MONETHIC

Wolf Game

Smart Contract Audit Report

Prepared for:
Wolf LLC

Date:
25.05.2024

Version:
Final, for public release

Table of Contents

About Monethic

Monethic is a young and thriving cybersecurity company with extensive experience in various fields, including Smart Contracts, Blockchain protocols (layer 0/1/2), wallets and off-chain components audits, as well as traditional security research, starting from penetration testing services, ending at Red Team campaigns. Our team of cybersecurity experts includes experienced blockchain auditors, penetration testers, and security researchers with a deep understanding of the security risks and challenges in the rapidly evolving IT landscape. We work with a wide range of clients, including fintechs, blockchain startups, decentralized finance (DeFi) platforms, and established enterprises, to provide comprehensive security assessments that help mitigate the risks of cyberattacks, data breaches, and financial losses.

At **Monethic**, we take a collaborative approach to security assessments, working closely with our clients to understand their specific needs and tailor our assessments accordingly. Our goal is to provide actionable recommendations and insights that help our clients make informed decisions about their security posture, while minimizing the risk of security incidents and financial losses.

About Client

Wolf Game is a blockchain-based strategy game where players manage digital assets like sheep and wolves. The game's economy revolves around earning $WOOL, the in-game currency. Players can stake their sheep to earn passive $WOOL or risk them in the game's risk-reward mechanics, where wolves have the chance to steal sheep and $WOOL. The game involves strategic decision-making and blockchain technology to facilitate unique gameplay experiences.

About Auditor

This audit was conducted at the client's request in the form of a "solo audit" by CEO & Co-Founder of Monethic.

Jakub Heba is a cybersecurity expert with over seven years of experience in the industry. For more than two years associated with blockchain technology as a Smart Contract and Blockchain auditor. He has conducted over 50 audits of various protocols, mostly related to Decentralized Finances. He specializes in the security of contracts written in Rust, such as CosmWasm, NEAR, Substrate, ink!, Scrypto or MultiversX (Elrond), as well as has a deep technical understanding of EVM and Solidity. He participated in assessments testing low-level aspects of blockchain technology, such as finality proof verifications, serialization libraries, as well as implementations of bridges between many different ecosystems. He has experience in auditing

Layer 0/1 Blockchains written in Rust and MOVE. Before moving to Web3, he was a Lead Security Researcher and Penetration Tester. He also specialized in low-level binary exploitation in both UNIX and Windows environments. Holder of OSCP, OSCE and Lead ISO27001 Auditor certifications.

Disclaimer

This report reflects a rigorous security assessment conducted on the specified product, utilizing industry-leading methodologies. While the service was carried out with the utmost care and proficiency, it is essential to recognize that no security verification can guarantee 100% immunity from vulnerabilities or risks.

Security is a dynamic and ever-evolving field. Even with substantial expertise, it is impossible to predict or uncover all future vulnerabilities. Regular and varied security assessments should be performed throughout the code development lifecycle, and engaging different auditors is advisable to obtain a more robust security posture.

This assessment is limited to the defined scope and does not encompass parts of the system or third-party components not explicitly included. It does not provide legal assurance of compliance with regulations or standards, and the client remains responsible for implementing recommendations and continuous security practices.

Scoping Details

The purpose of the assessment was to conduct Smart Contract Audit against Wolf Game Smart Contracts, available through the Github platform.

- https://github.com/pixel-vault/wolfgame-blast

The scope of testing includes the code sections listed below:

- `contracts/caveGame/*`

1. **CaveGame.sol**:
    - This is one of the game contracts, managing players' claims.
    - It verifies signed messages to allow players to claim rewards, such as NFTs.
    - It tracks claims to prevent double claiming.
2. **Gem.sol**:
    - Manages NFTs representing various items or entities in the game.
    - Enables minting and burning of these tokens.

    - Utilizes a more efficient ERC721 implementation to reduce transaction costs.
3. **PremiumBox.sol**:
    - Handles special premium boxes that may contain rare items or entities.
    - Allows purchasing these boxes using various tokens, integrating the payment system and in-game points.

Scope

The scope covered by the security assessment specifies that the Wolf Game contracts will be audited, the code of which has been shared on the GitHub platform with the **39b4bcf50965b83dd786a1edb703a4ef09dd3775** commit SHA hash.

No additional internal documentation was provided.

Timeframe

On May 24th Monethic was chosen for the security audit of Wolf Game. Work began immediately.

On May 25th, the report from the Smart Contract security assessment was delivered to the customer.

Vulnerability Classification

All vulnerabilities described in the report were thoroughly classified in terms of the risk they generate in relation to the security of the contract implementation. Depending on where they occur, their rating can be estimated on the basis of different methodologies.

In most cases, the estimation is done by summarizing the impact of the vulnerability and its likelihood of occurrence. The table below presents a simplified risk determination model for individual calculations.

		Impact		
	Severity	**High**	**Medium**	**Low**
Likelihood	**High**	**Critical**	**High**	**Medium**
	Medium	**High**	**Medium**	**Low**
	Low	**Medium**	**Low**	**Low**

Vulnerabilities that do not have a direct security impact, but may affect overall code quality, as well as open doors for other potential vulnerabilities, are classified as **INFORMATIVE**.

VULNERABILITIES SUMMARY

No.	Severity	Name	Status
1	Medium	`ethPurchaseBoxes` does not validate stale price	**Resolved**
2	Informative	Wrong assumption leads to redundant operations	**Resolved**
3	Informative	`Gem` contract is not using the `safeMint` function	**Resolved**
4	Informative	`AccessControlDefaultAdminRules` is superior over `AccessControl` implementation	**Acknowledged**

Technical summary

1. `ethPurchaseBoxes` does not validate stale price

Severity

Medium

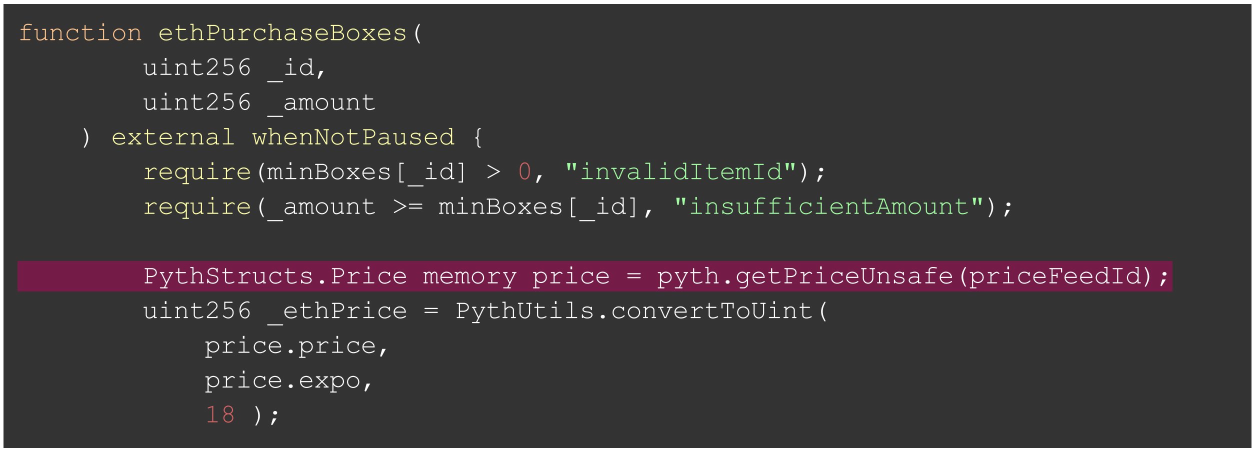
Location

`contracts/caveGame/PremiumBox.sol:105`

Description

It was observed that in `ethPurchaseBoxes` function, responsible for handling the purchase of boxes using ETH, current price retrieved is with usage of Pyth's `getPriceUnsafe` function. It differs from its safer counterpart, the `getPrice` function, in that it does not verify that the price data is recent enough to be considered valid.

The `getPriceUnsafe` documentation suggests to specify maximum time, for which price is treated as valid and recent. For comparison, the `getPrice` function requires it to be a minute or less. Currently, the contract does not implement such validation, which is problematic because very old data may distort the business logic and negatively affect NFT prices.



```
function ethPurchaseBoxes(
        uint256 _id,
        uint256 _amount
    ) external whenNotPaused {
        require(minBoxes[_id] > 0, "invalidItemId");
        require(_amount >= minBoxes[_id], "insufficientAmount");

        PythStructs.Price memory price = pyth.getPriceUnsafe(priceFeedId);
        uint256 _ethPrice = PythUtils.convertToUint(
            price.price,
            price.expo,
            18 );
```

Remediation

We suggest implementing validation that will implement a maximum time limit for which the price is considered valid and recent.

Status: Resolved

2. Wrong assumption leads to redundant operations

Severity

INFORMATIVE

Location

`contracts/caveGame/PremiumBox.sol`

Description

It was noticed that in the `PremiumBox` contract, two tokens are implemented and used - `USDB` and `USDBR`. In logic, they are treated as two separate entities, one as ERC20 and the other as ERC20 rebasing yield controller.

In fact, however, it is one and the same token, with `0x4300000000000000000000000000000000000003` address. The logic operating on both is therefore redundant and can be simplified.

```
contract PremiumBox is AccessControl, ReentrancyGuard, Pausable {
     IBlast public BLAST = IBlast(0x4300000000000000000000000000000000000002);
     IERC20Rebasing public USDBR =
         IERC20Rebasing(0x4300000000000000000000000000000000000003);
     IBlastPoints public BlastPoints =
         IBlastPoints(0x2536FE9ab3F511540F2f9e2eC2A805005C3Dd800);

     IERC20 public USDB;
     IPyth pyth;
```

Remediation

We suggest replacing two tokens with one as they refer to the same ERC20 token.

Status: Resolved

3. Gem contract is not using the safeMint function

Severity

INFORMATIVE

Location

contracts/caveGame/Gem.sol:38

Description

Gem contract is using _mint function to create new tokens. While _mint is a standard method for minting new tokens, it lacks certain safety checks that could lead to potential issues. The safer alternative, _safeMint, includes additional checks that mitigate these risks by ensuring that the recipient of the minted tokens is a valid address and capable of handling ERC721 tokens.

```
function mint(
        address _to,
        uint256 _nonce,
        uint256 _amount
    ) external onlyRole(MINT_ROLE) {
        unchecked {
            uint256 currentIndex = _nextTokenId();
            for (uint256 i = 0; i < _amount; i++) {
                nonce[currentIndex + i] = _nonce;
            }
        }
        _mint(_to, _amount);
    }
```

Remediation

Replace _mint with _safeMint to ensure that the recipient of the tokens can handle them properly.

Status: Resolved

4. `AccessControlDefaultAdminRules` is superior over `AccessControl` implementation

Severity

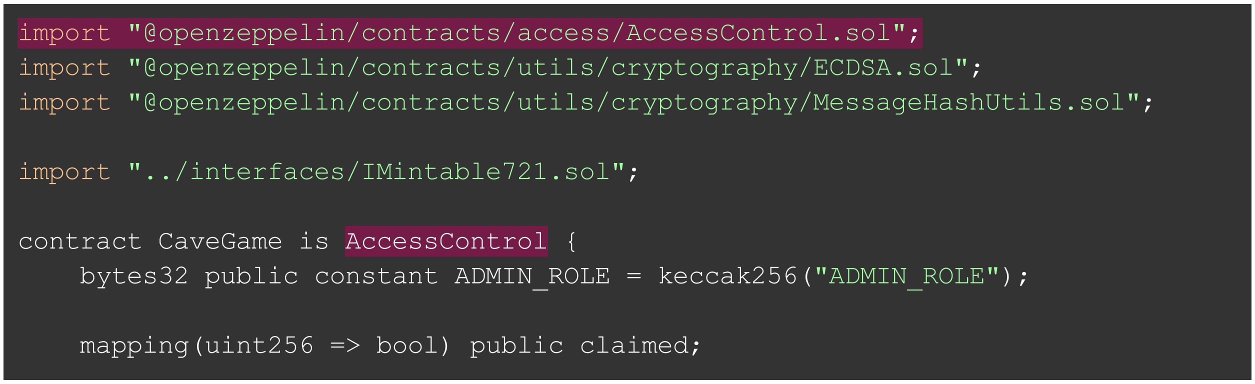
INFORMATIVE

Location

`contracts/caveGame/CaveGame.sol:10`

Description

It was found that the contract uses `AccessControl` for role-based access management. While `AccessControl` is effective, the alternative, `AccessControlDefaultAdminRules`, offers enhanced security features, related to the management of the default admin role.



```
import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";

import "../interfaces/IMintable721.sol";

contract CaveGame is AccessControl {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");

    mapping(uint256 => bool) public claimed;
```

Remediation

We suggest switching to `AccessControlDefaultAdminRules` to enhance the security of role-based access management.

Status: **Acknowledged**

END OF THE REPORT