



March 6, 2024

Express Audit Report for **ForceFinanceCoin [\$FFC]**

DISCLAIMER: This is an automatically generated audit performed with De.Fi Scanner tool. De.Fi smart contract auditing tool is intended to assist in identifying potential vulnerabilities or malicious functions in smart contracts. While this is done to our best effort and knowledge, please notice that no tool can guarantee complete accuracy or comprehensiveness in detecting all possible vulnerabilities.



Project Summary

Project Name	ForceFinanceCoin(ForceFinanceCoin)
Address	0xbf05c4023e735adb912e2cc34c0f391702efec34
Network	1

Issue ID	183
Severity	🎯 Optimization
Status	High
Description Code	string private _nameFallback;
Location	EIP712._nameFallback (EIP712.sol#51) should be constant

Issue ID	183
Severity	🎯 Optimization
Status	High
Description Code	string private _versionFallback;
Location	EIP712._versionFallback (EIP712.sol#52) should be constant

Issue ID	156
Severity	🕒 Low
Status	Medium
Description Code	<pre> function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256 result) { unchecked { // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1, then use // use the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is stored in two 256 // variables such that product = prod1 * 2^256 + prod0. uint256 prod0 = x * y; // Least significant 256 bits of the product uint256 prod1; // Most significant 256 bits of the product assembly { let mm := mulmod(x, y, not(0)) prod1 := sub(sub(mm, prod0), lt(mm, prod0)) } // Handle non-overflow cases, 256 by 256 division. if (prod1 == 0) { // Solidity will revert if denominator == 0, unlike the div opcode on its own. // The surrounding unchecked block does not change this fact. // See https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic. return prod0 / denominator; } // Make sure the result is less than 2^256. Also prevents denominator == 0. if (denominator <= prod1) { revert MathOverflowedMulDiv(); } //////////////////// // 512 by 256 division. //////////////////// // Make division exact by subtracting the remainder </pre>

Issue ID	156
Severity	🕒 Low
Status	Medium
Description Code	<pre> function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256 result) { unchecked { // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1, then use // use the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is stored in two 256 // variables such that product = prod1 * 2^256 + prod0. uint256 prod0 = x * y; // Least significant 256 bits of the product uint256 prod1; // Most significant 256 bits of the product assembly { let mm := mulmod(x, y, not(0)) prod1 := sub(sub(mm, prod0), lt(mm, prod0)) } // Handle non-overflow cases, 256 by 256 division. if (prod1 == 0) { // Solidity will revert if denominator == 0, unlike the div opcode on its own. // The surrounding unchecked block does not change this fact. // See https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic. return prod0 / denominator; } // Make sure the result is less than 2^256. Also prevents denominator == 0. if (denominator <= prod1) { revert MathOverflowedMulDiv(); } //////////////////// // 512 by 256 division. //////////////////// // Make division exact by subtracting the remainder </pre>

Issue ID	156
Severity	🕒 Low
Status	Medium
Description Code	<pre> function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256 result) { unchecked { // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1, then use // use the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is stored in two 256 // variables such that product = prod1 * 2^256 + prod0. uint256 prod0 = x * y; // Least significant 256 bits of the product uint256 prod1; // Most significant 256 bits of the product assembly { let mm := mulmod(x, y, not(0)) prod1 := sub(sub(mm, prod0), lt(mm, prod0)) } // Handle non-overflow cases, 256 by 256 division. if (prod1 == 0) { // Solidity will revert if denominator == 0, unlike the div opcode on its own. // The surrounding unchecked block does not change this fact. // See https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic. return prod0 / denominator; } // Make sure the result is less than 2^256. Also prevents denominator == 0. if (denominator <= prod1) { revert MathOverflowedMulDiv(); } //////////////////// // 512 by 256 division. //////////////////// // Make division exact by subtracting the remainder </pre>

Issue ID	156
Severity	🕒 Low
Status	Medium
Description Code	<pre> function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256 result) { unchecked { // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1, then use // use the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is stored in two 256 // variables such that product = prod1 * 2^256 + prod0. uint256 prod0 = x * y; // Least significant 256 bits of the product uint256 prod1; // Most significant 256 bits of the product assembly { let mm := mulmod(x, y, not(0)) prod1 := sub(sub(mm, prod0), lt(mm, prod0)) } // Handle non-overflow cases, 256 by 256 division. if (prod1 == 0) { // Solidity will revert if denominator == 0, unlike the div opcode on its own. // The surrounding unchecked block does not change this fact. // See https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic. return prod0 / denominator; } // Make sure the result is less than 2^256. Also prevents denominator == 0. if (denominator <= prod1) { revert MathOverflowedMulDiv(); } //////////////////// // 512 by 256 division. //////////////////// // Make division exact by subtracting the remainder </pre>

Issue ID	156
Severity	🕒 Low
Status	Medium
Description Code	<pre> function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256 result) { unchecked { // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1, then use // use the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is stored in two 256 // variables such that product = prod1 * 2^256 + prod0. uint256 prod0 = x * y; // Least significant 256 bits of the product uint256 prod1; // Most significant 256 bits of the product assembly { let mm := mulmod(x, y, not(0)) prod1 := sub(sub(mm, prod0), lt(mm, prod0)) } // Handle non-overflow cases, 256 by 256 division. if (prod1 == 0) { // Solidity will revert if denominator == 0, unlike the div opcode on its own. // The surrounding unchecked block does not change this fact. // See https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic. return prod0 / denominator; } // Make sure the result is less than 2^256. Also prevents denominator == 0. if (denominator <= prod1) { revert MathOverflowedMulDiv(); } //////////////////// // 512 by 256 division. //////////////////// // Make division exact by subtracting the remainder </pre>

Issue ID	156
Severity	🕒 Low
Status	Medium
Description Code	<pre> function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256 result) { unchecked { // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1, then use // use the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is stored in two 256 // variables such that product = prod1 * 2^256 + prod0. uint256 prod0 = x * y; // Least significant 256 bits of the product uint256 prod1; // Most significant 256 bits of the product assembly { let mm := mulmod(x, y, not(0)) prod1 := sub(sub(mm, prod0), lt(mm, prod0)) } // Handle non-overflow cases, 256 by 256 division. if (prod1 == 0) { // Solidity will revert if denominator == 0, unlike the div opcode on its own. // The surrounding unchecked block does not change this fact. // See https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic. return prod0 / denominator; } // Make sure the result is less than 2^256. Also prevents denominator == 0. if (denominator <= prod1) { revert MathOverflowedMulDiv(); } //////////////////// // 512 by 256 division. //////////////////// // Make division exact by subtracting the remainder </pre>


Issue ID	156
Severity	🕒 Low
Status	Medium
Description Code	<pre> function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256 result) { unchecked { // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1, then use // use the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is stored in two 256 // variables such that product = prod1 * 2^256 + prod0. uint256 prod0 = x * y; // Least significant 256 bits of the product uint256 prod1; // Most significant 256 bits of the product assembly { let mm := mulmod(x, y, not(0)) prod1 := sub(sub(mm, prod0), lt(mm, prod0)) } // Handle non-overflow cases, 256 by 256 division. if (prod1 == 0) { // Solidity will revert if denominator == 0, unlike the div opcode on its own. // The surrounding unchecked block does not change this fact. // See https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic. return prod0 / denominator; } // Make sure the result is less than 2^256. Also prevents denominator == 0. if (denominator <= prod1) { revert MathOverflowedMulDiv(); } //////////////////// // 512 by 256 division. //////////////////// // Make division exact by subtracting the remainder </pre>

Issue ID	156
Severity	🕒 Low
Status	Medium
Description Code	<pre> function mulDiv(uint256 x, uint256 y, uint256 denominator) internal pure returns (uint256 result) { unchecked { // 512-bit multiply [prod1 prod0] = x * y. Compute the product mod 2^256 and mod 2^256 - 1, then use // use the Chinese Remainder Theorem to reconstruct the 512 bit result. The result is stored in two 256 // variables such that product = prod1 * 2^256 + prod0. uint256 prod0 = x * y; // Least significant 256 bits of the product uint256 prod1; // Most significant 256 bits of the product assembly { let mm := mulmod(x, y, not(0)) prod1 := sub(sub(mm, prod0), lt(mm, prod0)) } // Handle non-overflow cases, 256 by 256 division. if (prod1 == 0) { // Solidity will revert if denominator == 0, unlike the div opcode on its own. // The surrounding unchecked block does not change this fact. // See https://docs.soliditylang.org/en/latest/control-structures.html#checked-or-unchecked-arithmetic. return prod0 / denominator; } // Make sure the result is less than 2^256. Also prevents denominator == 0. if (denominator <= prod1) { revert MathOverflowedMulDiv(); } //////////////////// // 512 by 256 division. //////////////////// // Make division exact by subtracting the remainder </pre>

Issue ID	184
Severity	🎯 Optimization
Status	High
Description Code	function pause() public onlyOwner { _pause(); }
Location	pause() should be declared external: - ForceFinanceCoin.pause() (ForceFinanceCoin.sol#20-22)

Issue ID	184
Severity	🎯 Optimization
Status	High
Description Code	function <code>unpause()</code> <code>public onlyOwner</code> { <code>_unpause()</code>; }
Location	<code>unpause()</code> should be declared external: - <code>ForceFinanceCoin.unpause()</code> (<code>ForceFinanceCoin.sol</code> #24-26)

Issue ID	184
Severity	🎯 Optimization
Status	High
Description Code	function mint (address to, uint256 amount) public onlyOwner { _mint(to, amount); }
Location	mint(address,uint256) should be declared external: - ForceFinanceCoin.mint(address,uint256) (ForceFinanceCoin.sol#28-30)

Issue ID	177
Severity	 Informational
Status	High
Description Code	<code>pragma solidity ^0.8.20;</code>
Location	Pragma version^0.8.20 (Ownable.sol#4) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.7



Issue ID	177
Severity	🎯 Informational
Status	High
Description Code	
Location	solc-0.8.20 is not recommended for deployment

