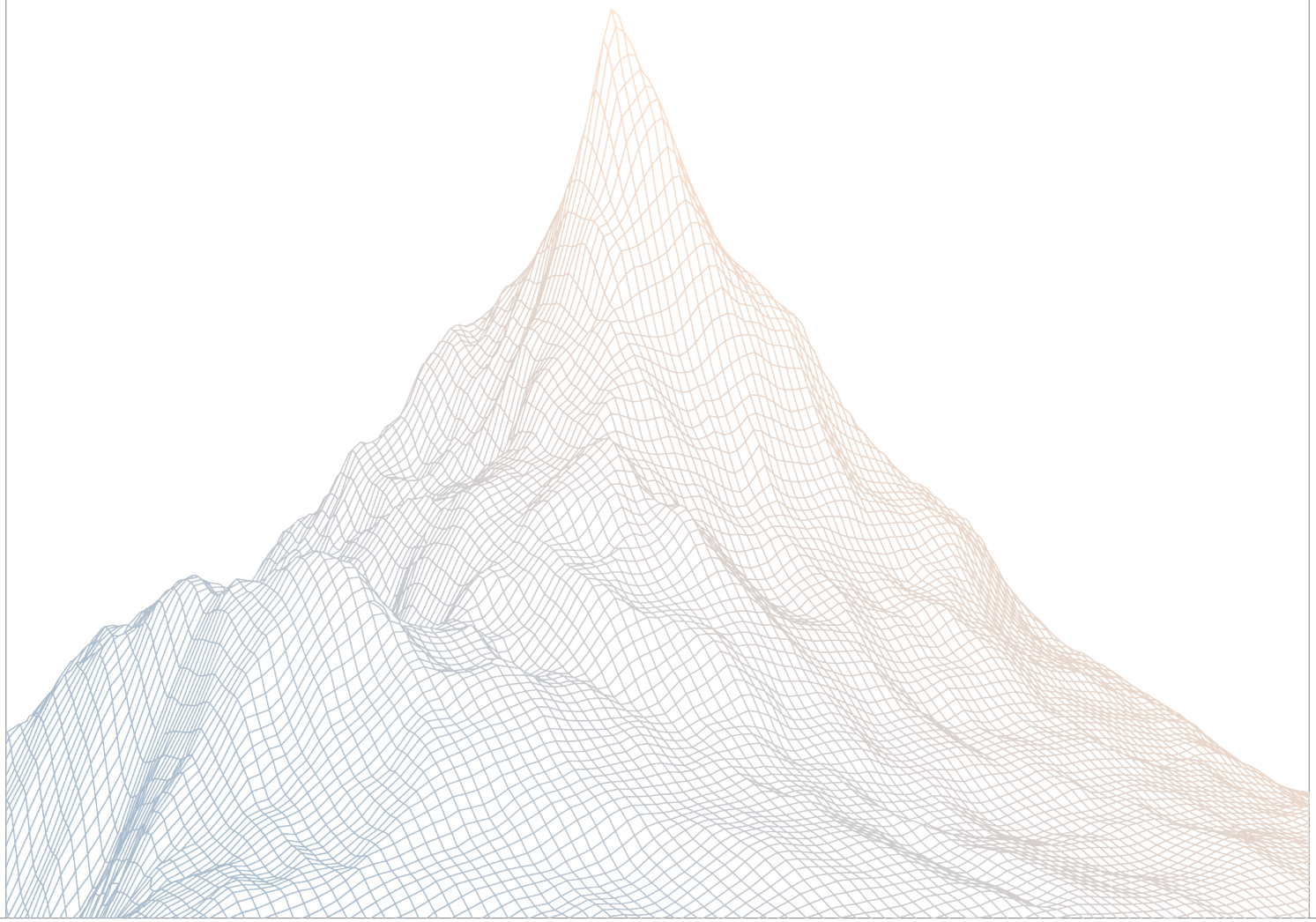
Zenith

Bro.fun

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: October 20th to October 21st, 2025

AUDITED BY: Arno
valuevalk

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Bro.fun

Bro.fun is a community-driven gaming platform that empowers anyone to create a community, send invites, and play games together. The platform is browser-based allowing for seamless access. Groups can string together a series of games from our ever-growing game library.

2.2 Scope

The engagement involved a review of the following targets:

Target	monad-contract-audit
Repository	https://github.com/Internet-Game/monad-contract-audit
Commit Hash	`95f5f6d71b5f7cd94659ed0463325724f185828b`
Files	`Monad.sol`

2.3 Audit Timeline

October 20, 2025	Audit start
October 21, 2025	Audit end
October 29, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	2
Informational	6
Total Issues	**9**

3

Findings Summary

ID	Description	Status
M-1	claimRakeback uses the same nonce mapping - referralNonces as claimReferral, which could lead to collisions	Resolved
L-1	Signature scheme lacks EIP-712 standard compliance	Acknowledged
L-2	Lack of _disableInitializers call in the constructor allows anyone to become owner of the implementation Contract, which could be used for honeypot/phishing and to steal leftover funds	Resolved
I-1	Unrestricted withdrawFunds lets owner withdraw winners' funds	Acknowledged
I-2	Incorrect import and usage of the OZ dependency	Acknowledged
I-3	Missing Explicit Balance Check in cashOut	Acknowledged
I-4	Missing event emission on critical owner-controlled state change	Acknowledged
I-5	Unused library and declared events can be removed	Resolved
I-6	If TransperantUpgradeable proxy is used, the deployer should never be the owner of the Monad.sol contract	Acknowledged

4

Findings

4.1 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] claimRakeback uses the same nonce mapping - `referralNonces` as claimReferral, which could lead to collisions

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- Monad.sol#L56-L57
- Monad.sol#L333
- Monad.sol#L365

Description:

Currently we are using the exact same mapping `mapping(address ⇒ uint256) public referralNonces;` for both `claimRakeback()` and `claimReferral()` functions:

```
    function claimReferral(uint256 amount, uint256 deadline,
    bytes calldata serverSignature) external nonReentrant {
        if (block.timestamp > deadline) revert SignatureExpired();
        if (amount == 0) revert ZeroPayoutAmount();
        if (amount > address(this).balance) revert InsufficientBalance();

        bytes32 messageHash =
@>>         keccak256(abi.encodePacked(claimReferralPrefix, amount,
    msg.sender, referralNonces[msg.sender], deadline));
@>>   referralNonces[msg.sender] += 1;
        _verifyAnyAdminSignature(messageHash, serverSignature);
        address recipient = msg.sender;
        (bool success,) = recipient.call{value: amount}("");
        if (!success) revert ReferralPaymentFailed();
        emit ReferralPaid(recipient, amount);
    }
```

and

```
    function claimRakeback(uint256 amount, uint256 deadline,
    bytes calldata serverSignature) external nonReentrant {
        if (block.timestamp > deadline) revert SignatureExpired();
        if (amount == 0) revert ZeroPayoutAmount();
        if (amount > address(this).balance) revert InsufficientBalance();

        bytes32 messageHash =
@>>         keccak256(abi.encodePacked(claimRakebackPrefix, amount,
    msg.sender, referralNonces[msg.sender], deadline));
@>>     referralNonces[msg.sender] += 1;
        _verifyAnyAdminSignature(messageHash, serverSignature);
        (bool success,) = payable(msg.sender).call{value: amount}("");
        if (!success) revert RakebackPaymentFailed();
        emit RakebackPaid(msg.sender, amount);
    }
```

Its originally marked that those nonces are for the referral only (as can be seen from the name as well):

```
@>> /// @notice Nonces for referral claims to prevent signature replay
--
@>> mapping(address => uint256) public referralNonces;
```

This can result in a potentially bad case:

If off-chain mechanism doesn't properly account that rakeback and referral use the same nonce, this could render some of the interactions useless and could result in a Player not being able to claim either rakeback or referral: Path:

1. Server issues rakeback with nonce 1 and referral with nonce 1
2. Player claims rakeback and nonce increases to 2 (in the contract)
3. Player cannot claim the referral because it needs to match the current nonce which is 2, but the signature is made with nonce 1.

This could be handled off-chain, if the server uses the same nonce for both. This is not fatal as the admin could re-issue the signature and still give the funds to the owed Player.

This could also have trivial impact on the UX, because the player has to jump back and forth between the functions in the order of the nonces.

Recommendations:

Decide if you would like to keep the nonces in 1 mapping and properly handle the collision case off-chain in the server side or if you would like to divide it into 2.

If you decide to keep it as 1: Rename the mapping to represent its actual purpose:

```
/// @notice Nonces for referral claims to prevent signature replay
mapping(address ⇒ uint256) public referralNonces;

/// @notice Nonces for referral & rakeback claims to prevent signature
    replay
mapping(address ⇒ uint256) public claimNonces;
```

Otherwise create 2 mappings and track them separately off-chain:

```
/// @notice Nonces for referral claims to prevent signature replay
mapping(address ⇒ uint256) public referralNonces;
/// @notice Nonces for referral claims to prevent signature replay
mapping(address ⇒ uint256) public referralNonces;

/// @notice Nonces for rakeback claims to prevent signature replay
mapping(address ⇒ uint256) public rakebackNonces;
```

After modifying the mapping, modify its reference in the claim functions as well.

Bro.fun: Resolved with `@9bdd66f072 ...`

Zenith: Verified.

4.2 Low Risk

A total of 2 low risk findings were identified.

[L-1] Signature scheme lacks EIP-712 standard compliance

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- monad.sol

Description:

The contract's signature verification in `_verifyAnyAdminSignature` uses manual `abi.encodePacked` concatenation and `ECDSA.toEthSignedMessageHash` (eth_sign style) across functions like `createGame`, `cashOut`, `markGameAsLost`, `claimReferral`, and `claimRakeback`. This lacks EIP-712 type safety and UX benefits. Critically, hashes omit `address(this)` and `block.chainid`, enabling replay attacks across contracts or chains if `messagePrefix` is reused. Eth_sign displays raw hex in wallets, increasing phishing risks.

Recommendations:

Switch to EIP-712 for typed data, including proper domain separation. Example implementation:

```
// Add to contract state
bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string
    name,string version,uint256 chainId,address verifyingContract)");
bytes32 public constant CREATE_GAME_TYPEHASH = keccak256("CreateGame(bytes32
    preliminaryGameId,bytes32 commitmentHash,address player,uint256
    betAmount,uint256 deadline)");
bytes32 public DOMAIN_SEPARATOR; // Set in initialize:
    keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256("Monad"),
    keccak256("1"), block.chainid, address(this)));

// In createGame (adapt similarly for others)
```

```
bytes32 structHash = keccak256(abi.encode(CREATE_GAME_TYPEHASH,
    preliminaryGameId, commitmentHash, msg.sender, msg.value, deadline));
bytes32 messageHash = keccak256(abi.encodePacked("\x19\x01",
    DOMAIN_SEPARATOR, structHash));
```

This binds signatures to the contract and chain, prevents replays, and improves wallet display. Use `abi.encode` over `encodePacked` for consistency.

Bro.fun: Acknowledged. The message prefix and signer will be different per chain.

[L-2] Lack of `_disableInitializers` call in the constructor allows anyone to become owner of the implementation Contract, which could be used for honeypot/phishing and to steal leftover funds

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- Monad.sol#L154-L166

Description:

`Monad.sol` is supposed to sit behind a proxy and has `initialize()` function, which comes with the `initializer` modifier from `Initializable.sol`, currently we lack the `_disableInitializers();` call, which allows an attacker to take over the implementation contract.

See docs of OZ: https://docs.openzeppelin.com/upgrades-plugins/writing-upgradeable

> Do not leave an implementation contract uninitialized. An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy. To prevent the implementation contract from being used, you should invoke the _disableInitializers function in the constructor to automatically lock it when it is deployed

In the current contract an attacker cannot do anything to the proxy, but they could us it for phishing/honeypot attacks and potentially force users to make bets and deposit native-tokens.

The attacker can also withdraw any funds donated/leftout in the `Monad.sol` contract's implementation.

Attack flow:

1. Proxy is created and it calls the initialize() function during its creation and links to the implementation address, for example `address(Monad)`
2. Users should use `address(Proxy)`
3. However any attacker is free to obtain full access to the actual legit implementation, i.e. `address(Monad)`, because initialiaze is free to be called directly towards the

implementation, because `_disableInitializers();` is missing in the `Monad.sol`'s constructor

Recommendations:

Add this into the contract:

```
constructor() {
    _disableInitializers();
}
```

Bro.fun: Resolved with `@f5abc3b6e6 ...`

Zenith: Verified.

4.3 Informational

A total of 6 informational findings were identified.

[I-1] Unrestricted `withdrawFunds` lets owner withdraw winners' funds

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- Monad.sol

Description:

`withdrawFunds` has no limit or accounting for user liabilities. The owner can withdraw the entire contract balance, including funds needed to pay winners/unclaimed prizes.

Recommendations:

Track obligations (e.g., totalWinningsOwed) and restrict withdrawals to excess funds only.

Bro.fun: Acknowledged.

[I-2] Incorrect import and usage of the OZ dependency

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- monad.sol

Description:

In OZ v5, `toEthSignedMessageHash` lives in `MessageHashUtils`, not in ECDSA

```
function _verifyAnyAdminSignature(bytes32 _hash, bytes calldata _signature)
    internal view {
  // Use OpenZeppelin's ECDSA library for robust verification
  bytes32 prefixedHash = ECDSA.toEthSignedMessageHash(_hash);
  address recoveredSigner = ECDSA.recover(prefixedHash, _signature);
  if (recoveredSigner == address(0) || !isAdmin[recoveredSigner]) {
    revert InvalidServerSignature();
  }
}
```

MessageHashUtils.sol#L48

Recommendations:

Import MessageHashUtils and call MessageHashUtils.toEthSignedMessageHash.

You could also consider using (sub V5), for example V4.9 for the non-upgradeable OZ contracts dependency, then no code change is needed, as the `ECDSA.sol` will contain the `toEthSignedMessageHash` but its cleaner to keep 1 version for both

Bro.fun: Acknowledged.

[I-3] Missing Explicit Balance Check in cashOut

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- monad.sol

Description:

The `cashOut` function attempts to transfer `payoutAmount` without explicitly checking if `address(this).balance ⩾ payoutAmount`. If insufficient, the low-level call fails and reverts with `PayoutFailed`, but this is inconsistent with explicit `InsufficientBalance` checks in `claimReferral`, `claimRakeback`, and `withdrawFunds`. This can lead to unclear errors.

Recommendations:

Add an explicit check before the transfer for consistency and better error handling:

```
// In cashOut, after line 246 (if (payoutAmount == 0) revert
    ZeroPayoutAmount();)
if (payoutAmount > address(this).balance) revert InsufficientBalance();
```

Bro.fun: Resolved with @ce38407a6a ...

Zenith: Verified.

[I-4] Missing event emission on critical owner-controlled state change

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- monad.sol

Description:

In `monad.sol`, `setGameCounter()` and `withdrawFunds()` modify critical state (`gameCounter` for sequencing, contract balance for treasury) without emitting events.

Recommendations:

Emit events to log changes. Proposed additions:

```
// Add events near other events
event GameCounterUpdated(uint256 previousCounter, uint256 newCounter,
    address indexed updater);
event FundsWithdrawn(address indexed initiator, address indexed recipient,
    uint256 amount);
```

```
// In setGameCounter()
function setGameCounter(uint256 newCounter) external onlyOwner {
    require(newCounter >= gameCounter, 'Cannot decrease gameCounter');
    uint256 previousCounter = gameCounter;
    gameCounter = newCounter;
    emit GameCounterUpdated(previousCounter, newCounter, msg.sender);
}
```

```
// In withdrawFunds() (around line 412)
(bool success, ) = recipient.call{value: amount}('');
require(success, 'Withdrawal failed');
emit FundsWithdrawn(msg.sender, recipient, amount);
```

Bro.fun: Acknowledged.

[I-5] Unused library and declared events can be removed

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Monad.sol#L20

Description:

The following library is imported and is set to `uint256` but it is never used in the logic:

```
contract Monad is Initializable, OwnableUpgradeable,
    ReentrancyGuardUpgradeable {
@>>     using Strings for uint256; //@audit this is unused
```

Additionally the following event is also unused

```
error NotGamePlayer(uint256 onChainGameId, address caller);
```

Both can be removed.

Recommendations:

```
import "@openzeppelin/contracts/utils/Strings.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol"; // Import
    ECDSA

/**
 * @title Monad
 * @dev Smart contract to manage Monad game
 *
 * The contract handles game creation, payout management, and provable
    fairness
 * for the Monad game, where players throw balls at targets to maximize
 * their potential winnings. Requires server-signed parameters for critical
    actions.
```

```
 */
contract Monad is Initializable, OwnableUpgradeable,
    ReentrancyGuardUpgradeable {
    using Strings for uint256;
```

```
    /// @notice Error when caller is not the player
error NotGamePlayer(uint256 onChainGameId, address caller);
```

Bro.fun: Resolved with @898a15f207 ...

Zenith: Verified.

[I-6] If TransperantUpgradeable proxy is used, the deployer should never be the owner of the Monad.sol contract

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- Monad.sol#L154-L155

Description:

Since the contract lacks normal definitions such as `authorizeUpgrade`, which are usually associated with `UUPSUpgradeable` schema, its likely that the other most popular option, i.e. `TransparentUpgradeableProxy` could be used for the proxy schema.

However there is a crucial invariant that must not be broken, when using `TransparentUpgradeableProxy`

The **owner** of the proxy, **cannot be** the **owner** of the contract.

Currently we enforce that the `owner` of the contract is the `msg.sender`, i.e. the deployer of the proxy contract. (because `initialize` is called DURING the creation of the proxy)

```
    function initialize(string memory _messagePrefix) public initializer {
@>>     __Ownable_init(msg.sender);
        __ReentrancyGuard_init();
        gameCounter = 0;
        messagePrefix = _messagePrefix;
        createGamePrefix = keccak256(abi.encodePacked(messagePrefix,
    ":createGame"));
        cashOutPrefix = keccak256(abi.encodePacked(messagePrefix,
    ":cashOut"));
        markGameAsLostPrefix = keccak256(abi.encodePacked(messagePrefix,
    ":markGameAsLost"));
        claimReferralPrefix = keccak256(abi.encodePacked(messagePrefix,
    ":claimReferral"));
        claimRakebackPrefix = keccak256(abi.encodePacked(messagePrefix,
    ":claimRakeback"));
        isAdmin[msg.sender] = true;
        emit AdminAdded(msg.sender);
    }
```

There are `TransparentUpgradeableProxy` implementations which enforce that the `msg.sender`, i.e. the deployer is **the proxy-owner**, it should be carefully checked that the proxy-owner **is not the deployer**, when using a specific proxy implementation. (others allow for you to set it, but in this case you should reserve a second admin and not use the same one, i.e. NOT the deployer)

The deployer will be set as owner of the `Monad` contract as this is already enforced.

If a case where they match happens, the admin functionality in the implementation contract would be unreachable: TransparentUpgradeableProxy.sol#L95-L105

```
    /**
@>>     * @dev If caller is the admin process the call internally, otherwise
    transparently fallback to the proxy behavior.
     */
    function _fallback() internal virtual override {
        if (msg.sender == _proxyAdmin()) {
@>>         if (msg.sig ≠
    ITransparentUpgradeableProxy.upgradeToAndCall.selector) {
                revert ProxyDeniedAdminAccess();
            } else {
                _dispatchUpgradeToAndCall();
            }
        } else {
            super._fallback();
        }
    }
```

Recommendations:

Be cautious with the described case and which `TransparentUpgradeableProxy,sol` you use and how you use it.

Potentially you could also change the `__Ownable_init(msg.sender);` to use a function-param passed `owner`, instead of setting it to `msg.sender`, to explicitly allow for any value to be set there.

Bro.fun: Acknowledged.