USE OF MULTIPLE PRAGMA VERSIONS

Description

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 project was found to be using multiple pragma versions across files which are not considered safe as they can be compiled with all the versions described.

Affected Files

- betOnBalaji.sol [L2]
- Lock.sol [L2]

Steps to fix

- It is also recommended to keep only one version of Solidity across all the contracts.

LONG NUMBER LITERALS

Description

Solidity supports multiple rational and integer literals, including decimal fractions and scientific notations. The use of very large numbers with too many digits was detected in the code that could have been optimized using a different notation also supported by Solidity.

The value `1000000` was detected on line `80`.

Affected Files

- betOnBalaji.sol [L80]

Steps to fix

- Scientific notation in the form of `2e10` is also supported, where the mantissa can be fractional but the exponent has to be an integer. The literal `MeE` is equivalent to `M * 10**E`. Examples include `2e10`, `2e10`, `2e-10`, `2.5e1`, as suggested in official solidity documentation
- https://docs.soliditylang.org/en/latest/types.html#rational-and-integer-literals

USE OF FLOATING PRAGMA

Description

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.

Affected Files

- Lock.sol [L2]
- betOnBalaji.sol [L2]

Steps to fix

- It is recommended to use a fixed pragma version, as future compiler versions may handle certain language constructions in a way the developer did not foresee.
- Using a floating pragma may introduce several vulnerabilities if compiled with an older version.
- The developers should always use the exact Solidity compiler version when designing their contracts as it may break the changes in the future.

OUTDATED COMPILER VERSION

Description

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

Affected Files

- Lock.sol [L2]
- betOnBalaji.sol [L2]

Steps to fix

- It is recommended to use a recent version of the Solidity compiler that should not be the most recent version, and it should not be an outdated version as well.
- Using very old versions of Solidity prevents the benefits of bug fixes and newer security checks.
- Consider using the solidity version 0.8.18, which patches most solidity vulnerabilities.

[RECOMMENDATION] Avoid block.timestamp Usage

Details of Vulnerability

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.

Steps to Fix

- It is recommended to use trusted external time sources, block numbers instead of timestamps, and/or utilizing multiple time sources to increase reliability. These practices can help mitigate risks of timestamp manipulation and inaccurate timing, increasing the reliability and security of the smart contract.

[RECOMMENDATION] Use ReentrancyGuard to Prevent Reentrancy Attacks

It is recommended to add a Re-entrancy Guard to the functions making external calls.

Inheriting from ReentrancyGuard will make the nonReentrant modifier available, which can be applied to functions to make sure there are no nested (reentrant) calls to them.

Note that because there is a single nonReentrant guard, functions marked as nonReentrant may not call one another. This can be worked around by making those functions private, and then adding external nonReentrant entry points to them.