FYI: Analyze LPManagement.sol Feb 10, 2025 2p U.S. PST

https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.26+commit.8a97fa7a.js

Remix 50:

Check-effects-interaction:Potential violation of Checks-Effects-Interaction pattern in LPManagement.withdraw(uint256,address): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.more

Pos: 265:4:
Block timestamp: Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.more
Pos: 101:27:
Block timestamp: Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.more
Pos: 117:28:
Gas costs: Gas requirement of function LPManagement.getETHUSDCExchangeRate is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 76:4:
Gas costs: Gas requirement of function LPManagement.setCommitment is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 91:4:
Gas costs: Gas requirement of function LPManagement.createCashCall is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 114:4:
Gas costs: Gas requirement of function LPManagement.getCashCalls is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 131:4:
Gas costs: Gas requirement of function LPManagement.makePayment is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 136:4:
Gas costs: Gas requirement of function LPManagement.executeCashCall is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 157:4:
Gas costs: Gas requirement of function LPManagement.revertExecution is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or

actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 169:4:
Gas costs: Gas requirement of function LPManagement.applyPenalty is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 182:4:
Gas costs: Gas requirement of function LPManagement.removeAdmin is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 211:4:
Gas costs: Gas requirement of function LPManagement.getAdmins is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 245:4:
Gas costs: Gas requirement of function LPManagement.withdraw is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 265:4:
For loop over dynamic array: Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.more
Pos: 221:8:
Similar variable names:LPManagement.makePayment(address,uint256) : Variables have very similar names "cashCall" and "cashCalls". Note: Modifiers are currently not considered by this static analysis.
Pos: 140:8:
Similar variable names:LPManagement.makePayment(address,uint256) : Variables have very similar names "cashCall" and "cashCalls". Note: Modifiers are currently not considered by this static analysis.
Pos: 140:36:
Similar variable names:LPManagement.makePayment(address,uint256) : Variables have very similar names "cashCall" and "cashCalls". Note: Modifiers are currently not considered by this static analysis.
Pos: 141:16:
Similar variable names:LPManagement.makePayment(address,uint256) : Variables have very similar names "cashCall" and "cashCalls". Note: Modifiers are currently not considered by this static analysis.
Pos: 144:17:
Similar variable names:LPManagement.makePayment(address,uint256) : Variables have very similar names "cashCall" and "cashCalls". Note: Modifiers are currently not considered by this static analysis.
Pos: 147:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 53:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 54:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 65:8:

Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 71:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 78:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 86:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 97:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 98:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 99:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 100:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 101:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 115:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 116:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 117:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 122:12:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 137:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 141:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 144:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 159:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug

in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 160:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 170:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 172:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 173:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 184:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 200:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 201:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 212:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 215:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 234:8:
Guard conditions: Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.more
Pos: 266:8:

Solhint:

Compiler version ^0.8.28 does not satisfy the ^0.5.8 semver requirement

Pos: 1:1
global import of path @openzeppelin/contracts/security/Pausable.sol is not allowed. Specify names to import individually or bind all exports of the module into a name (import "path" as Name)
Pos: 1:3
global import of path @openzeppelin/contracts/security/ReentrancyGuard.sol is not allowed. Specify names to import individually or bind all exports of the module into a name (import "path" as Name)
Pos: 1:4
global import of path @chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol is not allowed. Specify names to import individually or bind all exports of the module into a name (import "path" as Name)
Pos: 1:5

```
Explicitly mark visibility in function (Set ignoreConstructors to true if using solidity >=0.7.0)
Pos: 5:51
Error message for require is too long
Pos: 9:64
Error message for require is too long
Pos: 9:70
Error message for require is too long
Pos: 9:85
Error message for require is too long
Pos: 9:99
Error message for require is too long
Pos: 9:100
Avoid making time-based decisions in your business logic
Pos: 28:100
Error message for require is too long
Pos: 9:115
Error message for require is too long
Pos: 9:116
Avoid making time-based decisions in your business logic
Pos: 29:116
Error message for require is too long
Pos: 13:121
Error message for require is too long
Pos: 9:233
Code contains empty blocks
Pos: 46:271
```

Solidity Scan

```
Welcome to Remix 0.60.0

Your files are stored in indexedDB, 24.78 MB / 2 GB used

You can use this terminal to:
```

- Check transactions details and start debugging.
- Execute JavaScript scripts:
  - *Input a script directly in the command line interface*
  - *Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.exeCurrent()` in the command line interface*
  - *Right-click on a JavaScript file in the file explorer and then click `Run`*

```
The following libraries are accessible:
```

- web3.js
- ethers.js
- sol-gpt *<your Solidity question here>*

```
Type the library name to see available commands.

SolidityScan result for lp-dapp-1739157515355/LPManagement.sol:
```

#	NAME	SEVERITY	CONFIDENCE	DESCRIPTION	REMEDIATION
1.	INCORRECT ACCESS CONTROL	critical	1	Access control plays an important role in the segregation of privileges in smart contracts and other applications. If this is misconfigured or not properly validated on sensitive functions, it may lead to loss of funds, tokens and in some cases compromise of the smart contract. The contract is importing an access control library but the function is missing the modifier.	Not Available
2.	IMPROPER VALIDATION IN REQUIRE/ASSERT STATEMENTS	high	0	The contract is identified as having improper validation in **require** or **assert** statements when using user-passed arguments. Specifically, if user-supplied parameters are not properly validated within these checks, it can lead to vulnerabilities where incorrect or malicious inputs are not correctly handled. This can result in unexpected behavior, security flaws, and potential exploitation by attackers.	Not Available
3.	NONREENTRANT MODIFIER PLACEMENT	low	0	When using the **nonReentrant** modifier, it is crucial to place it before all other modifiers in a function. Placing it first ensures that all other modifiers are unable to bypass the reentrancy protection.	Not Available
4.	USE OF FLOATING PRAGMA	low	2	Solidity source files indicate the versions of the compiler they can be compiled with using a pragma directive at the top of the solidity file. This can either be a floating pragma or a specific compiler version. The contract was found to be using a floating pragma which is not considered safe as it can be compiled with all the versions described.	Not Available
5.	MISSING EVENTS	low	1	Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log — a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions. The contract was found to be missing these events on the function which would make it difficult or impossible to track these transactions off-chain.	Not Available
6.	USE SCIENTIFIC NOTATION	informational	0	Although the Solidity compiler can optimize exponentiation, it is recommended to prioritize idioms not reliant on compiler optimization. Utilizing scientific notation enhances code clarity, making it more self-explanatory and aligning with best practices in Solidity development.	Not Available

7.	MISSING UNDERSCORE IN NAMING VARIABLES	informational	0	Solidity style guide suggests using underscores as the prefix for non-external functions and state variables (private or internal) but the contract was not found to be following the same.	Not Available
8.	NAME MAPPING PARAMETERS	informational	0	After Solidity 0.8.18, a feature was introduced to name mapping parameters. This helps in defining a purpose for each mapping and makes the code more descriptive.	Not Available
9.	BLOCK VALUES AS A PROXY FOR TIME	informational	1	Contracts often need access to time values to perform certain types of functionality. Values such as **block.timestamp** and **block.number** can be used to determine the current time or the time delta. However, they are not recommended for most use cases. For **block.number**, as Ethereum block times are generally around 14 seconds, the delta between blocks can be predicted. The block times, on the other hand, do not remain constant and are subject to change for a number of reasons, e.g., fork reorganizations and the difficulty bomb. Due to variable block times, **block.number** should not be relied on for precise calculations of time.	Not Available
10.	MISSING INDEXED KEYWORDS IN EVENTS	informational	2	Events are essential for tracking off-chain data and when the event paraemters are **indexed** they can be used as filter options which will help getting only the specific data instead of all the logs.	Not Available
11.	UNUSED RECEIVE FALLBACK	informational	2	The contract was found to be defining an empty function. It is not recommended to leave them empty unless there's a specific use case such as to receive Ether via an empty **receive()** function.	Not Available
12.	USE CALL INSTEAD OF TRANSFER OR SEND	informational	2	The contract was found to be using **transfer** or **send** function call. This is unsafe as **transfer** has hard coded gas budget and can fail if the user is a smart contract.	Not Available

13.	USE SELFBALANCE() INSTEAD OF ADDRESS(THIS).BALANCE	gas	0	In Solidity, efficient use of gas is paramount to ensure cost-effective execution on the Ethereum blockchain. Gas can be optimized when obtaining contract balance by using **selfbalance()** rather than **address(this).balance** because it bypasses gas costs and refunds, which are not required for obtaining the contract's balance. Lines: ["266:266"]	To rectify this issue, developers are encouraged to replace instances of **address(this).balance** with **selfbalance()** wherever applicable. This optimization not only ensures streamlined gas operations but also contributes to substantial cost savings during contract execution.
14.	DEFINE CONSTRUCTOR AS PAYABLE	gas	0	Developers can save around 10 opcodes and some gas if the constructors are defined as payable. However, it should be noted that it comes with risks because payable constructors can accept ETH during deployment. Lines: ["52:61"]	It is suggested to mark the constructors as payable to save some gas. Make sure it does not lead to any adverse effects in case an upgrade pattern is involved.
15.	STORAGE VARIABLE CACHING IN MEMORY	gas	0	The contract is using the state variable multiple times in the function. **SLOADs** are expensive (100 gas after the 1st one) compared to **MLOAD/MSTORE** (3 gas each). Lines: ["114:128","136:154","211:230"]	Storage variables read multiple times inside a function should instead be cached in the memory the first time (costing 1 **SLOAD**) and then read from this cache to avoid multiple **SLOADs**.
16.	SPLITTING REQUIRE STATEMENTS	gas	1	Require statements when combined using operators in a single statement usually lead to a larger deployment gas cost but with each runtime calls, the whole thing ends up being cheaper by some gas units. Lines: ["117:117"]	It is recommended to separate the require statements with one statement/validation per line.

17.	AVOID RE-STORING VALUES	gas	0	The function is found to be allowing re-storing the value in the contract's state variable even when the old value is equal to the new value. This practice results in unnecessary gas consumption due to the `Gsreset` operation (2900 gas), which could be avoided. If the old value and the new value are the same, not updating the storage would avoid this cost and could instead incur a `Gcoldsload` (2100 gas) or a `Gwarmaccess` (100 gas), potentially saving gas. Lines: ["85:88","233:242"]	To optimize gas usage, add a check to compare the old value with the new value before updating the storage. Only perform the storage update if the new value is different from the old value. This approach will prevent unnecessary storage writes and reduce gas consumption.
18.	ASSIGNING TO STRUCTS CAN BE MORE EFFICIENT	gas	0	The contract is found to contain a struct with multiple variables defined in it. When a struct is assigned in a single operation, Solidity may perform costly storage operations, which can be inefficient. This often results in increased gas costs due to multiple SLOAD and SSTORE operations happening at once Lines: ["126:126"]	Instead of assigning all struct elements at once, initialize the struct as empty and assign each element individually. This can help in reducing gas costs by minimizing potentially expensive storage operations in a single transaction.
19.	ARRAY LENGTH CACHING	gas	2	During each iteration of the loop, reading the length of the array uses more gas than is necessary. In the most favorable scenario, in which the length is read from a memory variable, storing the array length in the stack can save about 3 gas per iteration. In the least favorable scenario, in which external calls are made during each iteration, the amount of gas wasted can be significant. Lines: ["221:227"]	Consider storing the array length of the variable before the loop and use the stored length instead of fetching it in each iteration.

20.	LONG REQUIRE/REVERT STRINGS	gas	2	The `require()` and `revert()` functions take an input string to show errors if the validation fails. This strings inside these functions that are longer than `32 bytes` require at least one additional `MSTORE`, along with additional overhead for computing memory offset, and other parameters. Lines: ["86:86","100:100","101:101","116:116","117:117","122:122","234:234"]	It is recommended to short the strings passed inside `require()` and `revert()` to fit under `32 bytes`. This will decrease the gas usage at the time of deployment and at runtime when the validation condition is met.
21.	CHEAPER CONDITIONAL OPERATORS	gas	0	During compilation, `x != 0` is cheaper than `x > 0` for unsigned integers in solidity inside conditional statements. Lines: ["78:78","86:86","99:99","116:116","141:141","159:159","172:172","184:184","120:120"]	Consider using `x != 0` in place of `x > 0` in `uint` wherever possible.
22.	CHEAPER INEQUALITIES IN REQUIRE()	gas	1	The contract was found to be performing comparisons using inequalities inside the `require` statement. When inside the `require` statements, non-strict inequalities `(>=, <=)` are usually costlier than strict equalities `(>, <)`. Lines: ["100:100","117:117","266:266"]	It is recommended to go through the code logic, and, if possible, modify the non-strict inequalities with the strict ones to save ~3 gas as long as the logic of the code is not affected.
23.	GAS OPTIMIZATION IN INCREMENTS	gas	0	`++i` costs less gas compared to `i++` or `i += 1` for unsigned integers. In `i++`, the compiler has to create a temporary variable to store the initial value. This is not the case with `++i` in which the value is directly incremented and returned, thus, making it a cheaper alternative. Lines: ["221:221"]	Consider changing the post-increments `(i++)` to pre-increments `(++i)` as long as the value is not used in any calculations or inside returns. Make sure that the logic of the code is not changed.

24.	CHEAPER INEQUALITIES IN IF()	gas	1	The contract was found to be doing comparisons using inequalities inside the if statement. When inside the **if** statements, non-strict inequalities **(>=, <=)** are usually cheaper than the strict equalities **(>, <)**. Lines: ["120:120"]	It is recommended to go through the code logic, and, if possible, modify the strict inequalities with the non-strict ones to save **~3** gas as long as the logic of the code is not affected.

Scan Summary:

Lines Analyzed: 218

Scan Score: 68.81

Issue Distribution: { "critical": 1, "gas": 31, "high": 2, "informational": 12, "low": 6, "medium": 0 }

For more details, go to SolidityScan.