hexens

ROCIFI SMART CONTRACT AUDIT REPORT

08.12.2022

CONTENTS

CONTENTS

SUMMARY

SEVERITY	NUMBER OF FINDINGS
CRITICAL	2
HIGH	4
MEDIUM	4
LOW	2
INFORMATIONAL	4

TOTAL: 16

SCOPE

The analyzed resources are located on:
https://github.com/RociFi/cydonia/commit/c2c4e23fbf87e4ec7426b083e3c1003a8f4640c2

The issues described in the report were fixed in the following commit:
https://github.com/RociFi/cydonia/commit/b0c910feb2e8de8ec431186fa78ab0f5117beac4

WEAKNESSES

This section contains the list of discovered weaknesses.

1. LOAN INTEREST PAYMENT BYPASS

SEVERITY: Critical

PATH: LoanManager.sol:L298-360

REMEDIATION: implement a withdrawal request system or don't allow a user to deposit and withdraw in the same block to protect against flash loans. Ideally the generated interest would be divided over the people who actually had a stake in the pool during the loan

STATUS: fixed

DESCRIPTION:

In **LoanManager.sol:repay** a user can repay the borrowed amount and generated interest of their loan. The interest is based on the duration and APR. The interest is also added into a pool's value with **loan.pool.updatePoolValue**. The pool's value determines the worth of a pool's shares, which are minted/burned upon depositing/withdrawing underlying tokens. Depositing and withdrawing can be done in a single transaction.

A user with a loan that has generated some interest can take out a big flash loan of the underlying token, deposit this into the loan's pool, repay the loan, withdraw the underlying token and pay back the flash loan. By repaying the loan, the user's shares would have increased in value to be almost worth the paid interest.

```
function repay(
  uint256 loanId,
  uint256 amount,
  string memory version
) external ifNotPaused nonReentrant checkVersion(version) {
  uint256 interestAccrued = getInterest(loanId, block.timestamp);
  [..]
  loan.pool.updatePoolValue(int256(interestAccrued - treasuryShare));
  [..]
}
function deposit(uint256 underlyingTokenAmount, string memory version)
  external
  checkVersion(version)
  ifNotPaused
  nonReentrant
{
  uint256 rTokenAmount = stablecoinToRToken(underlyingTokenAmount);
  poolValue += underlyingTokenAmount;
  _mint(msg.sender, rTokenAmount);
  underlyingToken.safeTransferFrom(msg.sender, address(this), underlyingTokenAmount);
  emit LiquidityDeposited(block.timestamp, msg.sender, underlyingTokenAmount, rTokenAmount);
}

function withdraw(uint256 rTokenAmount, string memory version)
  external
  checkVersion(version)
  ifNotPaused
  nonReentrant
{
  uint256 underlyingTokenAmount = rTokenToStablecoin(rTokenAmount);
  poolValue -= underlyingTokenAmount;
  _burn(msg.sender, rTokenAmount);
  underlyingToken.safeTransfer(msg.sender, underlyingTokenAmount);
  emit LiquidityWithdrawn(block.timestamp, msg.sender, underlyingTokenAmount, rTokenAmount);
}

function updatePoolValue(int256 value) external whenNotPaused onlyRole(Roles.LOAN_MANAGER) {
  int256 newPoolValue = int256(poolValue) + value;
  require(newPoolValue >= 0, Errors.POOL_VALUE_LT_ZERO);
  emit PoolValueUpdated(msg.sender, poolValue, uint256(newPoolValue), block.timestamp);
  poolValue = uint256(newPoolValue);
}
```

2. PARTIAL LIQUIDATION LEADS TO COLLATERAL-FREE LOANS

SEVERITY: Critical

PATH: LoanManager.sol:liquidate:L362-438

REMEDIATION: LoanManager.sol:liquidate (L362-438) should update the loan.frozenCollateralAmount when seizing the frozen collateral to correctly reflect the remaining frozen collateral

STATUS: fixed

DESCRIPTION:

In **LoanManager.sol:liquidate** both full and partial liquidations of loans are possible. The Loan Manager will seize the frozen collateral of the borrower for the liquidated amount. If it is a partial liquidation, the loan amount get updated with the remaining amount and the borrower can still continue repaying the loan afterwards.

However, a partial liquidation seizes the frozen collateral of the borrower and when the borrower fully repays the loan, they will get the entire original collateral amount unfrozen. This is a miscalculation.

For example:

1. User A loans 100 X underlying for 100 Y collateral.
2. User A defaults on the loan.
3. Now 100 Y is only worth 90 X, so the liquidation is partial.
4. This results in the loan manager seizing 100 Y and the loan having a remaining amount of 10 X.
5. The borrower starts a new loan for 90 X underlying and 100 Y collateral.
6. The borrower repays the old loan for 10 X.
7. This results in the Loan Manager unfreezing 100 Y for the borrower, making the second loan collateral-free.
8. The borrower nets 180 X underlying for only 100 Y collateral.

```
function liquidate(uint256 loanId, string memory version)
  external
  whenNotPaused
  nonReentrant
  checkVersion(version)
  onlyRole(Roles.LIQUIDATOR)
  returns (
    IERC20MetadataUpgradeable,
    IERC20MetadataUpgradeable,
    uint256,
    IPool
  )
{
  LoanLib.DelinquencyInfo memory info = getDelinquencyInfo(loanId);
  [..]
  loan.amount = info.notCovered;

  collateralManager.seize(
    msg.sender,
    loan.frozenCollateralToken,
    loan.borrower,
    info.toLiquidate
  );
  [..]
  LoanLib.Action statusAction = LoanLib.Action.LIQUIDATION_COVERED;

  if (loan.amount > 0) {
    statusAction = LoanLib.Action.LIQUIDATION_NOT_COVERED;
    loan.lastRepay = block.timestamp;
  }
  [..]
}
```

3. SIGNATURE REPLAY TO UNSET BAD SCORING OR BLOCK USER BORROWS

SEVERITY: High

PATH: ScoreDB.sol:updateScore:L118-127

REMEDIATION: implement a domain separator and nonce (in other words to fully comply with EIP712) in order to prevent replay attacks

STATUS: fixed

DESCRIPTION:

In the function **ScoreDB.sol:updateScore** a score message signature is first verified and then stored for the corresponding **NFCS** token **ID**. However, because there is no nonce check implemented, so an illicit actor can replay an older signature and consequently a better score back for themselves and create a new loan if the timestamp is still in the valid period. This issue can also be exploited by blocking other users from borrowing replaying their score message signatures and consequently setting their token **ID** score to an older one with an expired timestamp, effectively causing **getCreditScoreAndValidate** to always revert.

```
modifier verify(
  uint256 nfcsId,
  uint16 score,
  uint256 timestamp,
  bytes memory sig
) {
  require(score >= minScore && score <= maxScore,
Errors.SCORE_DB_UNKNOW_FETCHING_SCORE);
  // Recreate msg hash from inputs
  bytes32 hash = keccak256(abi.encodePacked(nfcsId, score, timestamp));
  require(
    hash.toEthSignedMessageHash().recover(sig) == nfcsSignerAddress,
    Errors.SCORE_DB_VERIFICATION
  );
  _;
}
function updateScore(
  uint256 nfcsId,
  uint16 score,
  uint256 timestamp,
  bytes memory sig,
  string memory version
) external whenNotPaused checkVersion(version) verify(nfcsId, score, timestamp, sig)
{
  nfcsIdToScore[nfcsId] = Score(timestamp, nfcsId, score);
  emit ScoreUpdated(block.timestamp, nfcsId, score);
}
```

4. ALLOWED COLLATERAL REMOVAL COULD LOCK FUNDS FOR A USER

SEVERITY: High

PATH: CollateralManager.sol:claimCollateral:L253-278

REMEDIATION: remove the check against the allowed collateral tokens list in CollateralManager.sol:claimCollateral so a user will always be able to withdraw their collateral

STATUS: fixed

DESCRIPTION:

In **CollateralManager.sol:claimCollateral** a user is able to withdraw their deposited collateral. The function checks the collateral token against an allowed tokens list. However, if a user still has a balance of that collateral token or has an active loan in the collateral token that is about to be removed then the user will be unable to withdraw the collateral funds.

This is both because of the check against the allowed tokens list and because the **CollateralManager.sol:removeCollaterals** (L164-170) function does not check whether there are any balances of the token or any active loans against the token.

On the other hand, both **CollateralManager.sol:unfreeze** (L200-213) and **CollateralManager.sol:seize** (L287-310) allow a token that may not be on the allowed tokens list, which is correct because there might be an active loan with such a token.

```
function claimCollateral(
  address user,
  IERC20MetadataUpgradeable token,
  uint256 amount
)
  external
  checkAmount(amount)
  checkFreezerOrUser(user)
  checkCollateralBalance(token, user, amount)
  ifNotPaused
{
  require(allowedCollaterals.includes[token],
Errors.COLLATERAL_MANAGER_TOKEN_NOT_SUPPORTED);
  [..]
}
```

5. BORROWER CAN LOSE OVERPAID ETH COLLATERAL

SEVERITY: High

PATH: LoanManager.sol:borrow:L222-290

REMEDIATION: replace collateralToAdd with msg.value in LoanManager.sol:borrow:L257 so that any overpaid ETH will get deposited into the user's collateral balance

we also recommend to add a check to see if the ETH is sufficient, for example:

require(msg.value >= collateralToAdd, "...");

STATUS: fixed

DESCRIPTION:

In **LoanManager.sol:borrow** the required amount of collateral is calculated with regard to the desired **borrow** amount (L242). If the user's collateral in the Collateral Manager is insufficient, the Loan Manager will try to transfer in the missing amount and add this to the **Collateral Manager** for the user (L251-261).

However, if the collateral is **ETH**, then the amount added is equal to the required collateral amount instead of **msg.value** (L257). If the user sends too much **ETH** (either due to fluctuating prices or a mistake), then the user loses the overpaid **ETH**.

```
function borrow(
  uint256 amount,
  IPool pool,
  IERC20MetadataUpgradeable collateral,
  uint256 ltv,
  uint256 duration,
  string memory version
) public payable ifNotPaused nonReentrant checkVersion(version) {
  ]..]
  uint256 collateralToFreeze = priceFeed.convert(
    (amount * 100 ether) / ltv,
    pool.underlyingToken(),
    collateral
  );
  uint256 userCollateralBalance = collateralManager.collateralToUserToAmount(
    collateral,
    msg.sender
  );
  uint256 collateralToAdd = userCollateralBalance >= collateralToFreeze
    ? 0
    : collateralToFreeze - userCollateralBalance;

  if (collateralToAdd > 0) {
    if (msg.value > 0) {
      collateralManager.addCollateral{value: collateralToAdd}(msg.sender, collateral, 0);
    } else {
      collateralManager.addCollateral(msg.sender, collateral, collateralToAdd);
    }
  }
  [..]
}
```

6. COLLATERAL WITHDRAWAL TO INCORRECT ADDRESS

SEVERITY: High

PATH: CollateralManager.Sol:claimCollateral:L253-278

REMEDIATION: change msg.sender to user on L271

STATUS: fixed

DESCRIPTION:

In **CollateralManager.Sol:claimCollateral** a user or loan manager can withdraw deposited collateral funds. However, if the collateral token is **WETH**, then the unwrapped **ETH** is sent to msg.sender instead of user. Any other token is sent to the user that has their balance lowered.

```
function claimCollateral(
  address user,
  IERC20MetadataUpgradeable token,
  uint256 amount
)
  external
  checkAmount(amount)
  checkFreezerOrUser(user)
  checkCollateralBalance(token, user, amount)
  ifNotPaused
{
  require(allowedCollaterals.includes[token],
Errors.COLLATERAL_MANAGER_TOKEN_NOT_SUPPORTED);

  collateralToUserToAmount[token][user] -= amount;

  // if token is wrapped native - unwrap it
  if (token == IERC20MetadataUpgradeable(address(wrapper))) {
      wrapper.withdraw(amount);
      (bool success, ) = payable(msg.sender).call{value: amount}("");
      require(success, Errors.COLLATERAL_MANAGER_NATIVE_TRANSFER);
  } else {
      token.safeTransfer(user, amount);
  }

  emit CollateralClaimed(user, token, amount);
}
```

7. SIGNATURE AND HASHING DOES NOT COMPLY WITH EIP712

SEVERITY: Medium

PATH: NFCS.sol:verifyAdd, ScoreDB.sol:verify

REMEDIATION: add a domain separator (in other words to fully comply with the EIP712 standard) in order to ensure that signatures from other contexts can not be replayed

STATUS: fixed

DESCRIPTION:

In the function **NFCS.sol:verifyAdd** the primary address and bundle signatures have no domain separator. Given that the message to be signed is just the address, chances to find a replayable signature are higher than in other, more specific cases.

A similar flaw is present in the **ScoreDB.sol:verify** function, where the impact is greater and described in Weakness 3.

```
function verifyAdd(
  address[] memory bundle,
  bytes[] memory signatures,
  uint256 tokenId
) internal {
  address primaryAddress = bundle[0];

  bytes memory primaryAddressBytes = abi.encodePacked(primaryAddress);

  bytes memory bundlePacked = primaryAddressBytes;

  for (uint256 i = 1; i < bundle.length; i++) {
    require(secondaryToPrimary[bundle[i]] == address(0), Errors.NFCS_ADDRESS_BUNDLED);

    require(
      primaryAddressBytes.toEthSignedMessageHash().recover(signatures[i]) == bundle[i],
      Errors.NFCS_WALLET_VERIFICATION_FAILED
    );

    secondaryToPrimary[bundle[i]] = primaryAddress;

    _tokenBundle[tokenId].push(bundle[i]);

    bundlePacked = abi.encodePacked(bundlePacked, bundle[i]);
  }

  require(
    bundlePacked.toEthSignedMessageHash().recover(signatures[0]) == primaryAddress,
    Errors.NFCS_WALLET_VERIFICATION_FAILED
  );
}
```

8. COLLATERAL MANAGER ALLOWS FREEZING OF UNSUPPORTED COLLATERAL

SEVERITY: Medium

PATH: CollateralManager.sol:freeze:L178-192

REMEDIATION: check that the collateral token is in the allowed tokens list

For example:

require(allowedCollaterals.includes[token], Errors.COLLATERAL_MANAGER_TOKEN_NOT_SUPPORTED);

STATUS: fixed

DESCRIPTION:

In **CollateralManager.sol:freeze** the Loan Manager can freeze a user's collateral when creating a loan. However, this function does not check the collateral token against the allowed tokens list. A user could therefore potentially create a loan against unsupported collateral that was previously allowed and deposited by the user.

```
function freeze(
  address user,
  IERC20MetadataUpgradeable token,
  uint256 amount
)
  external
  checkAmount(amount)
  checkCollateralBalance(token, user, amount)
  onlyRole(Roles.LOAN_MANAGER)
{
  collateralToUserToAmount[token][user] -= amount;
  collateralToFreezerToUserToAmount[token][msg.sender][user] +=
amount;

  emit CollateralFrozen(user, msg.sender, token, amount);
}
```

9. BUNDLE NONCE IS SET INCORRECTLY

SEVERITY: Medium

PATH: PATH: NFCS.sol:mintToken,addAddressToBundle

REMEDIATION: in the function mintToken a check should be added to validate that the bundle is not empty before setting the _bundleNonce. Furthermore, in the function addAddressToBundle the _bundleNonce should be set to true if it is false

STATUS: fixed

DESCRIPTION:

In the function **mintToken** the mapping **_bundleNonce** is set to true for the **NFCS** token **ID**, even if there was no bundle attached (e.g. only primary address is being passed).

A similar flaw is present in the function **addAddressToBundle**. A check is missing to ensure that if **_bundleNonce** is false then it should be set it to true.

NFCS.sol:mintToken

```
function mintToken(
  address[] memory bundle,
  bytes[] memory signatures,
  string memory version
) public override ifNotPaused checkVersion(version) {
  require(bundle.length > 0 && bundle.length == signatures.length,
Errors.ARGUMENTS_LENGTH);
  address primaryAddress = bundle[0];
  require(!_mintedNonce[primaryAddress], Errors.NFCS_TOKEN_MINTED);
  uint256 tokenId = _tokenIdCounter.current();
  _tokenBundle[tokenId].push(primaryAddress);
  verifyAdd(bundle, signatures, tokenId);
  _safeMint(primaryAddress, tokenId);
  _tokenIdCounter.increment();
  _mintedNonce[primaryAddress] = true;
  _bundleNonce[tokenId] = true;
  emit TokenMinted(block.timestamp, primaryAddress, tokenId, bundle);
}
```

NFCS.sol:addAddressToBundle

```
function addAddressToBundle(
  address[] memory bundle,
  bytes[] memory signatures,
  string memory version
) external ifNotPaused checkVersion(version) {
  require(bundle.length > 1 && bundle.length == signatures.length,
Errors.ARGUMENTS_LENGTH);
  address primaryAddress = bundle[0];
  uint256 tokenId = tokenOfOwnerByIndex(primaryAddress, 0);
  verifyAdd(bundle, signatures, tokenId);
  emit BundleUpdate(block.timestamp, primaryAddress, tokenId, bundle);
}
```

10. CENTRALIZATION RISK

SEVERITY: Medium

PATH: Pool.sol:approveLoanManager:L205-212

REMEDIATION: add the following check:

require(hasRole(Roles.LOAN_MANAGER, loanManager), "...")

STATUS: fixed

DESCRIPTION:

The admin can approve users' tokens in the pool to any address. This functionality is not limited to just loan managers and therefore poses a centralization risk.

```
/**
 * @dev Method to approve spending of underlyingToken by LoanManager
 * @notice needed for borrowing
 * @param loanManager LoanManager address
 * @param amount amount to approve
 */
function approveLoanManager(address loanManager, uint256 amount)
  external
  onlyRole(Roles.ADMIN)
{
  underlyingToken.approve(loanManager, amount);
  emit LoanManagerApproved(loanManager, amount);
}
```

11. REENTRANCY IN NFCS TOKEN MINTING

SEVERITY: Low

PATH: NFCS.sol:mintToken:L230-256

REMEDIATION: place the update of _mintedNonce and any other state-changing operations before the call to _safeMint. In other words, it is highly recommended to implement the Checks-Effects-Interactions pattern

STATUS: fixed

DESCRIPTION:

In the function **mintToken** reentrancy is possible via the call to **_safeMint**, which in-turn calls **_checkOnERC721Received** and passes the execution flow to the receiving contract (if the receiver is a contract). However, no impactful attack vectors were found for the scope but the possibility to hijack the flow is still present in the code as the **_mintedNonce** is updated only after the call.

```
function mintToken(
  address[] memory bundle,
  bytes[] memory signatures,
  string memory version
) public override ifNotPaused checkVersion(version) {
  require(bundle.length > 0 && bundle.length == signatures.length,
Errors.ARGUMENTS_LENGTH);
  address primaryAddress = bundle[0];
  require(!_mintedNonce[primaryAddress], Errors.NFCS_TOKEN_MINTED);
  uint256 tokenId = _tokenIdCounter.current();
  _tokenBundle[tokenId].push(primaryAddress);
  verifyAdd(bundle, signatures, tokenId);
  _safeMint(primaryAddress, tokenId);
  _tokenIdCounter.increment();
  _mintedNonce[primaryAddress] = true;
  _bundleNonce[tokenId] = true;
  emit TokenMinted(block.timestamp, primaryAddress, tokenId, bundle);
}
```

12. NFCS PRIMARY ADDRESS CAN ALSO BE SECONDARY ADDRESS AND VICE VERSA

SEVERITY: Low

PATH: NFCS.sol:verifyAdd:L190-220

REMEDIATION: add a require(primaryAddress != bundle[i]), "not same address") in the for-loop. And require(secondaryToPrimary[primaryAddress] == address(0), "already secondary") at the beginning of the function

STATUS: fixed

DESCRIPTION:

In **NFCS.sol:verifyAdd**, a user can propose a signed bundle to add secondary addresses.

However, the function does not check whether any of the secondary addresses is the primary address. And it does not check whether the primary address is an existing secondary address.

```
function verifyAdd(
  address[] memory bundle,
  bytes[] memory signatures,
  uint256 tokenId
) internal {
  address primaryAddress = bundle[0];
  bytes memory primaryAddressBytes = abi.encodePacked(primaryAddress);
  bytes memory bundlePacked = primaryAddressBytes;
  for (uint256 i = 1; i < bundle.length; i++) {
    require(secondaryToPrimary[bundle[i]] == address(0),
Errors.NFCS_ADDRESS_BUNDLED);

    require(
      primaryAddressBytes.toEthSignedMessageHash().recover(signatures[i]) ==
bundle[i],
      Errors.NFCS_WALLET_VERIFICATION_FAILED
    );
    secondaryToPrimary[bundle[i]] = primaryAddress;
    _tokenBundle[tokenId].push(bundle[i]);
    bundlePacked = abi.encodePacked(bundlePacked, bundle[i]);
  }

  require(
    bundlePacked.toEthSignedMessageHash().recover(signatures[0]) ==
primaryAddress,
    Errors.NFCS_WALLET_VERIFICATION_FAILED
  );
}
```

13. REDUNDANT INHERITANCE AND MODIFIERS

SEVERITY: Informational

PATH: NFCS.sol, Pool.sol:deposit, Pool.sol:withdraw

REMEDIATION: remove redundant inheritance and modifiers

STATUS: fixed

DESCRIPTION:

The contract **NFCS.sol** is derived from both **ERC721EnumerableUpgradeable** and **ERC721Upgradeable** contracts, although **ERC721EnumerableUpgradeable** itself inherits from **ERC721Upgradeable**.

In the functions **Pool.sol:deposit** and **Pool.sol:withdraw** the modifier **nonReentrant** (and as a result the **ReentrancyGuardUpgradeable** inheritance) is redundant as the token transfer calls are placed after state-changing operations (and therefore comply with the Check-Effects-Interaction pattern).

NFCS.sol

```
contract NFCS is
  Initializable,
  ERC721Upgradeable,
  ERC721EnumerableUpgradeable,
  PausableUpgradeable,
  OwnableUpgradeable,
  UUPSUpgradeable,
  NFCSInterface,
  Version
```

Pool.sol

```
contract Pool is
 IPool,
 Initializable,
 ERC20Upgradeable,
 SelectivePausable,
 AccessControlUpgradeable,
 UUPSUpgradeable,
 ReentrancyGuardUpgradeable,
 Version
...
function deposit(uint256 underlyingTokenAmount, string memory version)
  external
  checkVersion(version)
  ifNotPaused
  nonReentrant
{
  uint256 rTokenAmount = stablecoinToRToken(underlyingTokenAmount);
  poolValue += underlyingTokenAmount;
  _mint(msg.sender, rTokenAmount);
  underlyingToken.safeTransferFrom(msg.sender, address(this), underlyingTokenAmount);
  emit LiquidityDeposited(block.timestamp, msg.sender, underlyingTokenAmount, rTokenAmount);
}
function withdraw(uint256 rTokenAmount, string memory version)
  external
  checkVersion(version)
  ifNotPaused
  nonReentrant
{
  uint256 underlyingTokenAmount = rTokenToStablecoin(rTokenAmount);
  poolValue -= underlyingTokenAmount;
  _burn(msg.sender, rTokenAmount);
  underlyingToken.safeTransfer(msg.sender, underlyingTokenAmount);
  emit LiquidityWithdrawn(block.timestamp, msg.sender, underlyingTokenAmount, rTokenAmount);
}
```

14. FLOATING PRAGMA

SEVERITY: Informational

PATH: IVersion.sol, Version.sol, Errors.sol

REMEDIATION: change the compiler version to ^0.8.9.

STATUS: fixed

DESCRIPTION:

The interface **IVersion.sol**, the contract Version.sol, and the library **Errors.sol** use the compiler version ^0.8.4, while the rest of the contracts use ^0.8.9.

IVersion.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.4;

interface IVersion {
```

Version.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.4;

abstract contract Version is IVersion {
```

Errors.sol

```
// SPDX-License-Identifier: None
pragma solidity ^0.8.4;

library Errors {
```

15. NATSPEC INCOMPLETE DESCRIPTION

SEVERITY: Informational

PATH: CollateralManager.sol:checkFrozenCollateralBalance

REMEDIATION: add a description for the parameter

STATUS: fixed

DESCRIPTION:

The **NatSpec** comment for the modifier **checkFrozenCollateralBalance**

```
/**
 * @dev Modifier to verify that user has sufficient collateral balance
 * @param token collateral address
 * @param owner user address
 * @param amount needed amount of collateral
 */
modifier checkFrozenCollateralBalance(
  IERC20MetadataUpgradeable token,
  address freezer,
  address owner,
  uint256 amount
```

16. NATSPEC COMMENT MISTYPE

SEVERITY: Informational

PATH: NFCS.sol:mintToken

REMEDIATION: correct the comments

STATUS: fixed

DESCRIPTION:

There is a mistype present in the **NatSpec** comment for the function **mintToken**.

```
/**
   * @dev Verification of addresses bundle
   * @param bundle array of addresses; first address is a primary address
   * @param signatures signatures of bundle;
   * @notice first signature is from primary address that confirms bundle addresses
   * @notice other signatires are from bundle addresses that confirms primary
address
   * @param tokenId if of the NFCS token
   */
```