QuillAudits

Audit Report
August, 2022

For

BetSwirl

Table of Content

Executive Summary

Project Name	BetSwirl
Overview	BetSwirl is an online cryptocurrency gaming platform, fully decentralized and anonymous, where everyone will be able to enjoy a fair play, a fun time and an innovative gamer experience.
Timeline	July 4, 2022 - August 23, 2022
Method	Manual Review, Functional Testing, Automated Testing, etc.
Scope of Audit	The scope of this audit was to analyse Betswirl codebase for quality, security, and correctness. *https://github.com/BetSwirl/Smart-Contracts*
Initial Commit	46e638f8b0522d03cd212b2ed451d79526fb380c (Repo 1)
Fixed In	*https://github.com/BetSwirl/Smart-Contracts/commit/328f4aba2c265a0c5bbfffdbb62624daa63f341c*

17
Issues Found
🟥 High
🟨 Medium
🟩 Low
🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	1	2	0	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	2	3	7	2

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Compiler version not fixed
- ✓ Address hardcoded

Severus.finance - Audit Report

- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Using block.timestamp
- ✓ Multiple Sends
- ✓ Using SHA3
- ✓ Using suicide
- ✓ Using throw
- ✓ Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - Bank

High Severity Issues

A1. Centralization of payout

Description

Payout is used to pay a winning bet, and allocate the house edge fee. But this can only be called by GAME_ROLE admin. This could result in the admin denying the players their payouts for a long time and thus resulting in Denial of Service.

Moreover payout can be exploited if the GAME_ROLE is an EOA (Externally Owned Account) or is compromised by a malicious attacker. The attacker then chooses to payout and transfer as much profit he wants to transfer to himself.

Remediation

It is advised to review the business logic and make this function more decentralized so that there is less reliance on GAME_ROLE to make a payout to winning players as the GAME_ROLE can indefinitely delay the payout of winning players. A push and pull model can be used to transfer the fees but it is advised that the payout be automated in the smart contracts.

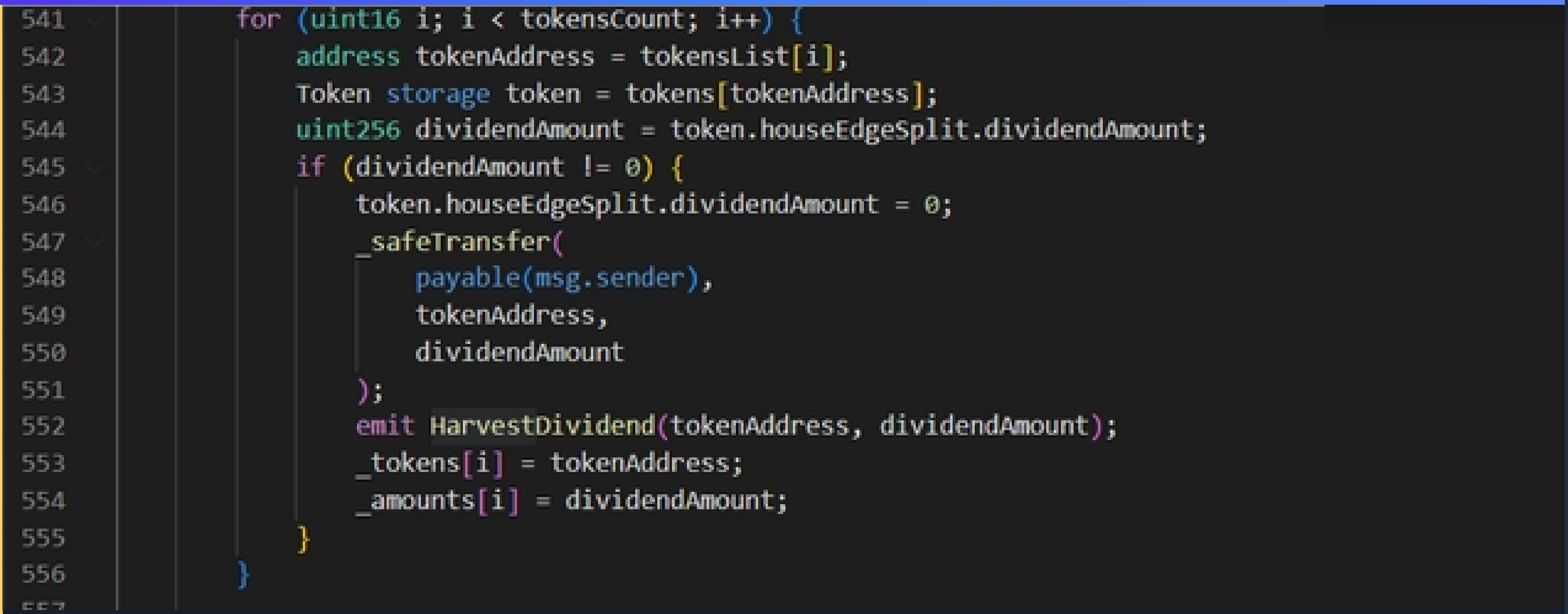
Status

Acknowledged

Client's' Comment: We acknowledge this centralization issue. Our Multi-sig owns the DEFAULT_ADMIN_ROLE which manages the GAME_ROLEs.

Medium Severity Issues

A2. Block gas limit



```
        for (uint16 i; i < tokensCount; i++) {
            address tokenAddress = tokensList[i];
            Token storage token = tokens[tokenAddress];
            uint256 dividendAmount = token.houseEdgeSplit.dividendAmount;
            if (dividendAmount != 0) {
                token.houseEdgeSplit.dividendAmount = 0;
                _safeTransfer(
                    payable(msg.sender),
                    tokenAddress,
                    dividendAmount
                );
                emit HarvestDividend(tokenAddress, dividendAmount);
                _tokens[i] = tokenAddress;
                _amounts[i] = dividendAmount;
            }
        }
```

Description

A for loop over dynamic array in harvestdividends() function exists. This can lead to out of gas issues if a lot of tokens are added via addTokens function.

Remediation

It is advised to add a limit on the number of tokens that can be added to avoid this issue.

Status

Acknowledged

Client's' Comment: We'll never have more tokens than the maximum based on the block size.

A3. Previous admin and/or partner can never withdraw

Description

If initially the admin of the contract who has the DEFAULT_ADMIN_ROLE, deposits a specific amount of token in the contract using the deposit function, he will not be able to withdraw it using withdraw() function.
For example, if the admin deposits 1000 tokens of say token A using deposit() function, then sets a partner for that token, the admin will lose all access to those 1000 tokens. Moreover the partner himself will be able to withdraw all the 1000 tokens instead. This scenario is also possible when an existing partner sets another new partner resulting in the new partner being able to withdraw all the tokens deposited by the previous partner for the same token.

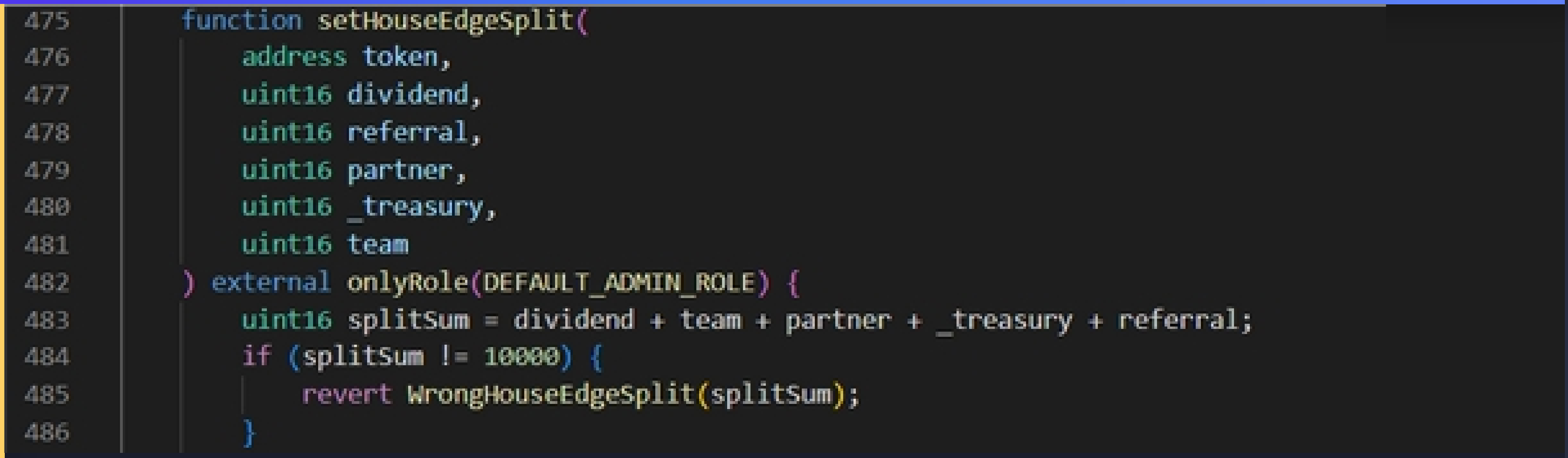
Remediation

Review the business logic and operational logic to avoid this issue. It is advised that previous depositors be notified that they will not be able to withdraw their amount once the partner is set and to withdraw their previous amount if they wish to. Special care should be taken to avoid this issue, because it could result in loss of the tokens for the previous admin and/or partner.

Status

Resolved

A4. Centralization of setHouseEdgeSplit()



```
    function setHouseEdgeSplit(
        address token,
        uint16 dividend,
        uint16 referral,
        uint16 partner,
        uint16 _treasury,
        uint16 team
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        uint16 splitSum = dividend + team + partner + _treasury + referral;
        if (splitSum != 10000) {
            revert WrongHouseEdgeSplit(splitSum);
        }
```

Description

setHouseedgeSplit determines the percentage of fees that each type of party will get. But this can be changed anytime by the admin without anyone agreeing or noticing the new changes and can thus lead to exploit to maximize profit.

Remediation

It is advised to add a timelock feature such as that of 1 day, to allow participants to opt out if they do not wish to participate in the new changes. Alternatively a DAO with voting and timelock mechanism can also be used in future for the same.

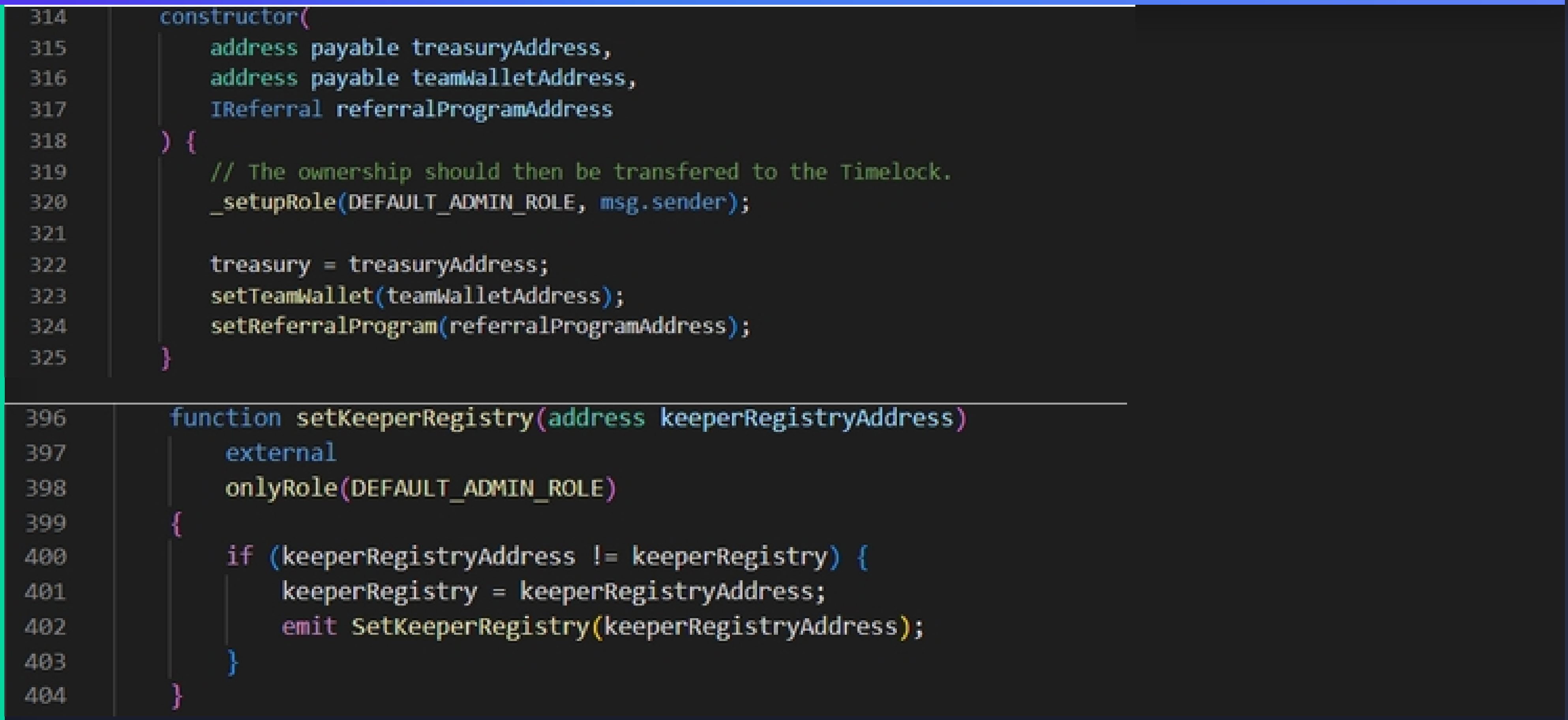
Status

Acknowledged

Comment: The client acknowledged this issue stating that this would be the case until the DAO level and that multisig is being currently utilized for the Bank contract (which currently has setHouseEdgeSplit() function).

Low Severity Issues

A5. Missing zero address checks



```
constructor(
    address payable treasuryAddress,
    address payable teamWalletAddress,
    IReferral referralProgramAddress
) {
    // The ownership should then be transfered to the Timelock.
    _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);

    treasury = treasuryAddress;
    setTeamWallet(teamWalletAddress);
    setReferralProgram(referralProgramAddress);
}
```

```
function setKeeperRegistry(address keeperRegistryAddress)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    if (keeperRegistryAddress != keeperRegistry) {
        keeperRegistry = keeperRegistryAddress;
        emit SetKeeperRegistry(keeperRegistryAddress);
    }
}
```

Description

There is a missing zero address check for the treasuryAddress parameter in the constructor. If manageBalanceOverflow() is called and treasury is a zero address, it could lead to funds being burnt.
Also there is a missing zero address check for keeperRegistryAddress parameter in setKeeperRegistry() function.

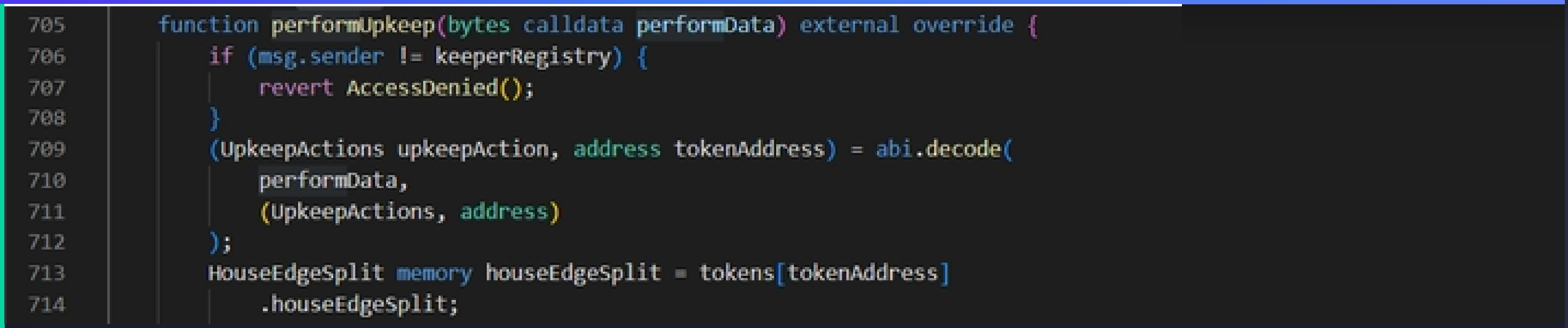
Remediation

It is advised to review the business logic and add the require checks if necessary.

Status

Resolved

A6. Not everyone can call performUpKeep()



```
function performUpkeep(bytes calldata performData) external override {
    if (msg.sender != keeperRegistry) {
        revert AccessDenied();
    }
    (UpkeepActions upkeepAction, address tokenAddress) = abi.decode(
        performData,
        (UpkeepActions, address)
    );
    HouseEdgeSplit memory houseEdgeSplit = tokens[tokenAddress]
        .houseEdgeSplit;
```

Description

According to comments on Chainlink keepers on the line: 29 on Github, anyone should be able to call the performUpkeep() function. But the performUpkeep() function in the Bank contract has been restricted to be called only by the keeperRegistry.

Remediation

It is advised to remove this restriction to allow anyone to call this function and make the necessary changes to be consistent with the Chainlink's recommendation in the comments.
https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/interfaces/KeeperCompatibleInterface.sol

Status

Resolved

A7. Partner is impossible to recover if access is lost

Description

If the admin accidentally sets the partner of a token to an address which no one has access to, it would be impossible to reset or change the partner address of that token forever. For example, there is token A and the admin sets a wrong address of the partner, such that the owner of that token is not having access to, it would never be possible again to set the correct address of that particular token. This scenario is also possible when an existing partner sets another new partner.

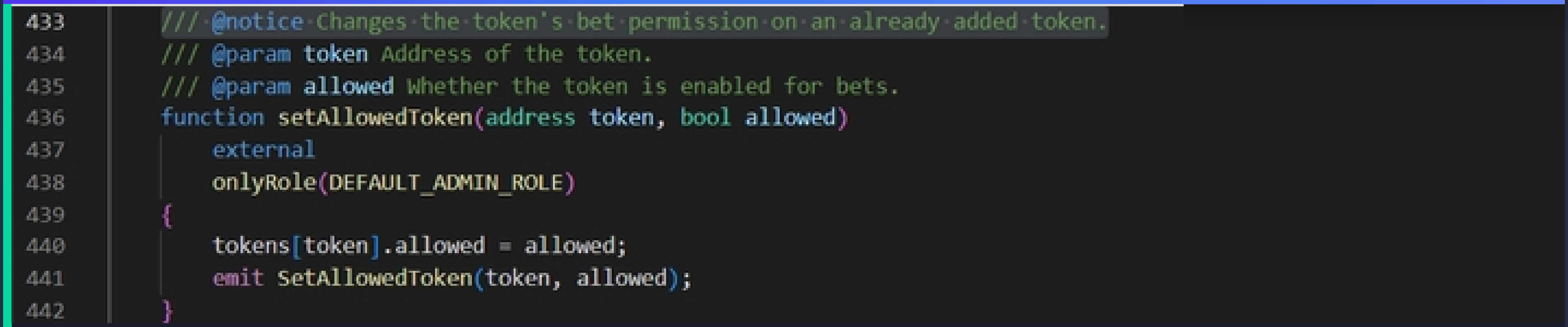
Recommendation

It is advised to make setting of partner address a two step process in which the first step is to propose to the new user the partnership of the token and the second step is the partnership acceptance step in which the user has to accept the partnership proposed to him. It is also advised to allow resetting/revoking of the partner address if the partnership acceptance by that address is pending for a long time thus mitigating the issue mentioned above.

Status

Resolved

A8. Inconsistency with comment and operational logic



```
433     /// @notice Changes the token's bet permission on an already added token.
434     /// @param token Address of the token.
435     /// @param allowed Whether the token is enabled for bets.
436     function setAllowedToken(address token, bool allowed)
437         external
438         onlyRole(DEFAULT_ADMIN_ROLE)
439     {
440         tokens[token].allowed = allowed;
441         emit SetAllowedToken(token, allowed);
442     }
```

Description

The comment on the line: 433 says
/// @notice Changes the token's bet permission on an already added token.

But there is no guarantee that a token would already be added because it is possible to use setAllowedToken() to enable a token for bets without actually adding the token using addToken(). For example it is possible to enable token X for betting by using setAllowedToken() without adding token X to tokensList using addToken(). This could lead to unintended issues.

Remediation

It is advised to either modify the code so that this scenario does not occur or review the business and operational logic and change the code comments accordingly.

Status

Resolved

Comment: The client changed the comment to be consistent with the operational and business logic.

Informational Issues

No issues found

B. Contract - Game

High Severity Issues

No issues found

Medium Severity Issues

B1. Denial of Service and Centralization Issue

Description

refundBet() allows the users to claim a refund in case of failed Chainlink VRF callback. But a malicious admin can use inCaseTokensGetStuck() function to withdraw all the refundable funds thus denying the users of any refund.

Recommendation

It is advised to allow inCaseTokensGetStuck() to be called only when all users have been refunded their amounts or a minimum percentage of users whose refunds are pending have been refunded their amount. Add necessary checks to ensure that inCaseTokensGetStuck() is not exploited.

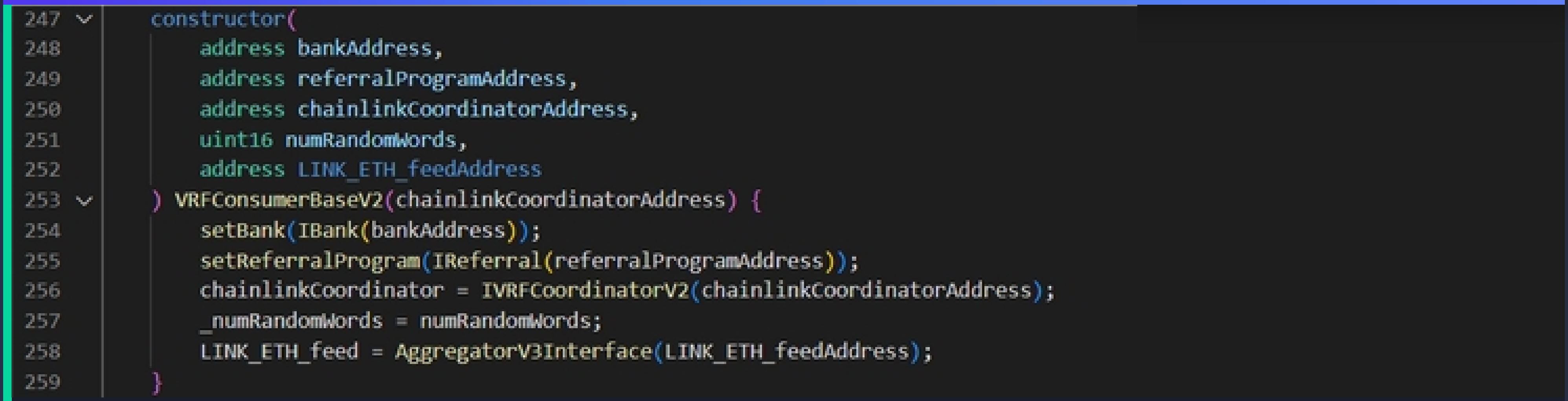
Status

Resolved

Comment: The function inCaseTokensGetStuck() has been removed from the contract.

Low Severity Issues

B2. Missing zero address checks



```
constructor(
    address bankAddress,
    address referralProgramAddress,
    address chainlinkCoordinatorAddress,
    uint16 numRandomWords,
    address LINK_ETH_feedAddress
) VRFConsumerBaseV2(chainlinkCoordinatorAddress) {
    setBank(IBank(bankAddress));
    setReferralProgram(IReferral(referralProgramAddress));
    chainlinkCoordinator = IVRFCoordinatorV2(chainlinkCoordinatorAddress);
    _numRandomWords = numRandomWords;
    LINK_ETH_feed = AggregatorV3Interface(LINK_ETH_feedAddress);
}
```

Description

There is a missing zero address check for LINK_ETH_feedAddress in the constructor.

Recommendation

It is advised to review the business logic and add the require checks if necessary.

Status

Resolved

B3. Ownership Renouncement

Description

When the renounceOwnership function is called accidentally by the admin of the contract, the contract immediately renounces ownership to address zero, after which it makes it impossible for the owner to call the admin onlyOwner functions.

Recommendation

It is recommended to override the renounce owner functionality to prevent the owner from calling this function or could fashion the renounce owner to work if one has successfully transferred ownership to the right address. If a multi-sig is to be utilized, it should be confirmed that two or more persons are required to sign before the execution of the renounce ownership function.

Status

Resolved

Comment: The client said that they would be utilizing multisig wallets for the same on each chain.

B4. Transfer Ownership Should be a two step process

Description

The transferOwnership() function in contract allows the current admin to transfer his privileges to another address. However, inside transferOwnership() , the newOwner is directly stored into the storage owner, after validating that the newOwner is a non-zero address, and immediately overwrites the current owner. This can lead to cases where the admin has transferred ownership to an incorrect address and wants to revoke the transfer of ownership or in the cases where the current admin comes to know that the new admin has lost access to his account.

Recommendation

Consider making transferOwnership a two step process in which the first step is to propose to the new owner the ownership of the contract and the second step is the ownership acceptance step in which the owner has to accept the ownership proposed to him.

Status

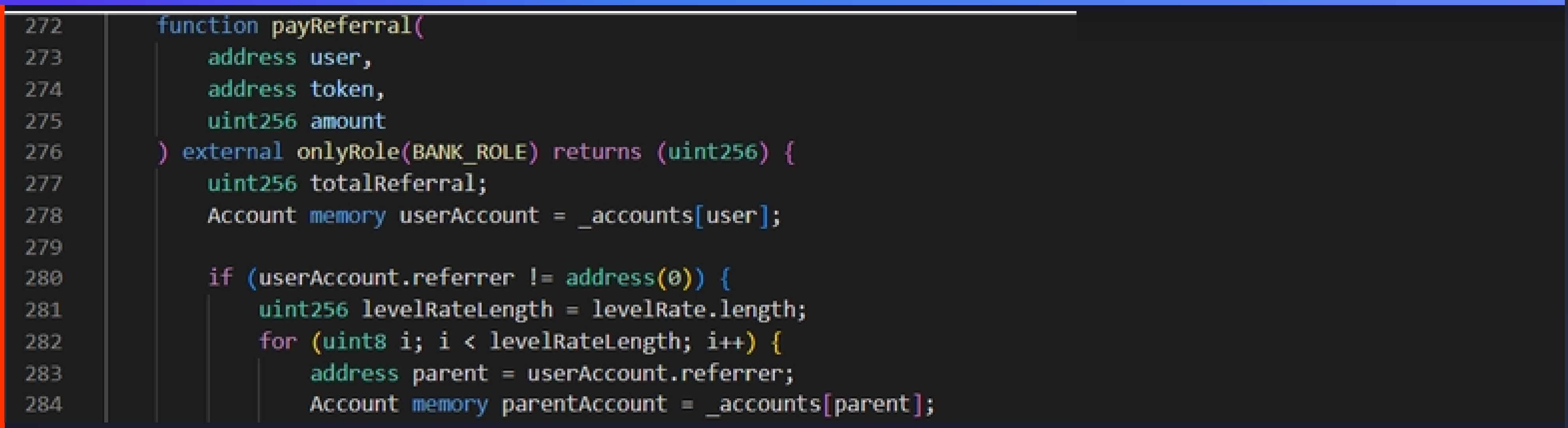
Resolved

Client comment: We are using multi-sig, so a double check is made by the "transferOwnership" transaction signatories and executor.

C. Contract - Referral

High Severity Issues

C1. Centralization of PayReferral



```
function payReferral(
    address user,
    address token,
    uint256 amount
) external onlyRole(BANK_ROLE) returns (uint256) {
    uint256 totalReferral;
    Account memory userAccount = _accounts[user];

    if (userAccount.referrer != address(0)) {
        uint256 levelRateLength = levelRate.length;
        for (uint8 i; i < levelRateLength; i++) {
            address parent = userAccount.referrer;
            Account memory parentAccount = _accounts[parent];
```

Description

payReferral can be exploited if the BANK_ROLE is an EOA(Externally Owned Account) or is compromised by a malicious attacker. The attacker then choose to payReferral to himself and add as much credit he wants to add to himself, thereby withdrawing the credits afterwards using withdrawCredits() and draining the contract.

Remediation

It is advised to review the business and operational logic to make the execution of this function more decentralized.

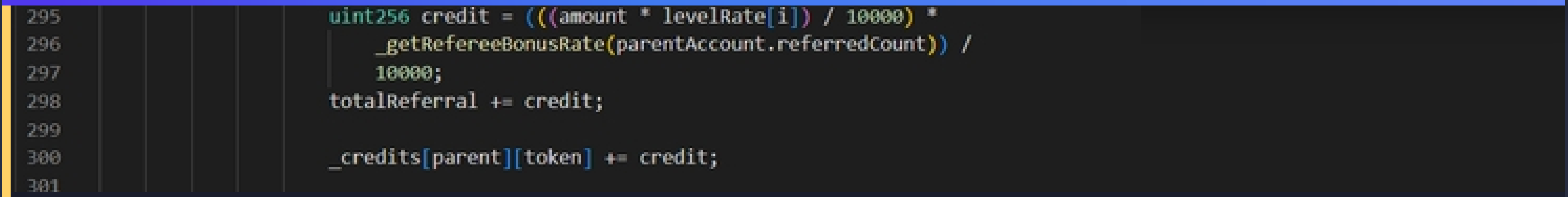
Status

Resolved

Comment: Referral code and logic has been removed from the contracts entirely.

Medium Severity Issues

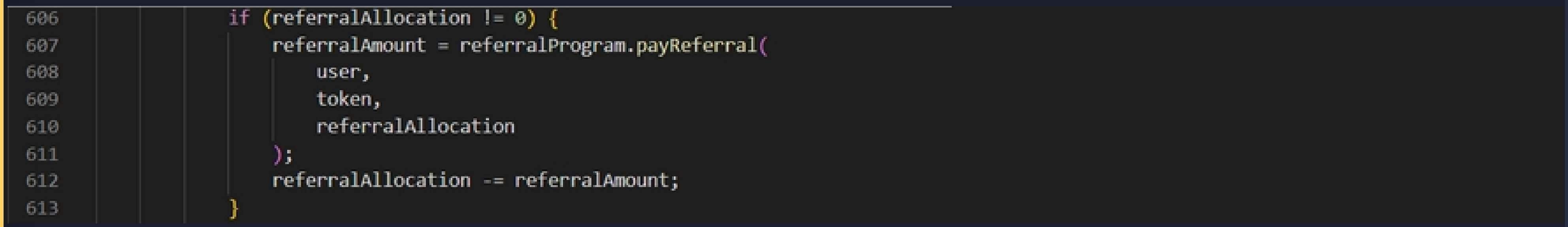
B2. Division before multiplication



```
295    uint256 credit = (((amount * levelRate[i]) / 10000) *
296        _getRefereeBonusRate(parentAccount.referredCount)) /
297        10000;
298    totalReferral += credit;
299
300    _credits[parent][token] += credit;
301
```

Description

On line: 295, there is division carried out before multiplication. If there is division carried out before multiplication such as in this case, it could result in loss of precision thus resulting in loss of value. If amount*levelRate is lesser than 10000, the result of it will be zero. On the other hand if multiplication would have been carried out before division by 10000, it would have resulted in a non-zero number. For example, let's say a token with 6 decimal places such as USDC is added for betting. And if it has minimum bet amount of 1000, then amount here would be nothing but
referralAllocation = (fees * tokenHouseEdge.referral) /10000;
which is passed on line: 610 to payReferral.



```
606    if (referralAllocation != 0) {
607        referralAmount = referralProgram.payReferral(
608            user,
609            token,
610            referralAllocation
611        );
612        referralAllocation -= referralAmount;
613    }
```

When we take another look at fees, it would be evident that referralAllocation could be a small number let's say 50 or so. This when multiplied with levelRate could result in a number less than 10000 such as 5000. Thus 5000/10000 would result in zero. If _getRefereeBonusRate() was multiplied with amount and levelRate before being divided by 10000, this may have resulted in a non-zero number.

Remediation

It is recommended to refactor the formula to
*(((amount * levelRate[i])) * _getRefereeBonusRate(parentAccount.referredCount) / 10000) / 10000*
to avoid any loss of value.

Status

Resolved

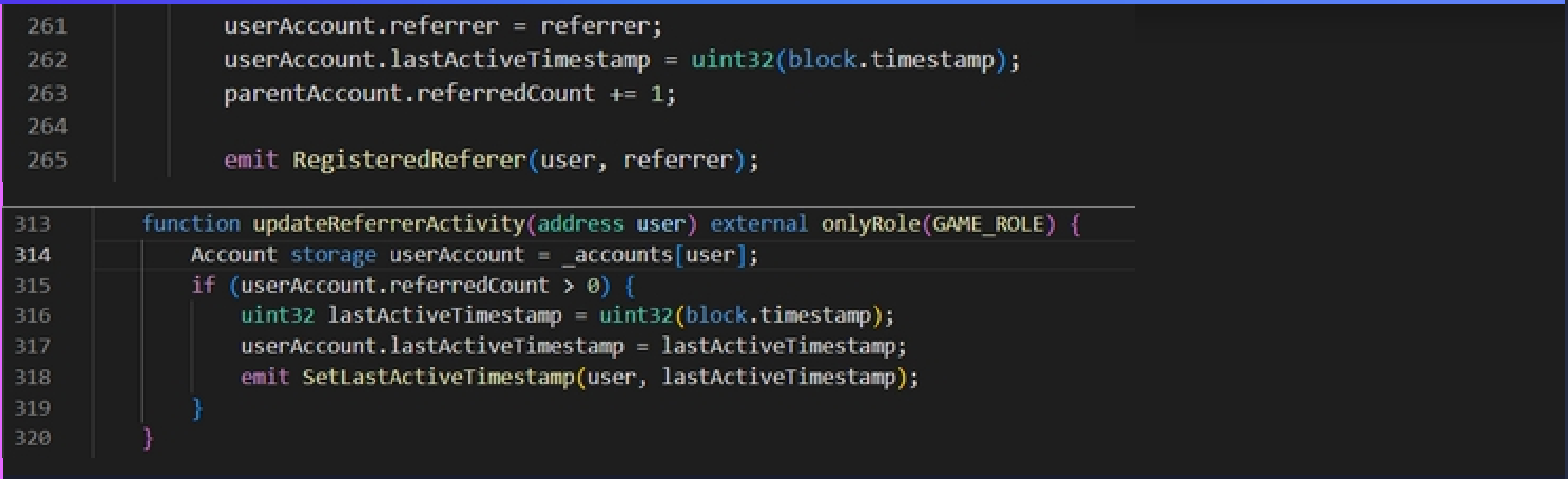
Client's’ Comment: Referral code and logic has been removed from the contracts entirely.

Low Severity Issues

No issues found

Informational Issues

B3. Unsafe downcasting to Uint32



```
        userAccount.referrer = referrer;
        userAccount.lastActiveTimestamp = uint32(block.timestamp);
        parentAccount.referredCount += 1;

        emit RegisteredReferer(user, referrer);
```

```
    function updateReferrerActivity(address user) external onlyRole(GAME_ROLE) {
        Account storage userAccount = _accounts[user];
        if (userAccount.referredCount > 0) {
            uint32 lastActiveTimestamp = uint32(block.timestamp);
            userAccount.lastActiveTimestamp = lastActiveTimestamp;
            emit SetLastActiveTimestamp(user, lastActiveTimestamp);
        }
    }
```

Description

There is unsafe downcasting done from uint to uint32. In case of overflows, Solidity does not revert on overflow which could lead to undesired bugs.

Recommendation

Although there are little chances for this to occur, it would be a best practice to use safecasting such as Openzeppelin’s Safecast library to safely downcast the value.

Status

Resolved

Comment: Referral code and logic has been removed from the contracts entirely.

D. Contract - CoinToss

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

E. Contract - Dice

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

F. Contract - Roulette

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

G. Common Issues

High Severity Issues

G1. Usage of Unsafe Low-level Calls

Description

Address.sendValue has been used throughout the codebase which utilizes low level calls. This coupled with not following the checks-effects interactions pattern can lead to some undiscovered bugs such as complex cross-function reentrant attacks if the functions share a common state between them (even non-reentrant modifier cannot mitigate such attacks). The following scenarios and usage of low level calls are recommended to be resolved
1) Address.sendValue which is a low-level call is used On line: 316 in _newBet() function, on line: 608 and line: 620 in refundBet() function.
2) _safeTransfer which utilizes Address.sendValue which uses a low-level call is used in payout() function on line: 661.
3) user.call on line: 312 in _resolveBet() function which is called in fulfillRandomWords() function.

Remediation

It is recommended to not use low level calls as they forward all gas and use the generic transfer or send. An attacker can use complex calls and execute fallback functions to exploit when the control is transferred to him via a low level call. Transfer and send forward minimal gas which does not allow exploits like reentrant attacks to occur.

Status

Resolved

Medium Severity Issues

No issues found

Low Severity Issues

No issues found

Informational Issues

G2. Floating compiler version

Description

There is a floating and unlocked solidity compiler version used throughout the codebase. This can lead to testing and debugging in a different compiler version and deploying the contract in a different version.

Recommendation

It is advised to fix the compiler version for each of the contract to avoid this issue.

Status

Resolved

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Betswirl. We performed our audit according to the procedure described above.

Some issues of High,Medium, Low and informational severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture.In the End, the majority of the issues have been resolved, and the BetSwirl team has acknowledged a few issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Betswirl Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Betswirl Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

500+
Audits Completed

$15B
Secured

500K
Lines of Code Audited

Follow Our Journey

Audit Report
August, 2022

For

BetSwirl

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com