTier	setTierAttribute()

GoldFeverItemTier.sol (L:153)	GoldFeverItemTier	getTierAttribute()
GoldFeverItemTier.sol (L:165)	GoldFeverItemTier	getItemAttribute()
GoldFeverItemTier.sol (L:177)	GoldFeverItemTier	collectFee()
GoldFeverLeasing.sol (L:79)	GoldFeverLeasing	setForwarderContract()
GoldFeverLeasing.sol (L:86)	GoldFeverLeasing	setMerchantContract()
GoldFeverLeasing.sol (L:93)	GoldFeverLeasing	setTaxationContract()
GoldFeverLeasing.sol (L:119)	GoldFeverLeasing	createItem()
GoldFeverLeasing.sol (L:179)	GoldFeverLeasing	cancelItem()
GoldFeverLeasing.sol (L:206)	GoldFeverLeasing	rentItemIncludeTax()
GoldFeverLeasing.sol (L:231)	GoldFeverLeasing	finalizeLeaseItem()
GoldFeverMarket.sol (L:69)	GoldFeverMarket	setForwarderContract()
GoldFeverMarket.sol (L:76)	GoldFeverMarket	setMerchantContract()
GoldFeverMarket.sol (L:83)	GoldFeverMarket	setTaxationContract()
GoldFeverMarket.sol (L:109)	GoldFeverMarket	createListing()
GoldFeverMarket.sol (L:166)	GoldFeverMarket	cancelListing()
GoldFeverMarket.sol (L:194)	GoldFeverMarket	buyListingIncludeTax()
GoldFeverMask.sol (L:61)	GoldFeverMask	setForwarderContract()
GoldFeverMask.sol (L:133)	GoldFeverMask	forgeMask()
GoldFeverMask.sol (L:313)	GoldFeverMask	unforgeMask()
GoldFeverMask.sol (L:398)	GoldFeverMask	updateForgeMaskFee()
GoldFeverMask.sol (L:407)	GoldFeverMask	updateUnforgeMaskFee()
GoldFeverMask.sol (L:416)	GoldFeverMask	updatePurchaseMaskCost()
GoldFeverMask.sol (L:428)	GoldFeverMask	withdrawCollectedFee()
GoldFeverMask.sol (L:437)	GoldFeverMask	getForgeMaskFee()
GoldFeverMask.sol (L:441)	GoldFeverMask	getUnforgeMaskFee()
GoldFeverMask.sol (L:445)	GoldFeverMask	getPurchaseMaskCost()

GoldFeverMask.sol (L:449)	GoldFeverMask	purchaseMask()
GoldFeverMask.sol (L:469)	GoldFeverMask	setCommissionRate()
GoldFeverMask.sol (L:481)	GoldFeverMask	getCommissionRate()
GoldFeverMaskBox.sol (L:38)	GoldFeverMaskBox	setForwarderContract()
GoldFeverMaskBox.sol (L:89)	GoldFeverMaskBox	createBoxes()
GoldFeverMaskBox.sol (L:138)	GoldFeverMaskBox	openBox()
GoldFeverMerchantRight.sol (L:67)	GoldFeverMerchantRight	setItemTypeContract()
GoldFeverMerchantRight.sol (L:233)	GoldFeverMerchantRight	getMerchantLimit()
GoldFeverMerchantRight.sol (L:265)	GoldFeverMerchantRight	updateCompanionLimit()
GoldFeverMerchantRight.sol (L:294)	GoldFeverMerchantRight	updateBoatAndPlaneLimit()
GoldFeverMerchantRight.sol (L:322)	GoldFeverMerchantRight	updateBuildingLimit()
GoldFeverMiningClaim.sol (L:58)	GoldFeverMiningClaim	setForwarderContract()
GoldFeverMiningClaim.sol (L:161)	GoldFeverMiningClaim	createMiningClaim()
GoldFeverMiningClaim.sol (L:195)	GoldFeverMiningClaim	supply()
GoldFeverMiningClaim.sol (L:208)	GoldFeverMiningClaim	addArenaHour()
GoldFeverMiningClaim.sol (L:216)	GoldFeverMiningClaim	setArenaHour()
GoldFeverMiningClaim.sol (L:224)	GoldFeverMiningClaim	getArenaHour()
GoldFeverMiningClaim.sol (L:228)	GoldFeverMiningClaim	setArenaHourPrice()
GoldFeverMiningClaim.sol (L:235)	GoldFeverMiningClaim	getArenaHourPrice()
GoldFeverMiningClaim.sol (L:240)	GoldFeverMiningClaim	buyArenaHour()
GoldFeverMiningClaim.sol (L:256)	GoldFeverMiningClaim	getMiningSpeed()
GoldFeverMiningClaim.sol (L:265)	GoldFeverMiningClaim	setMiningSpeed()
GoldFeverMiningClaim.sol (L:272)	GoldFeverMiningClaim	getMaxMiners()

GoldFeverMiningClaim.sol (L:281)	GoldFeverMiningClaim	setMaxMiners()
GoldFeverMiningClaim.sol (L:289)	GoldFeverMiningClaim	startArena()
GoldFeverMiningClaim.sol (L:342)	GoldFeverMiningClaim	closeArena()
GoldFeverMiningClaim.sol (L:366)	GoldFeverMiningClaim	registerAtArena()
GoldFeverMiningClaim.sol (L:388)	GoldFeverMiningClaim	cancelArenaRegistration()
GoldFeverMiningClaim.sol (L:405)	GoldFeverMiningClaim	enterArena()
GoldFeverMiningClaim.sol (L:441)	GoldFeverMiningClaim	leaveArena()
GoldFeverMiningClaim.sol (L:455)	GoldFeverMiningClaim	bankWithdraw()
GoldFeverMiningClaim.sol (L:514)	GoldFeverMiningClaim	withdrawNglFromSellingHour()
GoldFeverMiningClaim.sol (L:523)	GoldFeverMiningClaim	getArenaIdByMiningClaimId()
GoldFeverNativeGold.sol (L:31)	GoldFeverNativeGold	setForwarderContract()
GoldFeverNpc.sol (L:150)	GoldFeverNpc	setForwarderContract()
GoldFeverNpc.sol (L:157)	GoldFeverNpc	setMerchantContract()
GoldFeverNpc.sol (L:164)	GoldFeverNpc	setItemTierContract()
GoldFeverNpc.sol (L:171)	GoldFeverNpc	setBuildingManagerContract()
GoldFeverNpc.sol (L:178)	GoldFeverNpc	setGovernorContract()
GoldFeverNpc.sol (L:185)	GoldFeverNpc	setNpcHiringFee()
GoldFeverNpc.sol (L:193)	GoldFeverNpc	setTierStatus()
GoldFeverNpc.sol (L:244)	GoldFeverNpc	createHiring()
GoldFeverNpc.sol (L:323)	GoldFeverNpc	renewHiring()
GoldFeverNpc.sol (L:350)	GoldFeverNpc	upgradeHiring()
GoldFeverNpc.sol (L:372)	GoldFeverNpc	depositItem()
GoldFeverNpc.sol (L:387)	GoldFeverNpc	requestWithdrawal()
GoldFeverNpc.sol (L:410)	GoldFeverNpc	approveWithdrawal()
GoldFeverNpc.sol (L:466)	GoldFeverNpc	approveWithdrawals()
GoldFeverNpc.sol (L:532)	GoldFeverNpc	withdrawEarning()

GoldFeverNpc.sol (L:546)	GoldFeverNpc	payFee()
GoldFeverNpc.sol (L:601)	GoldFeverNpc	payBuildingServiceFee()
GoldFeverNpc.sol (L:634)	GoldFeverNpc	setTicketFee()
GoldFeverNpc.sol (L:650)	GoldFeverNpc	setRentFee()
GoldFeverNpc.sol (L:666)	GoldFeverNpc	cancelHiring()
GoldFeverNpc.sol (L:706)	GoldFeverNpc	assignManager()
GoldFeverNpc.sol (L:726)	GoldFeverNpc	unassignManager()
GoldFeverNpc.sol (L:742)	GoldFeverNpc	getPendingEarning()
GoldFeverNpc.sol (L:746)	GoldFeverNpc	getPercentageLimit()
GoldFeverNpc.sol (L:755)	GoldFeverNpc	setRentPercentageLimit()
GoldFeverNpc.sol (L:762)	GoldFeverNpc	setTicketPercentageLimit()
GoldFeverNpc.sol (L:769)	GoldFeverNpc	feeDecimals()
GoldFeverNpc.sol (L:878)	GoldFeverNpc	withdrawCompanyEarning()
GoldFeverRight.sol (L:86)	GoldFeverRight	setForwarderContract()
GoldFeverRight.sol (L:112)	GoldFeverRight	createRightOption()
GoldFeverRight.sol (L:151)	GoldFeverRight	purchaseRight()
GoldFeverRight.sol (L:248)	GoldFeverRight	finalizeRightPurchase()
GoldFeverRight.sol (L:274)	GoldFeverRight	getRightPurchase()
GoldFeverRight.sol (L:300)	GoldFeverRight	getRightOption()
GoldFeverSwap.sol (L:68)	GoldFeverSwap	setForwarderContract()
GoldFeverSwap.sol (L:94)	GoldFeverSwap	createOffer()
GoldFeverSwap.sol (L:138)	GoldFeverSwap	cancelOffer()
GoldFeverSwap.sol (L:163)	GoldFeverSwap	rejectOffer()
GoldFeverSwap.sol (L:188)	GoldFeverSwap	acceptOffer()
GoldFeverSwap.sol (L:225)	GoldFeverSwap	getOffer()
GoldFeverTaxation.sol (L:21)	GoldFeverTaxation	setTaxCollector()

GoldFeverTaxation.sol (L:28)	GoldFeverTaxation	setCompanyTax()
NativeMetaTransaction.sol (L:37)	NativeMetaTransaction	executeMetaTransaction()
NativeMetaTransaction.sol (L:90)	NativeMetaTransaction	getNonce()

5.25.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function as shown in the following example:

GoldFeverAuction.sol

```
110 function setForwarderContract(address forwarder_)
111     external
112     only(DEFAULT_ADMIN_ROLE)
113 {
114     _setTrustedForwarder(forwarder_);
115 }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspeX

CYBERSECURITY PROFESSIONAL SERVICE