Contract Audit

SQUID GAME

This audit report was undertaken by @adamdossa for the purpose of providing feedback to the UniTrade team. It has been written without any express or implied warranty.

This audit was done on the code committed to Github as:
https://github.com/SquidMoon/squid-game/commit/01a100c51606d647827fa479ca9fe83e91567f8c
which matched the branch `audit-nov-23` at the time of writing.

Disclosure

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. There is no owed duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

Classification

Comment - A note that highlights certain decisions taken and their impact on the contract.

Minor - A defect that does not have a material impact on the contract execution and is likely to be subjective.

Moderate - A defect that could impact the desired outcome of the contract execution in a specific scenario.

Major - A defect that impacts the desired outcome of the contract execution or introduces a weakness that may be exploited.

Critical - A defect that presents a significant security vulnerability or failure of the contract across a range of scenarios.

Audit

This audit covers the contract `SquidGame.sol` as well as a review of the underlying `Oracle.sol` and `PairOracle.sol` contracts.
Whilst the review covers the interaction between these contracts and UniSwapV2 / Chainlink VRF, it does not include a review of these underlying contracts which are well-used and established already.

Contract Behaviour

`SquidGame.sol` implements a game where a user can place a bet (ODD or EVEN). The bet stakes are limited to 1, 2 or 3 BNB (or native currency of the chain). The Chainlink VRF is then consulted, and used to generate a result of NONE, ODD or EVEN, and the user receives rewards (or not) as below. In addition, winners have a chance to receive an additional claim at the end of the game.

For the initial game, the payouts are as follows.

In the case of a NONE result

Chainlink fee (in BNB) is left on the SquidGame contract	50% of initial stake minus the Chainlink fee is sent back to the player in BNB
25% of initial stake is used to buy Squid and USDT (50 / 50) and added to the UniSwapV2 pool (LP tokens held by SquidGame contract)	25% of initial stake is used to buy Squid which is left on the SquidGame contract (added to general prize pool)

In the case of a WIN result (users bet matches Chainlink VRF generated result)

Chainlink fee (in BNB) is left on the SquidGame contract	Initial stake minus Chainlink fee and winners tax is returned to the player in BNB
5% (winners tax) of initial stake (in BNB) is swapped for Squid and left on the SquidGame contract (added to general prize pool)	An amount of Squid equal to the initial stake (minus fee and tax) is minted to the player in Squid

In the case of a LOSE result (users bet does not match the Chainlink VRF generated result)

Chainlink fee (in BNB) is left on the SquidGame contract	Initial stake in BNB minus the Chainlink fee is used to buy Squid which is then burnt

As per the above, the contract will accumulate:

BNB which should match the Chainlink VRF fees incurred

Squid tokens (in the NONE or WIN case)

These accumulated Squid tokens then form the final payout pool when the game completes, and are paid out as follows:

First, if prizeBoost is non-zero, then prizeBoost amount of Squid is minted to the contract and added to the general prize pool

The general prize pool (in Squid) is distributed 80/10/10 between winners top bottom winners (via contract owner) dev wallet

Assumptions

There are a few assumptions that are worth listing:

The CHAINLINK_FEE for BSC is fixed at 0.2 BNB and can't be modified by Chainlink.

The BNB <> LINK(peg), BNB <> USDT and USDT <> SQM UniSwapV2 pools exist and are liquid.

Issues

Moderate

General prize pool distribution to "bottom winners" is not on-chain

Description

10% of the general prize pool (in Squid) is reserved for "bottom winners" - however in the SquidGame contract this is actually sent to the contract owner. Presumably the intention is that this is then distributed onwards to those players considered "bottom winners" but this logic is not on-chain, so could be abused (i.e. the onward distribution not completed).

Mitigation

Calculate and payout the "bottom winners" proportion on-chain using a similar mechanic to the "top winners" approach.

Team Response

The logic to have a separate payout for "bottom winners" has been removed in commit:
https://github.com/SquidMoon/squid-game/commit/f3553ba31dcc0d39a35b39285b1dfadefab38670

Critical

Calculation of winners may exhaust block gas limits

Description

The calculation of the set of users considered "top winners" (i.e. have the highest aggregated winning score, where the score for an individual bet is the players initial stake) uses quickSort under the covers to order all winners.

The computational cost of quickSort depends on the number of winners to be considered (O(n*log(n)) average where n is the number of winners), so if the number of winners is high enough, this calculation may exhaust gas limits.

Mitigation

Limit the number of winners (or players) to a fixed value computed based on the gas cost of quickSort.

Team Response

The number of new players is now limited to 5,000 in commit:
https://github.com/SquidMoon/squid-game/commit/188890c4de82cb795239c665d3a11fa8cb8576c1

Critical

Winning bet may exhaust block gas limits

Description

When a player wins a bet, their score is updated via _createOrUpdateWinner . In the case of an existing player, this function loops over the whole winners array in order to find the index of the existing player so as to increment their score. Since the length of the winners array is unbounded, this could potentially exhaust the block gas limit.

Mitigation

The contract already tracks an `address` to `uint256` index in the `winners` array via:
`mapping(address => uint256) public winnerId;`

The contract can simply look up the existing players index in this mapping instead of looping over all winners. The code should ensure a player (address) only ever has a single entry in the `winners` array

Team Response

Mitigation approach was implemented in commit:
https://github.com/SquidMoon/squid-game/commit/84de77c1f1c5d3944a0ea2e25c244e2c71e1d288

Minor

Contract has centralised points of control

Description

Whilst it looks like this is by design, it is worth noting that the contract has several areas where the contract owner can modify the expected behaviour of the game. These include:

- the `distribute` function can only be called by the contract owner
- the `updateEnd` function can be used to make the game never end (thus never reaching the distribution of the general prize fund)
- the `updateSwapSlippage` function can be set so that every token swap is likely to fail (see separate issue immediately below)
- the `updatePrizeBoost` can be used to modify the `prizeBoost` during the game

Mitigation

I don't see any particular reason why `distribute` is only callable by the contract owner. There is a comment:
`Note: Only owner can call this to have enough to set prize boost properly` which I interpret to mean the contract owner wants enough time to be able to set the `priceBoost` before this function is called. An easy mitigation would be to make the `distribute` function callable by anyone, but only 48 hours after the game finishes, meaning the contract owner has enough time to set `prizeBoost` .

Note that even if `distribute` can be called by anyone, since it sends BNB to the contract owner, the contract owner can still cause this function to fail (by not having a payable fallback function). You could consider splitting out this portion of the `distribute` function to a separate function that just sends BNB and LINK on the contract back to the contract owner.

You could consider removing the `updateEnd` function, or limiting it to reasonable values.

You could consider removing the `updateSwapSlippage` and instead leaving it as a fixed value, although this also has some issues (see separate issue immediately below).

You could consider allowing the `updatePrizeBoost` to only be called once the game completes to make the intention (that it is calculated post-completion) clearer

Team Response

Centralised points of control were acknowledged, and the distribute function opened up with a delay at commit:
https://github.com/SquidMoon/squid-game/commit/656abaf047115404fbf87624758d0963248e6d30

Critical

Bet resolution can fail depending on UniSwapV2 liquidity

Description

When resolving bets, the contract automatically makes UniSwapV2 trades. These trades are subject to the usual UniSwapV2 slippage concerns, and the contract enforces that the slippage for any trade is less than `swapSlippage` (which is set by the contract owner).

However, in the case where slippage is higher than `swapSlippage` the bet resolution will revert and the player (and contract) won't receive their rewards, or their loss won't be accounted for.

This also opens an easy attack vector for the contract owner, which is to set `swapSlippage` to zero, forcing all bet resolutions to fail, in which case the contract will accumulate BNB (the players initial stake) which can then be claimed by the contract owner using `distribute`.

It is also the case that when swapping BNB <> SQM there are two underlying trades performed (BNB <> USDT, USDT <> SQM) and so this trade may want to consider having a high possible slippage to account for this additional hop.

Mitigation

There isn't an obvious mitigation that I can think of - setting a high `swapSlippage` will inevitably mean users transactions are sandwiched and the contract will leech value to sandwichers.

Possibly reasonable values of `swapSlippage` could be calculated based on the TWAP and current UniSwapV2 pool liquidity, but a sandwicher can always control pool liquidity and cause transactions to still fail.

A "retry" mechanism could be considered whereby if a bet resolution did fail, a user can retry at a later time when there is more liquidity.

For the last part of the issue, increasing the slippage for BNB <> SQM (vs. BNB <> LNK and BNB <> USDT) might be considered.

Team Response

Team has added the ability for a player, whose bet is not resolved, to be reimbursed in commit:
https://github.com/SquidMoon/squid-game/commit/83c57ab0a89b6dbc266ac6ab399126405b2dbf9d

It should be noted that the reimburse function can only be called by the contract owner, not an individual player

Minor

TWAP for SQM / USDT and BNB / USDT may be for different periods

Description

The most efficient route for SQN <> BNB is via USDT. The `Oracle.sol` contract wraps this up via its `consultBnbForSqm` function which consults both `thebnb_usdt` and `usdt_sqm` oracle pairs to return a crossed price.

However it is worth noting that the TWAPs consulted here can be independently updated (for example through a call to `consultBnbForUsdt`) and so may coverdifferent time periods, although always over at least 1 period (but may be over a longer period).

Ideally these two TWAPs would be synchronised when returning a crossed price to avoid any basis risk between the two pairs. Whether this has a material impactwould depend on the relative liquidity between these pools.

Mitigation

You could consider always updating both oracle pairs when consulting either the BNB <> USDT price, or the USDT <> SQM price.

Team Response

Team implemented the above mitigation ensuring that both oracles are always updated at the same time, in commit:
https://github.com/SquidMoon/squid-game/commit/175297e5f31da5db5551c243b7c6ef5a11036806

Minor

Setting swapSlippage via updateSwapSlippage could cause a subsequent overflow

Description

If the contract owner sets the `swapSlippage` value too high, then the `_amountOutAfterSlippage` and `_amountInAfterSlippage` functions could overflow.

Mitigation

Consider checking that the updated `swapSlippage` values are sensible (i.e. between 0 and 1) in `updateSwapSlippage`

Team Response

Team implemented the suggested mitigation to check for sensible values at commit:
https://github.com/SquidMoon/squid-game/commit/ec34b61e424d61d0e7f95573ea628275fba5af11

Comment

Miscellaneous code suggestions

Description

These are some miscellaneous code readability / gas improvements - they should be considered discretionary as they don't impact the logic of the contract, but you may want to consider small changes to resolve these.

- the `initialize` function could verify that `_oracle` is a valid address (i.e. not `address(0)`) as it does for the `devWallet` .
- the `Oracle.sol` contract could implement the `IOracle` interface to ensure compile time checks that it fulfils this interface.
- the `_amountOutAfterSlippage` / `_amountInAfterSlippage` functions can directly reference `swapSlippage` rather than taking it as a parameter.
- the `bet.result` value doesn't need to be explicitly stored as it is not referenced directly anywhere.
- the comment `This function will use all SQUID that its hold to buy BNB that will be used as pot` looks incorrect - the pot is denominatedin SQM and paid out in SQM, not BNB

Mitigation

These all have simple mitigations detailed above.

Team Response

Team comments:

- `IOracle` is not used because of solc versions conflicts (Game uses 0.8.9 and Oracle uses 0.6.6)
- `Keeping bet.result` because 1) the gas used isn't high 2) it can be useful for UI or user want to see this value on-chain

Remaining suggested mitigations were implmented in commit:
https://github.com/SquidMoon/squid-game/commit/7859c44705661007260a6c02df432b2ada290caa

Testing / Deployment

The contracts were deployed on a local hardhat BSC fork and test cases (including skipped) executed successfully.

Test cases look comprehensive although in particular excessive swap slippage isn't executed.

```
 adamdossa@Adams-MacBook-Pro squid-game % yarn test
yarn run v1.22.5
$ hardhat typechain && hardhat test
Generating typings for: 0 artifacts in dir: typechain for target: ethers-v5
Successfully generated 37 typings!
No need to generate any newer typings.


  Simulations
     simulations
GAME price before bets (in USD): 3.136344546170669586
> Bets ended
GAME price after bets (in USD): 3.159120259876489467
GAME tokens hold by contract: 649.548987217721651218
ETH to distribute (total): 0.000348618816396918
> Rewards distribuited
GAME price after distribution (in USD): 3.159120259876489467
✓ simulation #1 - base probabilityes, winners sell all (144521ms)


  SquidGame
    placeBet
       ✓ should place a bet (118ms)
    fulfillRandomness
       ✓ should lock all money when losing (890ms)
       ✓ should receive ETH back + tokens when winning (1474ms)
       ✓ should receive half ETH when result is none and use other half to add liquidity (1802ms)
       ✓ should accumulate score when winning twice (2440ms)
       ✓ should distribute probability according (115561ms)


  7 passing (5m)

Done in 317.40s
```