CD SECURITY

AUDIT REPORT

Crash Game
March 2025

Prepared by
tushar1698
0xluk3
zeroXchad

Introduction

A time-boxed security review of the **Crash Game** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Crash Game**

The Crash Game protocol is a blockchain-based gambling platform featuring a "crash" style game mechanic where players bet on when to cash out before a multiplier "crashes" to zero.

The protocol is designed to maintain fairness through its randomness generation, handle commission distribution through its affiliate system, and manage user funds through dedicated balance managers.

Severity classification

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

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 130acf2efc131eb34854f5e2faa3e2f1b257abee**

Scope

The following smart contracts were in scope of the audit:

- `src/contracts/*`

The following number of issues were found, categorized by their severity:

- Critical & High: 1 issues
- Medium: 2 issues
- Low: 17 issues

Findings Summary

ID	Title	Severity	Status
[H-01]	Global connection limit vulnerable to DoS attacks	High	Fixed
[M-01]	Overly restrictive API rate limiting	Medium	Fixed
[M-02]	Potential for cross-chain token replay attacks	Medium	Fixed
[L-01]	No slippage validation in collect function	Low	Fixed
[L-02]	Silent adjustment of transfer amounts masks accounting errors or can cause loss of funds	Low	Fixed
[L-03]	Use of unsafe `transferFrom` instead of `safeTransferFrom`	Low	Fixed
[L-04]	`safeIncreaseAllowance` might fail for certain tokens like USDT	Low	Fixed
[L-05]	Dependencies not pinned to exact versions	Low	Acknowledged
[L-06]	Unchecked Deadline Parameter in Swap Library	Low	Fixed
[L-07]	Insufficient Validation of Zero Random Number in Crash Game	Low	Fixed
[L-08]	Improper CORS Configuration	Low	Fixed
[L-09]	The `totalDeposits` not updated upon the `withdrawDeposit` call	Low	Fixed
[L-10]	Same TWAP usage in multiple swaps within the `collect` function	Low	Fixed
[L-11]	High default slippage within the `collectCommissions` function	Low	Fixed
[L-12]	The griefing attack possible for the `endRound` function	Low	Acknowledged
[L-13]	Lack of Content-Security-Policy	Low	Fixed
[L-14]	Logout does not terminate refresh JWT token	Low	Fixed
[L-15]	Lack of Strict-Transport-Security	Low	Fixed
[L-16]	Lack of anti-DDoS protection	Low	Fixed

ID	Title	Severity	Status
[L-17]	UniswapV3Pool's slot0 is prone to manipulation	Low	Acknowledged

Detailed Findings

[H-01] Global connection limit vulnerable to DoS attacks

Severity

Impact: High

Likelihood: Medium

Description

The application implements a global connection limit in the `connections.ts` middleware:

```
private static readonly MAX_ACTIVE_CONNECTIONS = 1000
```

While limiting connections is a good practice, the current implementation tracks connections globally rather than per IP address. This creates a vulnerability where an attacker could easily perform a Denial of Service (DoS) attack by creating multiple connections until the maximum limit is reached, preventing legitimate users from connecting. The lack of per-IP rate limiting at this layer makes it trivial for a malicious actor to consume all available connection slots.

Recommendations

Implement per-IP connection limits instead. Consider relying on Cloudflare or similar WAF for rate limiting purposes.

[M-01] Overly restrictive API rate limiting

Severity

Impact: Medium

Likelihood: Medium

Description

The application's rate limiting in `limit.ts` is configured to allow only 100 requests per IP address over a 15-minute window:

```
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Limit each IP to 100 requests per windowMs
});
```

This limit is likely too restrictive for normal application usage, especially for active users or scenarios where multiple users might share the same IP (such as corporate networks or NAT). This can lead to legitimate users being unable to access the service, resulting in a self-imposed denial of service.

Recommendations

Increase the rate limit to a more reasonable value (e.g., 500-1000 requests per 10 minutes). Consider implementing more granular rate limits for specific endpoints rather than a global limit.

[M-02] Potential for cross-chain token replay attacks

Severity

Impact: Medium

Likelihood: Low

Description

In the `auth.ts` middleware, JWT tokens are validated for a specific chain ID and service name:

```
if (
  decoded.chainId !== chainId ||
  decoded.serviceName !== expectedServiceName
) {
  logger.warn(
    `Token validation failed for user ${decoded.sub}: Mismatched chainId
or serviceName`
  );
  throw new Error("Authentication error: Mismatched chainId or
serviceName");
}
```

While this validation is good, there are potential replay vulnerabilities if:

- The same JWT secret is shared across different chains or service deployments
- The application architecture changes to support multiple chains simultaneously

In a multi-chain environment, a token generated for one chain could potentially be replayed on another if proper isolation isn't maintained throughout the system.

Recommendations

Ensure different JWT secrets are used for each chain and service. Include additional token validation specific to the target chain (e.g., chain-specific nonce). Implement token blacklisting after chain switching.

[L-01] No slippage validation in collect function

Description

In `Affiliate.sol` the `collect` function accepts a user-provided slippage parameter without any validation. This allows users to potentially specify extremely high slippage values (e.g., 100%), which could result in receiving significantly fewer tokens than expected during unfavorable market conditions. Malicious or inexperienced users could inadvertently set values that lead to substantial losses.

Recommendations

Implement a maximum slippage parameter (e.g., 10% or 20%). Add validation to ensure the provided slippage is within reasonable bounds. Consider implementing a default slippage value that users can override within limits.

[L-02] Silent adjustment of transfer amounts masks accounting errors or can cause loss of funds

Description

In `Affiliate.sol` (lines 569-579), the `_safeTransferNative` and `_safeTransferToken` functions silently adjust transfer amounts if the requested amount exceeds the balance:

```
if (amount > 0) {
  if (amount > balanace) {
    Address.sendValue(payable(to), balanace);
  } else {
    Address.sendValue(payable(to), amount);
  }
}
```

Rather than reverting, these functions will transfer whatever balance is available. This masks potential accounting errors and could create inconsistencies between expected and actual transferred amounts, potentially leading to funds being incorrectly accounted for.

On the other hand this is unlikely to happen, since the functions are used in a context of returning user funds, in a single transaction. However, in such case, a revert might be more safe for users than a withdraw cap.

Recommendations

Revert the transaction if the requested amount exceeds the available balance, or, at minimum, emit an event when the transferred amount differs from the requested amount.

[L-03] Use of unsafe `transferFrom` instead of `safeTransferFrom`

Description

In `BalanceManager.sol`, the `topUpUserBalance` function uses the unsafe `transferFrom` method instead of `safeTransferFrom`:

```
IERC20(asset).transferFrom(msg.sender, address(this), amount);
```

This can be problematic for tokens that don't revert on failure but return a boolean (which is then ignored), or for non-compliant tokens like USDT that might have different behavior compared to the ERC20 standard.

Recommendations

Replace `transferFrom` with `safeTransferFrom` to ensure compatibility with all ERC20 tokens.

[L-04] `safeIncreaseAllowance` might fail for certain tokens like USDT

Description

The Crash game aims to deposit and reward users with ERC-20 token, such as USDC. However, it was identified that presently solution is not prepared to support non-standard ERC-20 tokens, e.g. USDT. The USDT token does not return boolean value upon execution transfer or approval. Thus, making use of OpenZepplin's IERC20 interface will lead to direct transaction revert whenever such token will be used.

In terms of the Crash game two instances of discrepancies were identified: within the BalanceManager's constructor an usage of IERC20 approve and within the topUpUserBalance function an usage of the transferFrom. As a result, the protocol could not be deployed and used for e.g. USDT token.

```
constructor(
address asset*,
Affiliate affiliate*,
address accessManager,
uint256 minReserve*,
uint256 maxReserve*
) AccessManaged(accessManager) {
asset = asset*;
affiliate = affiliate*;

    // Approve affiliate system for infinite token transfers
    if (asset_ != address(0)) {
      IERC20(asset_).approve(address(affiliate), type(uint256).max);
//@audit revert
```

```
    }

...
function topUpUserBalance(
address user,
uint256 amount,
string calldata referrerId
) external payable whenNotPaused nonReentrant {
...
if (asset == address(0)) {
if (msg.value != amount) {
revert InvalidNativeAmount();
}
} else {
IERC20(asset).transferFrom(msg.sender, address(this), amount);
}
```

Recommendations

Consider adding full support for non-standard ERC20 tokens, so `approve` is replaced with `forceApprove` and the `transferFrom` is replaced with `safeTransferFrom`.

[L-05] Dependencies not pinned to exact versions

Description

Several dependencies in the `package.json` file are not pinned to exact versions, example:

```
"eslint-plugin-react-refresh": "^0.4.9",
"husky": "^9.1.4",
"typescript": "^5.5.4"
```

Using caret (^) or tilde (~) version ranges allows for automatic updates to minor or patch versions. This can lead to unexpected behavior or security issues if dependencies introduce breaking changes or vulnerabilities in newer versions.

Recommendations

Pin all dependencies to exact versions (remove ^ and ~ prefixes).

[L-06] Unchecked Deadline Parameter in Swap Library

Description

In the Swap.sol library (used by the Affiliate contract), there's an issue with the deadline parameter handling:

```
function sell(SwapParams memory params, bool useNative, bool toNative)
external returns (SwapResult memory result) {
    // ...
    IUniversalRouter(Constants.UNIVERSAL_ROUTER).execute{value: useNative
? params.amount : 0}(
      commands,
      inputs,
      params.deadline
    );
    // ...
}
```

Impact

The deadline parameter is passed directly to the Universal Router without validation. If a very large deadline is provided or if the deadline has already passed, the transaction would still proceed. This makes transactions vulnerable to sandwich attacks or unfavorable market conditions.

Recommendations

Add a high limit on the maximum value of the deadline or make the transaction deadline within a specific time range for users to select instead of passing an arbitrary value

[L-07] Insufficient Validation of Zero Random Number in Crash Game

Description

The `Crash.sol` contract doesn't adequately validate the case where the random number used for determining game outcomes is zero. While the contract checks that the random number doesn't exceed `Constants.BASIS`, it doesn't explicitly prevent a zero value, which could potentially trigger unexpected behavior in the crash point calculation.

In the `endRound` function of `Crash.sol`, there's an initial check for the upper bound of the random number and later validates that the random number matches the expected value derived from the secret and block hash:

```
function endRound(uint256 id, bytes32 secret, uint256 random, Exit[]
calldata exits) external restricted {
    // Ensure the provided number is within processable limits
    if (random > Constants.BASIS) revert InvalidRandomNumber();

    // ... [other code]
}

// Validate that the random number used is valid and based on the secret
and VRF number
```

```
if (uint256(keccak256(abi.encode(secret, round.baseHash))) %
Constants.BASIS != random)
    revert InvalidRandomNumber();
```

The modulo operation with `Constants.BASIS` naturally restricts the output to `[0, Constants.BASIS-1]`, meaning a value of 0 is possible and would pass validation. According to the game documentation in Crash-Game `Overview.md`, there's special handling for certain random values:

```
3. Crash Point Calculation:
   - Edge Case: If the random value is lower than the predefined house
edge, the crash point is set to a fixed value (e.g., 10000).
```

Impact

The impact is considered low because the probability of getting a random value of exactly 0 is very small (1 in 10,000 if using Constants.BASIS of 10000)

Recommendation

```
function endRound(uint256 id, bytes32 secret, uint256 random, Exit[]
calldata exits) external restricted {
    // Ensure the provided number is within processable limits (non-zero
and not greater than BASIS)
    if (random == 0 || random > Constants.BASIS) revert
InvalidRandomNumber();

    // ... [rest of the function]
}
```

[L-08] Improper CORS Configuration

Description

The current Cross-Origin Resource Sharing (CORS) configuration in the client-side code of the Web3 game permits requests with no Origin header (!origin) while enabling credentials: true without enforcing strict origin validation. This misconfiguration exposes the application to potential Cross-Site Request Forgery (CSRF) attacks, allowing malicious actors to execute unauthorized actions on behalf of authenticated users.

```
const corsOptions: cors.CorsOptions = {
  origin: (origin, callback) => {
    // Allow null origins and allowed origins
    if (!origin || allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
```

```
        callback(new Error("Not allowed by CORS"));
      }
    },
    credentials: true,
    methods: ["GET", "POST"],
    allowedHeaders: ["Content-Type", "Authorization"],
  };
```

Impact

Allowing requests without an `Origin` header could make the application susceptible to Cross-Site Request Forgery (CSRF) attacks. For example, an attacker could craft a malicious form on a different site that submits a request to the game's server, and if the browser automatically includes credentials (like cookies), the server might process it because no `Origin` header is required. While same-origin requests naturally lack an `Origin` header and are often legitimate, this permissiveness could be exploited if not paired with additional CSRF protections (e.g., CSRF tokens).

Recommendation

Revise the CORS configuration to enforce stricter origin checks and adopt additional security controls. Below is an improved configuration example:

```
const corsOptions: cors.CorsOptions = {
  origin: (origin, callback) => {
    if (allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
      callback(new Error("Not allowed by CORS"));
    }
  },
  credentials: true,
  methods: ["GET", "POST"],
  allowedHeaders: ["Content-Type", "Authorization"],
};
```

[L-09] The `totalDeposits` not updated upon the `withdrawDeposit` call

Description

The `totalDeposits` state variable is updated whenever the `topUpUserBalance` is called. However, it is not deducted within the `withdrawDeposit` function. Thus, after first `withdrawDeposit` call it can represent inaccurate value and the state of the protocol.

```
unction topUpUserBalance(
    address user,
    uint256 amount,
    string calldata referrerId
  ) external payable whenNotPaused nonReentrant {
...
    users[user].deposits.balance += amount;
    totalDeposits += amount;

    emit UserDeposit(user, amount);
  }
```

```
  function withdrawDeposit(uint256 amount) external nonReentrant {
    User storage user = users[msg.sender];
    uint256 available = user.deposits.balance - user.deposits.reserved;

    if (amount > available) revert InsufficientUserBalance();

    user.deposits.balance -= amount;
...
```

Recommendations

It is recommended to update the `totalDeposits` state variable within the `withdrawDeposit` function.

[L-10] Same TWAP usage in multiple swaps within the `collect` function

Description

The `estimateMaximumInputAmount` and `estimateMinimumOutputAmount` functions make use of the `getQuote` which uses `twap` as input parameter to determine the minimum output amount of the each swap action from the each hop of the `swapPath` defined in the config. This can affect the `collect` function directly, as it uses single `twap` period for every pool swap. As a result, this weakness can lead to the final quote amount deviation and providing the user with underestimated minimum output amount.

```
  function collect(address asset, uint256 twap, uint256 slippage, uint256
deadline) external {
    address referrer = msg.sender;
    uint256 collected = _referral.collect(referrer, asset);
    if (collected == 0) return;

    Config storage config = configuration[asset];
    Swap.SwapResult memory result;
```

```
    if (config.payoutMode == PayoutMode.Asset) {
      // Direct payout
      asset == address(0) ? _safeTransferNative(referrer, collected) :
_safeTransferToken(asset, referrer, collected);
    } else if (config.payoutMode == PayoutMode.Swap) {
      // Swap before payout (sell collected asset for payout token)
      result = Swap.sell(
        Swap.SwapParams({
          path: config.swapPath,
          recipient: referrer,
          amount: collected,
          slippage: slippage,
          twap: twap,
          deadline: deadline
        }),
        // Use native asset as input for the swap
        asset == address(0),
        // Automatically unwrap if the payout token is address(0)
        config.payoutToken == address(0)
      );
    } else {
      // We should never get here as we revert when no vault exists
      revert InvalidConfig();
    }

    emit PayedOut(
      referrer,
      asset,
      config.payoutMode == PayoutMode.Asset ? asset : config.payoutToken,
      collected,
      config.payoutMode == PayoutMode.Asset ? collected : result.received
    );
  }
```

```
  function getQuote(
    address inToken,
    address outToken,
    uint24 fee,
    uint256 twap,
    uint256 amount
  ) public view returns (uint256 quote, uint32 secondsAgo) {
    address pool = PoolAddress.computeAddress(Constants.FACTORY,
PoolAddress.getPoolKey(inToken, outToken, fee));

    (uint160 sqrtPriceX96, , , , , , ) = IUniswapV3Pool(pool).slot0();
    secondsAgo = uint32(twap * 60);
    uint32 oldest = OracleLibrary.getOldestObservationSecondsAgo(pool);

    if (oldest < secondsAgo) secondsAgo = oldest;

    if (secondsAgo > 0) {
```

```
        (int24 tick, ) = OracleLibrary.consult(pool, secondsAgo);
        sqrtPriceX96 = TickMath.getSqrtRatioAtTick(tick);
      }
      quote = OracleLibrary.getQuoteForSqrtRatioX96(sqrtPriceX96, amount,
inToken, outToken);
    }
```

Recommendations

Consider passing an array of `twap` periods, so each period correspond to each hop in swap path within the `collect` function.

[L-11] High default slippage within the `collectCommissions` function

Description

The `collectCommissions` function has defined hardcoded default slippage equal to the 5%, which can be considered too permissive as it allows to accept swaps with relatively high price fluctuations. On the contrary, the default slippage in Uniswap for volatile market is set to around 0.1%-0.5%.

```
export const collectCommissions = async (
  config: Config,
  pendingPayouts: PendingPayout[],
  affiliate: Address,
  chainId: number
) => {
  const calldata: Hex[] = []
  for (const pendingPayout of pendingPayouts) {
    if (pendingPayout.raw > 0n) {
      const data = encodeFunctionData({
        abi,
        functionName: 'collect',
        args: [
          pendingPayout.asset,
          5n,
          500n,
          await getDeadline(config, chainId)
        ]
      })

      calldata.push(data)
    }
  }
```

Recommendations

Consider setting more strict slippage in the `collectCommissions` function.

[L-12] The griefing attack possible for the endRound function

Description

The `endRound` function implements an assertion that it can be successfully called by the restricted user only before the round's timeout. While, the function cannot be manipulated directly, it can be prone to network attacks. If only either the malicious user can overload network with multiple transactions or the network is simply down, it is possible to delay the transaction execution, eventually leading to griefing attack. As a result, such user who could already know that it didn't win the round, could simply call the `recoverRound` to reclaim reserved deposit.

```
  function endRound(uint256 id, bytes32 secret, uint256 random, Exit[]
calldata exits) external restricted {
...
    // Should only allow to end a round before timeout
    if (block.timestamp >= round.timeout) revert RoundTimedOut();
...
  }
```

Recommendations

Consider reviewing business rules and the implementation having in mind possible griefing attack or simply incidents related to the unavailable network. It might be worth to resign from having such assertion within the `endRound` function.

[L-13] Lack of Content-Security-Policy

Description

The frontend lacks Content-Security-Policy (CSP) HTTP security header configured. The CSP is modern security control which defines resources, in particular JavaScript resources, a document is allowed to load. This is mainly used as a defense against cross-site scripting (XSS) attacks. Additionally, CSP has other features including defending against clickjacking and helping to ensure that a site's pages will be loaded over HTTPS.

Recommendations

It is recommended to implement CSP HTTP header and configure policy accordingly to the defined security business rules.

[L-14] Logout does not terminate refresh JWT token

Description

The `logout` functionality attempts to clear the `refreshToken` cookie from the user's web browser. However, it does not terminate the actual JWT token held by the cookie. Whenever such JWT token leaks or get disclosed the user might get only the false impression of successful logout and session termination. However, the leaked JWT token can be still used for maximal period of 7 days.

```
router.get("/logout", async (_req: Request, res: Response) => {
  try {
    res.clearCookie("refreshToken", {
      domain: API_DOMAIN,
      path: `${await getServiceName()}/auth/`,
    });
    logger.info("User logged out successfully");
    res.status(200).json({ message: "Logged out successfully" });
  } catch (error) {
    logger.error("Error during logout", { error });
    res.status(500).json({ error: "An error occurred during logout" });
  }
});
```

Recommendations

It is recommended to implement server-side JWT blacklisting collection that holds records of terminated session tokens.

[L-15] Lack of Strict-Transport-Security

Description

The frontend lacks Strict-Transport-Security (HSTS) HTTP security header configured. The HSTS informs browsers that the site should only be accessed using HTTPS, and that any future attempts to access it using HTTP should automatically be upgraded to HTTPS. Note that it is more secure than configuring a HTTP to HTTPS 301 redirect on your server, as the initial HTTP connection is still vulnerable to a man-in-the-middle attack.

Recommendations

It is recommended to implement HSTS HTTP header.

[L-16] Lack of anti-DDoS protection

Description

The frontend application lacks anti-DDoS protection, such as Cloudflare. The solution is deployed in AWS environment. Such default configuration may lead to dual risk: the application may encounter DoS attack

that can limit access to the website by legitimate users, and it may increase the cost of subscription maintenance, as higher traffic can consume more cloud resources.

Recommendations

It is recommended to consider implementing anti-DDoS protection.

[L-17] UniswapV3Pool's slot0 is prone to manipulation

Description

The `getQuote` function in `Swap.sol:323` uses UniswapV3Pool's `slot0` for price data without proper protections when the TWAP parameter is set to 0:

```
(uint160 sqrtPriceX96, , , , , , ) = IUniswapV3Pool(pool).slot0();
```

However the price at the `slo0` can be manipulated.

Recommendations

Always use a non-zero TWAP value by default, even if a user doesn't specify one. Implement additional safeguards like price deviation checks. Consider using Chainlink oracles or other more manipulation-resistant price sources.