MythX

Analysis 4582e60a-d2f9-4b53-9664-8d1d7141a276 MythX

Started	Fri Jun 24 2022 12:54:22 GMT+0000 (Coordinated Universal Time)
Finished	Fri Jun 24 2022 13:09:34 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Client Tool	Mythx-Vscode-Extension
Main Source File	/Contracts/Netf.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	3

ISSUES

LOW

SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.8.4"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source file
/contracts/netf.sol
Locations

```
1 // SPDX-License-Identifier: BUSL-1.1
2 pragma solidity ^0.8.4;
3
4 import {IERC20} from "./utils/interfaces/IERC20.sol";
```

LOW

SWC-108

State variable visibility is not set.

It is best practice to set the visibility of state variables explicitly. The default visibility for "__gap" is internal. Other possible visibility settings are public and private.

Source file
/contracts/netf.sol
Locations

```
291 }
292
293 uint[50] __gap;
294
295 }
```

LOW

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

SWC-123

Source file

/contracts/netf.sol

Locations

```
141
142 // Ensure token has 18
143 require(IERC20(token).decimals() == 18, "402: Token must have 18 decimals");
144
145 //Transfer the token from the protocol
```

Source file

/contracts/netf.sol

Locations

```
10  import {OwnableUpgradeable} from "../node_modules/@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
11
12  contract networkETF is Initializable, ContextUpgradeable, OwnableUpgradeable, PausableUpgradeable {
13
14  event Deposit(address indexed user,uint timestamp, uint amount);
15  event Withdraw(address indexed user, uint timestamp, uint amount);
16  event CalibrateToken(address indexed user, uint timestamp, address indexed tokenAddress, uint amount);
17  event CalibrateMynt(address indexed user, uint timestamp, uint amount);
18  event SubmitToken(address indexed provider, uint timestamp, address indexed tokenAddress, uint amount);
19  event SubmitMynt(address indexed provider, uint timestamp, uint amount);
20
21  nETFStructs.nFundManager private fundManager;
22
23
24  function initialize(uint cyclePeriod, uint cycleLength) initializer public payable {
25
26  require(msg.value > 0, "400: Initialization requires a non-zero deposit");
27
28  fundManager.cyclePeriod = cyclePeriod;
29  fundManager.cycleLength = cycleLength;
30
31  //Set bonds for fund & user
32  fundManager.users[_msgSender()].deposit = fMathPool.from_base_to_60x18(msg.value);
33  fundManager.users[_msgSender()].lastUpdated = block.timestamp;
34
35  fundManager.myntDeposited = fMathPool.from_base_to_60x18(msg.value);
36
37  __Ownable_init();
38  }
39
40  function pause() public onlyOwner {
41  _pause();
42  }
43
44  function unpause() public onlyOwner {
45  _unpause();
46  }
47
48  function deposit() whenNotPaused() public payable returns(bool) {
49
50  require(msg.value > 0, "400: Invalid amount");
51
52  //Set bonds for fund & user
53
```

```
fundManager.users[_msgSender()].deposit = fMathUD60x18.add(fundManager.users[_msgSender()].deposit, fMathPool.from_base_to_60x18(msg.value));
fundManager.users[_msgSender()].lastUpdated = block.timestamp;
fundManager.myntDeposited = fMathUD60x18.add(fundManager.myntDeposited, fMathPool.from_base_to_60x18(msg.value));

//Log event
emit Deposit(_msgSender(), block.timestamp, msg.value);

return true;
}

function withdraw(uint amount_) whenNotPaused() public payable returns(bool){

require(amount_ > 0, "400: Invalid amount");
require(amount_ <= fMathPool.to_uint(fundManager.users[_msgSender()].deposit), "401: Insufficient amount");

//Set bonds for fund & user
fundManager.users[_msgSender()].deposit = fMathUD60x18.sub(fundManager.users[_msgSender()].deposit, fMathPool.from_base_to_60x18(amount_));
fundManager.users[_msgSender()].lastUpdated = block.timestamp;
fundManager.myntDeposited = fMathUD60x18.sub(fundManager.myntDeposited, fMathPool.from_base_to_60x18(amount_));

// Send funds to user
payable(_msgSender()).transfer(amount_);

//Log event
emit Withdraw(_msgSender(), block.timestamp, amount_);

return true;
}



function calibrateToken(address user, address token) whenNotPaused() public {

//Ensure withdrawable is true
(fMath.UD60x18 memory userWithdrawableAmount, bool isWithdrawable, string memory reason) = _getUserExpectedTokenCalibration( user, token);
require(isWithdrawable, reason);

// Check & update token data if necessary
// Update token data for cycle
uint tokenLastUpdated = fundManager.tokenLastUpdated[token];
//If cycle bonds have not been allocated for this token yet, set to 0
if (tokenLastUpdated < (block.timestamp - fundManager.cycleLength)) {
fundManager.tokenBondsUsed[token] = fMathUD60x18.fromUint(0);
}
fundManager.tokenLastUpdated[token] = block.timestamp;

IERC20(token).transfer(user, fMathPool.to_uint(userWithdrawableAmount) );

// Update user's time token allocated & token data
fundManager.users[user].timeTokenAllocated[token] = block.timestamp;
fundManager.tokenBondsUsed[token] = fMathUD60x18.add(fundManager.tokenBondsUsed[token], fundManager.users[user].deposit);
fundManager.tokenBalances[token] = fMathUD60x18.sub(fundManager.tokenBalances[token], userWithdrawableAmount);

//Log event
emit CalibrateToken(user, block.timestamp, token, fMathPool.to_uint(userWithdrawableAmount));

}

function calibrateMynt(address payable user) whenNotPaused() public payable {

//Ensure withdrawable is true
(fMath.UD60x18 memory userWithdrawableAmount, bool isWithdrawable, string memory reason) =_getUserExpectedMyntCalibration( user);
```

```
require(isWithdrawable, reason);

// Check & update MYNT data if necessary
// Update MYNT data for cycle
uint myntLastUpdated = fundManager.myntLastUpdated;
//If cycle bonds have not been allocated for this token yet, set to 0
if (myntLastUpdated < (block.timestamp - fundManager.cycleLength)) {
fundManager.myntBondsUsed = fMathUD60x18.fromUint(0);
}
fundManager.myntLastUpdated = block.timestamp;

//Return user the tokens based on existing bonds, and update their timeTokenAllocated
user.transfer(fMathPool.to_uint(userWithdrawableAmount));

//Update user's time MYNT allocated & MYNT data
fundManager.users[user].timeMyntAllocated = block.timestamp;
fundManager.myntBondsUsed = fMathUD60x18.add(fundManager.myntBondsUsed, fundManager.users[user].deposit);
fundManager.myntBalance = fMathUD60x18.sub(fundManager.myntBalance, userWithdrawableAmount);

//Log event
emit CalibrateMynt(user, block.timestamp, fMathPool.to_uint(userWithdrawableAmount));
}

function submitToken(address token, uint amount) whenNotPaused() public {

//Require amount > 0
require(amount > 0, "401: Amount must be greater than 0");

// Ensure token has 18
require(IERC20(token).decimals() == 18, "402: Token must have 18 decimals");

//Transfer the token from the protocol
IERC20(token).transferFrom(_msgSender(), address(this), amount);

//Check if tokenExists, if not add to total & give uint
if (!fundManager.tokenExists[token]) {
fundManager.tokenExists[token] = true;
fundManager.tokenNumberToAddress[fundManager.totalTokensAvailable]= token;
fundManager.totalTokensAvailable += 1;
}

//Update token data
fundManager.tokenBalances[token] = fMathUD60x18.add(fundManager.tokenBalances[token], fMathPool.from_base_to_60x18(amount));

//Log event
emit SubmitToken(_msgSender(), block.timestamp, token, amount);
}

function submitMynt() whenNotPaused() public payable {
//Require msg value > 0
require(msg.value > 0, "401: Amount must be greater than 0");

//Update MYNT data
fundManager.myntBalance = fMathUD60x18.add(fundManager.myntBalance, fMathPool.from_base_to_60x18(msg.value));

//Log event
emit SubmitMynt(_msgSender(), block.timestamp, msg.value);
}


////////////////////////////////////////////////////////////////////////////////
```

```
177
178 function getTotalTokens() public view returns (uint) {
179 return fundManager.totalTokensAvailable;
180 }
181
182 function getTokenAddress(uint tokenNumber) public view returns (address) {
183 return fundManager.tokenNumberToAddress[tokenNumber];
184 }
185
186 function getTotalMyntDeposit() public view returns (uint) {
187 return fMathPool.to_uint(fundManager.myntDeposited);
188 }
189
190 function getTokenBalance(address token) public view returns (uint) {
191 return fMathPool.to_uint(fundManager.tokenBalances[token]);
192 }
193
194 function getMyntBalance() public view returns (uint) {
195 return fMathPool.to_uint(fundManager.myntBalance);
196 }
197
198 function isCalibrationOpen() public view returns (bool) {
199 return (block.timestamp % fundManager.cyclePeriod) < fundManager.cycleLength;
200 }
201
202 function getUserData(address user) public view returns (uint, uint) {
203 return (fMathPool.to_uint(fundManager.users[user].deposit), fundManager.users[user].lastUpdated);
204 }
205
206 function getUserExpectedTokenCalibration(address user, address token) public view
207 returns (uint amount, bool canWithdraw, string memory reason) {
208 fMath.UD60x18 memory amount_ud60x18;
209 (amount_ud60x18, canWithdraw, reason) = _getUserExpectedTokenCalibration(user, token);
210 return (fMathPool.to_uint(amount_ud60x18), canWithdraw, reason);
211 }
212
213 function getUserExpectedMyntCalibration(address user) public view
214 returns (uint amount, bool canWithdraw, string memory reason) {
215 fMath.UD60x18 memory amount_ud60x18;
216 (amount_ud60x18, canWithdraw, reason) = _getUserExpectedMyntCalibration(user);
217 return (fMathPool.to_uint(amount_ud60x18), canWithdraw, reason);
218 }
219
220
221 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
222 function _getUserExpectedTokenCalibration(address user, address token) internal view
223 returns (
224 fMath.UD60x18 memory amount,
225 bool canWithdraw,
226 string memory reason
227 ) {
228
229 canWithdraw = true;
230 //Ensure calibration cycle is open
231 if (!isCalibrationOpen()) {
232 canWithdraw = false;
233 reason = "401: Calibration cycle is closed";
234 }
235 //Ensure user has deposited before the previous calibration cycle
236 if( fundManager.users[user].lastUpdated >(block.timestamp - fundManager.cyclePeriod)){
237 canWithdraw = false;
238
```

```
reason = "403: User has not deposited before the previous calibration cycle";
}

//Ensure time timeTokenAllocated is before the previous calibration length
if(fundManager.users[user].timeTokenAllocated[token] >
(block.timestamp - fundManager.cycleLength)){
canWithdraw = false;
reason = "402: User has allocated tokens before in this calibration cycle";
}

//Return user the tokens based on existing bonds, and update their timeTokenAllocated
fMath.UD60x18 memory userBonds = fundManager.users[user].deposit;
fMath.UD60x18 memory totalTokenBonds = fMathUD60x18.sub(fundManager.myntDeposited, fundManager.tokenBondsUsed[token] );
fMath.UD60x18 memory tokenAvailable = fundManager.tokenBalances[token];

amount = fMathUD60x18.div(fMathUD60x18.mul(userBonds, tokenAvailable), totalTokenBonds);

return (amount, canWithdraw, reason);

}

function _getUserExpectedMyntCalibration(address user) internal view
returns (
fMath.UD60x18 memory amount,
bool canWithdraw,
string memory reason
) {

canWithdraw = true;
//Ensure calibration cycle is open
if (!isCalibrationOpen()) {
canWithdraw = false;
reason = "401: Calibration cycle is closed";
}
//Ensure user has deposited before the previous calibration cycle
if( fundManager.users[user].lastUpdated >
(block.timestamp - fundManager.cyclePeriod)){
canWithdraw = false;
reason = "403: User has not deposited before the previous calibration cycle";
}
//Ensure time timeTokenAllocated is before the previous calibration length
if(fundManager.users[user].timeMyntAllocated >
(block.timestamp - fundManager.cycleLength)){
canWithdraw = false;
reason = "402: User has allocated MYNT before in this calibration cycle";
}

//Return user the tokens based on existing bonds, and update their timeTokenAllocated
fMath.UD60x18 memory userBonds = fundManager.users[user].deposit;
fMath.UD60x18 memory totalMYNTBonds = fMathUD60x18.sub(fundManager.myntDeposited, fundManager.myntBondsUsed);
fMath.UD60x18 memory myntAvailable = fundManager.myntBalance;

amount = fMathUD60x18.div(fMathUD60x18.mul(userBonds, myntAvailable), totalMYNTBonds);

return (amount, canWithdraw, reason);
}

uint[50] __gap;

}
```