AUDIT REPORT

MegaPot

June 2024

Prepared by
0xnevi

Introduction

A time-boxed security review of the **Megapot** 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 **MegaPot**

Megapot contract is a lottery-based contract where users win jackpots based on liquidity funded by liquidity providers and tickets purchased by users. Liquidity providers earn a percentage yield from users' ticket entries.

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* - a0b633594bc5a1eb6cca45d68414d505c5b578a5**

Scope

The following smart contracts were in scope of the audit:

- `Lottery.sol`
- `BasedLottery.sol`

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

- Critical & High: 4 issues
- Medium: 4 issues
- Low: 4 issues
- Informational: 3 issues

Findings Summary

ID	Title	Severity
[C-01]	LPs will always lose their chance at winning the lottery for the first lottery run	Critical
[C-02]	Multiple for loops can result in permanent locking of funds	Critical
[C-03]	Unfair determination of winning ticket	Critical
[H-01]	Protocol may lose fees if `lpFeesTotal` is not at least 1 ETH	High
[M-01]	LPs cannot adjust risk percentage unless they deposit the minimum ticket price	Medium
[M-02]	In case of a force release of lottery lock, user purchased tickets are subjected to the next lottery	Medium
[M-03]	Users purchasing tickets can self-refer to gain a discount	Medium
[M-04]	Excess entropy fees paid will not be refunded	Medium
[L-01]	Possibility of locked funds if no winners are found	Low
[L-02]	First lottery run can be delayed by a day	Low
[L-03]	Lack of two-step admin transfer	Low
[L-04]	Donated ETH can be stuck in the contracts forever	Low
[I-01]	Implement events for state changes	Informational
[I-02]	Incomplete NatSpec documentation	Informational

ID	Title	Severity
[I-03]	Usage of floating pragma	Informational

Detailed Findings

[C-01] LPs will always lose their chance at winning the lottery for the first lottery run

Severity

Impact: High, for the first lottery, LPs lose the chance to win the lottery

Likelihood: High, it will always occur for the first lottery

Description

When running a lottery, if the `lpPoolTotal` is greater than `userPoolTotal`, then lps stand a chance to win the lottery too if the `winningTicket` exceeds the `ticketCountTotalBps`

```
        // Jackpot is won by LP's, so LP's get both the user pool and LP pool
        lastWinnerAddress = address(0);
        // Distribute user pool to the LP's
        distributeUserPoolToLps();
        returnLpPoolBackToLps();
        emit LotteryRun(
            lastLotteryEndTime,
            lastWinnerAddress,
            winningTicket,
            lpPoolTotal,
            0
        );
```

However, in `determineWinnerAndAdjustStakes()`, `stakeLps()` is never called after `distributeLpFeesToLps()` for the first lottery run, so it is guaranteed that lps will have zero chance to win a portion of the `userPoolTotal` since `userPoolTotal` will always be greater than `lpPoolTotal`.

Recommendations

Call `stakeLps()` after `distributeLpFeesToLps()` within `determineWinnerAndAdjustStakes()`

```
    function determineWinnerAndAdjustStakes(bytes32 randomNumber) private
{
        lastLotteryEndTime = block.timestamp;

        // No tickets bought
        if (ticketCountTotalBps == 0) {
```

```
            // Return LP Pool back to LPs
            returnLpPoolBackToLps();
            // Reset LP Pool before iniitalizing pool again
            lpPoolTotal = 0;
            stakeLps();
            emit LotteryRun(lastLotteryEndTime, address(0), 0, 0, 0);
            return;
        }

        // Distribute LP fees to LP's
        distributeLpFeesToLps();
+       stakeLPs()
```

Client

Acknowledged - design choice

[C-02] Multiple for loops can result in permanent locking of funds

Severity

Impact: High, LP and user funds can be permanently locked in the contract

Likelihood: High, based on current ticket prices, block gas limit can easily be reached

Description

In Megapot's lottery contract, there are mechanisms in place to limit the number of active lps and users such as

- Only allowing deposits/purchases of multiples of `ticketPrice`
- Implementing a maximum amount of `lpPoolCap`

The winners are determined via `determineWinnerAndAdjustStakes()`, which often involves multiple for loops within the following private functions potentially called which loops through all the active lps and users within the `activeLpAddresses` and `activeUserAddresses` respectively.

- `returnLpPoolBackToLps()`
- `stakeLps()`
- `distributeLpFeesToLps()`
- `findWinnerFromUsers()`
- `clearUserTicketPurchases()`

Given the current minimum ticket price of 0.001 ETH, attackers can maliciously create multiple addresses to deposit/purchase (via `lpDeposit()/purchaseTickets()`) tickets for just 0.001 ETH which can easily cause the 30 million block gas limit to be reached.

Take for example each SLOAD costs 2100 gas, so with 20000 active users, it will result in a minimum of 2100 * 20000 = ~42 million gas within `clearUserTicketPurchases()` which is also the only place where the `activeUserAddresses` is ever deleted.

If `determineWinnerAndAdjustStakes()` reverts due to reaching the block gas limit, it will result in the permanent locking of user and LP funds since there are no refund mechanisms within the protocol and the ability to withdraw deposited LP funds and lottery winnings is always dependent on the successful call to `determineWinnerAndAdjustStakes` during each lottery result.

Note: This can potentially affect `withdrawAllLP()` as well.

Recommendations

The fix is non-trivial. Consider implementing limits to a number of unique active players and lps and make sure to perform extensive tests to ensure block gas limits are never reached.

Client

Fixed

[C-03] Unfair determination of winning ticket

Severity

Impact: High, unfair determination to win lottery ticket

Likelihood: High, all lottery runs will be subjected to unfair determination of winning lottery ticket

Description

In the Megapot contract, the winning ticket is determined as follows:

```
function getWinningTicket(
    bytes32 rawRandomNumber,
    uint256 max
) private pure returns (uint256) {
    return (uint256(rawRandomNumber) % max) + 1;
}

function findWinnerFromUsers(
    uint256 winningTicket
) private view returns (address) {
    uint256 cumulativeTicketsBps = 0;
    for (uint256 i = 0; i < activeUserAddresses.length; i++) {
        address userAddress = activeUserAddresses[i];
        User memory user = usersInfo[userAddress];
        cumulativeTicketsBps += user.ticketsPurchasedTotalBps;
        if (winningTicket <= cumulativeTicketsBps) {
            return userAddress;
        }
    }
    // No winner found, this should never happen
    return address(0);
}
```

1. If `rawRandomNumber >= max`, `getWinningTicket` always return `1`
2. If `rawRandomNumber < max`, `getWinningTicket` always return `rawRandomNumber + 1`

In scenario 1, it is guaranteed that the winner will be the first user that bought tickets given the minimum `ticketsPurchasedTotalBps` is `9500` based on current fees set and ticket prices. Also note this value is not scaled up to BPS when comparing against `cumulativeTicketBps`.

In scenario 2, all of the cumulative tickets will be assigned as if the next user bought those tickets if `winningTicket` is still greater than `cumulativeTicketBps`. This favors users that have bought tickets earlier before the lottery run is initiated, which means the lottery run is essentially not completely random at all.

Recommendations

The fix is non-trivial. the protocol should consider refactoring how to determine a true winner based on a number of tickets bought and the random number returned by the entropy contract.

I suggest looking into the architecture of LooksRare YoloV2, which finds the winner by entry index instead of total tickets purchased as seen here

Client

Acknowledged - design choice, Since pyth random number acts as a seed, it is likely that the outcome would still be completely random.

[H-01] Protocol may lose fees if `lpFeesTotal` is not at least 1 ETH

Severity

Impact: Medium, requires lottery with less than 20 ETH worth of tickets bought

Likelihood: High, protocol may lose a material amount of fees

Description

During every lottery run, fees will be distributed via `distributeLpFeesToLps()` within `determineWinnerAndAdjustStakes` before results are determined.

The protocol will take 10% of lp fees only if the accumulated `lpFeesTotal` is greater than 1 ETH.

```
    if (protocolFeeAddress != address(0) && lpFeesTotal >= 1 ether) {
        uint256 protocolFee = lpFeesTotal / 10;
        lpFeesTotal -= protocolFee;
        protocolFeeClaimable += protocolFee;
    }
```

This essentially means the protocol will never accumulate fees if the accumulated lpFeesTotal is less than 1 ETH. Based on current `feeBPS` of 500 (5%) and ticket price of 0.001 ETH, it means a minimum of `1 ETH /`

`(0.001 ETH * 5%) * 0.001 ETH = 20 ETH` (~75000 USD at current prices) worth of lottery tickets must be bought per lottery round for protocol to accumulate fees.

This might happen often during each lottery round, resulting in a loss of protocol fees. 10 rounds of the lottery with less than 20 ETH of lottery tickets bought would result in 1 ETH lost.

Recommendations

Shift and compute protocol fees within `purchaseTickets`

```
function purchaseTickets(address referrer) public payable {
    require(
        msg.value > 0 && msg.value % ticketPrice == 0,
        "Invalid purchase amount, must be positive and a multiple of
minimum ticket size"
    );
    require(!lotteryLock, "Lottery is currently running!");

    // This is in "Bps", so a 0.001 ETH ticket purchase at feeBps = 500
(or 5%) would result in 9500
    uint256 ticketsPurchasedBps = (msg.value / ticketPrice) *
        (10000 - feeBps);

    User storage user = usersInfo[msg.sender];
    if (!user.active) {
        user.active = true;
        // Add new user address, will be reset when lottery ends
        activeUserAddresses.push(msg.sender);
    }

    // Calculate fees. Total fees are inclusive of referrer fees, if there
are any
    uint256 allFeeAmount = (msg.value * feeBps) / 10000;

    // Referrer fee is active only if a valid referrer address is
provided. otherwise, no referrer fee is taken from the total fee, aka LPs
get all the fees
    uint256 referralFeeAmount = (referrer != address(0))
        ? (msg.value * referralFeeBps) / 10000
        : 0;
    uint256 lpFeeAmount = allFeeAmount - referralFeeAmount;
+   uint256 protocolFee = lpFeeAmount * protocolFeeBps / 10000
+   uint256 lpFeeAmount = lpFeeAmount - protocolFee

    // Add entry to user pool. Note pool is post-fee
    userPoolTotal += (msg.value * (10000 - feeBps)) / 10000;

    // Add to total accumulators
    allFeesTotal += allFeeAmount;
    referralFeesClaimable[referrer] += referralFeeAmount;
    referralFeesTotal += referralFeeAmount;
    lpFeesTotal += lpFeeAmount;
```

```
    user.ticketsPurchasedTotalBps += ticketsPurchasedBps;
    ticketCountTotalBps += ticketsPurchasedBps;
+   protocolFeeClaimable += protocolFee;

    emit UserTicketPurchase(msg.sender, ticketsPurchasedBps);
}
```

Client

Acknowledged - design choice

[M-01] LPs cannot adjust risk percentage unless they deposit the minimum ticket price

Severity

Impact: Medium, users would have to commit a minimum ticket price (0.001 ETH) to adjust the risk percentage

Likelihood: High, because it occurs as long as an adjustment of risk percentage is intended

Description

In megapot, lps can decide how much of their deposited principal worth of ETH can be staked towards the lottery via the `riskPercentage` variable. The higher the amount staked, the higher the potential yield they will earn from a lottery run based on fees accumulated.

However, users cannot adjust risk percentage unless an additional lp worth the minimum ticket price is committed due to the second require check in `lpDeposit()`. This essentially means to change `riskPercentage` to a non-zero value, the lp would have to risk a minimal 0.001 ETH extra.

```
function lpDeposit(uint256 riskPercentage) public payable {
    require(
        riskPercentage > 0 && riskPercentage <= 100,
        "Invalid risk percentage"
    );
    require(
        msg.value % ticketPrice == 0,
        "Invalid deposit amount, must be a multiple of minimum ticket
size"
    );
    require(
        lpPoolTotal + msg.value <= lpPoolCap,
        "Deposit exceeds LP pool cap"
    );
    require(!lotteryLock, "Lottery is currently running!");

    LP storage lp = lpsInfo[msg.sender];
    if (!lp.active) {
        require(
```

```
            msg.value > 0,
            "Invalid deposit amount, must be strictly positive for new LP"
        );
        lp.active = true;
        // Add newly active LP address
        activeLpAddresses.push(msg.sender);
    }
    lp.principal += msg.value;
    lp.riskPercentage = riskPercentage;

    emit LpDeposit(msg.sender, msg.value, riskPercentage);
}
```

Recommendations

If lps already has a deposited position represented by `lp.principal > 0`, allow re-adjustments of risk percentage without paying minimum ticket price.

Client

Fixed

[M-02] In case of a force release of lottery lock, user purchased tickets are subjected to the next lottery

Severity

Impact: High, users' funds can be locked or subjected to another lottery run

Likelihood: Low, requires Pyth entropy contract to be non-functional

Description

In case of an emergency where entropy contracts revert or do not callback to `entropyCallback()`, the protocol can force unlock the `lotteryLock`. This allows the protocol to run another lottery presumably once the entropy contract can be utilized successfully again.

```
function forceReleaseLotteryLock() external onlyOwner {
    lotteryLock = false;
}
```

Note that LPs can only withdraw deposited funds and winning users can only claim winnings after a successful call to `determineWinnerAndAdjustStakes()` where the results are determined. However, if the entropy contract can never invoke the `entroyCallback()` again, these funds would be locked.

Additionally, even if the entropy contract is presumably functional again, this could potentially result in another unfair scenario where the previous expected lottery run results are never determined, and the

current ETH present within the user and lp total pots would be subjected to a completely different lottery run with potentially higher risks due to the possibility of more LPs deposit/tickets purchased.

Recommendations

Either acknowledge this as a design decision or implement admin-only functionalities to refund LPs and active users' funds for the failed lottery run.

Client

Acknowledged - design choice

[M-03] Users purchasing tickets can self-refer to gain a discount

Severity

Impact: Medium, allows a discount on tickets purchased and reduces lp fees

Likelihood: High, can be executed by any user purchasing tickets

Description

When purchasing tickets, if there is no referrer address, the fees are automatically delegated to lps

```
uint256 referralFeeAmount = (referrer != address(0))
    ? (msg.value * referralFeeBps) / 10000
    : 0;
uint256 lpFeeAmount = allFeeAmount - referralFeeAmount;
```

Users purchasing tickets can self-refer to their own address or an address they control to gain a discount via `purchaseTickets()` (2.5% discount based on intended refferalFeeBps). This reduces potential yield for LPs as well since `lpFeeAmount` added to `lpFeesTotal` would be lower.

Recommendations

- Acknowledge the limitation as a design decision
- Implement a check to ensure `referrer != msg.sender`
- Implement a whitelist of referrers

Client

Acknowledged - design choice

[M-04] Excess entropy fees paid will not be refunded

Severity

Impact: Medium, excess ETH paid for calling entropy is locked

Likelihood: Medium,

Description

In pyth `Entropy.sol` contract, it is explicitly noted that excess fees paid will not be refunded.

In `runLottery`, it is only checked that fees paid is greater than equal to `fee`. Hence, if the user overpays the fee for the entropy contract to run the lottery, consider refunding the excess back to the caller.

```
function runLottery(bytes32 userRandomNumber) external payable {
    // TIMELOCK
    require(
        block.timestamp >= lastLotteryEndTime + roundDurationInSeconds,
        "Lottery can only be run once a day"
    );

    require(!lotteryLock, "Lottery is currently running!");

    // acquire lottery lock
    lotteryLock = true;

    uint256 fee = entropy.getFee(entropyProvider);
    require(msg.value >= fee, "Insufficient gas to generate random
number");

    // Request the random number from the Entropy protocol. The call
returns a sequence number that uniquely
    // identifies the generated random number. Callers can use this
sequence number to match which request
    // is being revealed in the next stage of the protocol. Since we lock
the call to this function,
    // we don't need to care about the sequence number
    entropy.requestWithCallback{value: fee}(
        entropyProvider,
        userRandomNumber
    );

    emit LotteryRunRequested(msg.sender);
}
```

Recommendations

Refund excess fee paid to caller. Note that reentrancy is not possible because of the once a day lottery run checks as well as the `lotteryLock`.

```
function runLottery(bytes32 userRandomNumber) external payable {
    // TIMELOCK
    require(
        block.timestamp >= lastLotteryEndTime + roundDurationInSeconds,
        "Lottery can only be run once a day"
    );
```

```
    require(!lotteryLock, "Lottery is currently running!");

    // acquire lottery lock
    lotteryLock = true;

    uint256 fee = entropy.getFee(entropyProvider);
    require(msg.value >= fee, "Insufficient gas to generate random
number");
+   (bool success, ) = msg.sender.call{value: msg.value - fee}("");
+   require(success, "Transfer failed");


    // Request the random number from the Entropy protocol. The call
returns a sequence number that uniquely
    // identifies the generated random number. Callers can use this
sequence number to match which request
    // is being revealed in the next stage of the protocol. Since we lock
the call to this function,
    // we don't need to care about the sequence number
    entropy.requestWithCallback{value: fee}(
        entropyProvider,
        userRandomNumber
    );

    emit LotteryRunRequested(msg.sender);
}
```

Client

Fixed

[L-01] Possibility of locked funds if no winners are found

In `findWinnerFromUsers`, if no winners are found, it will return `address(0)`, and assign winnings to this address. This essentially locks the winning within the contract with no way of withdrawing it.

```
function findWinnerFromUsers(
    uint256 winningTicket
) private view returns (address) {
    uint256 cumulativeTicketsBps = 0;
    for (uint256 i = 0; i < activeUserAddresses.length; i++) {
        address userAddress = activeUserAddresses[i];
        User memory user = usersInfo[userAddress];
        cumulativeTicketsBps += user.ticketsPurchasedTotalBps;
        if (winningTicket <= cumulativeTicketsBps) {
            return userAddress;
        }
    }
    // No winner found, this should never happen
    return address(0);
}
```

However, if it so happens that no winners are found, these funds will be permanently locked in the contract, so consider returning the winner's address as an admin controlled address instead of `address(0)`. This way, admins can withdraw these winnings.

Client

Fixed

[L-02] First lottery run can be delayed by a day

Even when no lottery tickets are bought, `runLottery()` can still be called to invoke the 24 hour lottery check. Hence, the first lottery run can be delayed by a day by simply being the first user that calls `runLottery` once contracts are deployed

Consider only allowing calls to `runLottery()` only when there are tickets bought i.e. `ticketCountTotalBps != 0`

Client

Acknowledged - design choice

[L-03] Lack of two-step admin transfer

All of the contracts in scope inherit OZ's OwnableUpgradeable.sol contract. Single-step ownership transfers mean that if a wrong address was passed when transferring ownership or admin rights, ownership of the contract could be lost forever.

The ownership transfer should be done with great caution, so consider using Ownable2StepUpgradeable by OpenZeppelin to perform 2-step admin transfers.

Client

Fixed

[L-04] Donated ETH can be stuck in the contracts forever

There is a `receive()` fallback function implemented which allows the contracts to receive ETH directly.

However, there is currently no method to withdraw such funds, essentially meaning they will be stuck inside the contracts forever. Consider adding an admin-only function to withdraw excess donated funds or remove the `receive()` fallback method.

Client

Fixed

[I-01] Implement events for state changes

The following functions involve sensitive state changes to important variables used within the protocol. It is recommended to emit an explicit event when such variables are changed

- `setTicketPrice()`
- `setRoundDurationInSeconds()`
- `setReferralFeeBps()`
- `setFeeBps()`
- `setLpPoolCap()`
- `setProtocolFeeAddress()`
- `forceReleaseLotteryLock()`

[I-02] Incomplete NatSpec documentation

The Natspec docs on all external and public functions are missing - @param, @return, @dev.Consider implementing descriptive docs for each method and contract which will help users, developers, and auditors.

[I-03] Usage of floating pragma

Consider using a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. The protocol is currently using a floating pragma version of `^0.8.23`.

Additionally, consider using the latest available version of solidity.