SECURITY AUDIT REPORT

Metagame

DATE

15 November 2025

PREPARED BY

0xTeam.

WEB3 AUDITS

info@0xTeam.Space

0xTeamSpace

0xTeamSpace

Contents

0.0	Revision History & Version Control	3
1.0	Disclaimer	4
2.0	Executive Summary	5
3.0	Checked Vulnerabilities	7
4.0	Techniques , Methods & Tools Used	8
5.0	Technical Analysis	9
6.0	Auditing Approach and Methodologies Applied	40
7.0	Limitations on Disclosure and Use of this Report	41

Revision History & Version Control

Version	Date	Author's	Description
1.0	12 Nov 2025	Gurkirat, Manoj & Aditya	Initial Audit Report
2.0	15 Nov 2025	Gurkirat, Manoj & Aditya	Final Audit Report

0xTeam conducted a comprehensive Security Audit on the Metagame to ensure the overall code quality, security, and correctness. The review focused on ensuring that the code functions as intended, identifying potential vulnerabilities, and safeguarding the integrity of Metagame's operations against possible attacks.

Report Structure

The report is divided into two primary sections:

1. **Executive Summary** : Provides a high-level overview of the audit findings.
2. **Technical Analysis** : Offers a detailed examination of the Smart contracts code.

Note :
The analysis is static and manual, exclusively focused on the smart contract code. The information provided in this report should be used to assess the security, quality, and expected behavior of the code.

1.0 Disclaimer

This is a summary of our audit findings based on our analysis, following industry best practices as of the date of this report. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. The audit focuses on Smart contracts coding practices and any issues found in the code, as detailed in this report. For a complete understanding of our analysis, you should read the full report. We have made every effort to conduct a thorough analysis, but it's important to note that you should not rely solely on this report and cannot make claims against us based on its contents. We strongly advise you to perform your own independent checks before making any decisions. Please read the disclaimer below for more information.

DISCLAIMER: By reading this report, you agree to the terms outlined in this disclaimer. If you do not agree, please stop reading immediately and delete any copies you have. This report is for informational purposes only and does not constitute investment advice. You should not rely on the report or its content, and 0xTeam and its affiliates (including all associated companies, employees, and representatives) are not responsible for any reliance on this report.The report is provided "as is" without any guarantees. 0xTeam excludes all warranties, conditions, or terms, including those implied by law, regarding quality, fitness for a purpose, and use of reasonable care. Except where prohibited by law, 0xTeam is not liable for any type of loss or damage, including direct, indirect, special, or consequential damages, arising from the use or inability to use this report. The findings are solely based on the Smart contracts code provided to us.

2.0 Executive Summary

2.1 Overview

0xTeam has meticulously audited the Metagame Smart contracts project. The primary objective of this audit was to assess the security, functionality, and reliability of the Metagame's before their deployment on the blockchain. The audit focused on identifying potential vulnerabilities, evaluating the contract's adherence to best practices, and providing recommendations to mitigate any identified risks. The comprehensive analysis conducted during this period ensures that the Metagame is robust and secure, offering a reliable environment for its users.

2.2 Scope

The scope of this audit involved a thorough analysis of the Metagame Smart contracts, focusing on evaluating its quality, rigorously assessing its security, and carefully verifying the correctness of the code to ensure it functions as intended without any vulnerabilities.

Files in Examination:

Language	Solidity
In-Scope	• ~/src • ~/src/Metagame.sol • ~/src/MetagameConfig.sol • ~/src/MetagameErrors.sol • ~/src/MetagameStorage.sol
Github	https://github.com/zeroexcore/mph/blob/main/packages/evm/
Fixed Commit Hash	NA (Provided the ZIP Folder)

OUT-OF-SCOPE: External Smart contracts code, other imported code.

2.3 Audit Summary

Name	Verified	Audited	Vulnerabilities
Metagame	Yes	Yes	Refer Section 5.0

2.4 Vulnerability Summary

High	Medium	Low	Informational
3	0	2	7

2.5 Recommendation Summary

Severity

Issues	High	Medium	Low	Informational	Total (Σ)
Not Fixed					
Resolved	3		2	2	**7**
Acknowledged				5	**5**
Partially Resolved					
Total (Σ)	3		2	7	**12**

- **Open**: Unresolved security vulnerabilities requiring resolution.
- **Resolved**: Previously identified vulnerabilities that have been fixed.
- **Acknowledged**: Identified vulnerabilities noted but not yet resolved.
- **Partially Resolved**: Risks mitigated but not fully resolved.

2.6 Summary of Findings

ID	Title	Severity	Fixed
H-01	Missing Nonce in EIP-712 Signed Parameters Enables Signature Replay Attacks	High	✔
H-02	Invalid Token Handling During Mint Allows Arbitrary Token Address, Breaking Burn Logic	High	✔
H-03	Mint function Allows Accidental ETH Loss Due to a Missing Validation	High	✔
L-01	Functions not used internally could be marked as external	Low	✔
L-02	State Change Without Event Emission	Low	✔
I-01	Repeated Owner Checks In Administrative Functions	Info	📝
I-02	Solidity Pragma Should Be Specific, Not Wide	Info	✔
I-03	Redundant import of IERC20 interface in Metagame.sol	Info	✔
I-04	Missing Minimum-Fee Validation burn function in Allows Zero-Fee Operations in MATURE and REROLL Modes	Info	📝
I-05	Unrestricted Reroll Logic Allows Infinite Rerolls Without Checking Eligibility or State Tracking	Info	📝
I-06	User-Controlled Voucher Allowance Allows Unlimited Voucher Redemption	Info	📝
I-07	MATURE Burn Mode Provides No Meaningful Functionality and Is Bypassed by Existing CLOSE or CANCEL Behavior	Info	📝

✔ - Fixed ● - Partially Fixed ✘ - Not Fixed 📝 - Acknowledged

3.0 Checked Vulnerabilities

We examined Smart contracts for widely recognized and specific vulnerabilities. Below are some of the common vulnerabilities considered.

Category	Check Items
Source Code Review	➔ Reentrancy Vulnerabilities ➔ Ownership Control ➔ Time-Based Dependencies ➔ Gas Usage in Loops ➔ Transaction Sequence Dependencies ➔ Style Guide Compliance ➔ EIP Standard Compliance ➔ External Call Verification ➔ Mathematical Checks ➔ Type Safety ➔ Visibility Settings ➔ Deployment Accuracy ➔ Repository Consistency
Functional Testing	➔ Business Logic Validation ➔ Feature Verification ➔ Access Control and Authorization ➔ Escrow Security ➔ Token Supply Management ➔ Asset Protection ➔ User Balance Integrity ➔ Data Reliability ➔ Emergency Shutdown Mechanism

4.0 Techniques , Methods & Tools Used

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

- **Structural Analysis:**
  This involves examining the overall design and architecture of the smart contract. We ensure that the contract is logically organised, scalable, and follows industry best practices. This step is crucial for identifying potential structural issues that could lead to vulnerabilities or maintenance challenges in the future.

- **Static Analysis:**
  Static analysis is conducted using automated tools to scan the contract's codebase for common vulnerabilities and security risks without executing the code. This process helps identify issues such as reentrancy, arithmetic errors, and potential denial-of-service (DOS) vulnerabilities early on, allowing for quick remediation.

- **Code Review / Manual Analysis:**
  A manual, in-depth review of the smart contract's code is performed to verify the logic and ensure it matches the intended functionality as described in the project's documentation. During this phase, we also confirm the findings from the static analysis and check for any additional issues that may not have been detected by automated tools.

- **Dynamic Analysis:**
  Dynamic analysis involves executing the smart contract in various controlled environments to observe its behaviour under different conditions. This step includes running comprehensive test cases, performing unit tests, and monitoring gas consumption to ensure the contract operates efficiently and securely in real-world scenarios.

- **Tools and Platforms Used for Audit:**
  Utilising tools such as Remix , Slither, Aderyn, Solhint for static analysis, and platforms like Hardhat and Foundry for dynamic testing and simulation.

Note: The following values for "Severity" mean:

- High: Direct and severe impact on the funds or the main functionality of the protocol.
- Medium: Indirect impact on the funds or the protocol's functionality.
- Low: Minimal impact on the funds or the protocol's main functionality.
- Informational: Suggestions related to good coding practices and gas efficiency.

5.0 Technical Analysis

High

[H-01] Missing Nonce in EIP-712 Signed Parameters Enables Signature Replay Attacks

Severity
High

Location	Functions
`Metagame.sol`	`➔ metagame`

Issue Description

The protocol uses EIP-712 signatures to authorize sensitive operations like mint and burn.
However, the signed parameter structures MintParams and BurnParams **do not include a nonce or any unique identifier** that ensures one-time signature validity.

As a result, a valid signature can be **replayed indefinitely** by the same user, allowing repeated execution of the user's signature until the deadline expires.

Impact / Proof of Concept

This fundamentally breaks the trust model of EIP-712 authorization, as the Protocol loses control over how many times a given signature can be used.

Depending on the operation, this can cause:

- **Infinite authorized burns** (e.g., replaying a burn() signature).
- **Repeated minting with the same protocol approved parameters**.
- Users can suffer from funds loss, if they submit the same transaction twice.

In the POC below we can see that we have called mint() and burn() functions with the same signature.

```
function test_POC_Multiple_Burn_With_Single_Signature() public {
    address treasuryAddr = makeAddr("treasury");
    game.setTreasury(treasuryAddr);
    address user = makeAddr("user");
    vm.deal(user, 3 ether);

    // Mint native
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({ user: user, token: game.NATIVE(), amount: 1 ether, duration: block.timestamp + 30
    days, allowance: 0, deadline: block.timestamp + 1 hours});

    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));          Signature for Mint
    vm.prank(user);
    uint256 tokenId = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));                        Mint 1
    console.log("Mint 1 Successful and we got token id", tokenId);

    vm.prank(user);
    uint256 tokenId1 = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));                       Mint  2 with same signature
    console.log("Mint 2 is also Successful with the same signature and we got token id", tokenId1);

    // REROLL charges fee, emits Reroll
    uint256 userBefore = user.balance;
    uint256 treasuryBefore = treasuryAddr.balance;
    console.log("Balance of user BEFORE REROLL", userBefore);
    console.log("Balance of treasury BEFORE REROLL", treasuryBefore);

    uint256 fee = 0.5 ether;
    MetagameStorage.BurnParams memory b = MetagameStorage.BurnParams({
        user: user, tokenId: tokenId, mode: 4, fee: fee, deadline: block.timestamp + 1 hours
    });

    (uint8 bv, bytes32 br, bytes32 bs) = vm.sign(signerPrivateKey, game.getBurnParamsHash(b));          Signature for Burn
    vm.prank(user);
    game.burn{value: fee}(b, abi.encodePacked(br, bs, bv));                                              Burn 1
    console.log("REROLL 1 Successful");

    vm.prank(user);
    game.burn{value: fee}(b, abi.encodePacked(br, bs, bv));                                              Burn 2 with same signature
    console.log("REROLL 2 Successful with same signature");

    uint256 userAfter = user.balance;
    uint256 treasuryAfter = treasuryAddr.balance;
    console.log("Balance of user AFTER 2 REROLL's", userAfter);
    console.log("Balance of treasury AFTER 2 REROLL's", treasuryAfter);

    // Treasury received fees twice with the same signature
    assertEq(game.ownerOf(tokenId), user);
    assertEq(userAfter, userBefore - fee * 2, "User balance not reduced by expected amount");
    uint256 expected = treasuryBefore + fee * 2;
    assertEq(treasuryAfter, expected, "Treasury did not receive expected fees from repeated rerolls");
}
```

```
gurkiratsingh@Gurkirats-MacBook-Air evm % forge test --mt test_POC_Multiple_Burn_With_Single_Signature -vv
[⠊] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Metagame.t.sol:MetagameTest
[PASS] test_POC_Multiple_Burn_With_Single_Signature() (gas: 485188)
Logs:
  Mint 1 Successful and we got token id 1
  Mint 2 is also Successful with the same signature and we got token id 2
  Balance of user BEFORE REROLL 100000000000000000
  Balance of treasury BEFORE REROLL 0
  REROLL 1 Successful
  REROLL 2 Successful with same signature
  Balance of user AFTER 2 REROLL's 0
  Balance of treasury AFTER 2 REROLL's 1000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 15.19ms (4.20ms CPU time)

Ran 1 test suite in 21.64ms (15.19ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation

To prevent replay attacks, include a nonce in the signed parameter structures and track it on-chain.

A minimal example is presented below.

Example Fix:

```
// In storage
mapping(address => uint256) public nonces;

// In BurnParams / MintParams add `uint256 nonce;`

// Update hash functions to include nonce:
abi.encode(
   MetagameStorage.BURN_PARAMS_TYPE_HASH,
   params.user,
   params.tokenId,
   params.mode,
   params.fee,
   params.deadline,
   params.nonce
);

// In burn / mint verify:
if (params.nonce != nonces[params.user]) revert InvalidNonce();
nonces[params.user] += 1; // consume the nonce
```

Additionally:

- Use **shorter default deadlines** to reduce the attack window.
- Log nonce consumption events for transparency and off-chain monitoring.

Status - Fixed in Pull Request [ab3d3e8]

Re-Audit - POC Passed

Protocol has added a nonce tracker in the struct, and added nonce in the signature in getBurnParamsHash and getMintParamsHash functions and also included a check in the mint and burn functions.

And we can see in below images, 0xteam has tested with a POC that the bug doesn't exist as it is reverting with a valid error of StaleNonce in the terminal.

```
function test_POC_Multiple_Burn_With_Single_Signature() public {
    address treasuryAddr = makeAddr("treasury");
    game.setTreasury(treasuryAddr);
    address user = makeAddr("user");
    vm.deal(user, 3 ether);

    // Mint native
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({ user: user, token: game.NATIVE(),
    amount: 1 ether, duration: block.timestamp + 30 days, allowance: 0, deadline: block.timestamp + 1 hours,
    nonce: 1});

    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));
    vm.prank(user);
    uint256 tokenId = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));
    console.log("Mint 1 Successful and we got token id", tokenId);

    vm.prank(user);
    uint256 tokenId1 = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));
    console.log("Mint 2 is also Successful with the same signature and we got token id", tokenId1);
}
}
```

Mint 1

Mint 2 with same signature

```
gurkiratsingh@Gurkirats-MacBook-Air evm % forge test --mt test_POC_Multiple_Burn_With_Single_Signature -vv
[⠒] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Metagame.t.sol:MetagameTest
[FAIL: StaleNonce()] test_POC_Multiple_Burn_With_Single_Signature() (gas: 317516)
Logs:
  Mint 1 Successful and we got token id 1

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 12.85ms (3.11ms CPU time)

Ran 1 test suite in 18.97ms (12.85ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/Metagame.t.sol:MetagameTest
[FAIL: StaleNonce()] test_POC_Multiple_Burn_With_Single_Signature() (gas: 317516)
```

[H-02] Invalid Token Handling During Mint Allows Arbitrary Token Address, Breaking Burn Logic

Severity
High

Location	Functions
`Metagame.sol`	➔ `mint()`

Issue Description

The contract supports exactly two token types during mint:

- NATIVE (address(0))
- VOUCHER (fixed voucher address)

However, the mint() logic mistakenly treats any token address that is not equal to VOUCHER as NATIVE.

```
if (params.token == MetagameStorage.VOUCHER) {
   // voucher logic
   } else {
   // assumed NATIVE
}
```

There is **no validation** that:

```
params.token == MetagameStorage.NATIVE
```

A user could pass **any arbitrary token address** in the params, for example:

```
params.token = 0x123456...
```

The contract:

- accepts the mint
- collects native fee + amount
- stores incorrect token address in mintData[tokenId].token

During burn() or close() arbitrary token types cause mode validation to fail :

```
if (mintParams.token != NATIVE && mintParams.token != VOUCHER)
revert InvalidBurnPosition();
```

Which means the user's NFT position is closed or canceled etc, but the user's principal is permanently locked.

Impact / Proof of Concept

1. Users can accidentally mint invalid token types.
2. The user amount becomes permanently locked inside the contract.
3. NFT can never be burned or closed.

The attached POC shows a user minting with a non-NATIVE token address (i.e., not address(0)). An NFT is minted and the user's ETH balance is reduced as expected because of the minting, but when the position is closed by calling burn() or close() function the user's funds are not returned and NFT is burned. In short, burning the NFT does not restore the deducted balance, which results in a permanent loss of user funds. The user's initial balance is higher before minting and remains lower after burning.

```
function test_POC_Mint_Burn_Close_Wrong_Native_Address() public {
    address user = makeAddr("user");
    address randomAddress = makeAddr("randomAddr");
    vm.deal(user, 3 ether);

    // Mint a position with future maturity but with a random token address
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({
        user: user, token: randomAddress, amount: 1 ether, duration: block.timestamp + 7 days, allowance: 0,
        deadline: block.timestamp + 1 hours
    });

    uint256 balanceOfUserBeforeMint = user.balance;
    console.log("Balance of user BEFORE MINT", balanceOfUserBeforeMint);
    console.log("Passing a random address which is not equal to NATIVE token address(0) = ", randomAddress);

    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));
    vm.prank(user);
    uint256 tokenId = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));
    console.log("Balance of user AFTER MINT", user.balance);

    vm.warp(block.timestamp + 7 days);

    // Close after maturity should refund principal and burn
    uint256 userBefore = user.balance;
    MetagameStorage.BurnParams memory b = MetagameStorage.BurnParams({
        user: user, tokenId: tokenId, mode: 1, fee: 0, deadline: block.timestamp + 7 days + 1 hours});

    (uint8 bv, bytes32 br, bytes32 bs) = vm.sign(signerPrivateKey, game.getBurnParamsHash(b));
    vm.prank(user);
    game.burn(b, abi.encodePacked(br, bs, bv));

    uint256 balanceOfUserAfterBurn = user.balance;
    console.log("Balance of user AFTER BURN", balanceOfUserAfterBurn);

    console.log("Now balance of the User has not increased when passed a random address in mint after the
    burn of the NFT, which means that NFT is also burned and balance is also not returned");

    // Burned and principal NOT returned
    vm.expectRevert();
    game.ownerOf(tokenId);
    assertGt(balanceOfUserBeforeMint, balanceOfUserAfterBurn);
}
```

```
gurkiratsingh@Gurkirats-MacBook-Air evm % forge test --mt test_POC_Mint_Burn_Close_Wrong_Native_Address -vv
[⠊] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Metagame.t.sol:MetagameTest
[PASS] test_POC_Mint_Burn_Close_Wrong_Native_Address() (gas: 219480)
Logs:
  Balance of user BEFORE MINT 3000000000000000000
  Passing a random address which is not equal to NATIVE token address(0) =  0x302200519E6fF414a148af6Afd38E93e420D277C
  Balance of user AFTER MINT 2000000000000000000
  Burn is successfull
  Balance of user AFTER BURN 2000000000000000000
  Now balance of the User has not increased when passed a random address in mint after the burn of the NFT, which means that NFT i
s also burned and balance is also not returned

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.10ms (1.46ms CPU time)

Ran 1 test suite in 3.87ms (3.10ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation

Add strict token-type validation inside mint() to ensure users cannot pass arbitrary token addresses.

```
if(
params.token != MetagameStorage.NATIVE ||
params.token != MetagameStorage.VOUCHER) {
  revert InvalidTokenType();
}
```

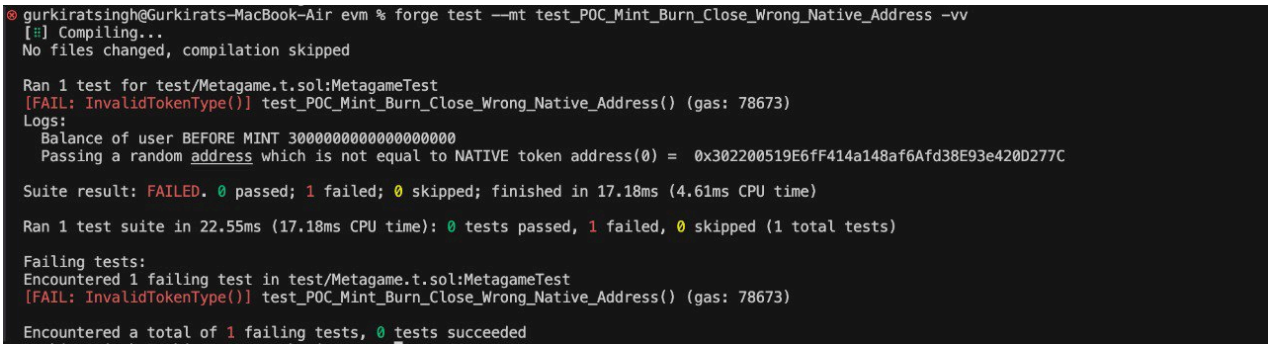
Status - Fixed in Pull Request [0676dc5]

Re-Audit - POC Passed

To protocol has introduced a new error in the error file and added specific check else if (params.token == MetagameStorage.NATIVE) { and also added an additional else statement, so that user will get an error if he passes any other address apart from VOUCHER and NATIVE token address

Here is a POC test which verifies that now the bug doesn't exist and is giving an error when someone passed any random address in the params.

```
function test_POC_Mint_Burn_Close_Wrong_Native_Address() public {
    address user = makeAddr("user");
    address randomAddress = makeAddr("randomAddr");
    vm.deal(user, 3 ether);

    // Mint a position with future maturity but with a random token address
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({
        user: user, token: randomAddress, amount: 1 ether, duration: block.timestamp + 7 days, allowance: 0,
        deadline: block.timestamp + 1 hours, nonce: 1
    });

    uint256 balanceOfUserBeforeMint = user.balance;
    console.log("Balance of user BEFORE MINT", balanceOfUserBeforeMint);
    console.log("Passing a random address which is not equal to NATIVE token address(0) = ", randomAddress);

    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));
    vm.prank(user);
    uint256 tokenId = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));
    console.log("Balance of user AFTER MINT", user.balance);

}
```


gurkiratsingh@Gurkirats-MacBook-Air evm % forge test --mt test_POC_Mint_Burn_Close_Wrong_Native_Address -vv
[⠊] Compiling...
No files changed, compilation skipped
Ran 1 test for test/Metagame.t.sol:MetagameTest
[FAIL: InvalidTokenType()] test_POC_Mint_Burn_Close_Wrong_Native_Address() (gas: 78673)
Logs:
Balance of user BEFORE MINT 3000000000000000000
Passing a random address which is not equal to NATIVE token address(0) = 0x302200519E6fF414a148af6Afd38E93e420D277C
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 17.18ms (4.61ms CPU time)
Ran 1 test suite in 22.55ms (17.18ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)
Failing tests:
Encountered 1 failing test in test/Metagame.t.sol:MetagameTest
[FAIL: InvalidTokenType()] test_POC_Mint_Burn_Close_Wrong_Native_Address() (gas: 78673)
Encountered a total of 1 failing tests, 0 tests succeeded

[H-03] Mint function Allows Accidental ETH Loss Due to a Missing Validation

Severity
High

Location	Functions
`Metagame.sol`	➔ `mint()`

Issue Description

When minting a voucher-backed position (params.token == MetagameStorage.VOUCHER), the protocol is designed to operate without any on-chain payment. The documentation states that voucher mints:

- do not require native currency
- do not transfer or hold ETH

However, in the voucher branch of the mint() function, the contract **does not check** that the user supplied msg.value == 0.

```
if (params.token == MetagameStorage.VOUCHER) {
    if ($.redeemedVouchers[msg.sender] + params.amount >
params.allowance) {
        revert InsufficientVouchers();
    }
     $.redeemedVouchers[msg.sender] += params.amount;
 }
```

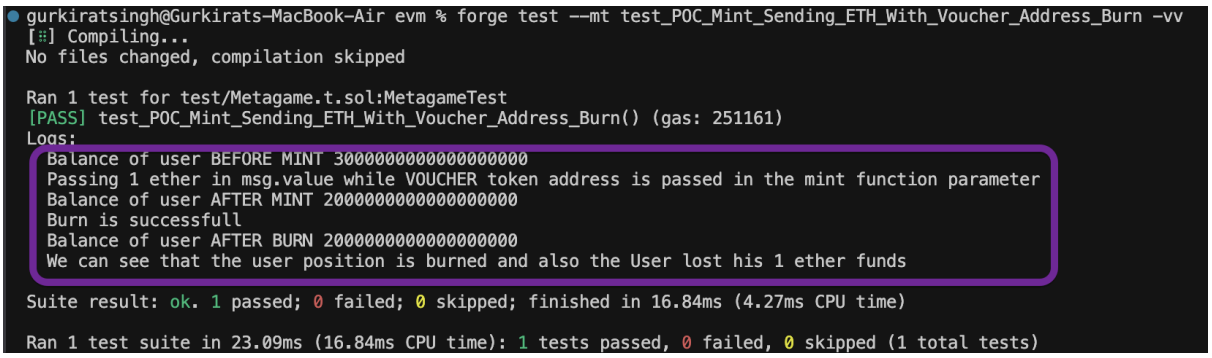
Because the function is payable, users can send arbitrary ETH, even accidentally when minting voucher positions.

Impact / Proof of Concept

- All ETH sent in this scenario becomes **permanently stuck in the contract balance**, since the voucher branch does not calculate the ETH transfer logic. This creates a direct and irreversible loss of user funds.
- No mechanism exists for recovering ETH because the voucher mint path performs no refunds and no return transfers.

In the below POC we can see that the user sent ETH when he has selected the VOUCHER token and balance from his wallet is deducted and after closing the position, the user's funds are forever locked in the smart contract.

```
function test_POC_Mint_Sending_ETH_With_Voucher_Address_Burn() public {
    address user = makeAddr("user");
    vm.deal(user, 3 ether);

    // Mint a position with future maturity but with VOUCHER token address
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({
        user: user, token: game.VOUCHER(), amount: 1 ether, duration: block.timestamp + 7 days, allowance: 3
        ether, deadline: block.timestamp + 1 hours
    });

    uint256 balanceOfUserBeforeMint = user.balance;
    console.log("Balance of user BEFORE MINT", balanceOfUserBeforeMint);

    console.log("Passing 1 ether in msg.value while VOUCHER token address is passed in the mint function
    parameter");

    // uint256 tokenId = game.mint(m, abi.encodePacked(mr, ms, mv));
    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));
    vm.prank(user);
    uint256 tokenId = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));

    console.log("Balance of user AFTER MINT", user.balance);
    vm.warp(block.timestamp + 7 days);

    // Close after maturity should refund principal and burn
    uint256 userBefore = user.balance;
    MetagameStorage.BurnParams memory b = MetagameStorage.BurnParams({
        user: user, tokenId: tokenId, mode: 1, fee: 0, deadline: block.timestamp + 7 days + 1 hours});

    (uint8 bv, bytes32 br, bytes32 bs) = vm.sign(signerPrivateKey, game.getBurnParamsHash(b));
    vm.prank(user);
    game.burn(b, abi.encodePacked(br, bs, bv));

    uint256 balanceOfUserAfterBurn = user.balance;
    console.log("Balance of user AFTER BURN", balanceOfUserAfterBurn);
    console.log("We can see that the user position is burned and also the User lost his 1 ether funds");

    // Burned and ETH is not returned to the user hence user funds are lost
    vm.expectRevert();
    game.ownerOf(tokenId);
    assertGt(balanceOfUserBeforeMint, balanceOfUserAfterBurn);
}
```



Recommendation

Add an explicit check in the voucher branch of the mint() function to prevent users from sending any ETH during voucher mints:

```
if (params.token == MetagameStorage.VOUCHER) {
  // vouchers
  require(msg.value == 0, "NoETHAllowedForVouchers");
  if ($.redeemedVouchers[msg.sender] + params.amount > params.allowance) {
    revert InsufficientVouchers();
  }
  $.redeemedVouchers[msg.sender] += params.amount;
}
```

Status - Fixed in Pull Request [a257178]

Re-Audit - POC Passed

Protocol has introduced a new check in the mint function in the VOUCHER branch of the function , protocol has added a check so that users can't send any ETH to the smart contract using VOUCHER token address.

Here is POC test proving that bug is resolved

```
function test_POC_Mint_Sending_ETH_With_Voucher_Address_Burn() public {
    address user = makeAddr("user");
    vm.deal(user, 3 ether);

    // Mint a position with future maturity but with VOUCHER token address
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({
        user: user, token: game.VOUCHER(), amount: 1 ether, duration: block.timestamp + 7 days, allowance: 3
        ether, deadline: block.timestamp + 1 hours, nonce:1
    });

    uint256 balanceOfUserBeforeMint = user.balance;
    console.log("Balance of user BEFORE MINT", balanceOfUserBeforeMint);

    console.log("Passing 1 ether in msg.value while VOUCHER token address is passed in the mint function
    parameter");

    // uint256 tokenId = game.mint(m, abi.encodePacked(mr, ms, mv));
    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));
    vm.prank(user);
    uint256 tokenId = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));

    console.log("Balance of user AFTER MINT", user.balance);
    vm.warp(block.timestamp + 7 days);

}
```

```
gurkiratsingh@Gurkirats-MacBook-Air evm % forge test --mt test_POC_Mint_Sending_ETH_With_Voucher_Address_Burn -vv
[⠊] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Metagame.t.sol:MetagameTest
[FAIL: ExcessValue()] test_POC_Mint_Sending_ETH_With_Voucher_Address_Burn() (gas: 77985)
Logs:
  Balance of user BEFORE MINT 3000000000000000000
  Passing 1 ether in msg.value while VOUCHER token address is passed in the mint function parameter

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.08ms (1.10ms CPU time)

Ran 1 test suite in 4.19ms (3.08ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/Metagame.t.sol:MetagameTest
[FAIL: ExcessValue()] test_POC_Mint_Sending_ETH_With_Voucher_Address_Burn() (gas: 77985)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Low

[L-01] Functions not used internally could be marked as external

Severity
LOW

Location	Functions/Variables
`Metagame.sol, MetagameConfig.sol`	`➔ certain functions`

Issue Description

Within the Metagame.sol, MetagameConfig.sol, certain functions are declared with the public visibility but are never invoked internally by other contract functions. In Solidity, functions that are intended to be accessed only from outside the contract should instead be declared as external. Using public unnecessarily increases the compiled bytecode size and gas costs, since the compiler generates additional code to allow internal calls.

Impact / Proof of Concept

- external functions are slightly more gas efficient than public functions when called from outside the contract, as calldata is read directly rather than being copied to memory.
- Using external signals to developers and auditors that the function is not intended for internal use, improving readability and maintainability.
- This is not a security issue, but a best practice for contract design and optimization.

Recommendation
Change the visibility of functions that are not called internally from public to external.

Status - Fixed in Pull Request 87b0d61

[L-02] State Change Without Event Emission

Severity
LOW

Location	Functions/Variables
`Metagame.sol`	`➔ emergencyWithdraw()`

Issue Description

The emergencyWithdraw() function allows the contract owner to withdraw the entire native balance from the contract. While the withdrawal logic is correct and access-controlled, the function **does not emit any event** when funds are withdrawn.

Impact / Proof of Concept

- Without events, off-chain systems (such as dApps, analytics dashboards, and monitoring systems) cannot reliably track or verify contract activity. This reduces transparency for users and stakeholders.
- Users and integrators rely on events to receive real time updates about their actions (e.g., successful participation, claim,or repayment). The absence of events can lead to confusion and bad UX.
- Many DeFi protocols and monitoring tools depend on events to trigger automated processes (e.g., updating balances, sending notifications). Lack of events can break these integrations.

Recommendation

Emit a withdrawal event:

```
event EmergencyWithdraw(address indexed owner, uint256 amount);

function emergencyWithdraw() external nonReentrant {
   Storage storage $ = _getOwnStorage();
   if ($.owner != msg.sender) revert Unauthorized();

   uint256 balance = address(this).balance;
   if (balance > 0) {
       (bool success,) = payable(msg.sender).call{value: balance}("");
       if (!success) revert FailedTransfer();
       emit EmergencyWithdraw(msg.sender, balance);
   }
}
```

This makes emergency fund movement trackable and aligns with common best practices in protocol design.

Status - Fixed in Pull Request [fec3314]

Info

[I-01] Repeated Owner Checks In Administrative Functions

Severity
INFO

Location	Functions
`MetagameConfig.sol`	`➔ Multiple Functions (listed below)`

Issue Description

The contract performs an explicit owner check inside every administrative function:

```
Storage storage $ = _getOwnStorage();
if ($.owner != msg.sender) revert Unauthorized();
```

This pattern appears across:

- setConfig
- setPaused
- setSigner
- setTreasury
- setFeeBPS
- setBaseURI

Each function embeds the same inline ownership check, resulting in:

- Duplicated logic
- Slightly larger bytecode size

Impact / Proof of Concept

- No direct impact on the smart contract, the contract becomes more gas efficient.
- Slightly higher deployment gas cost.

Recommendation

Introduce a dedicated onlyOwner modifier in MetagameConfig or a shared base contract:

```
modifier onlyOwner() {
    if (_getOwnStorage().owner != msg.sender) revert Unauthorized();
    _;
}
```

Now add this modifier to each of the administrative functions, so that duplicate code is reduced.

Status - Acknowledged

[I-02] Solidity Pragma Should Be Specific, Not Wide

Severity
INFO

Location	Contracts
`Metagame, MetagameConfig, MetagameErrors, MetagameStorage`	➔ `Metagame, MetagameConfig, MetagameErrors, MetagameStorage`

Issue Description

In the mentioned smart contracts, the pragma directive pragma solidity ^0.8.20; is used to specify the compiler version. This directive allows any compiler version greater than or equal to 0.8.20;.

Impact

Using a wide version in the Solidity pragma statement ^0.8.20 is discouraged. It's recommended to specify a particular version to ensure compatibility and avoid unexpected behavior due to potential breaking changes in future compiler versions.

Recommendation

Update the pragma statements in the contracts to specify a particular version of Solidity. For example:

```
pragma solidity 0.8.20;
```

Status - Fixed in Pull Request 164a52d

[I-03] Redundant import of IERC20 interface in Metagame.sol

Severity
INFO

Location	Contract
`Metagame.sol`	➔ `Metagame`

Issue Description

The IERC20 interface is imported in the Metagame.sol file but is not used anywhere within the contract. Redundant imports increase code clutter and may cause unnecessary compilation overhead.

```
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

Impact

No functional vulnerability, but reduces code cleanliness and readability.

Recommendation

Remove the unused import statement for IERC20 from Metagame.sol to maintain a cleaner and more efficient codebase.

Status - Fixed in Pull Request f68a569

[I-04] Missing Minimum-Fee Validation burn function in Allows Zero-Fee Operations in MATURE and REROLL Modes

Severity
INFO

Location	Contracts
`Metagame.sol`	`➔ burn()`

Issue Description

The burn function allows users to modify their position through four modes: CLOSE, CANCEL, MATURE, and REROLL.

The function does not enforce any minimum required fee for the MATURE and REROLL modes. It only checks:

- mintParams.amount < params.fee (upper bound)
- msg.value < params.fee (insufficient value)

However, there is **no lower-bound or minimum-fee requirement**. This means the caller can simply provide params.fee = 0, which is treated as valid by the function.

Impact/ Proof of Concept

1. **MATURE mode**
    - Users can set params.fee = 0, burn a position, receive **full principal**, and the treasury receives **0**, completely bypassing intended fee revenue.
2. **REROLL mode**
    - Users can set params.fee = 0, and as long as msg.value >= 0, both checks pass.
    - They can trigger Reroll without paying any fee.
    - This removes the economic barrier intended by the design and enables costless infinite rerolls.

This results in direct **revenue loss for the treasury** and enables **economic abuse** of system mechanics.

Here is POC for the Bug explaining that as the fee is set to zero users are still able to MATURE AND REROLL positions.

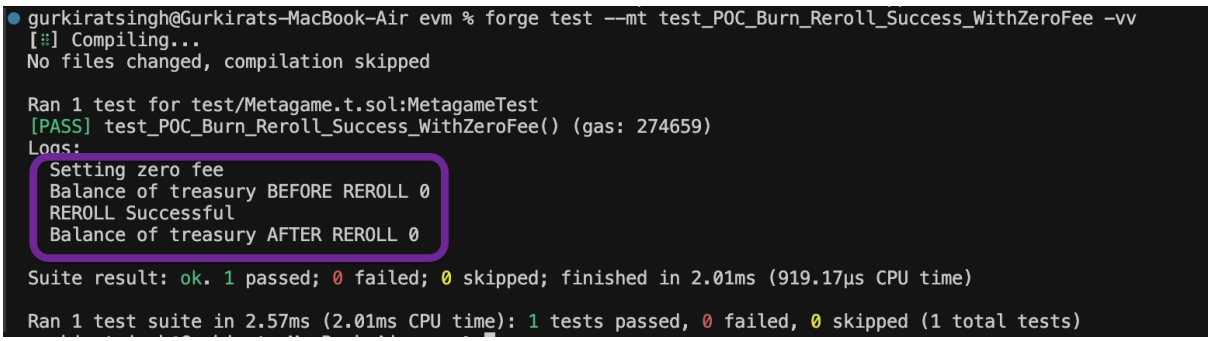
MATURE POSITION

```
function test_POC_Burn_Mature_Success_WithZeroFee() public {
    address treasuryAddr = makeAddr("treasury");
    game.setTreasury(treasuryAddr);
    address user = makeAddr("user");
    vm.deal(user, 2 ether);

    // Mint native
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({
        user: user,
        token: game.NATIVE(),
        amount: 1 ether,
        duration: block.timestamp + 1 days,
        allowance: 0,
        deadline: block.timestamp + 1 hours
    });
    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));
    vm.prank(user);
    uint256 tokenId = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));

    vm.warp(block.timestamp + 2 days);

    // Mature with zero fee
    console.log("Setting zero fee");
    uint256 fee = 0;
    uint256 userBefore = user.balance;
    uint256 treasuryBefore = treasuryAddr.balance;
    console.log("Balance of treasury BEFORE MATURE", treasuryBefore);

    MetagameStorage.BurnParams memory b = MetagameStorage.BurnParams({
        user: user, tokenId: tokenId, mode: 3, fee: fee, deadline: block.timestamp + 2 days + 1 hours});
    (uint8 bv, bytes32 br, bytes32 bs) = vm.sign(signerPrivateKey, game.getBurnParamsHash(b));
    vm.prank(user);
    game.burn(b, abi.encodePacked(br, bs, bv));
    console.log("MATURE Successful");

    uint256 treasuryAfter = treasuryAddr.balance;
    console.log("Balance of treasury AFTER MATURE", treasuryAfter);

    // Token burned, user got amount, treasury got zero fee
    vm.expectRevert();
    game.ownerOf(tokenId);
    assertEq(treasuryAddr.balance, treasuryBefore);
```

```
gurkiratsingh@Gurkirats-MacBook-Air evm % forge test --mt test_POC_Burn_Mature_Success_WithZeroFee -vv
[⠊] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Metagame.t.sol:MetagameTest
[PASS] test_POC_Burn_Mature_Success_WithZeroFee() (gas: 231020)
Logs:
  Setting zero fee
  Balance of treasury BEFORE MATURE 0
  MATURE Successful
  Balance of treasury AFTER MATURE 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.22ms (1.61ms CPU time)

Ran 1 test suite in 4.18ms (3.22ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

REROLL POSITION

```
function test_POC_Burn_Reroll_Success_WithZeroFee() public {
    address treasuryAddr = makeAddr("treasury");
    game.setTreasury(treasuryAddr);
    address user = makeAddr("user");
    vm.deal(user, 3 ether);

    // Mint native
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({
        user: user,
        token: game.NATIVE(),
        amount: 1 ether,
        duration: block.timestamp + 30 days,
        allowance: 0,
        deadline: block.timestamp + 1 hours
    });
    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));
    vm.prank(user);
    uint256 tokenId = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));

    // REROLL charges fee, emits Reroll, does not burn
    console.log("Setting zero fee");
    uint256 fee = 0;
    uint256 treasuryBefore = treasuryAddr.balance;

    console.log("Balance of treasury BEFORE REROLL", treasuryAddr.balance);

    MetagameStorage.BurnParams memory b = MetagameStorage.BurnParams({
        user: user, tokenId: tokenId, mode: 4, fee: fee, deadline: block.timestamp + 1 hours
    });
    (uint8 bv, bytes32 br, bytes32 bs) = vm.sign(signerPrivateKey, game.getBurnParamsHash(b));
    vm.prank(user);
    game.burn{value: fee}(b, abi.encodePacked(br, bs, bv));
    console.log("REROLL Successful");

    uint256 treasuryAfter = treasuryAddr.balance;
    console.log("Balance of treasury AFTER REROLL", treasuryAfter);

    // Not burned, treasury received zero fees
    assertEq(game.ownerOf(tokenId), user);
    assertEq(treasuryAfter, treasuryBefore);
}
```



Recommendation

Implement a minimum fee requirement or validate fees against protocol defined parameters to ensure users cannot supply params.fee = 0 for MATURE and REROLL modes.

Status - Acknowledged

Re-Audit

Client Comment - (Handled Off-Chain)

Contract does not impose constraints on absence of fees:

- Fees could be zero, depends on game design / external business logic.
- Burn params are signed by a trusted party.

[I-05] Unrestricted Reroll Logic Allows Infinite Rerolls Without Checking Eligibility or State Tracking

Severity
INFO

Location	Contracts
`Metagame.sol`	`➔ burn()`

Issue Description

The burn() function's REROLL mode allows users to perform rerolls indefinitely without any eligibility checks or updates to position state (mintParams).Which means users can unknowingly pay fees multiple times while achieving nothing.

This breaks expected business logic and can lead to inconsistent or unintended contract state.

Impact / Proof of Concept

- Users can call REROLL multiple times without maturing their position.
- There is no mechanism to define how long a reroll lasts or when it can be triggered again.
- The function emits a Reroll event and transfers the fee to the treasury, but it **does not update duration, maturity, or cooldown**.

In the POC below we can see the user has rerolled multiple times without any checks or any update in the state, so the user pays fees multiple times without achieving anything.

```
function test_POC_Burn_Mutiple_Reroll_Success_For_A_User() public {
    address treasuryAddr = makeAddr("treasury");
    game.setTreasury(treasuryAddr);
    address user = makeAddr("user");
    vm.deal(user, 3 ether);

    // Mint native
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({user: user,token: game.NATIVE(),amount: 1 ether,duration: block.timestamp + 30 days,allowance: 0,deadline: block.timestamp + 1
    hours});

    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));
    vm.prank(user);
    uint256 tokenId = game.mint{value: 1 ether}(m, abi.encodePacked(mr, ms, mv));

    // REROLL charges fee, emits Reroll
    uint256 fee = 0.5 ether;
    uint256 treasuryBefore = treasuryAddr.balance;

    console.log("Balance of treasury BEFORE REROLL", treasuryAddr.balance);
    MetagameStorage.BurnParams memory b = MetagameStorage.BurnParams({
        user: user, tokenId: tokenId, mode: 4, fee: fee, deadline: block.timestamp + 1 hours
    });

    (uint8 bv, bytes32 br, bytes32 bs) = vm.sign(signerPrivateKey, game.getBurnParamsHash(b));
    vm.prank(user);
    game.burn{value: fee}(b, abi.encodePacked(br, bs, bv));
    console.log("REROLL 1 Successful");
    uint256 treasuryAfter1 = treasuryAddr.balance;
    console.log("Balance of treasury AFTER REROLL 1", treasuryAfter1);

    (uint8 bv1, bytes32 br1, bytes32 bs1) = vm.sign(signerPrivateKey, game.getBurnParamsHash(b));
    vm.prank(user);
    game.burn{value: fee}(b, abi.encodePacked(br1, bs1, bv1));
    console.log("REROLL 2 Successful");
    uint256 treasuryAfter2 = treasuryAddr.balance;
    console.log("Balance of treasury AFTER REROLL 2", treasuryAfter2);

    (uint8 bv3, bytes32 br3, bytes32 bs3) = vm.sign(signerPrivateKey, game.getBurnParamsHash(b));
    vm.prank(user);
    game.burn{value: fee}(b, abi.encodePacked(br3, bs3, bv3));
    console.log("REROLL 3 Successful");
    uint256 treasuryAfter3 = treasuryAddr.balance;
    console.log("Balance of treasury AFTER REROLL 3", treasuryAfter3);

    // Treasury received  multiple fees
    assertEq(game.ownerOf(tokenId), user);
    assertGt(treasuryAfter3, treasuryBefore);
}
```

REROLL 1

REROLL 2

REROLL 3

```
gurkiratsingh@Gurkirats-MacBook-Air evm % forge test --mt test_POC_Burn_Mutiple_Reroll_Success_For_A_User -vv
[⠊] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Metagame.t.sol:MetagameTest
[PASS] test_POC_Burn_Mutiple_Reroll_Success_For_A_User() (gas: 382033)
Logs:
  Balance of treasury BEFORE REROLL 0
  REROLL 1 Successful
  Balance of treasury AFTER REROLL 1 500000000000000000
  REROLL 2 Successful
  Balance of treasury AFTER REROLL 2 1000000000000000000
  REROLL 3 Successful
  Balance of treasury AFTER REROLL 3 1500000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.49ms (1.40ms CPU time)

Ran 1 test suite in 2.87ms (2.49ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation

The protocol should clearly define the intended economic behavior for REROLL. Possible fixes could include:

- Introduce **eligibility conditions** (e.g., require maturity or minimum holding period before reroll).
- Add a **reroll timestamp** or **counter** in storage to track last reroll and prevent spam.
- Update mintParams.duration or create a new variable (e.g., rerollExpiry) to define the new active duration post-reroll.
- Consider limiting rerolls to **one per position** or per defined epoch.

Status - Acknowledged

Re-Audit

Client Comment - (Handled Off-Chain)

The purpose of REROLL is to track user payment being made on chain. To credit user ability to re-play a minigame (offchain). Re-roll fees are assigned and tracked separately, all burn params must be signed by a trusted party.

[I-06] User-Controlled Voucher Allowance Allows Unlimited Voucher Redemption

Severity
INFO

Location	Functions
`Metagame.sol`	➔ `mint()`

Issue Description

The voucher minting flow is intended to enforce a strict redemption limit using:

```
if ($.redeemedVouchers[msg.sender] + params.amount > params.allowance)
{
    revert InsufficientVouchers();
}
```

However, the allowance used in this comparison is taken **directly from user-supplied MintParams**, which is signed but not verified.

This creates a flawed trust model:

- In reality, a malicious user can request a signature with an arbitrarily high allowance.

Impact / Proof of Concept

- Users can obtain more voucher backed positions than intended.

In the POC below we can see that the user only supplies MintParams while calling the mint() function, so the user can set very high numbers in the allowance and mint as much as vouchers he wants and hence staking a lot of NFT amount in the smart contract.

```
function test_POC_Mint_With_High_Allowance_Set_By_User() public {
    address user = makeAddr("user");
    vm.deal(user, 3 ether);

    console.log("Passing 1000 ether as allowance and 500 ether as amount");
    // Mint a position
    MetagameStorage.MintParams memory m = MetagameStorage.MintParams({
        user: user, token: game.VOUCHER(), amount: 500 ether, duration: block.timestamp + 7 days,
        allowance: 1000 ether, deadline: block.timestamp + 1 hours
    });

    uint256 balanceOfUserBeforeMint = user.balance;

    (uint8 mv, bytes32 mr, bytes32 ms) = vm.sign(signerPrivateKey, game.getMintParamsHash(m));
    vm.prank(user);
    uint256 tokenId = game.mint(m, abi.encodePacked(mr, ms, mv));

    console.log("Mint Successfull");
}
```

```
gurkiratsingh@Gurkirats-MacBook-Air evm % forge test --mt test_POC_Mint_With_High_Allowance_Set_By_User -vv
[⠊] Compiling...
No files changed, compilation skipped

Ran 1 test for test/Metagame.t.sol:MetagameTest
[PASS] test_POC_Mint_With_High_Allowance_Set_By_User() (gas: 274501)
Logs:
  Passing 1000 ether as allowance and 500 ether as amount
  Mint Successfull

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.92ms (618.33µs CPU time)

Ran 1 test suite in 2.84ms (1.92ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommendation

Enforce on-chain allowance system rather than trusting user supplied values.

Status - Acknowledged

Re-Audit

Client Comment - (Handled Off-Chain)

Contract does not impose constraints on external (offchain) allowance management:

- Fees could be zero, depends on game design / external business logic.
- Mint params are estimated and signed by a trusted party.

[I-07] MATURE Burn Mode Provides No Meaningful Functionality and Is Bypassed by Existing CLOSE or CANCEL Behavior

Severity
INFO

Location	Functions
`Metagame.sol`	`➔ burn ()`

Issue Description

The burn() function supports four modes: CLOSE, CANCEL, MATURE, and REROLL. According to documentation, MATURE mode is intended to:

- return the principal minus a fee
- send the fee to the treasury
- serve as a fee-based exit path after maturity

However, logic reveals that MATURE mode is **redundant**, because CLOSE and CANCEL mode already provide the same functionality **without charging any fee**.

Since there are only two options that a user could choose once for the position of the stake, one is before Mature and one is after Mature, so both these options are available without paying any fee so why would the user pay any fee and choose MATURE mode in the burn function.

There is **no scenario** in which a rational user would select MATURE.

The result is that the entire MATURE pathway is never economically optimal or required.

Impact / Proof of Concept

- The MATURE mode fee mechanism becomes dead code and will never be used.
- The treasury never receives maturity fees, breaking the fee model described in documentation.

Recommendation

The protocol should clearly define the intended economic behavior for matured exits. Possible fixes could include:

- **If fees are not intended at maturity:** remove MATURE and rely entirely on CLOSE for matured positions.
- **If fees are intended at maturity:** restrict or redesign CLOSE so users cannot bypass the fee by choosing a free path.

Status - Acknowledged

Re-Audit

Client Comment - (Handled Off-Chain)

Contract defines distinct actions (with it's underlying economic behavior):

- MATURE burn events are tracked and used off-chain.
- MATURE, CANCEL and CLOSE fees and intents are inherently different.

Clarification w/ Example

User staked 100 KARRAT for 5 days. After 1 day user decided to close the position to withdraw:

- If CANCEL, user notional will be reduced by the fee amount, that is dynamically calculated based on position age: 1 day. Cancellation would mean that user is explicitly opting out of participation in the game. They wont get off-chain reward associated w/ staking KARRAT.
- If MATURE, user notional will be reduced by the fee amount, that is dynamically calculated based on time left to maturity: 4 days. Maturity would mean that user is explicitly paying extra fee to close their position before staking duration has been reached. They will receive off-chain reward associated w/ staking KARRAT.

6.0 Auditing Approach and Methodologies Applied

The Solidity smart contract was audited using a comprehensive approach to ensure the highest level of security and reliability. Careful attention was given to the following key areas to ensure the overall quality of the code:

- **Code quality and structure**: We conducted a detailed review of the codebase to identify any potential issues related to code structure, readability, and maintainability. This included analysing the overall architecture of the Solidity smart contract and reviewing the code to ensure it follows best practices and coding standards.
- **Security vulnerabilities**: Our team used manual techniques to identify any potential security vulnerabilities that could be exploited by attackers. This involved a thorough analysis of the code to identify any potential weaknesses, such as buffer overflows, injection vulnerabilities, signatures, and deprecated functions.
- **Documentation and comments**: Our team reviewed the code documentation and comments to ensure they accurately describe the code's intended behaviour and logic. This helps developers to better understand the codebase and make modifications without introducing new issues.
- **Compliance with best practices**: We checked that the code follows best practices and coding standards that are recommended by the Solidity community and industry experts. This ensures that the Solidity smart contract is secure, reliable, and efficient.

Our audit team followed OWASP and Ethereum (Solidity) community security guidelines for this audit. As a result, we were able to identify potential issues and provide recommendations to improve the smart contract's security and performance.

Throughout the audit of the smart contracts, our team placed great emphasis on ensuring the overall quality of the code and the use of industry best practices. We meticulously reviewed the codebase to ensure that it was thoroughly documented and that all comments and logic aligned with the intended behaviour. Our approach to the audit was comprehensive, methodical, and aimed at ensuring that the smart contract was secure, reliable, and optimised for performance.

6.1 Code Review / Manual Analysis

Our team conducted a manual analysis of the Solidity smart contracts to identify new vulnerabilities or to verify vulnerabilities found during static and manual analysis. We carefully analysed every line of code and made sure that all instructions provided during the onboarding phase were followed. Through our manual analysis, we were able to identify potential vulnerabilities that may have been missed by automated tools and ensure that the smart contract was secure and reliable.

6.2 Tools Used for Audit

In the course of our audit, we leveraged a suite of tools to bolster the security and performance of our program. While our team drew on their expertise and industry best practices, we also integrated various tools into our development environment. Noteworthy among them are Remix,Slither, Aderyn, Solhint for Static Analysis and Hardhat & Foundry for Dynamic Analysis. This holistic approach ensures a thorough analysis, uncovering potential issues that automated tools alone might overlook. 0xTeam takes pride in utilising these tools, which significantly contribute to the quality, security, and maintainability of our codebase

7.0 Limitations on Disclosure and Use of this Report

This report contains information concerning potential details of the Metagame Project and methods for exploiting them. 0xTeam recommends that special precautions be taken to protect the confidentiality of both this document and the information contained herein. Security Assessment is an uncertain process, based on past experiences, currently available information, and known threats. All information security systems, which by their nature are dependent on human beings, are vulnerable to some degree. Therefore, while 0xTeam considers the major security vulnerabilities of the analysed systems to have been identified, there can be no assurance that any exercise of this nature will identify all possible vulnerabilities or propose exhaustive and operationally viable recommendations to mitigate those exposures. In addition, the analysis set forth herein is based on the technologies and known threats as of the date of this report. As technologies and risks change over time, the vulnerabilities associated with the operation of the Metagame Smart contracts Code Base described in this report, as well as the actions necessary to reduce the exposure to such vulnerabilities, will also change. 0xTeam makes no undertaking to supplement or update this report based on changed circumstances or facts of which 0xTeam becomes aware after the date hereof, absent a specific written agreement to perform the supplemental or updated analysis. This report may recommend that 0xTeam use certain software or hardware products manufactured or maintained by other vendors. 0xTeam bases these recommendations upon its prior experience with the capabilities of those products. Nonetheless, 0xTeam does not and cannot warrant that a particular product will work as advertised by the vendor, nor that it will operate in the manner intended. This report was prepared by 0xTeam for the exclusive benefit of Metagame and is proprietary information. The Non-Disclosure Agreement (NDA) in effect between 0xTeam and Metagame governs the disclosure of this report to all other parties including product vendors and suppliers.