shieldify

PenguinGotchi

SECURITY REVIEW

Date: 28 May 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Rust, Go, Vyper, Move and Cairo.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About PenguinGotchi

PenguinGotchi is a blockchain-based virtual pet adventure where you nurture penguins, embark on thrilling missions, and strategically manage resources to earn rewards! Your penguins depend on your daily care to stay alive and thrive. Feed them, embark on thrilling adventures, level them up, and strategically earn valuable $GOTCHI rewards. Balance immediate rewards with long-term compounding to maximize your success.

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 7 days with a total of 168 hours dedicated by the Shieldify team.

Overall, the code is well-written. The audit report identified two High-severity, one Medium-severity and one Low-severity issues, primarily related to predictable randomness, inability to purchase new items and incorrect validation.

The PenguinGotchi team has been highly responsive to the Shieldify research team's inquiries and promptly implemented all recommendations.

5.1 Protocol Summary

Project Name	**PenguinGotchi**
Repository	pengu-sc
Type of Project	GameFi
Audit Timeline	7 days
Review Commit Hash	cd9e0fd078f55dbd1067be0fd4b25002ac59a425
Fixes Review Commit Hash	dc13e67c7033771a47223c2d5599bac00e4b0df9

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/Treasury.sol	161
src/TestToken.sol	24
src/PenguinManager.sol	538
src/Game.sol	252
src/Gotchi.sol	91
src/utils/Constants.sol	4
src/utils/CoSignable.l.sol	14
src/utils/GameLogic.sol	41
Total	**1125**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **2**
- **Medium** issues: **1**
- **Low** issues: **1**

ID	Title	Severity	Status
[H-01]	Predictable Randomness in Activity Outcomes	High	Fixed
[H-02]	User Can Be Permanently Denied from Purchasing New Penguins Due to Incorrect Alive Penguin Count Validation	High	Fixed
[M-01]	Maximum Level Bypass in `fastUpgradePenguin()`	Medium	Fixed
[L-01]	Wrong conditional logic affecting player registration mechanism in `buyPenguin()`	Low	Fixed

7. Findings

[H-01] Predictable Randomness in Activity Outcomes

Severity

High Risk

Description

The randomness generation in `PenguinManager.doActivity()` is predictable due to reliance on `block.timestamp` and inputs controlled by the user. If the caller of `Game.doActivity()` is a contract, it can simulate and predict activity outcomes before execution, then only proceed when favorable results are expected.

```
uint256 dRandomNumber = uint256(keccak256(abi.encodePacked(params.
    penguinId, block.timestamp, params.salt)))
     % (100 * PERCENT_DENOMINATOR);
uint256 sRandomNumber = uint256(keccak256(abi.encodePacked(block.
    timestamp, params.salt, params.penguinId)))
     % (100 * PERCENT_DENOMINATOR);
uint256 dropRandomNumber = uint256(keccak256(abi.encodePacked(params.salt
    , params.penguinId, block.timestamp)))
     % (100 * PERCENT_DENOMINATOR);
```

The vulnerability is severe because: 1. The signature in `Game.doActivity()` has no expiration time 2. The `salt` and `penguinId` are user-controlled

```
//@audit only check signer, no timeout
modifier validSignature(bytes32 paramsHash, bytes calldata signature) {
    address signer = CoSignable.isValidSignature(paramsHash, signature);

    if (!isSigner(signer)) {
        revert SignerIsNotAuthorized(signer);
    }
    _;
}
```

For example:

1. Attacker creates a malicious contract to buy a penguin and implements the following attack flow:

```
function executeAttack(uint256 activityId, uint256[] calldata penguinIds,
    bytes calldata salt, bytes calldata signature) external {
// Simulate the random numbers using the same inputs
   uint256 dRandomNumber = calculateDRandomNumber(penguinIds[0], block.
      timestamp, salt);
   uint256 sRandomNumber = calculateSRandomNumber(block.timestamp, salt,
      penguinIds[0]);
   uint256 dropRandomNumber = calculateDropRandomNumber(salt, penguinIds
      [0], block.timestamp);

// Check if outcomes are favorable (e.g., no death, guaranteed success,
   desired item drop)
   Activity memory activity = game.getActivity(activityId);
   if (dRandomNumber >= activity.deathChance && sRandomNumber < activity.
      successChance) {
// Only execute when conditions are favorable
       game.doActivity(activityId, penguinIds, salt, signature);
   }
// else revert
}
```

2. The attacker obtains a valid signature for the contract from an authorized signer to execute `oActivity()` as a regular user.
3. The attacker repeatedly calls their contract with different transaction timestamps until they find a favourable outcome.
4. When a favourable simulation is found, the transaction executes, guaranteeing:
    - No penguin deaths
    - Successful activities
    - Preferred item drops (backpack or shield)
    - Maximum rewards
5. The attacker can repeat this process indefinitely, as signatures have no expiration.
6. Attackers can leverage MEV infrastructure like flashbot—if the outcome isn't what they want, they just `revert`. Since failed transactions don't hit the chain, these services charge no fees, drastically lowering attack costs.

Location of Affected Code

File: Game.sol

File: PenguinManager.sol

Impact

The attacker can wait for their desired outcome before executing `doActivity()`, which breaks the game's randomness and creates unfairness.

Recommendation

Use a more robust randomness solution like Chainlink VRF, or add an expiration time to signatures to make attacks harder.

Team Response

Fixed.

[H-02] User Can Be Permanently Denied from Purchasing New Penguins Due to Incorrect Alive Penguin Count Validation

Severity

High Risk

Description

The vulnerability exists in the penguin purchase mechanism within the `buyPenguin()` function and its associated validation logic. The contract maintains two separate data structures to track penguin ownership: `playerOwnedPenguins` mapping, which stores all penguins ever owned by a player (including dead ones), and `alivePenguins` mapping, which should theoretically track only living penguins. However, the purchase validation logic incorrectly uses `playerOwnedPenguins` instead of `alivePenguins` to determine if a user can acquire additional penguins.

The `checkCanGetExtraPenguin` modifier calls `playerOwnedPenguins[owner].length` to count the user's penguins and compares this against `settings.limits.maxAlivePenguins`. This count includes all penguins that have ever been owned by the user, regardless of their current state. When penguins die during activities in the `doActivity()` function, they are correctly removed from the alivePenguins mapping via `alivePenguins[params.player].remove(params.penguinId)`, but they remain in the `playerOwnedPenguins` array permanently. This creates a discrepancy where the validation logic considers dead penguins as part of the alive count, effectively reducing the user's ability to purchase new penguins below the intended limit.

In extreme scenarios, if the `maximum alive penguin limit` is set to a relatively low number (such as 10 as mentioned in test cases), users could find themselves completely unable to participate in the game after a series of unfortunate activity outcomes, despite the game's intended design.

Location of Affected Code

File: src/PenguinManager.sol#L168

```
modifier checkCanGetExtraPenguin(address owner) {
    uint256[] memory penguinIds = playerOwnedPenguins[owner];
    uint256 penguinCount = penguinIds.length;

    if (penguinCount >= settings.limits.maxAlivePenguins) {
        revert MaximumPenguinsAcquired(penguinCount, settings.limits.
            maxAlivePenguins);
    }
    _;
}
```

Impact

This vulnerability creates a permanent denial of service condition that fundamentally breaks the core functionality of the penguin management system. Users who experience multiple penguin deaths through activities will progressively lose their ability to purchase replacement penguins, eventually reaching a state where they cannot acquire any new penguins despite having zero living ones. This effectively locks users out of the primary game mechanic permanently.

Recommendation

The immediate fix requires modifying the `checkCanGetExtraPenguin()` modifier to use the correct data structure for counting alive penguins. Replace the current validation logic that uses `playerOwnedPenguins[owner].length` with `alivePenguins[owner].length()` since the `alivePenguins` mapping properly tracks only living penguins and is correctly maintained throughout the penguin lifecycle.

Team Response

Fixed.

[M-01] Maximum Level Bypass in `fastUpgradePenguin()`

Severity

Medium Risk

Description

The `fastUpgradePenguin()` function in PenguinManager.sol lacks the `underMaxLevel()` modifier that is present in the regular `upgradePenguin()` function. This allows penguins to be upgraded beyond the maximum level configured in the system.

```
function fastUpgradePenguin(uint256 penguinId)
    external
    whenNotPaused
    penguinExists(penguinId)
    onlyPenguinOwner(msg.sender, penguinId)
    penguinAlive(penguinId)
{
    _rewardPayment(_fastUpgradeCost(penguins[penguinId]));

    if (
        block.timestamp < penguins[penguinId].lastFastUpgradeTime +
           settings.intervals.fastUpgradeTimeout
            && penguins[penguinId].lastFastUpgradeTime != 0
    ) {
        revert PenguinCannotUpgrade(
            penguins[penguinId].lastFastUpgradeTime + settings.intervals.
               fastUpgradeTimeout, block.timestamp
        );
    }

    Penguin storage penguin = penguins[penguinId];
    penguin.lastFastUpgradeTime = block.timestamp;
    penguin.lastUpgradeTime = block.timestamp;
    penguin.payedActions = 0;
    penguin.level++;
}
```

The maximum level is set to 10 in the deployment script:

```
settings = Settings({
// ... other settings ...
    limits: LimitSettings({ maxAlivePenguins: 10, maxLevel: 10 }),
// ... other settings ...
});
```

The `_fastUpgradeCost()` function is designed with level 10 as the maximum level to be charged:

```
function _fastUpgradeCost(Penguin storage penguin) internal view returns
   (uint256 cost) {
    if (penguin.level < 4) {
        cost = settings.taxes.fastUpgradeTax;
    } else if (penguin.level >= 4 && penguin.level < 7) {
        cost = settings.taxes.fastUpgradeTax * 120 / 100;
    } else if (penguin.level >= 7 && penguin.level < 10) {
        cost = settings.taxes.fastUpgradeTax * 140 / 100;
    } else if (penguin.level == 10) {
        cost = settings.taxes.fastUpgradeTax * 2;
    }
}
```

But later in `fastUpgradePenguin()`, the `penguin.level` is still plus 1, which means the level becomes 11.

For example:

1. A user has a penguin at level 10 (the maximum intended level)
2. The user calls `fastUpgradePenguin()` on their penguin
3. The penguin's level is increased to 11
4. When the user performs an activity with this penguin in the `Game` contract, the reward calculation attempts to access `rewardByLevelPercentage[10]` (since level is 11)
5. This index might not be populated because the `setRewardByLevelPercentage()` function prevents setting rewards for level 10:

```
if (activity.appliesRewardMultiplier) {
    basePlusLevelBonus = (reward * rewardByLevelPercentage[penguin.level
       - 1]) / PERCENT_DENOMINATOR;
```

```
function setRewardByLevelPercentage(
    uint256 level,
    uint256 percentage
)
    public
    onlyAdmin
    validPercentage(percentage)
{
    uint256 maxLevel = penguinManager.maxLevel();
    if (level >= maxLevel) {
        revert InvalidLevelForReward(level, maxLevel);
    }
    rewardByLevelPercentage[level] = percentage;
}
```

6. As a result, the player with a level 11 penguin will receive no bonus rewards from activities.

Location of Affected Code

File: PenguinManager.sol

File: Game.sol

Impact

Players can upgrade to level 11 via `fastUpgradePenguin()`, but its functionality clashes with other protocol features, like being unable to claim the activity bonus.

Recommendation

Cap the max level at 10 in `fastUpgradePenguin()`, and update `_fastUpgradeCost()` to make the final tier level 9 instead of 10.

Team Response

Fixed.

[L-01] Wrong conditional logic affecting player registration mechanism in `buyPenguin()`

Severity

Low Risk

Description

The `buyPenguin()` function in the `PenguinManager` contract contains a logic error in its player registration mechanism that results in dead code execution. The vulnerability stems from an inverted conditional check at lines where the function attempts to initialize new players. The problematic code block reads `if (_players.contains(player))` followed by player initialization logic, which creates a logical impossibility since players can only be added to the `_players` EnumerableSet within this very conditional block.

The function logic attempts to initialize new players by adding them to the `_players` set and creating a new empty array for `playerOwnedPenguins[player]`, but the conditional check executes this initialization only when the player already exists in the set. Since no other function in the contract adds players to the _players set, and the only `_players.add(player)` call occurs within the conditional block that requires the player to already exist, the condition `_players.contains(player)` will perpetually evaluate to false. This creates unreachable code that never executes during runtime.

The intended behavior appears to be initializing new players upon their first penguin purchase, but the current implementation fails to achieve this goal.

Location of Affected Code

File: PenguinManager.sol

```
function buyPenguin() external whenNotPaused checkCanGetExtraPenguin(msg.
    sender) nonReentrant returns (uint256) {
        IERC20 paymentToken = treasury.paymentToken();
        address player = msg.sender;
// @audit this never gets executed
        if (_players.contains(player)) {
            _players.add(player);
            playerOwnedPenguins[player] = new uint256[](0);
        }

        uint256 price = nextPenguinPrice();
}
```

Impact

There is no direct security exploitation here, but the intended behavior of initializing new players upon their first penguin purchase fails here.

Recommendation

The immediate fix requires inverting the conditional logic from `if (_players.contains(player))` to `if (!_players.contains(player))` to properly detect and initialize new players

Team Response

Fixed.

shieldify

Thank you!