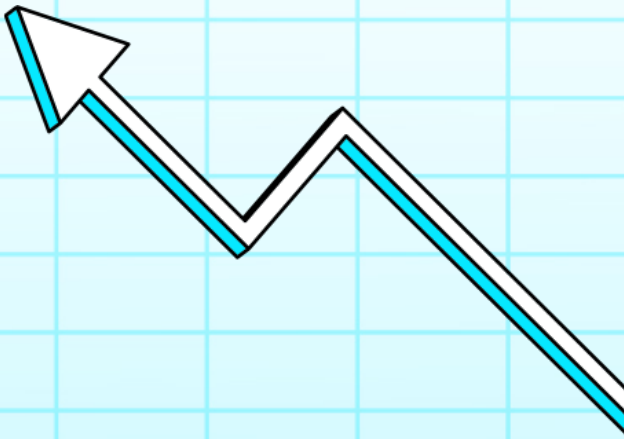
shieldify



Tollan Universe

SECURITY REVIEW

Date: 19 December 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, Vyper, Rust, Cairo, Move and Go.

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 Tollan Universe

Play Bullet Heaven Survivors-mode, survive countless waves of enemies, acquire elemental powers and defeat deadly Bosses while competing for a prize pool that grows as more players take part in the Season.

Learn more: here

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 1 day, with a total of 16 hours dedicated to the audit by two researchers from the Shieldify team.

Overall, the code is well-written. The audit report identified one Medium and four Low severity issues. They're mainly related to duplicate processing of the same logical request and different generation misconfigurations.

The Tollan Universe team provided exceptional support and promptly implemented the suggested recommendations from the Shieldify researchers.

5.1 Protocol Summary

Project Name	**Tollan Universe**
Repository	tollan-rng-smart-contract
Type of Project	GameFi - Elemental affinities using Proof of Play's vRNG
Security Review Timeline	1 day
Review Commit Hash	afacb9a2f31a0b4a38a7e72b915fed5937cf27eb
Fixes Review Commit Hash	e020c74e79dcc49a8e98823a62a1cfcc3310a2bf

5.2 Scope

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

File	**nSLOC**
contracts/TollanRandomConsumerMax.sol	130
Total	**130**

6. Findings Summary

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

- **Medium** issues: 1
- **Low** issues: 4

ID	Title	Severity	Status
[M-01]	`mockRequestRandom()` Can Bypass `Single-Processing` Invariant and Generate Multiple Results for the Same Logical Request (Event-Level Replay)	Medium	Fixed
[L-01]	Misleading `elementIds` Configuration Is Ignored During Result Generation	Low	Fixed
[L-02]	Multiple Requests Can Produce Identical Outcomes Within a Short Window	Low	Fixed
[L-03]	Invalid `setBuildConfiguration()` Input Silently Skews Wave Probabilities Instead of Reverting	Low	Fixed
[L-04]	Permissionless `requestRandom()` Enables Griefing via VRF Requests	Low	Fixed

[M-01] `mockRequestRandom()` Can Bypass `Single-Processing` Invariant and Generate Multiple Results for the Same Logical Request (Event-Level Replay)

Severity

Medium Risk

Description

The `mockRequestRandom()` bypasses the "single-processing" invariant enforced on the real VRF path. It never checks or sets `processedRequests`, and it directly calls `_handleRandom()` after emitting events.

This means the owner can generate an arbitrary number of result events for the same `buildId` (and even for the same `fakeRequestId` pattern if they can control inputs/timing), without any on-chain "already processed" protection.

Location of Affected Code

File: contracts/TollanRandomConsumerMax.sol#L42-L50

```
function mockRequestRandom(uint256 buildId) external onlyOwner returns (
    uint256, uint256) {
     require(configs[buildId].waveCount > 0, "No config");

     uint256 fakeRequestId = uint256(keccak256(abi.encodePacked(block.
        timestamp, msg.sender, buildId)));
     uint256 fakeRandom = uint256(keccak256(abi.encodePacked(block.
        prevrandao, block.timestamp, buildId)));

     requestToBuildId[fakeRequestId] = buildId;
     emit RandomNumberRequested(fakeRequestId, buildId);
     emit RandomNumberReceived(fakeRequestId, fakeRandom);

     _handleRandom(fakeRequestId, fakeRandom); // <-- no processedRequests
         gating
     return (fakeRequestId, fakeRandom);
}
```

Impact

If any downstream component (indexer/backend/game logic) treats `ResultGeneratedIDs` as authoritative without separating mock vs real, the owner (or a compromised owner key) can emit multiple conflicting "final" results and effectively rewrite outcomes off-chain.

Recommendation

Enforce the same single-processing guard used in `randomNumberCallback()`:

- Add `require(!processedRequests[fakeRequestId], "Request already processed")`
- Set `processedRequests[fakeRequestId] = true` before/after generation (prefer after successful `_handleRandom()` as in the first issue's fix)
- Optionally emit a distinct event for mock mode (or remove mock from production builds).

Team Response

Fixed.

[L-01] Misleading `elementIds` Configuration Is Ignored During Result Generation

Severity

Low Risk

Description

The `elementIds` are accepted as part of `BuildConfiguration` but are never stored or used. Only the length of the array is persisted (`elementCount`), and all result generation implicitly assumes element IDs are `0..elementCount-1` .

This makes the configuration API misleading and can cause off-chain systems or admins to believe specific IDs are being used when they are not.

Location of Affected Code

File: contracts/TollanRandomConsumerMax.sol

```
// setBuildConfiguration
require(config.elementIds.length > 0 && config.elementIds.length <= 4, "
    Elements 1-4");
configs[buildId] = Config(uint8(config.waves.length), uint8(config.
    elementIds.length));

// _handleRandom
uint256 elemCount = cfg.elementCount;
picked[j] = uint8((indices >> (r * 8)) & 0xFF);

// getBuildConfiguration
for (uint8 i = 0; i < cfg.elementCount; i++) elementIds[i] = i;
```

Recommendation

Either store and use the actual `elementIds` array during result generation, or remove `elementIds` from the configuration interface entirely to avoid false assumptions.

Team Response

Fixed.

[L-02] Multiple Requests Can Produce Identical Outcomes Within a Short Window

Severity

Low Risk

Description

The contract derives all wave results from the VRF-provided `randomNumber`:

```
uint256 state = randomNumber;    //@audit [L] - risk of repeatable outcome
```

The PoP VRF returns the same `randomNumber` for multiple requests with the same `buildId` within a short 3-second window. Those distinct requests will deterministically generate identical outcomes, meaning that not every user will get unique randomness.

Location of Affected Code

File: contracts/TollanRandomConsumerMax.sol

Recommendation

Consider PoP documentation for deriving a per-request seed to always guarantee unique randomness:

```
-     uint256 state = randomNumber;
+     uint256 state = uint256(keccak256(abi.encodePacked(requestId,
    randomNumber)));
```

Team Response

Fixed.

[L-03] Invalid `setBuildConfiguration()` Input Silently Skews Wave Probabilities Instead of Reverting

Severity

Low Risk

Description

The `setBuildConfiguration()` allows a wave configuration where `config.waves[i].length` exceeds `elementIds.length`, as long as total probabilities sum to `10_000`. While validation passes, `_handleRandom()` later caps the rolled `count` to `elemCount`, silently increasing the probability of the highest valid count.

Root cause

The `setBuildConfiguration()` validates `config.waves[i][j].count` is within [1, 4], but does not enforce
`config.waves[i].length == elementIds.length`.

In `_handleRandom()`, rolls resolving to an invalid higher count are capped:

```
if (count > elemCount) count = elemCount;
```

Location of Affected Code

File: contracts/TollanRandomConsumerMax.sol

Impact

A misconfigured build does not revert and instead affects randomness, increasing the probability of rolling the highest possible count and altering wave outcomes.

Example

For `elemCount = 3` and wave config:

```
[
  { "count": 1, "chance": 2500 },
  { "count": 2, "chance": 2500 },
  { "count": 3, "chance": 2500 },
  { "count": 4, "chance": 2500 }
]
```

The effective probability becomes:

```
[
  { "count": 1, "chance": 2500 },
  { "count": 2, "chance": 2500 },
  { "count": 3, "chance": 5000 }
]
```

Proof of Concept

Add the following test inside `TollanRandomConsumerMax.test.ts`:

```
it("invalid setBuildConfiguration input does not fail but incorrectly
   increases probability for count 3", async () => {
  // ============================== START HELPER FUNCTIONS
     ==============================
  const MASK_256 = (1n << 256n) - 1n;

  const splitMixZWave0 = (randomNumber: bigint) => {
    let state = randomNumber & MASK_256;
    state = (state + 0x9e3779b97f4a7c15n) & MASK_256;
    let z = state;
    z = ((z ^ (z >> 30n)) * 0xbf58476d1ce4e5b9n) & MASK_256;
    z = ((z ^ (z >> 27n)) * 0x94d049bb133111ebn) & MASK_256;
    z = (z ^ (z >> 31n)) & MASK_256;
    return z;
  };

  const rollForSeedWave0 = (seed: bigint) => Number(splitMixZWave0(seed)
     % 10000n);

  const findSeedInRollRange = (minIncl: number, maxIncl: number) => {
    for (let seed = 1n; seed < 200000n; seed++) {
      const roll = rollForSeedWave0(seed);
      if (roll >= minIncl && roll <= maxIncl) return seed;
    }
    throw new Error(`No seed found for roll range [${minIncl}, ${maxIncl
       }] in search bound`);
  };

  // ============================== END HELPER FUNCTIONS
     ==============================

  // 1. Configure wave distribution for 4 elements
  const { consumer, mockVrf } = await loadFixture(deployFixture);
  const waveWithAllCounts = [
    [
      { count: 1, chance: 2500 },
      { count: 2, chance: 2500 },
      { count: 3, chance: 2500 },
      { count: 4, chance: 2500 },
    ],
  ];

  //2. Configure 2 builds - 1 with only 3 elementIds, 2 with 4 elementIds
  const buildWith3Elements = 1;
  const buildWith4Elements = 2;
  await consumer.setBuildConfiguration(buildWith3Elements, {
    elementIds: [100, 101, 102],
    waves: waveWithAllCounts,
  });
```

```
  await consumer.setBuildConfiguration(buildWith4Elements, {
    elementIds: [100, 101, 102, 103],
    waves: waveWithAllCounts,
  });

  // 3. Find a random number such that roll falls into last 25% bucket:
     count=4 when roll in [7500..9999]
  const seedThatSelectsCount4 = findSeedInRollRange(7500, 9999);
  const roll = rollForSeedWave0(seedThatSelectsCount4);
  // Sanity: prove it's actually in the count=4 bucket
  expect(roll).to.be.greaterThanOrEqual(7500);
  expect(roll).to.be.lessThanOrEqual(9999);


  // 4. Request randomness for both builds
  await consumer.requestRandom(buildWith3Elements); // requestId 1
  await consumer.requestRandom(buildWith4Elements); // requestId 2


  // 5. Fulfill random numbers and extract waves
  const extractGeneratedWavesFromReceipt = async (tx: any) => {
    const receipt = await tx.wait();
    const log = receipt?.logs.find((l: any) => {
      try {
        return consumer.interface.parseLog(l)?.name === "
           ResultGeneratedIDs";
      } catch {
        return false;
      }
    });
    const parsed = consumer.interface.parseLog(log as any);
    return parsed.args.all as any; // uint8[][]
  };
  const generatedWavesElem3 = await extractGeneratedWavesFromReceipt(
    await mockVrf.fulfillRandomNumber(1, seedThatSelectsCount4)
  );
  const generatedWavesElem4 = await extractGeneratedWavesFromReceipt(
    await mockVrf.fulfillRandomNumber(2, seedThatSelectsCount4)
  );

  //6. Validate that a roll falling in [t3;t4) returns 3 due to cap check
  expect(generatedWavesElem3[0].length).to.equal(3); // capped from 4 ->
     3
  expect(generatedWavesElem4[0].length).to.equal(4); // no cap, true
     count=4
});
```

Recommendation

Validate at configuration time that no wave defines probabilities for selecting more elements than exist in the build (i.e., enforce `count <= elementIds.length`). This prevents silent probability fold-

ing at runtime and ensures the configured distribution matches the actual element set.

Add the following check inside the per-wave loop in `setBuildConfiguration()`:

```
function setBuildConfiguration(uint256 buildId, BuildConfiguration
    calldata config) external onlyOwner {
     require(buildId != 0, "Zero buildId");
     require(config.elementIds.length > 0 && config.elementIds.length <=
        4, "Elements 1-4");
     require(config.waves.length > 0 && config.waves.length <= 64, "Waves
        1-64");

     configs[buildId] = Config(uint8(config.waves.length), uint8(config.
        elementIds.length));

     uint256 slotsNeeded = (config.waves.length + 4) / 5;
     uint256[] memory packed = new uint256[](slotsNeeded);

     for (uint256 i = 0; i < config.waves.length; i++) {
         WaveConfiguration[] calldata opts = config.waves[i];
+        require(opts.length <= config.elementIds.length, "Opts count
    higher than elementIds size");
         uint16[5] memory chances;
         uint256 total = 0;

         for (uint256 j = 0; j < opts.length; j++) {
             uint8 cnt = opts[j].count;
             require(cnt >= 1 && cnt <= 4, "Count 1-4");
             require(chances[cnt] == 0, "Duplicate");
             chances[cnt] = opts[j].chance;
             total += opts[j].chance; }
         require(total == 10000, "Sum 10000");

         uint256 t1 = chances[1];
         uint256 t2 = t1 + chances[2];
         uint256 t3 = t2 + chances[3];

         packed[i / 5] |= ((t1 | (t2 << 16) | (t3 << 32)) << ((i % 5) *
            48)); }

     delete packedWaves[buildId];
     for (uint256 i = 0; i < packed.length; i++) {
         packedWaves[buildId].push(packed[i]);
     }
}
```

Team Response

Fixed.

[L-04] Permissionless `requestRandom()` enables griefing via VRF requests

Severity

Low Risk

Description

The `requestRandom(buildId)` is callable by anyone. A malicious actor can spam VRF requests to exhaust the VRF credit limit, hit rate limits, or flooding contract with requests.

Root cause

The `requestRandom()` has no access control, rate limit or any anti-spam mechanism, yet it triggers an external VRF request and persists request tracking state.

Location of Affected Code

File: contracts/TollanRandomConsumerMax.sol#L35-L40

```
function requestRandom(uint256 buildId) external returns (uint256) {
    // @audit permissionless, vrf spam / creditline risk
     require(configs[buildId].waveCount > 0, "No config");
     uint256 requestId = vrfSystem.requestRandomNumberWithTraceId(0);
     requestToBuildId[requestId] = buildId;
     emit RandomNumberRequested(requestId, buildId);
     return requestId;
}
```

Impact

- Exhaust VRF credit line resulting in protocol loss of funds
- Flooding requests that might result in operational overload and excessive emission of events

Recommendation

- Consider restricting who can call `requestRandom()` via whitelisting the gamer contract to avoid spam by an external party.
- Consider implementing a rate-limiting mechanism per user.

Team Response

Fixed.

shieldify

Thank you!