April 19, 2024

## Express Audit Report for

# EraApe [EAPE]



De.Fi

# Project Summary

| | |
|---|---|
| Project Name | TokenERC20(EraApe ) |
| Address | 0x2e2992688ee4b2b7229118f2f4cfd9b8ab13c520 |
| Network | 137 |

De.Fi

| | |
|---|---|
| Issue ID | 183 |
| Severity | Optimization |
| Status | High |
| Description Code | **bytes32 private** _PERMIT_TYPEHASH_DEPRECATED_SLOT; |
| Location | ERC20PermitUpgradeable._PERMIT_TYPEHASH_DEPRECATED_SLOT (draft-ERC20PermitUpgradeable.sol#40) should be constant |

| | |
|---|---|
| Issue ID | 103 |
| Severity | Informational |
| Status | High |
| Description Code | **pragma solidity** ^0.8.0; |
| Location | Different versions of Solidity is used:<br>- Version used: ['^0.8.0', '^0.8.1', '^0.8.11', '^0.8.2']<br>- ^0.8.0 (AccessControlEnumerableUpgradeable.sol#4)<br>- ^0.8.0 (AccessControlUpgradeable.sol#4)<br>- ^0.8.0 (IAccessControlEnumerableUpgradeable.sol#4)<br>- ^0.8.0 (IAccessControlUpgradeable.sol#4)<br>- ^0.8.0 (IVotesUpgradeable.sol#3)<br>- ^0.8.2 (Initializable.sol#4)<br>- ^0.8.0 (ReentrancyGuardUpgradeable.sol#4)<br>- ^0.8.0 (ERC20Upgradeable.sol#4)<br>- ^0.8.0 (IERC20Upgradeable.sol#4)<br>- ^0.8.0 (ERC20BurnableUpgradeable.sol#4)<br>- ^0.8.0 (ERC20VotesUpgradeable.sol#4)<br>- ^0.8.0 (IERC20MetadataUpgradeable.sol#4)<br>- ^0.8.0 (draft-ERC20PermitUpgradeable.sol#4)<br>- ^0.8.0 (draft-IERC20PermitUpgradeable.sol#4)<br>- ^0.8.1 (AddressUpgradeable.sol#4)<br>- ^0.8.0 (ContextUpgradeable.sol#2-4)<br>- ^0.8.0 (CountersUpgradeable.sol#4)<br>- ^0.8.0 (MulticallUpgradeable.sol#4)<br>- ^0.8.0 (StringsUpgradeable.sol#4)<br>- ^0.8.0 (ECDSAUpgradeable.sol#4)<br>- ^0.8.0 (draft-EIP712Upgradeable.sol#4)<br>- ^0.8.0 (ERC165Upgradeable.sol#4)<br>- ^0.8.0 (IERC165Upgradeable.sol#4)<br>- ^0.8.0 (MathUpgradeable.sol#4)<br>- ^0.8.0 (SafeCastUpgradeable.sol#4)<br>- ^0.8.0 (EnumerableSetUpgradeable.sol#4)<br>- ^0.8.0 (IERC20.sol#2)<br>- ^0.8.0 (IPlatformFee.sol#2)<br>- ^0.8.0 (IPrimarySale.sol#2)<br>- ^0.8.11 (IThirdwebContract.sol#2)<br>- ^0.8.0 (IWETH.sol#2)<br>- ^0.8.11 (ITokenERC20.sol#2) |

| | |
|---|---|
| Issue ID | 156 |
| Severity | Low |
| Status | Medium |
| Description Code | |

```
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; // Least significant 256 bits of the product
uint256 prod1; // Most significant 256 bits of the product
assembly {
let mm := mulmod(x, y, not(0))
prod0 := mul(x, y)
prod1 := sub(sub(mm, prod0), lt(mm, prod0))
}
// Handle non-overflow cases, 256 by 256 division.
if (prod1 == 0) {
return prod0 / denominator;
}
// Make sure the result is less than 2^256. Also prevents denominator == 0.
require(denominator > prod1);
//////////////////////////////////////////////
// 512 by 256 division.
//////////////////////////////////////////////
// Make division exact by subtracting the remainder from [prod1 prod0].
uint256 remainder;
assembly {
// Compute remainder using mulmod.
remainder := mulmod(x, y, denominator)
// Subtract 256 bit number from 512 bit number.
```

| | |
|---|---|
| Issue ID | 156 |
| Severity | Low |
| Status | Medium |
| Description Code | |

```
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; // Least significant 256 bits of the product
uint256 prod1; // Most significant 256 bits of the product
assembly {
let mm := mulmod(x, y, not(0))
prod0 := mul(x, y)
prod1 := sub(sub(mm, prod0), lt(mm, prod0))
}
// Handle non-overflow cases, 256 by 256 division.
if (prod1 == 0) {
return prod0 / denominator;
}
// Make sure the result is less than 2^256. Also prevents denominator == 0.
require(denominator > prod1);
//////////////////////////////////////////////
// 512 by 256 division.
//////////////////////////////////////////////
// Make division exact by subtracting the remainder from [prod1 prod0].
uint256 remainder;
assembly {
// Compute remainder using mulmod.
remainder := mulmod(x, y, denominator)
// Subtract 256 bit number from 512 bit number.
```

| | |
|---|---|
| Issue ID | 156 |
| Severity | Low |
| Status | Medium |
| Description Code | |

```
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; // Least significant 256 bits of the product
uint256 prod1; // Most significant 256 bits of the product
assembly {
let mm := mulmod(x, y, not(0))
prod0 := mul(x, y)
prod1 := sub(sub(mm, prod0), lt(mm, prod0))
}
// Handle non-overflow cases, 256 by 256 division.
if (prod1 == 0) {
return prod0 / denominator;
}
// Make sure the result is less than 2^256. Also prevents denominator == 0.
require(denominator > prod1);
//////////////////////////////////////////////
// 512 by 256 division.
//////////////////////////////////////////////
// Make division exact by subtracting the remainder from [prod1 prod0].
uint256 remainder;
assembly {
// Compute remainder using mulmod.
remainder := mulmod(x, y, denominator)
// Subtract 256 bit number from 512 bit number.
```

| | |
|---|---|
| Issue ID | 156 |
| Severity | Low |
| Status | Medium |
| Description Code | |

```
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; // Least significant 256 bits of the product
uint256 prod1; // Most significant 256 bits of the product
assembly {
let mm := mulmod(x, y, not(0))
prod0 := mul(x, y)
prod1 := sub(sub(mm, prod0), lt(mm, prod0))
}
// Handle non-overflow cases, 256 by 256 division.
if (prod1 == 0) {
return prod0 / denominator;
}
// Make sure the result is less than 2^256. Also prevents denominator == 0.
require(denominator > prod1);
//////////////////////////////////////////////
// 512 by 256 division.
//////////////////////////////////////////////
// Make division exact by subtracting the remainder from [prod1 prod0].
uint256 remainder;
assembly {
// Compute remainder using mulmod.
remainder := mulmod(x, y, denominator)
// Subtract 256 bit number from 512 bit number.
```

| | |
|---|---|
| Issue ID | 156 |
| Severity | Low |
| Status | Medium |
| Description Code | |

```
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; // Least significant 256 bits of the product
uint256 prod1; // Most significant 256 bits of the product
assembly {
let mm := mulmod(x, y, not(0))
prod0 := mul(x, y)
prod1 := sub(sub(mm, prod0), lt(mm, prod0))
}
// Handle non-overflow cases, 256 by 256 division.
if (prod1 == 0) {
return prod0 / denominator;
}
// Make sure the result is less than 2^256. Also prevents denominator == 0.
require(denominator > prod1);
//////////////////////////////////////////////
// 512 by 256 division.
//////////////////////////////////////////////
// Make division exact by subtracting the remainder from [prod1 prod0].
uint256 remainder;
assembly {
// Compute remainder using mulmod.
remainder := mulmod(x, y, denominator)
// Subtract 256 bit number from 512 bit number.
```

| | |
|---|---|
| Issue ID | 156 |
| Severity | Low |
| Status | Medium |
| Description Code | |

```
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; // Least significant 256 bits of the product
uint256 prod1; // Most significant 256 bits of the product
assembly {
let mm := mulmod(x, y, not(0))
prod0 := mul(x, y)
prod1 := sub(sub(mm, prod0), lt(mm, prod0))
}
// Handle non-overflow cases, 256 by 256 division.
if (prod1 == 0) {
return prod0 / denominator;
}
// Make sure the result is less than 2^256. Also prevents denominator == 0.
require(denominator > prod1);
//////////////////////////////////////////////
// 512 by 256 division.
//////////////////////////////////////////////
// Make division exact by subtracting the remainder from [prod1 prod0].
uint256 remainder;
assembly {
// Compute remainder using mulmod.
remainder := mulmod(x, y, denominator)
// Subtract 256 bit number from 512 bit number.
```

| | |
|---|---|
| Issue ID | 156 |
| Severity | Low |
| Status | Medium |
| Description Code | |

```
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; // Least significant 256 bits of the product
uint256 prod1; // Most significant 256 bits of the product
assembly {
let mm := mulmod(x, y, not(0))
prod0 := mul(x, y)
prod1 := sub(sub(mm, prod0), lt(mm, prod0))
}
// Handle non-overflow cases, 256 by 256 division.
if (prod1 == 0) {
return prod0 / denominator;
}
// Make sure the result is less than 2^256. Also prevents denominator == 0.
require(denominator > prod1);
//////////////////////////////////////////////
// 512 by 256 division.
//////////////////////////////////////////////
// Make division exact by subtracting the remainder from [prod1 prod0].
uint256 remainder;
assembly {
// Compute remainder using mulmod.
remainder := mulmod(x, y, denominator)
// Subtract 256 bit number from 512 bit number.
```

| | |
|---|---|
| Issue ID | 156 |
| Severity | Low |
| Status | Medium |
| Description Code | |

```
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; // Least significant 256 bits of the product
uint256 prod1; // Most significant 256 bits of the product
assembly {
let mm := mulmod(x, y, not(0))
prod0 := mul(x, y)
prod1 := sub(sub(mm, prod0), lt(mm, prod0))
}
// Handle non-overflow cases, 256 by 256 division.
if (prod1 == 0) {
return prod0 / denominator;
}
// Make sure the result is less than 2^256. Also prevents denominator == 0.
require(denominator > prod1);
//////////////////////////////////////////////
// 512 by 256 division.
//////////////////////////////////////////////
// Make division exact by subtracting the remainder from [prod1 prod0].
uint256 remainder;
assembly {
// Compute remainder using mulmod.
remainder := mulmod(x, y, denominator)
// Subtract 256 bit number from 512 bit number.
```

De.Fi

| | |
|---|---|
| Issue ID | 177 |
| Severity | Informational |
| Status | High |
| Description Code | **pragma solidity** ^0.8.0; |
| Location | Pragma version^0.8.0 (AccessControlEnumerableUpgradeable.sol#4) allows old versions |

De.Fi

| | |
|---|---|
| Issue ID | 177 |
| Severity | Informational |
| Status | High |
| Description Code | **pragma solidity** ^0.8.2; |
| Location | Pragma version^0.8.2 (Initializable.sol#4) allows old versions |

De.Fi

| | |
|---|---|
| Issue ID | 177 |
| Severity | Informational |
| Status | High |
| Description Code | **pragma solidity** ^0.8.1; |
| Location | Pragma version^0.8.1 (AddressUpgradeable.sol#4) allows old versions |

De.Fi

| | |
|---|---|
| Issue ID | 177 |
| Severity | Informational |
| Status | High |
| Description Code | **pragma solidity** ^0.8.11; |
| Location | Pragma version^0.8.11 (IThirdwebContract.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6/0.8.7 |

De.Fi

| | |
|---|---|
| Issue ID | 177 |
| Severity | Informational |
| Status | High |
| Description Code | |
| Location | solc-0.8.12 is not recommended for deployment |

De.Fi

| | |
|---|---|
| Issue ID | 173 |
| Severity | Informational |
| Status | High |
| Description Code | **function sendValue**(**address payable** recipient, **uint256** amount) **internal** {<br>require(**address**(this).balance >= amount, "Address: insufficient balance");<br>(**bool** success, ) = recipient.call{value: amount}("");<br>require(success, "Address: unable to send value, recipient may have reverted");<br>} |
| Location | Low level call in AddressUpgradeable.sendValue(address,uint256) (AddressUpgradeable.sol#60-65):<br>- (success) = recipient.call{value: amount}() (AddressUpgradeable.sol#63) |

De.Fi

| | |
|---|---|
| Issue ID | 173 |
| Severity | Informational |
| Status | High |
| Description Code | **function functionCallWithValue**(<br>**address** target,<br>**bytes memory** data,<br>**uint256** value,<br>**string memory** errorMessage<br>) **internal returns** (**bytes memory**) {<br>require(**address**(this).balance >= value, "Address: insufficient balance for call");<br>require(isContract(target), "Address: call to non-contract");<br>(**bool** success, **bytes memory** returndata) = target.call{value: value}(data);<br>**return** verifyCallResult(success, returndata, errorMessage);<br>} |
| Location | Low level call in AddressUpgradeable.functionCallWithValue(address,bytes,uint256,string) (AddressUpgradeable.sol#128-139):<br>- (success,returndata) = target.call{value: value}(data) (AddressUpgradeable.sol#137) |

De.Fi

| | |
|---|---|
| Issue ID | 173 |
| Severity | Informational |
| Status | High |
| Description Code | **function functionStaticCall**(<br>**address** target,<br>**bytes memory** data,<br>**string memory** errorMessage<br>) **internal view returns** (**bytes memory**) {<br>require(isContract(target), "Address: static call to non-contract");<br>(**bool** success, **bytes memory** returndata) = target.staticcall(data);<br>**return** verifyCallResult(success, returndata, errorMessage);<br>} |
| Location | Low level call in AddressUpgradeable.functionStaticCall(address,bytes,string) (AddressUpgradeable.sol#157-166):<br>- (success,returndata) = target.staticcall(data) (AddressUpgradeable.sol#164) |

De.Fi

| | |
|---|---|
| Issue ID | 173 |
| Severity | Informational |
| Status | High |
| Description Code | **function** _functionDelegateCall(**address** target, **bytes memory** data) **private returns** (**bytes memory**) {<br>require(AddressUpgradeable.isContract(target), "Address: delegate call to non-contract");<br>// solhint-disable-next-line avoid-low-level-calls<br>(**bool** success, **bytes memory** returndata) = target.delegatecall(data);<br>**return** AddressUpgradeable.verifyCallResult(success, returndata, "Address: low-level delegate call failed");<br>} |
| Location | Low level call in MulticallUpgradeable._functionDelegateCall(address,bytes) (MulticallUpgradeable.sol#37-43):<br>- (success,returndata) = target.delegatecall(data) (MulticallUpgradeable.sol#41) |

De.Fi

| | |
|---|---|
| Issue ID | 173 |
| Severity | Informational |
| Status | High |
| Description Code | **function sendValue**(**address payable** recipient, **uint256** amount) **internal** {<br>require(**address**(this).balance >= amount, "Address: insufficient balance");<br>(**bool** success, ) = recipient.call{ value: amount }("");<br>require(success, "Address: unable to send value, recipient may have reverted");<br>} |
| Location | Low level call in TWAddress.sendValue(address,uint256) (TWAddress.sol#60-65):<br>- (success) = recipient.call{value: amount}() (TWAddress.sol#63) |

De.Fi

| | |
|---|---|
| Issue ID | 173 |
| Severity | Informational |
| Status | High |
| Description Code | **function functionCallWithValue**(<br>**address** target,<br>**bytes memory** data,<br>**uint256** value,<br>**string memory** errorMessage<br>) **internal returns** (**bytes memory**) {<br>require(**address**(this).balance >= value, "Address: insufficient balance for call");<br>require(isContract(target), "Address: call to non-contract");<br>(**bool** success, **bytes memory** returndata) = target.call{ value: value }(data);<br>**return** verifyCallResult(success, returndata, errorMessage);<br>} |
| Location | Low level call in TWAddress.functionCallWithValue(address,bytes,uint256,string) (TWAddress.sol#128-139):<br>- (success,returndata) = target.call{value: value}(data) (TWAddress.sol#137) |

De.Fi

| | |
|---|---|
| Issue ID | 173 |
| Severity | Informational |
| Status | High |
| Description Code | **function functionStaticCall**(<br>**address** target,<br>**bytes memory** data,<br>**string memory** errorMessage<br>) **internal view returns** (**bytes memory**) {<br>require(isContract(target), "Address: static call to non-contract");<br>(**bool** success, **bytes memory** returndata) = target.staticcall(data);<br>**return** verifyCallResult(success, returndata, errorMessage);<br>} |
| Location | Low level call in TWAddress.functionStaticCall(address,bytes,string) (TWAddress.sol#157-166):<br>- (success,returndata) = target.staticcall(data) (TWAddress.sol#164) |

De.Fi

| | |
|---|---|
| Issue ID | 173 |
| Severity | Informational |
| Status | High |
| Description Code | **function functionDelegateCall**(<br>**address** target,<br>**bytes memory** data,<br>**string memory** errorMessage<br>) **internal returns** (**bytes memory**) {<br>require(isContract(target), "Address: delegate call to non-contract");<br>(**bool** success, **bytes memory** returndata) = target.delegatecall(data);<br>**return** verifyCallResult(success, returndata, errorMessage);<br>} |
| Location | Low level call in TWAddress.functionDelegateCall(address,bytes,string) (TWAddress.sol#184-193):<br>- (success,returndata) = target.delegatecall(data) (TWAddress.sol#191) |

De.Fi

| | |
|---|---|
| Issue ID | 193 |
| Severity | Critical |
| Status | Medium |
| Description Code | |
| Location | Contract: TokenERC20 have proxy upgradeability |

De.Fi

| | |
|---|---|
| Issue ID | 186 |
| Severity | Critical |
| Status | Medium |
| Description Code | **function mintTo**(**address** to, **uint256** amount) **public virtual** {<br>require(hasRole(MINTER_ROLE, _msgSender()), "not minter.");<br>_mintTo(to, amount);<br>} |
| Location | Mint function: TokenERC20.mintTo(address,uint256) (TokenERC20.sol#165-168)<br>- in internal call: _mintTo(to,amount)<br>- In expression: _balances[account] += amount |

De.Fi

| | |
|---|---|
| Issue ID | 167-a |
| Severity | Low |
| Status | Medium |
| Description Code | function initialize(<br>address _defaultAdmin,<br>string memory _name,<br>string memory _symbol,<br>string memory _contractURI,<br>address[] memory _trustedForwarders,<br>address _primarySaleRecipient,<br>address _platformFeeRecipient,<br>uint256 _platformFeeBps<br>) external initializer {<br>__ReentrancyGuard_init();<br>__ERC2771Context_init_unchained(_trustedForwarders);<br>__ERC20Permit_init(_name);<br>__ERC20_init_unchained(_name, _symbol);<br>contractURI = _contractURI;<br>primarySaleRecipient = _primarySaleRecipient;<br>platformFeeRecipient = _platformFeeRecipient;<br>require(_platformFeeBps <= MAX_BPS, "exceeds MAX_BPS");<br>platformFeeBps = uint128(_platformFeeBps);<br>_setupRole(DEFAULT_ADMIN_ROLE, _defaultAdmin);<br>_setupRole(TRANSFER_ROLE, _defaultAdmin);<br>_setupRole(MINTER_ROLE, _defaultAdmin);<br>_setupRole(TRANSFER_ROLE, address(0));<br>} |
| Location | TokenERC20.initialize(address,string,string,string,address[],address,address,uint256) (TokenERC20.sol#89-115) should emit an event for:<br>- platformFeeBps = uint128(_platformFeeBps) (TokenERC20.sol#109) |

De.Fi

| | |
|---|---|
| Issue ID | 168 |
| Severity | Low |
| Status | Medium |
| Description Code | **address** _primarySaleRecipient, |
| Location | TokenERC20.initialize(address,string,string,string,address[],address,address,uint256)._primarySaleRecipient (TokenERC20.sol#95) lacks a zero-check on :<br>- primarySaleRecipient = _primarySaleRecipient (TokenERC20.sol#105) |

De.Fi

| | |
|---|---|
| Issue ID | 168 |
| Severity | Low |
| Status | Medium |
| Description Code | **address** _platformFeeRecipient, |
| Location | TokenERC20.initialize(address,string,string,string,address[],address,address,uint256)._platformFeeRecipient (TokenERC20.sol#96) lacks a zero-check on :<br>- platformFeeRecipient = _platformFeeRecipient (TokenERC20.sol#106) |

De.Fi

| | |
|---|---|
| Issue ID | 168 |
| Severity | Low |
| Status | Medium |
| Description Code | **function setPrimarySaleRecipient**(**address** _saleRecipient) **external onlyRole**(DEFAULT_ADMIN_ROLE) { |
| Location | TokenERC20.setPrimarySaleRecipient(address)._saleRecipient (TokenERC20.sol#189) lacks a zero-check on :<br>- primarySaleRecipient = _saleRecipient (TokenERC20.sol#190) |

De.Fi

| | |
|---|---|
| Issue ID | 168 |
| Severity | Low |
| Status | Medium |
| Description Code | **function setPlatformFeeInfo**(**address** _platformFeeRecipient, **uint256** _platformFeeBps) |
| Location | TokenERC20.setPlatformFeeInfo(address,uint256)._platformFeeRecipient (TokenERC20.sol#195) lacks a zero-check on :<br>- platformFeeRecipient = _platformFeeRecipient (TokenERC20.sol#202) |

De.Fi

| | |
|---|---|
| Issue ID | 113 |
| Severity | Low |
| Status | Medium |
| Description Code | **function** _functionDelegateCall(**address** target, **bytes memory** data) **private returns** (**bytes memory**) {<br>require(AddressUpgradeable.isContract(target), "Address: delegate call to non-contract");<br>// solhint-disable-next-line avoid-low-level-calls<br>(**bool** success, **bytes memory** returndata) = target.delegatecall(data);<br>**return** AddressUpgradeable.verifyCallResult(success, returndata, "Address: low-level delegate call failed");<br>} |
| Location | MulticallUpgradeable._functionDelegateCall(address,bytes) (MulticallUpgradeable.sol#37-43) has external calls inside a loop: (success,returndata) = target.delegatecall(data) (MulticallUpgradeable.sol#41) |

De.Fi

| | |
|---|---|
| Issue ID | 160 |
| Severity | Informational |
| Status | Medium |
| Description Code | (**uint256** oldWeight, **uint256** newWeight) = _writeCheckpoint(_checkpoints[dst], _add, amount); |
| Location | ERC20VotesUpgradeable._moveVotingPower(address,address,uint256).oldWeight_scope_0 (ERC20VotesUpgradeable.sol#226) is a local variable never initialized |

De.Fi

| | |
|---|---|
| Issue ID | 160 |
| Severity | Informational |
| Status | Medium |
| Description Code | (**uint256** oldWeight, **uint256** newWeight) = _writeCheckpoint(_checkpoints[dst], _add, amount); |
| Location | ERC20VotesUpgradeable._moveVotingPower(address,address,uint256).newWeight_scope_1 (ERC20VotesUpgradeable.sol#226) is a local variable never initialized |