BLOCKSEC

Security Audit Report for monad-game-contract

Date: November 28, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	dapdap
Target	monad-game-contract

Version History

Version	Date	Description
1.0	November 28, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-11-28 13:02:02+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of monad-game-contract of dapdap.

This audit covers contracts corresponding to three distinct game systems. The contract `SpaceInvaders` serves as the on-chain settlement layer for an adjudicated betting game, which implements a manager-trusted model. The initial game state is cryptographically committed and stored on-chain, and the final reward distribution is authorized via manager signatures. The other two contracts, `ChartVoyager` and `lucky777`, are both responsible for handling user deposits and administrator withdrawals.

Note this audit only focuses on the smart contracts in the following directories/files:

- ChartVoyager/ChartVoyager.sol
- lucky777/lucky777.sol
- Space Invaders/SpaceInvaders.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
monad-game-contract	Version 1	0e0bfbaadab5802ae8c17d85c52a729d1ddc1dc0
	Version 2	a028bbf28490dcbebc85b71165ae204a3ee3b691

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

[1] https://github.com/dapdaps/monad-game-contract

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

- Access control
- Permission management
- Whitelist and blacklist mechanisms
- Initialization consistency
- Improper use of the proxy system
- Reentrancy
- Denial of Service (DoS)
- Untrusted external call and control flow
- Exception handling
- Data handling and flow
- Events operation
- Error-prone randomness
- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact	*High*	High	Medium
	Low	Medium	Low
		High	*Low*
		Likelihood	

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **two** potential security issues. Besides, we have **three** recommendations and **three** notes.

- Medium Risk: 2
- Recommendation: 3
- Note: 3

ID	Severity	Description	Category	Status
1	Medium	Potential signature replay attack	Security Issue	Fixed
2	Medium	Circumvention of contract check in function `deposit()`	Security Issue	Fixed
3	-	Remove redundant code	Recommendation	Fixed
4	-	Add an explicit error message in function `batchWithdraw()`	Recommendation	Confirmed
5	-	Emit an event for function `setGameCounter()`	Recommendation	Fixed
6	-	Settlement risks in the function `endGame()`	Note	-
7	-	Potential centralization risks	Note	-
8	-	The dependency on off-chain logic	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Potential signature replay attack

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `SpaceInvaders`, functions `startGame()`, `endGame()`, and `markGameAsFailed()` rely on a manager signature for authorization. However, the signed messages do not include the current contract address and the chain ID. If the `prefixMessage` and the mapping `manager` are reused across all chains and contract deployments, a valid signature generated for a game can be reused on another chain or another contract deployment, and pass the signature verification.

```
function startGame(
    string calldata tempGameId,
    bytes32 gameSeedHash,
    string calldata algoVariant,
    string calldata gameConfig,
    uint256 deadline,
    bytes calldata managerSignature
) external payable {
    require(block.timestamp <= deadline, "Signature expired");
    bytes32 messageHash = keccak256(
        abi.encode(
            string.concat(prefixMessage, ":startGame"),
            tempGameId,
            gameSeedHash,
            algoVariant,
            gameConfig,
            msg.sender,
            msg.value,
            deadline
        )
    );
    _verifyAnyManagerSignature(messageHash, managerSignature);
```

Listing 2.1: Space Invaders/SpaceInvaders.sol

```
function endGame(
    uint256 chainGameId,
    uint256 rewardAmount,
    string calldata gameState,
    string calldata gameSeed,
    uint256 deadline,
    bytes calldata managerSignature
) external nonReentrant {
    Game storage game = games[chainGameId];
    if (game.participant == address(0)) {
        revert GameNotFound(chainGameId);
    }
    if (game.status != GameStatus.Ongoing) {
        revert GameNotOngoing(chainGameId);
    }
    require(rewardAmount > 0, "RewardZero");
    address participantAddress = game.participant;

    if (!isManager[msg.sender]) {
        require(msg.sender == participantAddress, "Not authorized");
        require(block.timestamp <= deadline, "Signature expired");
        bytes32 messageHash = keccak256(
            abi.encode(
                string(abi.encodePacked(prefixMessage, ":endGame")),
                chainGameId,
                rewardAmount,
                gameState,
                gameSeed,
                deadline
            )
        );
        _verifyAnyManagerSignature(messageHash, managerSignature);
    }
```

Listing 2.2: Space Invaders/SpaceInvaders.sol

```
function markGameAsFailed(
    uint256 chainGameId,
    string calldata gameState,
    string calldata gameSeed,
    uint256 deadline,
    bytes calldata managerSignature
) public nonReentrant {
    Game storage game = games[chainGameId];
    if (game.participant == address(0)) {
        revert GameNotFound(chainGameId);
    }
    if (game.status != GameStatus.Ongoing) {
        revert GameNotOngoing(chainGameId);
    }
    address participantAddress = game.participant;

    if (!isManager[msg.sender]) {
        require(msg.sender == participantAddress, "Not authorized");
        require(block.timestamp <= deadline, "Signature expired");
        bytes32 messageHash = keccak256(
            abi.encode(
                string(
                    abi.encodePacked(prefixMessage, ":markGameAsFailed")
                ),
                chainGameId,
                gameState,
                gameSeed,
                deadline
            )
        );
        _verifyAnyManagerSignature(messageHash, managerSignature);
    }
```

Listing 2.3: Space Invaders/SpaceInvaders.sol

Impact Malicious actors can replay signatures to forge game execution and obtain rewards.

Suggestion Include `address(this)` and `block.chainid` to the signed message.

Clarification from BlockSec Note that EOA accounts using EIP-7702 can initiate deposits through delegated contract logic. The protocol needs to take this into consideration in backend logic.

2.1.2 Circumvention of contract check in function `deposit()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `ChartVoyager`, function `deposit()` checks the code size to block smart contract callers. However, this check can be circumvented when a contract invokes the function in its own constructor. This flaw allows contracts to circumvent the intended restriction, which only permits deposits from Externally Owned Accounts (EOAs). As a result, contracts can deposit funds during deployment, violating the core design assumption of EOA-only interactions.

```
function deposit(uint256 amount) external payable nonReentrant {

    uint256 codeSize;
    assembly {
        codeSize := extcodesize(caller())
    }
    require(codeSize == 0, "Contracts are not allowed to deposit");
```

Listing 2.4: ChartVoyager/ChartVoyager.sol

Impact Contracts can deposit during construction, violating the EOA-only restriction.

Suggestion Revise the logic accordingly.

2.2 Recommendation

2.2.1 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is an unused error `NotGameParticipant` in contract `SpaceInvaders`. It is recommended to remove the unused code for better code readability.

```
error NotGameParticipant(uint256 chainGameId, address caller);
```

Listing 2.5: Space Invaders/SpaceInvaders.sol

Suggestion Remove the redundant code.

2.2.2 Add an explicit error message in function `batchWithdraw()`

Status Confirmed

Introduced by `Version 1`

Description Function `batchWithdraw()` processes an array `recipients` and `amounts` in a loop and checks inputs for each withdrawal operation. However, when a check fails, the entire transaction reverts with a generic error (e.g., "Insufficient contract balance"). This design makes it difficult to identify exactly which transaction in the batch caused the failure. It is recommended to implement better error reporting that indicates the specific account (i.e., `recipient`) where the failure occurred.

```
function batchWithdraw(address[] calldata recipients, uint256[] calldata amounts) external
     onlyRole(ADMIN_ROLE) nonReentrant {
    require(recipients.length == amounts.length, "Recipients and amounts length mismatch");
    require(recipients.length > 0, "No recipients provided");

    for (uint256 i = 0; i < recipients.length; i++) {
        address payable recipient = payable(recipients[i]);
        uint256 amount = amounts[i];

        require(recipient != address(0), "Recipient cannot be zero address");
        require(amount > 0, "Withdraw amount must be greater than 0");
        require(address(this).balance >= amount, "Insufficient contract balance");


        recipient.sendValue(amount);


        withdrawn[recipient] += amount;


        emit Withdraw(recipient, amount);
    }
}
```

Listing 2.6: ChartVoyager/ChartVoyager.sol

Suggestion Include the specific recipient address in all validation error messages within the loop.

2.2.3 Emit an event for function `setGameCounter()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The owner could invoke privileged function `setGameCounter()` to set the storage variable `gameCounter`. However, the function does not emit an event to log such updates.

```
function setGameCounter(uint256 newCounter) external onlyOwner {
    require(newCounter >= gameCounter, "Cannot decrease gameCounter");
    gameCounter = newCounter;
}
```

Listing 2.7: Space Invaders/SpaceInvaders.sol

Suggestion Emit an event in the function.

2.3 Note

2.3.1 Settlement risks in the function `endGame()`

Introduced by `Version 1`

Description In contract `SpaceInvaders`, function `endGame()`reverts if the participant is unable to accept native tokens. This scenario can occur when an EOA participant uses EIP-7702 and delegates to a contract that lacks a payable fallback or receive function. Consequently, the game may remain in the `Ongoing` status rather than transitioning to a settled state. Off-chain systems should use the on-chain success or failure of the `endGame()`execution as the basis for deciding whether a game reward is settled.

```
        game.status = GameStatus.Victory;
        game.rewardAmount = rewardAmount;
        game.gameState = gameState;
        game.gameSeed = gameSeed;

        (bool success, ) = payable(participantAddress).call{
            value: rewardAmount
        }("");
        if (!success) {
            revert RewardTransferFailed(chainGameId, rewardAmount);
        }
```

Listing 2.8: Space Invaders/SpaceInvaders.sol

2.3.2 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `owner`, `manager`) can conduct sensitive operations, which introduces potential centralization risks. For example, contracts `ChartVoyage`, `lucky777`, and `SpaceInvaders` include privileged withdrawal functions like `withdrawBalance()` that allow the owner to withdraw arbitrary amounts of native tokens from the contract. The contract `SpaceInvaders` implements function `payAffiliate()` that enables managers to make unlimited affiliate payments and `endGame()` where managers can unilaterally decide reward amounts and terminate games. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol. Crucially, the game's integrity requires that the seed remains confidential and inaccessible to players prior to the game's completion.

2.3.3 The dependency on off-chain logic

Introduced by `Version 1`

Description The project relies heavily on off-chain logic for critical game functionality and verification:

1. The contracts `ChartVoyage` and `lucky777` serve primarily as financial gateways rather than core game logic.

For contract `ChartVoyage`, users deposit funds to participate in the game, and admins can withdraw funds. The contract records the deposit and withdrawal operations. For contract `lucky777`, users deposit to obtain game currency, with admins having withdrawal privileges. The actual game logic and game currency records occur off-chain, and the contract only triggers specific events.

2. The provable fairness is ensured off-chain. In contract `SpaceInvaders`, variable `gameSeedHash` stores the hash of `gameSeed`, `gameConfig`, and `algoVersion`, and is generated upon game creation. The `gameSeed` determines the death tile locations and should only be revealed when a game ends. Once revealed, the `gameSeed` must be verified against the `gameSeed` to prove fair play. Currently, such verification is performed by players off-chain.

3. The uniqueness of randomness is ensured off-chain. In contract `SpaceInvaders`, variable `gameSeedHash` is the hash of `gameSeed`, `gameConfig`, and `algoVersion`.

The randomness `gameSeed` is generated off-chain, while `gameConfig` and `algoVersion` can be identical across different games. The `gameSeed` must be unique across all games to avoid collision in `gameSeedHash`. Otherwise, this could allow players to predict the `gameSeed` and win games by recognizing a previously observed hash.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS