# Competitive Security Assessment

## Game7Hyperplay

Oct 27th, 2023

Secure3

secure3.io

# Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

# Overview

### Project Detail

| | |
|---|---|
| **Project Name** | Game7Hyperplay |
| **Platform & Language** | Solidity |
| **Codebase** | • audit codebase - https://gitfront.io/r/user-3808373/2DuWU5ScT8f4/achievo-contracts/blob/contracts/GameSummary.sol<br>• final commit - to be provided |
| **Audit Methodology** | • Audit Contest<br>• Business Logic and Code Review<br>• Privileged Roles Review<br>• Static Analysis |

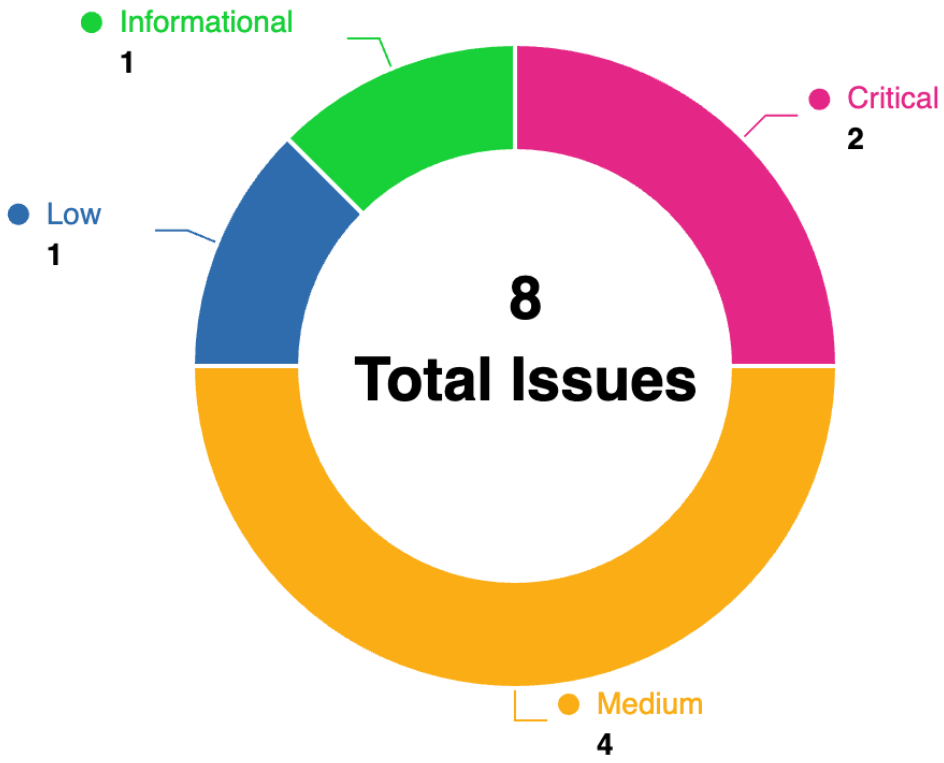
### Code Vulnerability Review Summary

| Vulnerability Level | Total | Reported | Acknowledged | Fixed | Mitigated | Declined |
|---|---|---|---|---|---|---|
| Critical | 2 | 2 | 0 | 0 | 0 | 0 |
| Medium | 4 | 4 | 0 | 0 | 0 | 0 |
| Low | 1 | 1 | 0 | 0 | 0 | 0 |
| Informational | 1 | 1 | 0 | 0 | 0 | 0 |

# Audit Scope

| File | SHA256 Hash |
| --- | --- |
| ./GameSummary.sol | 9d3cc55398755a48bd85123b4af0e8412697e736f74e1b5cb32d3518a0dcfb97 |

# Code Assessment Findings




| ID | Name | Category | Severity | Client Response | Contributor |
|---|---|---|---|---|---|
| G7H-1 | There exists collisions for `tokenId` crafted with `storeId` and `gameId` | Logical | Critical | Reported | 0xac, slowfrog, Yaodao |
| G7H-2 | Potential risk of using signature repeatedly | Signature Forgery or Replay | Critical | Reported | 0xac, Yaodao, yekong, slowfrog |
| G7H-3 | Potentially add achievements to the burned token | Logical | Medium | Reported | Yaodao |

| | | | | | |
|---|---|---|---|---|---|
| G7H-4 | Inconsistent and Unrestricted Achievement Update Vulnerability | Logical | Medium | Reported | yekong |
| G7H-5 | Unable to call the `safeTransferFrom()` and `safeBatchTransferFrom()` again due to transferring the token without fully updating the token's playerGameData information | Logical | Medium | Reported | Yaodao |
| G7H-6 | Potential incorrect number of tokens burned due to the given amount via the parameter | Logical | Medium | Reported | Yaodao |
| G7H-7 | Lack of the exist check for the `gameSummary` in `mintGameSummary()` | Logical | Low | Reported | Yaodao |
| G7H-8 | `verifySignature(...)` if statement can be simplified | Gas Optimization | Informational | Reported | plasmablocks |

# G7H-1:There exists collisions for `tokenId` crafted with `storeId` and `gameId`

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Critical | Reported | 0xac, slowfrog, Yaodao |

## Code Reference

- code/GameSummary.sol#L64-L71
- code/GameSummary.sol#L197-L211
- code/GameSummary.sol#L213-L226
- code/GameSummary.sol#L238-L247

```
64:function concat(uint256 storeId, uint256 gameId) private pure returns (uint256) {
65:    string memory gameIdStr = Strings.toString(gameId);
66:    string memory storeIdStr = Strings.toString(storeId);
67:    string memory zero = "0";
68:    string memory concatenatedString = string(abi.encodePacked(storeIdStr, zero, gameIdStr));
69:    uint256 concatenatedUint = stringToUint(concatenatedString);
70:    return concatenatedUint;
71:  }

197:function createCommonGameSummary(
198:    uint256 storeId,
199:    uint256 gameId,
200:    string memory name,
201:    string memory onChainURI,
202:    string memory externalURI,
203:    uint256 totalAchievements
204:  ) public onlyRole(GAME_CREATOR_ROLE) {
205:    require(gameId > 0, "GameId must be greater than 0");
206:    require(storeId > 0, "StoreId must be greater than 0");
207:    uint256 tokenId = concat(storeId, gameId);
208:    require(commonGameSummaries[tokenId].storeId == 0, "Token already exists");
209:    commonGameSummaries[tokenId] = GameSummary(storeId, gameId, name, onChainURI, externalURI, t
otalAchievements);
210:    emit GameSummaryMinted(msg.sender, tokenId, totalAchievements);
211:  }

213:function mintGameSummary(
214:    address player,
215:    uint256 gameId,
216:    uint256 achievementsLength,
217:    uint256 storeId,
218:    bool soulBound
219:  ) private {
220:    require(storeId > 0, "StoreId must be greater than 0");
221:    uint256 tokenId = concat(storeId, gameId);
222:    require(playerGameData[player][tokenId].tokenId == 0, "Token already exists");
223:    _mint(player, tokenId, 1, "");
224:    playerGameData[player][tokenId] = PlayerGameData(tokenId, achievementsLength, soulBound);
225:    emit PlayerGameSummaryMinted(player, tokenId, achievementsLength);
226:  }
```

```
238:function mintGameSummaryWithSignature(
239:    uint256 gameId,
240:    uint256 achievementsLength,
241:    uint256 storeId,
242:    uint256 nonce,
243:    bytes memory signature
244:  ) public nonReentrant notGameSummaryMintPaused {
245:    require(verifySignature(nonce, signature), "Invalid signature");
246:    mintGameSummary(msg.sender, gameId, achievementsLength, storeId, true);
247:  }
```

## Description

**0xac :** Different inputs to the concat() function have the same output: consider two different numbers: storeId = 10 and gameId = 201; and storeId = 1002 and gameId = 1. These two sets of inputs will produce a result of 100201 after being processed by the concat function. This is what happens with collisions.

Example: For input (10,201): storeIdStr = "10" gameIdStr = "201" Concatenated string = "10" + "0" + "201" = "100201" Convert to number = 100201

For input (1002,1): storeIdStr = "1002" gameIdStr = "1" Concatenated string = "1002" + "0" + "1" = "100201" Convert to number = 100201

As you can see, although the two sets of inputs are different, they both produce the same output

**slowfrog :** `tokenId` is used as the key for both `commonGameSummaries` and `playerGameData`, and it is crafted by concatenating `storeId`, `0`, and `gameId` in order with `abi.encodePacked`. But we could craft collision for this concatenated string:

```
string memory s = string(abi.encodePacked("1","0","101"));
//  → s
// Type: string
// ├ UTF-8: 10101

string memory _s = string(abi.encodePacked("101","0","1"));
// → _s
// Type: string
// ├ UTF-8: 10101
```

we could see with `storeId` as `1` and `gameId` as `101`, we would get `tokenId` `10101`. While with `storeId` as `101` and `gameId` as `1`, we get the same `tokenId` `10101`. If the first case happened first, we would never be able to `createCommonSummary` and `mintGameSummary` for the second case.

**Yaodao :** The `gameId` and `storeId` are concatenated by 0 and generate the `tokenId`. Since the `gameId` and `storeId` in the contract only need to be greater than 0, and the `tokenId` means that it contains information about the game and the store.

```
  function concat(uint256 storeId, uint256 gameId) private pure returns (uint256) {
    string memory gameIdStr = Strings.toString(gameId);
    string memory storeIdStr = Strings.toString(storeId);
    string memory zero = "0";
    string memory concatenatedString = string(abi.encodePacked(storeIdStr, zero, gameIdStr));
    uint256 concatenatedUint = stringToUint(concatenatedString);
    return concatenatedUint;
  }
```

So the `tokenId` for the different game and store may be same. For example:

1. storeId = 1 and gameId = 101 => tokenId = 10101
2. storeId = 101 and gameId = 1 => tokenId = 10101

As a result, the same `tokenId` for different game and store will make the information recorded in the contract be incorrect.

## Recommendation

**0xac :** It is recommended to use a fixed length prefix: In this way, no matter what the actual values of storeId and gameId are, their combination will be unique.

```
function combineIds(uint256 storeId, uint256 gameId) public pure returns (uint256) {
    // Ensure both storeId and gameId do not exceed uint128's maximum value
    require(storeId <= type(uint128).max, "storeId exceeds uint128 maximum value");
    require(gameId <= type(uint128).max, "gameId exceeds uint128 maximum value");

    // Extract the last 16 bytes (128 bits) from storeId and gameId
    bytes16 storeIdBytes = bytes16(uint128(storeId));
    bytes16 gameIdBytes = bytes16(uint128(gameId));

    // Combine both 16-byte values into a single 32-byte value
    bytes32 combined = bytes32((uint256(uint128(storeIdBytes)) << 128) | uint128(gameIdBytes));

    // Convert combined 32-byte value to uint256 and return
    return uint256(combined);
}
```

**slowfrog :** use `keccak256(abi.encode(storeId, gameId))` instead:

```
function getTokenId(uint256 storeId, uint256 gameId) private pure returns (uint256) {
    uint256 tokenId = uint256(keccak256(abi.encode(storeId, gameId)));
    return tokenId
}
```

**Yaodao :** Recommend adding unique information in the logic of `tokenId` to avoid the condition.

## Client Response

TBD

# G7H-2:Potential risk of using signature repeatedly

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| **Signature Forgery or Replay** | **Critical** | **Reported** | **0xac, Yaodao, yekong, slowfrog** |

## Code Reference

- code/GameSummary.sol#L110-L124
- code/GameSummary.sol#L117-L124
- code/GameSummary.sol#L187-L195
- code/GameSummary.sol#L238-L247
- code/GameSummary.sol#L261-L272
- code/GameSummary.sol#L289-L303

```
110:function recoverAddress(uint256 nonce, bytes memory signature) private view returns (address) {
111:    bytes32 message = keccak256(abi.encodePacked(msg.sender, nonce));
112:    bytes32 hash = ECDSA.toEthSignedMessageHash(message);
113:    address signer = ECDSA.recover(hash, signature);
114:    return signer;
115:  }
116:
117:  function verifySignature(uint256 nonce, bytes memory signature) public view returns (bool) {
118:    address signer = recoverAddress(nonce, signature);
119:    if (whitelistSigners[signer]) {
120:      return true;
121:    } else {
122:      return false;
123:    }
124:  }

117:function verifySignature(uint256 nonce, bytes memory signature) public view returns (bool) {
118:    address signer = recoverAddress(nonce, signature);
119:    if (whitelistSigners[signer]) {
120:      return true;
121:    } else {
122:      return false;
123:    }
124:  }

187:function updatePlayerAchievementsWithSignature(
188:    uint256 tokenId,
189:    uint256 newAchievements,
190:    uint256 nonce,
191:    bytes memory signature
192:  ) public nonReentrant notGameSummaryMintPaused {
193:    require(verifySignature(nonce, signature), "Invalid signature");
194:    addPlayerAchievements(msg.sender, tokenId, newAchievements);
195:  }

238:function mintGameSummaryWithSignature(
239:    uint256 gameId,
240:    uint256 achievementsLength,
241:    uint256 storeId,
```

```
242:     uint256 nonce,
243:     bytes memory signature
244:   ) public nonReentrant notGameSummaryMintPaused {
245:     require(verifySignature(nonce, signature), "Invalid signature");
246:     mintGameSummary(msg.sender, gameId, achievementsLength, storeId, true);
247:   }

261:function batchPlayerUpdateAchievementsWithSignature(
262:     uint256[] calldata tokenIds,
263:     uint256[] calldata newAchievements,
264:     uint256 nonce,
265:     bytes memory signature
266:   ) public nonReentrant notGameSummaryMintPaused {
267:     require(verifySignature(nonce, signature), "Invalid signature");
268:     require(tokenIds.length == newAchievements.length, "The players and newAchievements arrays m
ust have the same length");
269:     for (uint i = 0; i < tokenIds.length; i++) {
270:       addPlayerAchievements(msg.sender, tokenIds[i], newAchievements[i]);
271:     }
272:   }

289:function batchMintGameSummaryWithSignature(
290:     uint256[] calldata gameIds,
291:     uint256[] calldata newAchievements,
292:     uint256[] calldata storeIds,
293:     uint256 nonce,
294:     bytes memory signature
295:   ) public notGameSummaryMintPaused nonReentrant {
296:     require(verifySignature(nonce, signature), "Invalid signature");
297:     require(gameIds.length == storeIds.length, "The gameIds and storeIds arrays must have the sa
me length");
298:     require(gameIds.length == newAchievements.length, "The gameIds and newAchievements arrays mu
st have the same length");
299:
300:     for (uint i = 0; i < gameIds.length; i++) {
301:       mintGameSummary(msg.sender, gameIds[i], newAchievements[i], storeIds[i], true);
302:     }
303:   }
```

## Description

**0xac :** In GameSummary contract, the `verifySignature()` function is called by `updatePlayerAchievementsWithSignature()`, `mintGameSummaryWithSignature()`, `batchPlayerUpdateAchievementsWithSignature`

`()`, `batchMintGameSummaryWithSignature()` functions to verify the signature. However, `verifySignature(uint256 nonce, bytes memory signature)` function receives the `nonce` parameter from user. Onece a user get a signature, it can use this signature any number of times with its special `nonce`. Attacker can use this `nonce` and `signature` cuple to call `mintGameSummary()` and `addPlayerAchievements()` with any parameter designed by him.

```
function recoverAddress(uint256 nonce, bytes memory signature) private view returns (address) {
  bytes32 message = keccak256(abi.encodePacked(msg.sender, nonce));
  bytes32 hash = ECDSA.toEthSignedMessageHash(message);
  address signer = ECDSA.recover(hash, signature);
  return signer;
}

function verifySignature(uint256 nonce, bytes memory signature) public view returns (bool) {
  address signer = recoverAddress(nonce, signature);
  if (whitelistSigners[signer]) {
    return true;
  } else {
    return false;
  }
}
```

**Yaodao :** In the functions `updatePlayerAchievementsWithSignature()`, `mintGameSummaryWithSignature()`, `batchPlayerUpdateAchievementsWithSignature()` and `batchMintGameSummaryWithSignature()`, the `nonce` and `signature` will be passed via params and verified whether it is valid.

However, the `nonce` and `signature` are only checked whether it is valid but no checked whether they have been used. As a result, the caller can use the same `nonce` and `signature` to call these functions repeatedly.

```
function verifySignature(uint256 nonce, bytes memory signature) public view returns (bool) {
  address signer = recoverAddress(nonce, signature);
  if (whitelistSigners[signer]) {
    return true;
  } else {
    return false;
  }
}
```

The function `updatePlayerAchievementsWithSignature()` is used to add new achievements to the player. The user can repeatedly call this function to add far more achievements to the player than expected if there is no limit on the times of a signature can be used.

The similar risks exist on the functions `mintGameSummaryWithSignature()`, `batchPlayerUpdateAchievementsWithSignature()` and `batchMintGameSummaryWithSignature()`.

Hence, it's necessary to use a map to record the use of `signature` and check whether the `signature` has been used.

**Yaodao :** In the functions `updatePlayerAchievementsWithSignature()`, `mintGameSummaryWithSignature()`, `batchPlayerUpdateAchievementsWithSignature()` and `batchMintGameSummaryWithSignature()`, the `nonce` and `signature` will be passed via params and verified whether it is valid.

However, only the `nonce` and `signature` are used in the function `verifySignature()` to check the `signature`, the game information like `gameId`, `storeId`, `newAchievements`, etc is not checked.

```
function verifySignature(uint256 nonce, bytes memory signature) public view returns (bool) {
  address signer = recoverAddress(nonce, signature);
  if (whitelistSigners[signer]) {
    return true;
  } else {
    return false;
  }
}
```

Hence, the `signature` may be incorrect used by the user to call another function.

For example, the `nonce` and `signature` are used for the user to call `updatePlayerAchievementsWithSignature()` to update the achievement of the player. But the user can use the `nonce` and `signature` to call the function `mintGameSummaryWithSignature()`.

**yekong :** The Solidity smart contract in question is designed to manage game summaries and player achievements using ERC1155 tokens on the Ethereum blockchain. A critical logical vulnerability has been identified in the methods that handle the updating of player achievements based on signed messages (updatePlayerAchievementsWithSignature and batchPlayerUpdateAchievementsWithSignature). These methods allow a player to update their achievements by providing a signature, presumably signed by a trusted authority. However, the recoverAddress function, which is used to extract the signer's address from the signature, only takes the player's address and a nonce as the message. This design flaw makes it possible for a malicious player to reuse a valid signature to update achievements for different games (different tokenIds) without any restrictions, as long as the nonce remains constant.

**slowfrog :** `verifySignature` only checks that the `msg.sender` and `nonce` is authenticated by whitelist signers. But as the signature is given before the `msg.sender` signs the all function arguments including signature, this means `msg.sender` could replace other important arguments with unintended values:

```
function updatePlayerAchievementsWithSignature(
    uint256 tokenId,
    uint256 newAchievements,
    uint256 nonce,
    bytes memory signature
  ) public nonReentrant notGameSummaryMintPaused {
    require(verifySignature(nonce, signature), "Invalid signature");
    addPlayerAchievements(msg.sender, tokenId, newAchievements);
  }
```

For example, `msg.sender` could pass arbitrary `tokenId` and `newAchievements` since they are not protected by the signature.

## Recommendation

**0xac :** Suggest to increase the nonce of user afer the signature was used.

```
// address(user) => nonce
mapping(address => uint256) public userNonce;

function recoverAddress(uint256 nonce, bytes memory signature) private view returns (address) {
    bytes32 message = keccak256(abi.encodePacked(msg.sender, nonce));
    bytes32 hash = ECDSA.toEthSignedMessageHash(message);
    address signer = ECDSA.recover(hash, signature);
    return signer;
  }

function verifySignature(bytes memory signature) public view returns (bool) {
    uint256 nonce = userNonce[msg.sender];
    address signer = recoverAddress(nonce, signature);
    if (whitelistSigners[signer]) {
      userNonce[msg.sender] += 1;
      return true;
    } else {
      return false;
    }
  }
```

**Yaodao :** Recommend adding a map to record the use of `signature` and checking the map at the begining of the calls of these functions.

**Yaodao :** Recommend adding logic to add game information into the `signature` to avoid the incorrect use of `signature.`

**yekong :** To mitigate this vulnerability, the message that is signed and subsequently verified should include all relevant parameters that the signed action is authorized to perform. Specifically, the `recoverAddress` function should also take

the `tokenId` and `newAchievements` (and any other relevant parameters) as input, ensuring that the signature explicitly authorizes the update for the specific game and achievements amount.

1. Update the `recoverAddress` function to include all relevant parameters:

```
function recoverAddress(address player, uint256 tokenId, uint256 newAchievements, uint256 nonc
e, bytes memory signature) private view returns (address) {
  bytes32 message = keccak256(abi.encodePacked(player, tokenId, newAchievements, nonce));
  bytes32 hash = ECDSA.toEthSignedMessageHash(message);
  address signer = ECDSA.recover(hash, signature);
  return signer;
}
```

2. Update the `verifySignature` function and all its usages to pass the additional parameters:

```
function verifySignature(address player, uint256 tokenId, uint256 newAchievements, uint256 nonc
e, bytes memory signature) public view returns (bool) {
  address signer = recoverAddress(player, tokenId, newAchievements, nonce, signature);
  return whitelistSigners[signer];
}
```

3. Ensure that all calls to `verifySignature` within the contract are updated to pass the correct parameters.
   **slowfrog :** encode all the arguments with `abi.encode` and whitelist signers should sign the encoded bytes:

```
function recoverAddress(uint256 _nonce, bytes memory _msg, bytes memory signature) private view retu
rns (address) {
    bytes32 message = keccak256(abi.encodePacked(msg.sender, nonce, _msg));
    bytes32 hash = ECDSA.toEthSignedMessageHash(message);
    address signer = ECDSA.recover(hash, signature);
    return signer;
  }

function updatePlayerAchievementsWithSignature(
    uint256 tokenId,
    uint256 newAchievements,
    uint256 _nonce,
    bytes memory signature
  ) public nonReentrant notGameSummaryMintPaused {
    bytes memory _msg = abi.encodePacked(tokenId, newAchievements);
    require(verifySignature(_nonce, _msg, signature), "Invalid signature");
    addPlayerAchievements(msg.sender, tokenId, newAchievements);
  }
```

## Client Response

TBD

# G7H-3:Potentially add achievements to the burned token

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Medium | Reported | Yaodao |

## Code Reference

- code/GameSummary.sol#L167-L177
- code/GameSummary.sol#L324-L331

```
167:function addPlayerAchievements (
168:    address player,
169:    uint256 tokenId,
170:    uint256 newAchievements
171:  ) private {
172:    require(tokenId > 0, "TokenId must be greater than 0");
173:    require(playerGameData[player][tokenId].tokenId != 0, "Token doesn't exists");
174:    PlayerGameData storage playerData = playerGameData[player][tokenId];
175:    playerData.achievementsMinted += newAchievements;
176:    emit PlayerGameSummaryUpdated(player, tokenId, playerData.achievementsMinted);
177:  }

324:function burn(uint256 tokenId, uint256 amount) public nonReentrant {
325:    require(playerGameData[msg.sender][tokenId].tokenId != 0, "Token doesn't exists");
326:    if(playerGameData[msg.sender][tokenId].soulBounded) {
327:      revert("You can't burn this token");
328:    }
329:    _burn(msg.sender, tokenId, amount);
330:    playerGameData[msg.sender][tokenId].achievementsMinted = 0;
331:  }
```

## Description

**Yaodao :** The function `burn()` is used to burn the token. The achievementsMinted will be cleared to `0`, but other information will not be cleared.

The private function `addPlayerAchievements()` is used to add `newAchievements` to the `playerData`. The private function `addPlayerAchievements()` is called by functions `adminUpdatePlayerAchievements()`, `updatePlayerAchievementsWithSignature()`, `adminBatchPlayerUpdateAchievements()` and `batchPlayerU-`

`pdateAchievementsWithSignature()`. All above functions not check whether the `tokenId` is burned because the informaiton of `tokenId` is not cleared in the function `burn()`.

As a result, the admin or the user with signature can add achievements to the burned token.

```
function burn(uint256 tokenId, uint256 amount) public nonReentrant {
  require(playerGameData[msg.sender][tokenId].tokenId != 0, "Token doesn't exists");
  if(playerGameData[msg.sender][tokenId].soulBounded) {
    revert("You can't burn this token");
  }
  _burn(msg.sender, tokenId, amount);
  playerGameData[msg.sender][tokenId].achievementsMinted = 0;
}
```

## Recommendation

**Yaodao :** Recommend clearing the token information when burning the token.

## Client Response

TBD

# G7H-4:Inconsistent and Unrestricted Achievement Update Vulnerability

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Medium | Reported | yekong |

## Code Reference

- code/GameSummary.sol#L167-L177

```
167:function addPlayerAchievements (
168:    address player,
169:    uint256 tokenId,
170:    uint256 newAchievements
171:  ) private {
172:    require(tokenId > 0, "TokenId must be greater than 0");
173:    require(playerGameData[player][tokenId].tokenId != 0, "Token doesn't exists");
174:    PlayerGameData storage playerData = playerGameData[player][tokenId];
175:    playerData.achievementsMinted += newAchievements;
176:    emit PlayerGameSummaryUpdated(player, tokenId, playerData.achievementsMinted);
177:  }
```

## Description

**yekong :** The smart contract has a logical inconsistency and a lack of restriction on the addPlayerAchievements function, which can be exploited by a malicious user. This function is declared as private and is meant to be called only within the contract. It updates the achievements of a player associated with a specific game token. However, the function can be indirectly accessed through the adminUpdatePlayerAchievements and updatePlayerAchievementsWithSignature functions without any checks on the maximum limit of achievements, leading to a potential overflow of achievements.

Moreover, the addPlayerAchievements function does not check whether the new achievements added would exceed the total number of achievements defined in the GameSummary of the common game. This inconsistency could lead to a situation where a player has more achievements than actually possible, deeming the tracking of achievements unreliable and potentially exploitable for benefits tied to the number of achievements.

## Recommendation

**yekong :** To mitigate this vulnerability and ensure the integrity of the achievements data:

Add Achievements Limit Check: Introduce a check in the addPlayerAchievements function to ensure that the sum of existing achievements and new achievements does not exceed the total achievements specified in the GameSummary.

Validate Game Summary: Ensure that the game summary associated with the token exists before allowing updates to player achievements. This can be done by checking if the storeId in the GameSummary is non-zero.

Restrict Direct Updates: If the achievements are meant to be updated only through specific administrative functions or via signatures, ensure that the addPlayerAchievements function is not directly accessible, even indirectly, without proper validations and restrictions.

## Client Response

TBD

# G7H-5:Unable to call the `safeTransferFrom()` and `safeBatchTransferFrom()` again due to transferring the token without fully updating the token's playerGameData information

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Medium | Reported | Yaodao |

## Code Reference

- code/GameSummary.sol#L305-L309
- code/GameSummary.sol#L311-L318

```
305:function safeTransferFrom(address _from, address _to, uint256 _id, uint256 _amount, bytes memory
_data) public virtual override {
306:    require(playerGameData[_from][_id].tokenId != 0, "Token doesn't exists");
307:    require(!playerGameData[_from][_id].soulBounded, "You can't transfer this token");
308:    super.safeTransferFrom(_from, _to, _id, _amount, _data);
309:  }

311:function safeBatchTransferFrom(address _from, address _to, uint256[] memory _ids, uint256[] memo
ry _amounts, bytes memory _data) public virtual override {
312:    for (uint i = 0; i < _ids.length; i++) {
313:      require(playerGameData[_from][_ids[i]].tokenId != 0, "Token doesn't exists");
314:
315:      require(!playerGameData[_from][_ids[i]].soulBounded, "You can't transfer this token");
316:    }
317:    super.safeBatchTransferFrom(_from, _to, _ids, _amounts, _data);
318:  }
```

## Description

**Yaodao :** The functions `safeTransferFrom()` and `safeBatchTransferFrom()` are used to transfer the token to the `to` address from the `from` address. In these two functions, the `playerGameData[_from][_ids[i]].tokenId` and `playerGameData[_from][_ids[i]].soulBounded` will be checked.

However, the `playerGameData` of the token for both `from` and `to` addresses are not updated, only the token is transferred.

As a result, the `to` address can't transfer the token received again when the `playerGameData[_to][_ids[i]].tokenId` or `playerGameData[_to][_ids[i]].soulBounded` can't pass the check.

## Recommendation

**Yaodao :** Recommend fully updating the `playerGameData` of the token for both `from` and `to` addresses.

## Client Response

TBD

# G7H-6:Potential incorrect number of tokens burned due to the given amount via the parameter

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Medium | Reported | Yaodao |

## Code Reference

- code/GameSummary.sol#L213-L226
- code/GameSummary.sol#L324-L331
- code/GameSummary.sol#L333-L344

```
213:function mintGameSummary(
214:    address player,
215:    uint256 gameId,
216:    uint256 achievementsLength,
217:    uint256 storeId,
218:    bool soulBound
219:  ) private {
220:    require(storeId > 0, "StoreId must be greater than 0");
221:    uint256 tokenId = concat(storeId, gameId);
222:    require(playerGameData[player][tokenId].tokenId == 0, "Token already exists");
223:    _mint(player, tokenId, 1, "");
224:    playerGameData[player][tokenId] = PlayerGameData(tokenId, achievementsLength, soulBound);
225:    emit PlayerGameSummaryMinted(player, tokenId, achievementsLength);
226:  }

324:function burn(uint256 tokenId, uint256 amount) public nonReentrant {
325:    require(playerGameData[msg.sender][tokenId].tokenId != 0, "Token doesn't exists");
326:    if(playerGameData[msg.sender][tokenId].soulBounded) {
327:      revert("You can't burn this token");
328:    }
329:    _burn(msg.sender, tokenId, amount);
330:    playerGameData[msg.sender][tokenId].achievementsMinted = 0;
331:  }

333:function burnBatch(uint256[] memory tokenIds, uint256[] memory amounts) public nonReentrant {
334:    for (uint i = 0; i < tokenIds.length; i++) {
335:      require(playerGameData[msg.sender][tokenIds[i]].tokenId != 0, "Token doesn't exists");
336:      if(playerGameData[msg.sender][tokenIds[i]].soulBounded) {
337:        revert("You can't burn this token");
338:      }
339:    }
340:    _burnBatch(msg.sender, tokenIds, amounts);
341:    for (uint i = 0; i < tokenIds.length; i++) {
342:      playerGameData[msg.sender][tokenIds[i]].achievementsMinted = 0;
343:    }
344:  }
```

## Description

**Yaodao :** The mint of tokens is only called in the function `mintGameSummary()`, and the amount to mint is the fixed value 1. As a result, the token amount of each tokenId is always 1.

However, in the burn function, the amount to burn is the given amount via the parameter.

This may lead to two bad conditions.

1. The amount is over 1, the call of `_burn()` or `_burnBatch()` will fail.
2. The amount is 0, the call will success but the 1 token is still exist and the users' achievements information will be cleared.

## Recommendation

**Yaodao :** Recommend using the fixed amount like mint, instead of the given amount via the parameter.

## Client Response

TBD

# G7H-7:Lack of the exist check for the `gameSummary` in `mintGameSummary()`

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Logical | Low | Reported | Yaodao |

## Code Reference

- code/GameSummary.sol#L197-L211
- code/GameSummary.sol#L213-L226

```
197:function createCommonGameSummary(
198:    uint256 storeId,
199:    uint256 gameId,
200:    string memory name,
201:    string memory onChainURI,
202:    string memory externalURI,
203:    uint256 totalAchievements
204:  ) public onlyRole(GAME_CREATOR_ROLE) {
205:    require(gameId > 0, "GameId must be greater than 0");
206:    require(storeId > 0, "StoreId must be greater than 0");
207:    uint256 tokenId = concat(storeId, gameId);
208:    require(commonGameSummaries[tokenId].storeId == 0, "Token already exists");
209:    commonGameSummaries[tokenId] = GameSummary(storeId, gameId, name, onChainURI, externalURI, t
otalAchievements);
210:    emit GameSummaryMinted(msg.sender, tokenId, totalAchievements);
211:  }

213:function mintGameSummary(
214:    address player,
215:    uint256 gameId,
216:    uint256 achievementsLength,
217:    uint256 storeId,
218:    bool soulBound
219:  ) private {
220:    require(storeId > 0, "StoreId must be greater than 0");
221:    uint256 tokenId = concat(storeId, gameId);
222:    require(playerGameData[player][tokenId].tokenId == 0, "Token already exists");
223:    _mint(player, tokenId, 1, "");
224:    playerGameData[player][tokenId] = PlayerGameData(tokenId, achievementsLength, soulBound);
225:    emit PlayerGameSummaryMinted(player, tokenId, achievementsLength);
226:  }
```

## Description

**Yaodao :** The function `mintGameSummary()` is used to mint the token for the `gameSummary` which is created by the function `createCommonGameSummary()`.

However, there is no logic to check whether the `gameSummary` exists for the token to mint.

As a result, it is possible that the gameSummary corresponding to the mint's token does not exist. The token will be a meaningless token.

## Recommendation

**Yaodao :** Recommend adding the logic to check whether the `gameSummary` exists.

## Client Response

TBD

# G7H-8: `verifySignature(...)` if statement can be simplified

| Category | Severity | Client Response | Contributor |
| --- | --- | --- | --- |
| Gas Optimization | Informational | Reported | plasmablocks |

## Code Reference

- code/GameSummary.sol#L117-L124

```
117:function verifySignature(uint256 nonce, bytes memory signature) public view returns (bool) {
118:    address signer = recoverAddress(nonce, signature);
119:    if (whitelistSigners[signer]) {
120:      return true;
121:    } else {
122:      return false;
123:    }
124:  }
```

## Description

**plasmablocks :** The `verifySignature(...)` method derives an address from the `nonce`, `signature` and `msg.sender` and determines if that address is whitelisted by checking the `whitelistSigners` mapping.
Instead of using an `if` block the value of the mapping can be directly returned.

## Recommendation

**plasmablocks :** Update the `verifySignature(...)` method to the following:

```
function verifySignature(uint256 nonce, bytes memory signature) public view returns (bool) {
   address signer = recoverAddress(nonce, signature);
   return whitelistSigners[signer];
}
```

## Client Response

TBD

# Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.