EURxb.finance platform: Smart Contracts audit report

By: SFXDX Team
For: EURxb Finance

EURxb.finance platform: Smart Contracts audit report

The following Audit was prepared by the Sfxdx team and the purpose of this audit is to analyze the codebase of ten smart contracts for quality of the code, security threats and compliance with the business requirements.

Authors:
Maxim Prishchepo,
Stanislav Golovin.

This document describes the compliance of EURxb.finance platform`s Smart Contracts with the logical, architectural and security requireme The following Audit was prepared by the Sfxdx team and the purpose this audit is to analyze the codebase of ten smart contracts for quality the code, security threats and compliance with the business requirem

During the audit, we classified the shortcomings into fou categories of severity:

- Informational - shortcomings that are described in the document for informational purposes only. They are not required for correction, do not violate the intended architecture and business logic.

- Low - shortcomings that do not violate the requirements, but can be eliminated to improve the quality of the code. Their elimination can be considered as a recommendation.

- Medium - shortcomings that are not serious, but should be better eliminated. Their elimination can be considered as a recommendation.

- High - shortcomings that lead to the fact that the smart contract does not meet the requirements. For example, business logic violation, inconsistency with the planned architecture etc. This type of shortcomings must be eliminated from the smart contract code.

Based on the results of the audit, the following conclusion can be ma

Generally the implemented business logic and architecture of smart contracts meets initial requirements. All smart contracts do not have security or other critical issues.
Mostly informational - severity shortcomings were found. For example or extra checks.
Three high-severity issues were found that should be corrected in the nearest future.
No medium - severity issues were found.
Also few low-severity shortcomings were detected and eliminating th can improve the code’s quality.

A more detailed description of the founded issues is provided in the document. The document is attached.

EURxb.finance Audit report

Introduction

Git repository with contracts:

https://github.com/EURxbfinance/SmartBond/

Audited contracts:

EurxbSecurityAssetToken
EurxbBondToken
AllowList
OperatorVote
MultiSignature
DDP
EURxb
XBE
Staking Manager
Router

Below are summary conclusions regarding the contracts codebase considering the business logic analysis, correctness of logic implementation, security analysis, gas efficiency analysis and style guide principles.

As an appendix to this report the “Test Coverage” report and “Static Analysis” report are also attached.

Please note the naming of the XBG token was changed to XBE prior to launch. Any references to XBG in code has been changed to XBE.

EurxbSecurityAssetToken

The contract codebase audit is current as of December 11, 2020

Commit: https://github.com/EURxbfinance/SmartBond/commit/5d90682b29848e28a2bacfc2abc563046d4d473e

The "EurxbSecurityAssetToken" contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	3
Low	-	1
Medium	-	0
High	-	0

The shortcomings are detailed below.

Modified ERC721 Template

Severity - Low

The "EurxbSecurityAssetToken" contract fully implements the ERC721 standard interface, but takes into account the specifics of the required business logic.
Pay attention to the fact that the "EurxbSecurityAssetToken" contract is inherited not from the widely accepted ERC721 template contract from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol), but from the ERC721 custom template:

```
import "./templates/ERC721.sol";
```

This approach clearly indicates, at least, not an optimal approach to the implementation of logic. It is quite possible that in the end the logic will be consistent and not contain bugs. But at the same time the applied approach itself leads to unnecessary complexity in the code, including the complexity of the code readability and perception. And in the case of the subsequent addition of new logic, this initial complexity will interfere and may lead to the fact that bugs will be made.

In this case, the ERC721 custom template is slightly different from the ERC721 template contract from OpenZeppelin and does not violate the basic template logic.

The main difference, for the sake of which its own implementation of the ERC721 template is used, is the changed internal function “_transfer”, which lacks the following check:

```
require(ownerOf(tokenId) == from, "ERC721: transfer of token that is not own");
```

Why this is done becomes clear from the additional private (not even internal) function “_safeTransferFrom” of the “EurxbSecurityAssetToken” contract, where “_msgSender ()” is used as the first argument.
Further this argument is checked against that the function call was actually made from MultiSignature or from “EurxbBondToken” contract, which are allowed to transfer tokens.
And this is where the initial complexity of the chosen approach leads to the misperception of the “from” parameter of the “_safeTransferFrom” function, which is first passed to the internal function “_safeTransfer” of the ERC721 template, and then passed to the internal function “_transfer” of the ERC721 template.
It is this parameter that is validated in the check that was removed, mistakenly believing that the value of the “from” parameter contains the address of the Multior the “EurxbBondToken” contract, which themselves are not the owners of the tokens and therefore the check will always fail.
But in fact, the value of the “from” parameter contains the address from which the token should be transferred, and the removed check should confirm that this address is the current owner of the token, so the transfer is valid.
Inconsistency becomes possible in the “_holderTokens” data structure without this check, since if the address of a non-current token owner was passed in the “from” parameter and this is not checked, then:
- the operation “_holderTokens [from] .remove (tokenId)” in the “_transfer” function will not delete the token record from the current token owner (and “revert” will not happen);
- the next operation “_holderTokens [to] .add (tokenId)” will add a token record to the new token owner.
Thus, a record about the same token will be present at once for two owners.
So, the “_transfer” function with the implementation that is present in the ERC721 template contract from OpenZeppelin does not conflict with the specifics of the required business logic of the “EurxbSecurityAssetToken” contract.

The rest of the differences - there are only two of them:
The public "getter"-functions “getApproved” and “isApprovedForAll” have a “virtual” keyword added to imply that these functions will be overridden at the “EurxbSecurityAssetToken” contract level. But in the end, the logic of these functions has not been changed in the “EurxbSecurityAssetToken” contract, so such a “virtual” keyword is not needed in these functions.
Accordingly, functions “getApproved” and “isApprovedForAll” do not conflict with the specifics of the required business logic of the “EurxbSecurityAssetToken” contract.

As a result, the “EurxbSecurityAssetToken” contract does not need to use the custom ERC721 template, but rather the ERC721 template contract from OpenZeppelin.

Also in the current implementation of the “EurxbSecurityAssetToken” contract:
- The public function “transferFrom” calls the private function “_safeTransferFrom”, which in turn calls the private function “_safeTransfer”. Those in fact, the “transferFrom” function implements the secure option for tokens transferring. Given the specifics of the required business logic, this point does not

really matter. But in order to best match the logic of the ERC721 template contract from OpenZeppelin, it is recommended to implement the unsafe option for tokens transferring in the “transferFrom” function.
- The private function “_isApproved” is called in just one place - inside the private function “_safeTransferFrom”. It follows from this that the logic of the “_isApproved” function could have been implemented inside the “_safeTransferFrom” function and not as a separate function.

Taking into account all of the above, the following implementation is proposed for the “EurxbSecurityAssetToken” contract (when inheriting from the ERC721 template contract from OpenZeppelin):

```
function transferFrom(address from, address to, uint256 tokenId) public override(ERC721,
IEurxbSecurityAssetToken)
   {
       require(
           hasRole(TokenAccessRoles.transferer(),  _msgSender()),
           "user is not allowed to transfer"
       );

       require(
           IAllowList(_allowList).isAllowedAccount(to),
           "user is not allowed to receive tokens"
       );

       if (_msgSender() != _bond) {
           require(_isApproved(to, tokenId), "transfer was not approved");
       }

       _transfer(from, to, tokenId);

       if (_msgSender() != _bond) {
           IERC721(_bond).transferFrom(from, to, tokenId);
       }
   }

   function safeTransferFrom(address from, address to, uint256 tokenId, bytes memory
_data) public override
   {
       require(
           hasRole(TokenAccessRoles.transferer(),  _msgSender()),
           "user is not allowed to transfer"
       );

       require(
           IAllowList(_allowList).isAllowedAccount(to),
           "user is not allowed to receive tokens"
       );
```

```
        if (_msgSender() != _bond) {
            require(_isApproved(to, tokenId), "transfer was not approved");
        }

        _safeTransfer(from, to, tokenId, _data);

        if (_msgSender() != _bond) {
            IERC721(_bond).safeTransferFrom(from, to, tokenId, _data);
        }
    }

    function _isApproved(address to, uint256 tokenId) private view returns (bool)
    {
        require(_exists(tokenId), "token does not exist");
        address owner = ownerOf(tokenId);
        return (getApproved(tokenId) == to ||
            isApprovedForAll(owner, to));
    }
```

In this case, the “safeTransferFrom (address from, address to, uint256 tokenId)” function (without the “_data” parameter) is not redefined at the “EurxbSecurityAssetToken” contract level, but is used from the ERC721 template contract.
In this case, the “transferFrom” function implements the unsafe option for token transferring, and the “_isApproved” function is called in several places (in the “transferFrom” and “safeTransferFrom” functions), so its existence makes sense.

It can be seen that the proposed implementation fully follows the approach that is incorporated in the logic of the ERC721 template contract from OpenZeppelin. And as a result, unnecessary complexity and confusion of logic may be removed from the contract.

Contract initialization - constructor

The initial initialization of the “EurxbSecurityAssetToken” contract takes place in the constructor, where the value of “baseURI” and the addresses MultiSignature contract, “bond” (“EurxbBondToken” contract), “allowList” (“AllowList” contract) are set.
Further the roles “MINTER_ROLE”, “BURNER_ROLE”, “TRANSFERER_ROLE” are assigned to the “MultiSig” account and role “TRANSFERER_ROLE” is also assigned to the “bond” account.

No check for null values of arguments

Severity - Informational

There is no check in the constructor that the passed address arguments are not null.
The possible check implementation:

```
require(multisig != address(0), "multisig address is invalid");
require(bond != address(0), "bond address is invalid");
require(allowList != address(0), "list address is invalid");
```

Features: privileged operations

The “EurxbSecurityAssetToken” contract meets the following requirements:
- token minting and token burning can only be done by the MultiSig account;
- the users cannot transfers their tokens themselves;
- the token transfer from the account of the current owner to the account of the new owner can be done by MultiSig (if the current owner has allowed such a transfer);
- the token transfer from the account of the current owner to the account of the new owner can be done by the “EurxbBondToken” contract (if the redemption of the corresponding EurxbBond token is done by the new owner).

To implement the above requirements, the “EurxbSecurityAssetToken” contract inherits from the “AccessControl” template contract from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol), defining the required roles “MINTER_ROLE”, “BURNER_ROLE”, “TRANSFERER_ROLE” in the “TokenAccessRoles.sol” library:
- the public functions “mint” and “burn” can only be called by the MultiSig account, which is assigned the roles “MINTER_ROLE” and “BURNER_ROLE” in the contract constructor;
- the public functions of tokens transferring “transferFrom” and “safeTransferFrom” can only be called by the MultiSig account or by the “EurxbBondToken” contract, which are assigned the “TRANSFERER_ROLE” role in the contract constructor.

No bugs were found in the implementation of this feature, but there is one informational note.

Uninitialized role "DEFAULT_ADMIN_ROLE"

Severity - Informational

The role “DEFAULT_ADMIN_ROLE” remains uninitialized in the contract “AccessControl” from which the contract “EurxbSecurityAssetToken” is inherited.
This means that the set values for the roles "MINTER_ROLE", "BURNER_ROLE", "TRANSFERER_ROLE" are static, and the list of members of each role cannot be changed.
If such a "static-roles" requirement is dictated by the business requirements then inheritance from the “AccessControl” contract is redundant since the setter-functions implemented in it are not used (setter-functions to change the list of role principals).

Features: List of allowed users

The “EurxbSecurityAssetToken” contract meets the following requirements:
- token minting and token transfer are possible only to a user account that was previously added to the allow list by the MultiSig account;
- the current owner of the token / tokens can specify the account of the new owner from the allow list, to whom the token / tokens should be transferred.

To implement the above requirements, the “EurxbSecurityAssetToken” contract uses the “AllowList” external contract which address is set in the contract constructor.
As a result, the logic of the public function "mint" and the private function "_safeTransferFrom" (called from the public functions "transferFrom" and "safeTransferFrom") contains a check that the user account is in the allow list:

```
require(
    IAllowList(_allowList).isAllowedAccount(to),
    "user is not allowed to receive tokens"
);
```

No bugs were found in the implementation of this feature.

Features: Approved transfers

The “EurxbSecurityAssetToken” contract meets the following requirement:
- The MultiSig can transfer a token from the current owner's account to the new owner's account only if the current owner has allowed such a transfer.

To implement the above requirement, the "EurxbSecurityAssetToken" contract uses the existing functionality of the functions:
- “approve”;
- “getApproved”;
- “setApprovalForAll”;
- “isApprovedForAll”.

The current owner of the token allows a token transfer to the new owner using the “approve” or “setApprovalForAll” function.
The MultiSig can then transfer the token.
When transferring a token, a check takes place, which must confirm that the transfer is valid. This verification logic is moved to a separate private function “_isApproved”:

```
function _isApproved(address to, uint256 tokenId) private view returns (bool)
{
    require(_exists(tokenId), "token does not exist");
    address owner = ownerOf(tokenId);
    return (getApproved(tokenId) == to ||
        isApprovedForAll(owner, to));
}
```

It can be seen from the code above that the check uses the “getApproved” and “isApprovedForAll” functions.

No bugs were found in the implementation of this feature.

Features: AutoId for TokenId

The “EurxbSecurityAssetToken” contract uses an approach for auto-generating “tokenURI” when minting tokens - “tokenURI” is a concatenation of the “baseURI” string, which is set in the constructor, and the unique token identifier “tokenId”, which is the value of the incremental counter.
For this, the “EurxbSecurityAssetToken” contract uses the “Counters.sol” utility from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Counters.sol), which implements the logic of the incremental counter.

No bugs were found in the implementation of this feature.

Features: used variables and data structures

The “EurxbSecurityAssetToken” contract uses the following additional variables and data structures (not implied by the the ERC721 standard):

```
mapping(uint256 => uint256) private _values;

        uint256 private _totalValue;

        address private _bond;

        address private _allowList;
```

Missing getter-functions

Severity - Informational

Not all of the above variables and data structures have corresponding getter-functions.
In particular, there are no getter-functions for the “_values”, “_bond” and “_allowList”.

EurxbBondToken

The contract codebase audit is current as of December 17, 2020

Commit: https://github.com/EURxbfinance/SmartBond/commit/d7de32925e66ae26222817c196666640868845ae

The “EurxbBondToken” contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	6
Low	-	2
Medium	-	0
High	-	0

The shortcomings are detailed below.

Modified ERC721 Template

Severity - Low

The same approach as in the “EurxbSecurityAssetToken” contract was applied here.
The “EurxbBondToken” contract fully implements the ERC721 standard interface, but takes into account the specifics of the required business logic (the implementation of these specific features will be outlined below).
And for this, the “EurxbBondToken” contract is inherited not from the widely accepted ERC721 template contract from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol), but from the ERC721 custom template:

import "./templates/ERC721.sol";

This is the same ERC721 custom template that is used in the “EurxbSecurityAssetToken” contract, and the differences between this template and the ERC721 template contract from OpenZeppelin are already described in the corresponding section Modified ERC721 Template of the “EurxbSecurityAssetToken” contract:

- the modified internal function “_transfer”, which lacks the following check:

require(ownerOf(tokenId) == from, "ERC721: transfer of token that is not own");

- the “virtual” keyword has been added in the public getter-functions “getApproved” and “isApprovedForAll” which implies that these functions will be overridden.

The disadvantages of using the approach with a custom ERC721 template have also already been described (in the corresponding section Modified ERC721 Template of the “EurxbSecurityAssetToken” contract):
As a result, this approach leads to a misperception of the “from” parameter of the “_safeTransferFrom” function, mistakenly believing that the value of the “from” parameter contains the address of the "EurxbSecurityAssetToken" contract or the "DDP" contract, which themselves are not the owners of the tokens and therefore the check will always fail.
But in fact, the value of the “from” parameter contains the address from which the token should be transferred, and the removed check should confirm that this address is the current owner of the token, so the transfer is valid.
Inconsistency becomes possible in the “_holderTokens” data structure without this check, since if the address of a non-current token owner was passed in the “from” parameter and this is not checked, then:
- the operation “_holderTokens [from] .remove (tokenId)” in the “_transfer” function will not delete the token record from the current token owner (and “revert” will not happen);
- the next operation “_holderTokens [to] .add (tokenId)” will add a token record to the new token owner.
Thus, a record about the same token will be present at once for two owners.
So, the “_transfer” function with the implementation that is present in the ERC721 template contract from OpenZeppelin does not conflict with the specifics of the required business logic of the “EurxbBondToken” contract.

As for the public getter-functions “getApproved” and “isApprovedForAll” in the ERC721 custom template:
The public setter-functions “approve” and “setApprovalForAll” from the ERC721 standard should not be available to users due to the specifics of the “EurxbBondToken” contract business logic.
And the implementation of the getter-functions “getApproved” and “isApprovedForAll” can be left as it is in the ERC721 template contract from OpenZeppelin, since the setter-functions “approve” and “setApprovalForAll” do not allow to modify the corresponding data structures anyway.
Accordingly, implementation of the getter-functions “getApproved” and “isApprovedForAll” in the ERC721 template contract from OpenZeppelin do not conflict with the specifics of the “EurxbBondToken” contract business logic.

As a result, the “EurxbBondToken” contract does not need to use the custom ERC721 template, but rather the ERC721 template contract from OpenZeppelin.

Also in the current implementation of the “EurxbBondToken” contract, the public function “transferFrom” calls the private function “_safeTransferFrom”, which in turn calls the private function “_safeTransfer”. Those in fact, the “transferFrom” function implements the secure option for tokens transferring. Given the specifics of the required business logic, this point does not really matter. But in order to best match the logic of the ERC721 template contract from OpenZeppelin, it is recommended to implement the unsafe option for tokens transferring in the “transferFrom” function.

Taking into account all of the above, the following implementation is proposed for the “EurxbBondToken” contract (when inheriting from the ERC721 template contract from OpenZeppelin):

```
   function transferFrom(address from, address to, uint256 tokenId) public override(ERC721,
 IEurxbBondToken)
```

```
  {
    require(
      hasRole(TokenAccessRoles.transferer(),  _msgSender()),
      "user is not allowed to transfer"
    );

    _transfer(from, to, tokenId);

    if (_msgSender() != _sat) {
      IERC721(_sat).transferFrom(from, to, tokenId);
    }
  }

  function safeTransferFrom(address from, address to, uint256 tokenId, bytes memory _data)
public override
  {
    require(
      hasRole(TokenAccessRoles.transferer(),  _msgSender()),
      "user is not allowed to transfer"
    );

    _safeTransfer(from, to, tokenId, _data);

    if (_msgSender() != _sat) {
      IERC721(_sat).safeTransferFrom(from, to, tokenId, _data);
    }
  }
```

In this case, the “safeTransferFrom (address from, address to, uint256 tokenId)” function (without the “_data” parameter) is not redefined at the “EurxbBondToken” contract level, but is used from the ERC721 template contract.
In this case, the “transferFrom” function implements the unsafe option for token transferring.

It can be seen that the proposed implementation fully follows the approach that is incorporated in the logic of the ERC721 template contract from OpenZeppelin. And as a result, unnecessary complexity and confusion of logic may be removed from the contract.

Contract initialization - constructor and “configure” function

The initial initialization of the “EurxbBondToken” contract is divided into two steps:
- the “_baseURI” variable is initialized in the contract constructor;
- in the “configure” function (the function can be called once by the contract deployer), the addresses of the “EurxbSecurityAssetToken” contract and “DDP” contract are set and further the roles “MINTER_ROLE”, “BURNER_ROLE”, “TRANSFERER_ROLE” are assigned to them.

Unused parameter “allowList”

Severity - Informational

The “configure” function has a parameter “allowList”, which is not used anywhere in the code (logic) of the “EurxbBondToken” contract. Accordingly, this parameter is unnecessary and the “configure” function should not contain it:

```
function configure(address sat, address ddp) external initializer {
```

There is no check for contract initialization completeness before calling writable functions

Severity - Low

Since the “configure” function is responsible for initializing the “EurxbBondToken” contract, then such writable functions as “mint”, “burn”, “transferFrom”, “safeTransferFrom” must check the contract initialization completeness before executing their logic.
The initialization state is stored in the private variable "_isContractInitialized" of the "Initializable" contract, from which the "EurxbBondToken" contract is inherited:

```
import "./templates/Initializable.sol";
```

Such a kind of check in writable functions might look like this:

```
require(_isContractInitialized, "contract has not been initialized");
```

Using an unified approach when initializing a contract

Severity - Informational

The initial initialization of the “EurxbBondToken” contract occurs both using the contract constructor and using the “configure” function.
The same approach is used in some other contracts, while the rest of the contracts use only the constructor.

The profit of using such an approach is not explicitly visible and, most likely, this was done for the convenience of deploying contracts. But because of this, the complexity of the logic and code perception increases.
To eliminate unnecessary complexity and confusion of logic, it is recommended to use, if applicable, the unified approach for contracts initialization (for all contracts, not just for “EurxbBondToken” contract) either using the contract constructor or using a separate initializing function (the “configure” function).

Features: privileged operations

The “EurxbBondToken” contract meets the following requirements:
- token minting can only be done by the “EurxbSecurityAssetToken” contract;
- token burning can only be done by the “DDP” contract;
- the users cannot transfers their tokens themselves;
- the token transfer from the account of the current owner to the account of the new owner can be done by the “EurxbSecurityAssetToken” contract (internal call: MultiSig -> EurxbSecurityAssetToken -> EurxbBondToken, if the current owner has allowed such a transfer);
- the token transfer from the account of the current owner to the account of the new owner can be done by the “DDP” contract (if the redemption of the corresponding EurxbBond token is done by the new owner).

To implement the above requirements, the “EurxbBondToken” contract inherits from the “AccessControl” template contract from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol), defining the required roles “MINTER_ROLE”, “BURNER_ROLE”, “TRANSFERER_ROLE” in the “TokenAccessRoles.sol” library:
- the public function “mint” can only be called by the “EurxbSecurityAssetToken” contract, which is assigned the “MINTER_ROLE” role in the “configure” function of the contract;
- the public function “burn” can only be called by the “DDP” contract, which is assigned the “BURNER_ROLE” role in the “configure” function of the contract;
- the public functions of tokens transferring “transferFrom” and “safeTransferFrom” can only be called by the “EurxbSecurityAssetToken” contract or by the “DDP” contract, which are assigned the “TRANSFERER_ROLE” role in the “configure” function of the contract.

No bugs were found in the implementation of these features, but there is one informational note.

Uninitialized role "DEFAULT_ADMIN_ROLE"

Severity - Informational

The role “DEFAULT_ADMIN_ROLE” remains uninitialized in the contract “AccessControl” from which the contract “EurxbBondToken” is inherited.
This means that the set values for the roles "MINTER_ROLE", "BURNER_ROLE", "TRANSFERER_ROLE" are static, and the list of members of each role cannot be changed.
If such a "static-roles" requirement is dictated by the business requirements then inheritance from the “AccessControl” contract is redundant since the setter-functions implemented in it are not used (setter-functions to change the list of role principals).

Features: “approve”, “getApproved”, “setApprovalForAll”, “isApprovedForAll” functions

The “EurxbBondToken” contract meets the following requirement:
- the setter-functions “approve” and “setApprovalForAll” of the ERC721 standard should not be available to users.
To implement the above requirement, the implementations of the “approve” and “setApprovalForAll” setter-functions are redefined at the “EurxbBondToken” contract level:

```
revert("method is not supported");
```

Implementations of the “getApproved” and “isApprovedForAll” functions

Severity - Informational

The implementations of the “getApproved” and “isApprovedForAll” getter-functions are also redefined at the “EurxbBondToken” contract level:

```
revert("method is not supported");
```

But the implementation of these getter-functions can be left as it is in the ERC721 template contract from OpenZeppelin, since the setter-functions “approve” and “setApprovalForAll” do not allow to modify the corresponding data structures anyway.

As a result, this will allow inheritance for the “EurxbBondToken” contract from the ERC721 template contract from OpenZeppelin, rather than from the custom ERC721 template.

Features: used variables and data structures

The “EurxbBondToken” contract uses the following additional variables and data structures (not implied by the the ERC721 standard):

```
struct BondInfo {
    uint256 value;
    uint256 interestPerSec;
    uint256 maturityEnds;
}

mapping(uint256 => BondInfo) private _bondInfo;

uint256 private _totalValue;

address private _sat;

address private _ddp;
```

Unused variable “interestPerSec” in the “BondInfo” data structure

Severity - Informational

The “BondInfo” data structure stores additional info for each EurxbBond token. This info is set during the EurxbBond token minting (using the “mint” function).

The info includes the calculation and setting a value for the “interestPerSec” variable.
This value means the EurxbBond token's income which it generates per 1 second:

```
uint256 interestPerSec = value
        .mul(INTEREST_PERCENT).div(365 days).div(100);

        _bondInfo[tokenId] = BondInfo(
        {
                value: value,
                interestPerSec: interestPerSec,
                maturityEnds: maturityEnds
        });
```

But variable “interestPerSec” is not used anywhere in the code (logic) of the “EurxbBondToken” contract. So the variable is not necessary, and the “BondInfo” data structure should not contain it:

```
struct BondInfo {
        uint256 value;
        uint256 maturityEnds;
        }
```

Missing getter-functions

Severity - Informational

Not all of the above variables have corresponding getter-functions.
In particular, there are no getter-functions for the “_sat”, “_ddp”.

Features: “hasToken” function

The external function “hasToken” is defined in the “EurxbBondToken” contract.
But this function is not used anywhere in the code (logic) of the “EurxbBondToken” contract.
However, the “hasToken” function is used by the “EurxbSecurityAssetToken” and “DDP” contracts to perform the necessary checks.
So this function is necessary for inter-contract interactions.

No bugs were found in the implementation of this feature.

AllowList

Interface implementation

The “AllowList” contract inherits from its two interfaces “IAllowList” and “IAllowListChange”.

There is no import of file for the interface "IAllowListChange"

Severity - Informational

The “IAllowList” interface file was imported into the “AllowList” contract, but the “IAllowListChange” interface file was not imported:

```
import "./interfaces/IAllowList.sol";

contract AllowList is IAllowList, IAllowListChange, Context {
```

At the same time, the “IAllowListChange” interface is declared inside the “IAllowList.sol” file for the “IAllowList” interface. That is an anti-pattern.
Also, the profit of using such an approach is not quite clear - only one function “isAllowedAccount” is declared in the “IAllowList” interface, and only two functions are declared in the “IAllowListChange” interface: “allowAccount” and “disallowAccount”.

It is recommended to combine all the declared functions into the “IAllowList” interface, which will reduce the logic confusion and improve the code readability:
- remove the declaration of the “IAllowListChange” interface from the file “IAllowList.sol”, but keep the declared functions “allowAccount” and “disallowAccount” (as functions of the “IAllowList” interface);
- leave inheritance of the “AllowList” contract only from “IAllowList” interface:

```
contract AllowList is IAllowList, Context {
```

Features: privileged operations

In the “AllowList” contract, the “allowAccount” (adding the user's address to the allowed list) and “disallowAccount” (removing the user's address from the allowed list) functions can be called only by the administrator whose address is set in the variable “_admin”.

No bugs were found in the implementation of this feature, but there are several informational notes.

No check for null values of arguments

Severity - Informational

There is no check in the “allowAccount” and “disallowAccount” functions that the address passed in the argument “account” is not null.
The possible check implementation:

```
require(account != address(0), "account address is invalid");
```

Static Administrator Role

Severity - Informational

The administrator's address is set in the contract constructor. At the same time, the contract does not contain a corresponding function for setting the address of a new administrator. As a result, the set administrator address cannot be changed.
If such a “static-administrator” requirement is not dictated by the business requirements, then perhaps it makes sense for the “AllowList” contract to inherit from the “Ownable” template contract from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol), which will allow the current administrator to transfer their privileged rights to a new administrator if necessary.

This recommendation implies adding functionality to the contract to transfer the administrator role to a new account. At the same time, this kind of functionality can be implemented using another way, and not by inheriting from the “Ownable” template contract from OpenZeppelin. This should be a decision of the contract development team.

Features: used variables and data structures

The “AllowList” contract uses only one variable and one data structure:

```
address _admin;

mapping(address => bool) private _allowList;
```

Missing a visibility-specifier for the declared variable

Severity - Informational

The “_admin” variable is not explicitly scoped.

Missing getter-functions

Severity - Informational

The corresponding getter-functions are not implemented for the above variable and data structure.

OperatorVote

The contract codebase audit is current as of December 17, 2020

Commit: https://github.com/EURxbfinance/SmartBond/commit/d7de32925e66ae26222817c196666640868845ae

The “OperatorVote” contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	3
Low	-	1
Medium	-	0
High	-	0

The shortcomings are detailed below.

Contract initialization - constructor

The initial initialization of the “OperatorVote” contract takes place in the constructor, where the value of “_votesThreshold” is set and the address list of “_founders” is formed.

No check for null values of arguments

Severity - Informational

There is no check in the constructor that the passed arguments are not null.
The possible check implementation:

```
        require(_votesThreshold != uint256(0), "votes threshold is invalid");
...
        for (uint256 i = 0; i < founders.length; i++) {
require(founders[i] != address(0), "founder address is invalid");
        _founders[founders[i]] = true;
        }
```

Features: privileged operations

The contract “OperatorVote” defines two modifiers “onlyFounders()” and “onlyOperator()”.
At the same time, in the “OperatorVote” contract itself only the “onlyFounders()” modifier is used in the “voteOperator” function.
The other modifier “onlyOperator()” is not used anywhere in the contract code (logic).

But this modifier is used in the privileged functions of the “MultiSignature” contract, which inherits from the “OperatorVote” contract.

The “onlyFounders()” modifier checks that the caller of the privileged function “voteOperator” of the “OperatorVote” contract is in the “_founders” list.
Thereby checking the ability to vote for the operator candidates.
The “onlyOperator()” modifier checks that the caller of the privileged function of the “MultiSignature” contract is the acting operator.
The operator address is stored in the “_operator” private variable of the “OperatorVote” contract and can be set using the “voteOperator” function of the “OperatorVote” contract by voting by the founders.

One minor bug was found in the implementation of this feature, and there is also an informational note.

Event does not work according to the expected logic

Severity - Low

The “OperatorChanged” Event logs the information about the operator change and this info is recorded to the transaction log. The two arguments are passed to be logged by the event: the address of the previous operator and the address of the new operator:

```
event OperatorChanged(address oldOperator, address newOperator);
```

In the current implementation the “OperatorChanged” event is generated only after the address of the new operator has been written to the private variable “_operator” in the “voteOperator” function:

```
_operator = candidate;
        emit OperatorChanged(_operator, candidate);
```

As a result, the same address of the new operator is passed to the event as both arguments, i.e. the event contains the address of the new operator as the address of the previous operator, which does not match the logic of the event.

It is required to swap the above lines of code in the “voteOperator” function to fix the bug:

```
emit OperatorChanged(_operator, candidate);
        _operator = candidate;
```

Or fix can be more detailed in the code, but it also requires a little more gas to execute:

```
        address oldOperator = _operator;
_operator = candidate;
        emit OperatorChanged(oldOperator, candidate);
```

Static “_founders” list

Severity - Informational

The founders list is set in the constructor of the “OperatorVote” contract and stored in the appropriate data structure:

```
mapping(address => bool) private _founders;
```

At the same time, the contract does not contain a corresponding setter-function for changing the founders list (adding a new founder to the list / removing the current founder from the list), or setting a completely new list for founders (as implemented in the constructor).
As a result, the set founders list cannot be changed.
Also, due to the static nature of the founders list, the “AddedFounders” Event doesn't make sense:

```
event AddedFounders(address[] founders);
```

If the founders list cannot be changed, then this event runs only once - in the constructor during the contract deployment. But in this case, it is more rational to be able to get the founders list through the corresponding getter-function which is not in the contract.

If such a “static-list” requirement is not dictated by the business requirements, then perhaps it makes sense for the “OperatorVote” contract to add an appropriate setter-function which allows to change the founders list / set a completely new founders list (the exact implementation depends on the contract development team decision).

Features: used variables and data structures

The “OperatorVote” contract uses the following variables and data structures:

```
        address private _operator;
        uint256 private _votesThreshold;

        mapping(address => bool) private _founders;
        mapping(address => address[]) private _candidates;
```

The contract contains getter-functions for the above variables (“_operator” and “_votesThreshold”), as well as getter-function to get the current number of votes cast by the founders for any operator candidate:

```
function getNumberVotes(address candidate) external view returns (uint256) {
 return _candidates[candidate].length;
}
```

Missing getter-functions

Severity - Informational

At the same time, the following getter-functions are missing in the contract:
- to get the current list of founders;
- to check if the address is in the founders list;
- to get a list of founders who voted for a certain operator candidate.

MultiSignature

The contract codebase audit is current as of December 17, 2020

Commit: https://github.com/EURxbfinance/SmartBond/commit/d7de32925e66ae26222817c196666640868845ae

The “MultiSignature” contract inherits from the “OperatorVote” contract outlined above.

The “MultiSignature” contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	4
Low	-	1
Medium	-	0
High	-	0

The shortcomings are detailed below.

Contract initialization - constructor and “configure” function

The initial initialization of the “MultiSignature” contract is divided into two steps:
- the required arguments are passed to the contract constructor to further initialization of the inherited contract “OperatorVote” (“founders”, “votesThreshold”);
- in the “configure” function (the function can be called once by the contract deployer), the addresses of the contracts “AllowList”, “DDP” and “EurxbSecurityAssetToken” are set.

No check for null values of arguments

Severity - Informational

There is no check in the “configure” function that the passed address arguments are not null.
The possible check implementation:

```
require(allowList != address(0), "list address is invalid");
require(ddp != address(0), "ddp address is invalid");
require(sat != address(0), "sat address is invalid");
```

There is no check for contract initialization completeness before calling writable functions

Severity - Low

Since the “configure” function is responsible for initializing the “MultiSignature” contract, then all writable functions (“allowAccount”, “disallowAccount”, “mintSecurityAssetToken”, “burnSecurityAssetToken”, “transferSecurityAssetToken”, “setClaimPeriod”) must check the contract initialization completeness before executing their logic.
The initialization state is stored in the private variable “_isContractInitialized” of the “Initializable” contract, from which the “MultiSignature” contract is inherited:

```
import "./templates/Initializable.sol";
```

Such a kind of check in writable functions might look like this:

```
require(_isContractInitialized, "contract has not been initialized");
```

Using an unified approach when initializing a contract

Severity - Informational

The initial initialization of the “MultiSignature” contract occurs both using the contract constructor and using the “configure” function.
The same approach is used in some other contracts, while the rest of the contracts use only the constructor.

The profit of using such an approach is not explicitly visible and, most likely, this was done for the convenience of deploying contracts. But because of this, the complexity of the logic and code perception increases.
To eliminate unnecessary complexity and confusion of logic, it is recommended to use, if applicable, the unified approach for contracts initialization (for all contracts, not just for “MultiSignature” contract) either using the contract constructor or using a separate initializing function (the “configure” function).

Features: privileged operations

The “MultiSignature” contract uses the “onlyOperator()” modifier, which is defined at the level of the inherited “OperatorVote” contract.
This modifier is used when calling all functions of the “MultiSignature” contract (“allowAccount”, “disallowAccount”, “mintSecurityAssetToken”, “burnSecurityAssetToken”, “transferSecurityAssetToken”, “setClaimPeriod”), which, in turn, execute internal calls to the corresponding functions of the “AllowList”, “SecurityAssetToken” and “DDP” contracts.

No check for null values of arguments

Severity - Informational

Each of the privileged functions does not have a check that the passed arguments are not null.

It makes sense to implement such a check in all functions, regardless of whether there is such a check in the corresponding functions of the "AllowList", "SecurityAssetToken" and "DDP" contracts or not.
In case of a null value of the argument such an approach will allow to make the "revert" at the level of the “MultiSignature” contract's function, and not as a result of an unsuccessful internal call to the function of another contract.

Features: used variables and data structures

The “MultiSignature” contract uses the following variables:

```
address private _allowList;

address private _ddp;

address private _sat;
```

Missing getter-functions

Severity - Informational

The corresponding getter-functions are not implemented for the above variables.

DDP

The contract codebase audit is current as of December 17, 2020

Commit:
https://github.com/EURxbfinance/SmartBond/commit/d7de32925e66ae26222817c196666640868845ae

The "DDP" contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	9
Low	-	1
Medium	-	0
High	-	0

The shortcomings are detailed below.

Modified ERC721 Template

Severity - Informational

The business logic of the "DDP" contract implies interaction with the "EurxbBondToken" contract, what's why the interface of the "EurxbBondToken" contract and the custom template ERC721 are imported into the "DDP" contract:

```
import "./templates/ERC721.sol";
import "./interfaces/IBondToken.sol";
```

This is the same ERC721 custom template that is used in the "EurxbSecurityAssetToken" and "EurxbBondToken" contracts, and the differences between this template and the ERC721 template contract from OpenZeppelin are already described in the corresponding section Modified ERC721 Template of the "EurxbSecurityAssetToken" contract.
Also it was already concluded in the sections describing the contracts "EurxbSecurityAssetToken" and "EurxbBondToken" that the business logic of these contracts corresponds to the logic of the ERC721 template contract from OpenZeppelin, and there is no need to use the custom ERC721 template.

As a result, it is recommended to import the interface of the ERC721 template contract from OpenZeppelin instead of importing the custom ERC721 template:

```
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
```

Contract initialization - constructor and “configure” function

The initial initialization of the “DDP” contract is divided into two steps:
- in the contract constructor, the role “ADMIN_ROLE” is assigned to the passed address argument;
- in the “configure” function (the function can be called once by the contract deployer), the addresses of the “EurxbBondToken”, “EURxb” and “AllowList” contracts are set.

No check for null values of arguments

Severity - Informational

There is no check in the constructor that the passed address argument is not null.
The possible check implementation:

```
require(admin != address(0), "admin address is invalid");
```

There is no check in the “configure” function that the passed address arguments are not null.
The possible check implementation:

```
require(bond != address(0), "bond address is invalid");
require(eurxb != address(0), "eurxb address is invalid");
require(allowList != address(0), "allowList address is invalid");
```

There is no check for contract initialization completeness before calling writable functions

Severity - Low

Since the “configure” function is responsible for initializing the “DDP” contract, then the writable functions “deposit” and “withdraw” must check the contract initialization completeness before executing their logic.
The initialization state is stored in the private variable “_isContractInitialized” of the “Initializable” contract, from which the “DDP” contract is inherited:

```
import "./templates/Initializable.sol";
```

Such a kind of check in writable functions might look like this:

```
require(_isContractInitialized, "contract has not been initialized");
```

Using an unified approach when initializing a contract

Severity - Informational

The initial initialization of the “DDP” contract occurs both using the contract constructor and using the “configure” function.
The same approach is used in some other contracts, while the rest of the contracts use only the constructor.

The profit of using such an approach is not explicitly visible and, most likely, this was done for the convenience of deploying contracts. But because of this, the complexity of the logic and code perception increases.
To eliminate unnecessary complexity and confusion of logic, it is recommended to use, if applicable, the unified approach for contracts initialization (for all contracts, not just for “DDP” contract) either using the contract constructor or using a separate initializing function (the “configure” function).

Features: privileged operations

The “DDP” contract meets the following requirements:
- the deposit of the EurxbBond tokens can only be done by the “EurxbBondToken” contract;
- only the MultiSig can set the time period duration after the maturity of the EurxbBond token ends, after which it becomes possible to buy the EurxbBond token by another user.

To implement the above requirements, the "DDP" contract:
- inherits from the “AccessControl” template contract from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol), defining the required role “ADMIN_ROLE” in the “TokenAccessRoles.sol” library. The public function “setClaimPeriod” can only be called by an account that has been assigned the role “ADMIN_ROLE” in the contract constructor;
- checks that the call to the public function "deposit" is done by the "EurxbBondToken" contract.

No bugs were found in the implementation of this feature, but there are several informational notes.

Uninitialized role "DEFAULT_ADMIN_ROLE"

Severity - Informational

The role “DEFAULT_ADMIN_ROLE” remains uninitialized in the contract “AccessControl” from which the contract “DDP” is inherited.
This means that the set value for the role "ADMIN_ROLE" is static, and the list of members of this role cannot be changed.
If such a “static-role” requirement is dictated by the business requirements then inheritance from the “AccessControl” contract is redundant since the corresponding setter-function implemented in it is not used (setter-function to change the list of role principals).

Using the single role

Severity - Informational

Only one role “ADMIN_ROLE” is assigned in the contract “DDP” from the inherited “AccessControl” template contract from OpenZeppelin. This role is used for the required check in the privileged function “setClaimPeriod”.
But the rest of the privileged contract functions (“deposit” and “withdraw”) do not use any roles for the required checks.
Considering the fact which was mentioned previously, - that the “ADMIN_ROLE” role is static, and the list of members of this role cannot be changed, - inheritance from the “AccessControl” template contract from OpenZeppelin is definitely redundant.

Perhaps it is make sense to use an additional private variable instead of the role “ADMIN_ROLE” in the “DDP” contract, which can also be initialized in the constructor, and further used to perform the required check in the “setClaimPeriod” privileged function.

Using a misleading name for the role "ADMIN_ROLE"

Severity - Informational

In the logic of the “AccessControl” contract (from which the “DDP” contract is inherited), the administrative functionality implies the ability to manage the list of role / roles principals.
Initially, this right belongs to the role “DEFAULT_ADMIN_ROLE”, which is defined by the default in the contract and actually acts as the administrator for all other contract roles.
If it's required, a person with the “DEFAULT_ADMIN_ROLE” role can assign a separate administrator for each of the contract roles.

Thus, the name of the role “ADMIN_ROLE” is misleading and adds additional confusion in understanding the general logic, since this role does nothing in terms of role administration.

Since only the MultiSig has the ability to execute the privileged function “setClaimPeriod” according to the business requirements, it would be more logical to name the role using the “MultiSig” name.
Or, in case of using the corresponding private variable instead of the role as was suggested above, it would be more logical to name the variable using the “MultiSig” name.

There is no check for null values of arguments in the privileged functions

Severity - Informational

The “setClaimPeriod” function lacks a check for the passed argument, which sets the value for the private variable “_claimPeriod”.

The “deposit” function also lacks a check for the passed arguments, but in this case a successful call to this privileged function is only possible from the “EurxbBondToken” contract, which means that all required checks should be performed at the level of the “EurxbBondToken” contract.

Two internal calls are made to the same external contract to perform an atomic operation.

Severity - Informational

This note does not apply to the logic of the “DDP” contract itself (since here only external functions of another contract are called), but refers to the implementation of the corresponding logic in the “EURxb” contract.

According to the logic, the operations “mint” and “addNewMaturity” together represent an atomic operation.
Thus, the call to the “addNewMaturity” function should be impossible without the previously successfully completed “mint” operation.
And it is in this sequence these functions are called from the “DDP” contract in the “deposit” function:

```
IEURxb(_eurxb).mint(to, value);
IEURxb(_eurxb).addNewMaturity(value, maturity);
```

So, actually, there is no error in the logic execution here.

But it would be more logical to make the “addNewMaturity” function not external, but internal in the “EURxb” contract.
And further this internal function “addNewMaturity” should be called during the executing of the “mint” function, because in total they represent an atomic operation.

As for the corresponding internal call from the “deposit” function of the “DDP” contract, - all that's required is to leave the call to only one external function “mint” of the “EURxb” contract:

```
IEURxb(_eurxb).mint(to, value);
```

Thus, the atomicity of these two operations will be controlled at the level of the “EURxb” contract, and not at the level of the external “DDP” contract.

A similar situation with the execution of the operations “burn” and “removeMaturity”, - according to the logic, these operations together represent an atomic operation.
And although the call to the “removeMaturity” function should not be executed in all cases after the “burn” operation, the same rule applies here - the call to the “removeMaturity” function should be impossible without the previously successfully completed “burn” operation.
In the current implementation the check if is it required or not to execute the “removeMaturity” operation after the “burn” operation (as well as the call to the “removeMaturity” function itself if such an operation is required) is done at the level of the “DDP” contract in the “withdraw” function:

```
IEURxb(_eurxb).burn(user, value);
        if (maturity > block.timestamp) {
            IEURxb(_eurxb).removeMaturity(value, maturity);
        }
```

Once again, actually, there is no error in the logic execution here.

But it would be more logical to make the “removeMaturity” function not external, but internal in the “EURxb” contract.
And further the necessity of calling this internal function “removeMaturity” as well as the call to the “removeMaturity” function itself should be processed during the executing of the “burn” function, because in total they represent an atomic operation.

As for the corresponding internal call from the “withdraw” function of the “DDP” contract, - all that's required is to leave the call to only one external function “burn” of the “EURxb” contract:

```
IEURxb(_eurxb).burn(user, value);
```

Thus, the atomicity of these two operations will be controlled at the level of the “EURxb” contract, and not at the level of the external “DDP” contract.

Features: used variables and data structures

The “DDP” contract uses the following variables:

```
address private _bond;
address private _eurxb;
address private _allowList;

uint256 _claimPeriod = 30 days;
```

Missing getter-functions

Severity - Informational

Not all of the above variables have corresponding getter-functions.
In particular, there are no getter-functions for the “_bond”, ”_eurxb”, “_allowList”.

EURxb

The contract codebase audit is current as of January 2, 2021

Commit: https://github.com/EURxbfinance/SmartBond/commit/170bc825bcf212c46b2c36c77b7009530da95878

The “EURxb” contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	11
Low	-	3
Medium	-	0
High	-	1

The shortcomings are detailed below.

Modified ERC20 Template

Severity - Informational

The “EURxb” contract fully implements the ERC20 standard interface, but takes into account the specifics of the required business logic.
In particular, to implement the specifics it is required to redefine the “balanceOf” function of the ERC20 standard (to implement the logic of real-time interest bearing).
Because of this requirement, the “EURxb” contract does not inherit from the widely accepted ERC20 template contract from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol), since the “balanceOf” function does not have a “virtual” keyword in this template contract. And this means that the “balanceOf” function cannot be overridden to implement the custom logic.
As a result, the “EURxb” contract is inherited from the ERC20 custom template:

```
import "./templates/OverrideERC20.sol";
```

In this case, the ERC20 custom template is slightly different from the ERC20 template contract from OpenZeppelin and does not violate the basic template logic.
As mentioned, this ERC20 custom template has added a “virtual” keyword for the “balanceOf” function:

```
function balanceOf(address account) public view override virtual returns (uint256) {
return _balances[account];
}
```

In addition, there is another difference, which is that the scope for the data structure "_balances" has been changed from "private" to "internal":

```
mapping (address => uint256) internal _balances;
```

But in this case, such a change is unnecessary, because this customization is not used anywhere in the logic (code) of the "EURxb" contract.
As a result, the scope should be left "private" for the data structure "_balances", as defined in the ERC20 template contract from OpenZeppelin.

Contract initialization - constructor and "configure" function

The initial initialization of the "EURxb" contract is divided into two steps:

In the constructor of the contract, the role "ADMIN_ROLE" is assigned to the address passed in the argument, and the values for the following variables are set:
- "_AnnualInterest" (annual interest bearing for EURxb token holding);
- "_ExpIndex" (initial value of the index used in calculating the real-time interest);
- "_CountMaturity" (initiating value for pagination, which is used in order not to exceed the gas limit during the calculation of actual value for the "_expIndex" variable).

In the "configure" function (the function can be called once by the contract deployer), the address of the "DDP" contract is set and further the roles "MINTER_ROLE" and "BURNER_ROLE" are assigned to this address.

No check for null values of arguments

Severity - Informational

There is no check in the constructor that the passed address argument is not null.
The possible check implementation:

```
require(admin != address(0), "admin address is invalid");
```

There is no check in the "configure" function that the passed address argument is not null.
The possible check implementation:

```
require(ddp != address(0), "ddp address is invalid");
```

There is no check for contract initialization completeness before calling writable functions

Severity - Low

Since the “configure” function is responsible for initializing the “EURxb” contract, then the writable functions: “mint”, “burn”, “addNewMaturity” and “removeMaturity” must check the contract initialization completeness before executing their logic.
The initialization state is stored in the private variable “_isContractInitialized” of the “Initializable” contract, from which the “EURxb” contract is inherited:

```
import "./templates/Initializable.sol";
```

Such a kind of check in writable functions might look like this:

```
require(_isContractInitialized, "contract has not been initialized");
```

Using an unified approach when initializing a contract

Severity - Informational

The initial initialization of the “EURxb” contract occurs both using the contract constructor and using the “configure” function.
The same approach is used in some other contracts, while the rest of the contracts use only the constructor.

The profit of using such an approach is not explicitly visible and, most likely, this was done for the convenience of deploying contracts. But because of this, the complexity of the logic and code perception increases.
To eliminate unnecessary complexity and confusion of logic, it is recommended to use, if applicable, the unified approach for contracts initialization (for all contracts, not just for “EURxb” contract) either using the contract constructor or using a separate initializing function (the “configure” function).

Features: privileged operations

The “EURxb” contract meets the following requirements:
- the token minting and token burning of the EURxb tokens can only be done by the “DDP” contract;
- only the “DDP” contract can set the actual values for the real-time interest bearing (the amount of the participating EURxb tokens and the period during which this amount of the participating EURxb tokens provides the real-time interest);
- only the MultiSig can set the value for pagination, which is used in order not to exceed the gas limit during the calculation of actual value for the “_expIndex” variable.

To implement the above requirements, the “EURxb” contract:

Inherits from the “AccessControl” template contract from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol), defining the required roles “MINTER_ROLE”, “BURNER_ROLE”, “ADMIN_ROLE” in the “TokenAccessRoles.sol” library:
- the public functions “mint” and “burn” can only be called by the “DDP” contract which has been assigned the roles “MINTER_ROLE” and “BURNER_ROLE” in the “configure” function;
- the public function “setCountMaturity” can only be called by an account that has been assigned the role “ADMIN_ROLE” in the contract constructor.

Also, the public functions “addNewMaturity” and “removeMaturity” can only be called by the “DDP” contract, but such a check is implemented without using a role.

No bugs were found in the implementation of this feature, but there are several informational notes.

Uninitialized role "DEFAULT_ADMIN_ROLE"

Severity - Informational

The role “DEFAULT_ADMIN_ROLE” remains uninitialized in the contract “AccessControl” from which the contract “EURxb” is inherited.
This means that the set values for the roles “MINTER_ROLE”, “BURNER_ROLE”, “ADMIN_ROLE” are static, and the list of members of these roles cannot be changed.
If such a “static-roles” requirement is dictated by the business requirements then inheritance from the “AccessControl” contract is redundant since the corresponding setter-functions implemented in it are not used (setter-functions to change the list of roles principals).

Using an unified approach when assigning privileges

Severity - Informational

The privileged functions “mint”, “burn”, “addNewMaturity” and “removeMaturity” can only be called by the “DDP” contract.
But at the same time, a different approaches are used to implement the corresponding privileges:
- the “DDP” contract is assigned the roles “MINTER_ROLE” and “BURNER_ROLE” to be able to call the “mint” and “burn” functions;
- the “onlyDDP” modifier is used in the “addNewMaturity” and “removeMaturity” functions to ensure that the functions can only be called by the “DDP” contract. Thus, the roles functionality is not used in this case.

The profit of using different approaches is not explicitly visible, but as a result, the code looks more complicated, so the confusion of logic increases.

To eliminate unnecessary complexity and confusion of logic, it is recommended to use, if applicable, the unified approach for privileges implementation either using the roles functionality or using modifiers.

Using a misleading name for the role "ADMIN_ROLE"

Severity - Informational

In the logic of the “AccessControl” contract (from which the “EURxb” contract is inherited), the administrative functionality implies the ability to manage the list of role / roles principals.
Initially, this right belongs to the role “DEFAULT_ADMIN_ROLE”, which is defined by the default in the contract and actually acts as the administrator for all other contract roles.
If it's required, a person with the “DEFAULT_ADMIN_ROLE” role can assign a separate administrator for each of the contract roles.

Thus, the name of the role “ADMIN_ROLE” is misleading and adds additional confusion in understanding the general logic, since this role does nothing in terms of role administration.

Since only the MultiSig has the ability to execute the privileged function “setCountMaturity” according to the business requirements, it would be more logical to name the role using the “MultiSig ” name.

Perhaps it is make sense to use an additional private variable instead of the role “ADMIN_ROLE” in the “EURxb” contract, which can also be initialized in the constructor, and further used to perform the required check in the “setCountMaturity” privileged function.
And here again, it would be logical to name the variable using the “MultiSig” name.

There is no check for null values of arguments in the privileged functions

Severity - Informational

The “addNewMaturity” and “removeMaturity” functions lack a check for the passed arguments, but in this case a successful call to these privileged functions is only possible from the “DDP” contract, which means that all required checks should be performed at the level of the “DDP” contract.

The atomic operation logic is split between two privileged functions: “mint” and “addNewMaturity”

Severity - Informational

According to the logic, the operations “mint” and “addNewMaturity” together represent an atomic operation.
Thus, the call to the “addNewMaturity” function should be impossible without the previously successfully completed “mint” operation.
And it is in this sequence these functions are called from the “DDP” contract in the “deposit” function.

But it would be more logical to make the “addNewMaturity” function not external, but internal in the “EURxb” contract.
And further this internal function “addNewMaturity” should be called during the executing of the “mint” function in the “EURxb” contract, because in total they represent an atomic operation.

As for the corresponding internal call from the “deposit” function of the “DDP” contract, - all that's required is to leave the call to only one external function “mint” of the “EURxb” contract.

Thus, the atomicity of these two operations will be controlled at the level of the “EURxb” contract, and not at the level of the external “DDP” contract.

The atomic operation logic is split between two privileged functions: “burn” и “removeMaturity”

Severity - Informational

According to the logic, the operations “burn” and “removeMaturity” together represent an atomic operation.
And although the call to the “removeMaturity” function should not be executed in all cases after the “burn” operation, the same rule applies here - the call to the “removeMaturity” function should be impossible without the previously successfully completed “burn” operation.
In the current implementation the check if is it required or not to execute the “removeMaturity” operation after the “burn” operation (as well as the call to the “removeMaturity” function itself if such an operation is required) is done at the level of the “DDP” contract in the “withdraw” function.

But it would be more logical to make the “removeMaturity” function not external, but internal in the “EURxb” contract.
And further the necessity of calling this internal function “removeMaturity” as well as the call to the “removeMaturity” function itself should be processed during the executing of the “burn” function in the “EURxb” contract, because in total they represent an atomic operation.

As for the corresponding internal call from the “withdraw” function of the “DDP” contract, - all that's required is to leave the call to only one external function “burn” of the “EURxb” contract.

Thus, the atomicity of these two operations will be controlled at the level of the “EURxb” contract, and not at the level of the external “DDP” contract.

Features: “accrueInterest” and “_calculateInterest” functions

The public function “accrueInterest” is designed to calculate the actual value of the “_expIndex” variable, which in turn is used to calculate the real-time interest.
This function can be called at any time and by any user (using their own gas). Thus, an unlimited ability to update the value of the “_expIndex” (the key parameter for calculating the real-time interest) is provided.
This function is also called before each minting operation, each burning operation and each token transfer operation in order to calculate and accrue real-time interest to the token holders participating in the operation (before performing the operation).

The very logic of calculating the value of the “_expIndex” variable is moved to the additional internal function “_calculateInterest” for ease of use.
This approach allows calculating the actual value of the “_expIndex” variable not only in the writable function “accrueInterest”, which stores a new value to the “_expIndex” variable, but also in the read function “balanceByTime”, which allows the calculation of actual tokens balance for the users, taking into account real-time interest.

Using pagination in the “accrueInterest” function

Severity - Low

The pagination is used during the function execution (the limit for pagination is the set value of the "_countMaturity" variable).
The pagination is required in order not to exceed the gas limit in case of a large number of time ranges in the "_list" data structure which must be taken into account when calculating the actual value of the "_expIndex" variable.
Such a situation can happen only if in the "EurxbSecurityAssetToken" contract (and, accordingly, in the "EurxbBondToken" contract), a large number of nft-tokens with the same maturity were issued in a fairly short period of time.
Thus, the maturity of all these tokens will also end almost simultaneously, which will lead to the necessity to take into account a large number of time ranges in the "_list" data structure when calculating the current value of the "_expIndex" variable.

The situation is frankly hypothetical considering that the issue of tokens in the "EurxbSecurityAssetToken" contract is controlled and executed by MultiSig .
Especially considering the fact already mentioned that the call to the "accrueInterest" function occurs at each minting operation, each burning operation and each token transfer operation, ie. as often as possible. And besides of this, the "accrueInterest" function can be called at any time by any user.

If, nevertheless, such a situation occurs, then due to the presence of pagination, it can lead to incorrect calculations of real-time interest for token holders participating in the first operation after the situation occurs ("mint", "burn", "transfer" or "transferFrom" operation).

As a result, the pagination looks redundant in the "accrueInterest" function (and in the whole contract).

Implementation violation in the logic for calculating the real-time interest in the "_calculateInterest" function

Severity - High

The "_calculateInterest" function contains the following code:

```
uint256 period = timestampNow.sub(lastAccrualTimestamp);
        if (period < 60) {
        return prevIndex;
        }
```

This code means that the calculation of actual value for the "_expIndex" variable (the key parameter for calculating the real-time interest) can execute no more than once a minute.
This definitely violates the required logic of calculating the real-time interest (in terms of business logic), which implies the ability to calculate the current token balance for users in real time.
Accordingly, it should be possible to calculate the actual value of the "_expIndex" variable at any given time.

In addition, the logic of the "accrueInterest" function, from which the "_calculateInterest" function is called, ends with the following code (after the call to the "_calculateInterest" function):

```
_accrualTimestamp = block.timestamp;
```

This means that the timestamp of the last call of the public function “accrueInterest” will always be written to the variable “_accrualTimestamp”, regardless of whether the actual value of the variable “_expIndex” was calculated or not due to the condition “if (period < 60)”.

As a result, the combination of the above “accrueInterest” function logic (line 332 in the “EURxb” contract code) with the above “_calculateInterest” function logic (lines 350-353 in the “EURxb” contract code) leads that the calculation of actual value for the “_expIndex” variable may be blocked for significantly longer time than one minute.

For instance:
User A executed the “transfer” function which in turn called other functions in the following order: “transfer” -> “_transfer” -> “accrueInterest” -> “_calculateInterest”.
As a result, the actual value for the “_expIndex” variable was calculated and the current timestamp was written to the private variable “_accrualTimestamp”.

50 seconds after the previous event, user B also executed the “transfer” function, so the same functions were called and in the same order.
But this time the calculation of actual value for the “_expIndex” variable was not executed due to the condition “if (period < 60)” in the logic of the “_calculateInterest” function.
As a result, the previous value of the “_expIndex” variable was used for real-time interest calculations.
But despite this the current timestamp was written to the private variable “_accrualTimestamp” in the logic of the “accrueInterest” function, which blocked the ability to calculate the actual value of the “_expIndex” variable for another one minute.

Those the calculation of actual value for the “_expIndex” variable can be done at best 1 minute 50 seconds after the previous calculation.
End of example.

If the “EURxb” contract is actively used by the users (e.g. the “transfer” and “transferFrom” functions), then the ability of calculating actual value for the “_expIndex” variable may be blocked even longer than in the example above.

This bug also opens up the possibility for a malicious attack, since the “accrueInterest” function is public and any user can call it at any time.
By executing this function, say, every 50 seconds, a malicious user can block the ability of calculating actual value for the “_expIndex” variable for a long time, thereby blocking the real-time interest bearing feature actually (although each such call requires payment of gas from the malicious user side).

Using pagination in the contract

Severity - Informational

As mentioned above, the “accrueInterest” function uses pagination.
And it was also concluded that the usage of pagination is redundant.

As a result, the following are redundant in the contract itself:
- private variable “_countMaturity”;
- getter-function “countMaturity”;
- setter-function “setCountMaturity”;
- the role "ADMIN_ROLE" and it's assignment and usage as well.

Features: “balanceOf” and “balanceByTime” functions

The public function “balanceOf” of the ERC20 standard returns the current tokens balance of the user, taking into account the real-time interest.
To do this, the “balanceOf” function calls another public function “balanceByTime” and passes the “block.timestamp” to it as the argument value.
The “balanceByTime” function uses the timestamp value passed as the argument to calculate the tokens balance of a user which will be actual at the specified timestamp (taking into account the real-time interest).

The timestamp check is missing in the “balanceByTime” function

Severity - Low

There is no check in the “balanceByTime” function that the passed in the argument timestamp value does not refer to the past time.
The possible check implementation:

```
require(timestamp >= block.timestamp, "timestamp is invalid");
```

Without this check, the function call will be processed as usual.
But the returned tokens balance of the user will be incorrect, since the current value of the “_expIndex” variable will be used in calculations, and not a value that was actual at the specified timestamp.

Features: used variables and data structures

The “EURxb” contract uses the following variables and data structures:

```
uint256 private _countMaturity;
uint256 private _totalActiveValue;
uint256 private _annualInterest;
uint256 private _accrualTimestamp;
uint256 private _expIndex;

mapping(address => uint256) private _holderIndex;

LinkedList.List private _list;
mapping(uint256 => uint256) private _deletedMaturity;
```

```
address private _ddp;
```

Missing getter-functions

Severity - Informational

The corresponding getter-functions are not implemented for the “_ddp” variable and for “_holderIndex” and “_deletedMaturity” data structures.

XBE

The contract codebase audit is current as of February 10, 2021

Commit: https://github.com/EURxbfinance/SmartBond/commit/b770b4b66d041633d4f40e8b53a3e5c8fe1c6b18

The “XBE” contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	4
Low	-	0
Medium	-	0
High	-	0

The shortcomings are detailed below.

ERC20 Template

The “XBE” contract fully implements the ERC20 standard interface.
For that it inherits from the widely accepted ERC20 template contract from OpenZeppelin (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol):

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

No bugs were found in the implementation of this feature.

Contract initialization - constructor

The initialization of the “XBE” contract takes place in the constructor, where:
- the passed “initialSupply” argument is used to set initial values for “_totalSupply” and the contract deployer's token balance (through a call of the “_mint” function);
- the values for the "_name" and "_symbol" variables are set;
- the contract deployer's address is set as an address of the governance account.

No check for null values of arguments

Severity - Informational

There is no check in the constructor that the passed argument is not null.
The possible check implementation:

```
require(initialSupply != uint256(0), "initialSupply amount is invalid");
```

Features: privileged operations

The “XBE” contract meets the following requirements:
- the current governance account can set the address of a new governance account;
- the current governance account can modify the membership of the minters list (add and remove accounts to/from the minters list);
- only the members of the minters list can mint additional XBE tokens.

To implement the above requirements, the “XBE” contract performs the required checks in the privileged functions “setGovernance”, “addMinter”, “removeMinter”, “mint” and sets the contract deployer address as the initial governance account (in the private variable “governance”) during the contract deployment.
It is worth noting here that the current governance account can, if necessary, transfer the privileged right to a new account - this logic is defined in the “setGovernance” privileged function.

No bugs were found in the implementation of this feature, but there are several informational notes.

No check for provided arguments

Severity - Informational

There is no check for null value for the passed address argument “_governance” in the function “setGovernance”.
The possible check implementation:

```
require(_governance != address(0), "governance address is invalid");
```

There are no checks for null values for the passed address argument “_minter” in the functions “addMinter” and “removeMinter”.
The possible check implementation:

```
require(_minter != address(0), "minter address is invalid");
```

There are no checks for null values for the passed arguments “account” and “amount” in the function “mint”.
The possible check implementation:

```
require(account != address(0), "account address is invalid");
require(amount != uint256(0), "amount is invalid");
```

Features: used variables and data structures

The “XBE” contract uses the “governance” variable and “minters” data structure.
The contract contains the corresponding getter-functions for the above variable and data structure.

Naming convention for private variables and data structures

Severity - Informational

It’s recommended to follow the naming convention for naming private variables and data structures:
- “_governance” instead of “governance”;
- “_minters” instead of “minters”.

Events for the sensitive operations

Severity - Informational

It’s recommended to emit events after the sensitive functions execution.
In this case should be logged such governance-related operations as change governance account and add or remove minter account.
So the “setGovernance”, “addMinter”, “removeMinter” functions should emit the corresponding events.
It is worth noting here that the “mint” function already emits the required event through the “_mint” function implementation in the ERC20 standard.

StakingManager

The contract codebase audit is current as of February 19, 2021

Commit:
https://github.com/EURxbfinance//SmartBond/commit/a75dcc25c598f519673fa1573664e271a57a83b0

The “StakingManager” contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	3
Low	-	1
Medium	-	0
High	-	0

The shortcomings are detailed below.

Contract initialization - constructor and “configure” function

The initial initialization of the “StakingManager” contract is divided into two steps:

In the constructor of the contract, the address of the contract “XBE” (the passed “xbg” argument) and the timestamp of the staking period beginning (the passed “startTime” argument) are set.

In the “configure” function (the function can be called once by the contract deployer), the following occurs:
- the required amount of the XBE tokens (“XBG_AMOUNT”) are transferred to the contract address. This XBE tokens amount is the total amount of rewards which will be distributed for all participating pools during the entire staking period;
- addresses of the participating Uniswap pools (USDT-EURxb, BUSD-EURxb) and Balancer pools (USDC-EURxb, DAI-EURxb) are set. Further, the amount of rewards (in XBE tokens) which is available for each day of the staking period is set for the participating pools.

No check for provided arguments

Severity - Informational

There are no checks in the constructor:
- that the passed address argument is not null.
The possible check implementation:

```
require(xbg != address(0), "xbg address is invalid");
```

- that the passed timestamp argument is not in the past.
The possible check implementation:

```
require(startTime > block.timestamp, "startTime is invalid");
```

There is no check in the “configure” function that the passed address arguments are not null.
The possible check implementation:

```
        for (uint i = 0; i < 4; ++i) {
            address pool = pools[i];
    require(pool != address(0), "pool address is invalid");
...
}
```

There is no check for contract initialization completeness before calling writable functions

Severity - Low

Since the “configure” function is responsible for initializing the “StakingManager” contract, then the writable functions “addStake” и “claimReward” must check the contract initialization completeness before executing their logic.
The initialization state is stored in the private variable “_isContractInitialized” of the “Initializable” contract, from which the “StakingManager” contract is inherited:

```
import "./templates/Initializable.sol";
```

Such a kind of check in writable functions might look like this:

```
require(_isContractInitialized, "contract has not been initialized");
```

Using an unified approach when initializing a contract

Severity - Informational

The initial initialization of the “StakingManager” contract occurs both using the contract constructor and using the “configure” function.
The same approach is used in some other contracts, while the rest of the contracts use only the constructor.

The profit of using such an approach is not explicitly visible and, most likely, this was done for the convenience of deploying contracts. But because of this, the complexity of the logic and code perception increases.
To eliminate unnecessary complexity and confusion of logic, it is recommended to use, if applicable, the unified approach for contracts initialization (for all contracts, not just for “StakingManager” contract) either using the contract constructor or using a separate initializing function (the “configure” function).

Features: “addStake” function

The “addStake” function allows any user to stake LP-tokens of one of the allowed pools (Uniswap - USDT-EURxb, BUSD-EURxb; Balancer - USDC-EURxb, DAI-EURxb).

The staking operation can be done an unlimited number of times by a user, but only during the staking period, which lasts 7 days from the beginning (the start time is the value of the private variable “_startTime”).
During the staking operation, the required amount of pool LP-tokens are transferred from the balance of the calling user to the balance of the “StakingManager” contract (the values of the “amount” and “pool” passed arguments are used).
It can be noted that the user address is also passed to the function in the “user” argument.
This address value is further used for staking accounting, but not used during transferring the required amount of pool LP-tokens from the user account to the contract account and the “_msgSender” is used instead.
Such an approach is used due to the fact that the “addStake” function can be called not by the user, but by the “Router” contract.
And in this case, the pool LP-tokens is transferred from the balance of the “Router” contract (“_msgSender”) while the value of the “user” argument is used for staking accounting thus the “Router” contract processes the staking operation on behalf of the user.
Next, the transferred amount of pool LP-tokens is taken into account in the data structures “_dailyAccumulator” (for the required pool) and “_stakes” (for the participating user).

The “StakerAdded” event is generated upon completion of the staking operation.
It is worth noting here that the event uses the value “day + 1” as a timestamp for the staking operation.
This is done for the convenience of information perception by the users.
For instance:
A staking operation was executed on the first day of the staking period. So the corresponding event contains the “1” value as the operation timestamp, although the contract itself starts counting days of the staking period from zero value. Thus the “0” is used in the contract data structures to refer to the first day of the staking period.

No bugs were found in the implementation of this feature.

Features: “claimReward” function

The “claimReward” function allows the user to finish the staking and claim their pool LP-tokens and their accrued reward in the form of XBE tokens.
It is worth noting here that for the user the staking ends at once for all pools for which the user staked. If a user staked using several pools then the user cannot finish the staking for only one of the pools.

The user can finish the staking and claim their pool LP-tokens and their accrued reward before the end of the staking period. But this is possible only after a day has passed from the execution of the staking operation by the user.
If the “claimReward” function was called by a non-staking user, the call will fail.

During the function execution the required reward in the form of XBE tokens is calculated for the user.
Since the staking ends at once for all pools for which the user staked, the calculation takes into account all the required pools.
The calculation for each pool takes into account the time of each staking operation (if there were several staking operations) since the amount of the accrued reward depends on the day of the staking period on which the user staked.
Then the pool / pools LP-tokens that belong to the user are transferred from the balance of the “StakingManager” contract to the user's balance.
After that, the accrued reward in the form of XBE tokens is transferred from the balance of the “StakingManager” contract to the user's balance.

The “StakerHasClaimedReward” event is generated upon staking ends (the corresponding LP-tokens and accrued reward were transferred to the user).

No bugs were found in the implementation of this feature.

Features: “calculateReward” function

The “calculateReward” function allows a user who has already staked to make a preliminary calculation of the owed staking reward (the owed amount of XBE tokens).
It is worth noting here that the function accepts the value of an optional argument “timestamp” as the time for which it is necessary to make a calculation of staking reward.
Thus it is possible for the user to make a preliminary calculation of their staking reward which will be accrued at the particular time of the staking period.
In this case the staking reward calculation algorithm does not differ from the algorithm used in the “claimReward” function.

No bugs were found in the implementation of this feature.

Features: “currentDay” function

The “currentDay” function allows any user to know what is the current day of the staking period.
In this case, the return value “0” means that the staking period is not active, ie. either the staking period has not yet begun, or has already ended.

It should be noted here that the value “day + 1” is used as the return value for the active staking period. This is done for the convenience of information perception by the users (as in the case of the “StakerAdded” event mentioned above).
For instance:
The function returns the “1” value for the first day of the staking period, although the contract itself starts counting days of the staking period from zero value. Thus the “0” is used in the contract data structures to refer to the first day of the staking period.

No bugs were found in the implementation of this feature.

Features: used variables and data structures

The “StakingManager” contract uses the following variables and data structures:

```
address[4] private _pools;

mapping(address => bool) private _allowListOfPools;

mapping(address => mapping(address => uint256[7])) private _stakes;

mapping(address => Accumulator[7]) private _dailyAccumulator;

IERC20 private _tokenXbg;

uint256 private _startTime;
```

The contract contains getter-functions for the above variables (“_tokenXbg” and “_startTime”), as well as getter-functions for some of the above data structures (“_pools”, “_stakes” and “_dailyAccumulator”).

Missing getter-function

Severity - Informational

The corresponding getter-function is not implemented for the “_allowListOfPools” data structure.

Router

The contract codebase audit is current as of February 22, 2021

Commit:
https://github.com/EURxbfinance//SmartBond/commit/a75dcc25c598f519673fa1573664e271a57a83b0

The “Router” contract audit results includes the defined shortcomings of the following categories:

Severity		Count
Informational	-	8
Low	-	2
Medium	-	0
High	-	2

The shortcomings are detailed below.

Contract initialization - constructor and “configure” function

The initial initialization of the “Router” contract is divided into two steps:

In the constructor of the contract, the timestamp of the staking period beginning (the passed “startTime” argument) is set. Also, the following are set:
- the address of the “StakingManager” contract (the passed argument “stakingManager”);
- the address of the USDT token contract (the passed argument “tUSDT”);
- the address of the USDC token contract (the passed argument “tUSDC”);
- the address of the BUSD token contract (the passed argument “tBUSD”);
- the address of the DAI token contract (the passed argument “tDAI”);
- the address of the EURxb token contract (the passed argument “tEURxb”);
- the address of the team pool (the passed argument “teamAddress”), to which the USDT, USDC, BUSD, DAI tokens are transferred after their exchange for the EURxb token.

In the “configure” function (the function can be called once by the contract deployer), the address of the “UniswapV2Router02” contract (the passed argument “uniswapRouter”) is set to interact with the Uniswap protocol.

No check for provided arguments

Severity - Informational

There is no check in the constructor that the passed address arguments are not null.
The possible check implementation:

```
require(stakingManager != address(0), "stakingManager address is invalid");
require(tUSDT != address(0), "tUSDT address is invalid");
require(tUSDC != address(0), "tUSDC address is invalid");
require(tBUSD != address(0), "tBUSD address is invalid");
require(tDAI != address(0), "tDAI address is invalid");
require(tEURxb != address(0), "tEURxb address is invalid");
require(teamAddress != address(0), "teamAddress address is invalid");
```

Also there is no check in the constructor that the passed timestamp argument is not in the past.
The possible check implementation:

```
require(startTime > block.timestamp, "startTime is invalid");
```

The value of the “_startTime” variable is set through a constructor parameter

Severity - Informational

The timestamp of the staking period beginning is set in the “StakingManager” contract with which the “Router” contract interacts.

The “Router” contract deployment follows after the “StakingManager” contract deployment since the “stakingManager” parameter is used in the constructor of the “Router” contract.
Thus, in the constructor of the “Router” contract, the timestamp value of the staking period beginning can be obtained from the public getter-function “startTime” of the “StakingManager” contract, and there is no need to set this value through the constructor parameter.
Also there is no check in the current implementation that this value matches the value set in the “StakingManager” contract.

There is no check for contract initialization completeness before calling writable functions

Severity - Low

Since the “configure” function is responsible for initializing the “Router” contract to interact with the Uniswap protocol, then the writable functions “setUniswapPair” и “addLiquidity” must check the contract initialization completeness before executing their logic.
The initialization state is stored in the private variable “_isContractInitialized” of the “Initializable” contract, from which the “Router” contract is inherited:

```
import "./templates/Initializable.sol";
```

Such a kind of check in writable functions might look like this:

```
require(_isContractInitialized, "contract has not been initialized");
```

Using an unified approach when initializing a contract

Severity - Informational

The initial initialization of the “Router” contract occurs both using the contract constructor and using the “configure” function.
The same approach is used in some other contracts, while the rest of the contracts use only the constructor.

The profit of using such an approach is not explicitly visible and, most likely, this was done for the convenience of deploying contracts. But because of this, the complexity of the logic and code perception increases.
To eliminate unnecessary complexity and confusion of logic, it is recommended to use, if applicable, the unified approach for contracts initialization (for all contracts, not just for “Router” contract) either using the contract constructor or using a separate initializing function (the “configure” function).

Features: privileged operations

The “Router” contract meets the following requirements:
- only the administrator can set addresses for the Balancer pools with which the contract interacts;
- only the administrator can set addresses for the Uniswap pools with which the contract interacts;
- only the administrator can change the address of the team pool (set in the contract constructor) to which USDT, USDC, BUSD, DAI tokens are transferred after they are exchanged for the EURxb token;
- only the administrator can permanently close the ability to use the “addLiquidity” function for users, but only after the staking period ends, which lasts 7 days from the beginning (value of the private variable “_startTime”).

To implement the above requirements, the “Router” contract inherits from the “Ownable” template contract from OpenZeppelin
(https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol)
and sets the contract deployer address as the administrator (in the private variable “_owner”) during the contract deployment.

External-functions “setBalancerPool”, “setUniswapPair”, “setTeamAddress”, “closeContract” use the “onlyOwner” modifier to check that the caller is the administrator.
It is worth noting here that the current administrator can, if necessary, transfer the privileged right to a new administrator - this logic is defined in the “Ownable” template contract from OpenZeppelin.

There is one bug found in the implementation of this feature, and there are several informational notes.

No check for provided arguments

Severity - Informational

There are no checks for null values for the passed address arguments “token” and “pool” in the functions “setBalancerPool” and “setUniswapPair”.

The possible check implementation:

```
require(token != address(0), "token address is invalid");
require(pool != address(0), "pool address is invalid");
```

Also in the functions “setBalancerPool” and “setUniswapPair” there are no checks that the address value in the passed “token” argument is the supported token by the contract (supported tokens are set in the constructor).
The possible check implementation:

```
require(token == _tUSDT || token == _tBUSD, "token address is invalid");
require(token == _tUSDC || token == _tDAI, "token address is invalid");
```

In the “setTeamAddress” function there is no check for null value for the passed address in the “team” argument.
The possible check implementation:

```
require(team != address(0), "team address is invalid");
```

Duplicate check in the “closeContract” function

Severity - Informational

There is an unnecessary check in the function:

```
require(block.timestamp >= _startTime, "The time has not come yet");
```

Because this condition is already checked in the previous check:

```
require(_startTime + 7 days < block.timestamp, "Time is not over");
```

Implementation violation in the business logic in the “closeContract” function

Severity - Low

When the function is called by the current administrator, the EURxb tokens are transferred from the contract balance to the balance of the current administrator:

```
uint256 balance = _tEURxb.balanceOf(address(this));
```

```
if (balance > 0) {
_tEURxb.transfer(_msgSender(), balance);
}
```

But according to the business requirements, the remainder of EURxb tokens should be transferred to the balance of the team pool (the address of which is stored in the private variable “_teamAddress”), and not to the balance of the current administrator.

Features: “addLiquidity” function

The “addLiquidity” function allows any user to perform an operation to add liquidity to one of the supported pairs / pools (Uniswap - USDT-EURxb, BUSD-EURxb; Balancer - USDC-EURxb, DAI-EURxb).

If there is a sufficient EURxb tokens amount on the balance of the “Router” contract, then:
- half of the deposited tokens amount will be exchanged for the EURxb tokens at the current rate of the required pool / pair;
- liquidity will be added to the required pool / pair in the appropriate proportions of both tokens.

In addition, if the operation of adding liquidity occurs during the staking period, then the “Router” contract will staking the LP-tokens of the required pool / pair to the “StakingManager” contract on behalf of the user.
If at the time of the liquidity addition operation the staking period has already ended, then the corresponding LP-tokens will be transferred to the user's balance.

The “addLiquidity” function uses the following internal functions to execute the above logic:
- the “_addLiquidityUniswap” function for processing an operation to add liquidity to one of the supported Uniswap pairs (USDT-EURxb, BUSD-EURxb);
- the “_addLiquidityBalancer” function for processing an operation to add liquidity to one of the supported Balancer pools (USDC-EURxb, DAI-EURxb).

No bugs were found in the implementation of this feature.

Features: “_addLiquidityUniswap” function

As mentioned above, the “_addLiquidityUniswap” function is used to process an operation to add liquidity to one of the supported Uniswap pairs (USDT-EURxb, BUSD-EURxb).

The “TransferHelper” library from Uniswap is used to perform operations with tokens (“transfer”, “transferFrom”, “approve”):

```
import "@uniswap/lib/contracts/libraries/TransferHelper.sol";
```

Thus, a more secure interaction with token contracts is implemented.

The additional internal function “_getUniswapReservesRatio” is used to determine the current exchange rate between the tokens in the corresponding Uniswap pair.

This function returns the current values of the reserves in the Uniswap pair, or returns the initializing values of the reserves if the current reserves values are zero. The initializing values of the reserves takes into account the decimals of the tokens.

The function implements:
- a calculation of the EURxb tokens amount, which at current rate is equals to the half of the tokens amount deposited by the user;
- if the EURxb tokens amount is insufficient on the “Router” contract balance, then the calculation adjusts the required amount of user's tokens (using the current rate) which will be further used in the operation of adding liquidity;
- execution of "approve" operations for the required EURxb tokens amount and for the required amount of tokens deposited by the user. Thus the “UniswapV2Router02” contract is allowed to execute "transferFrom" operations to claim these tokens;
- performing the operation of adding liquidity through the “addLiquidity” function in the “UniswapV2Router02” contract (using the appropriate proportions of both tokens);
- transferring the required amount of tokens which were exchanged for the EURxb tokens to the team pool account;
- if the staking period is active, then performing the staking operation through the “addStake” function in the “StakingManager” contract on behalf of the user. During the staking operation, an amount of LP-tokens obtained as a result of adding liquidity to the corresponding Uniswap pair is used;
- if the staking period is over, then transferring the amount of LP-tokens obtained as a result of adding liquidity to the corresponding Uniswap pair to the user's balance.

Implementation violation in the logic of working with user funds

Severity - High

If there is not enough EURxb tokens on the “Router” contract balance to exchange them for a half amount of tokens deposited by the user, then the “_addLiquidityUniswap” function calculates the required amount of user's tokens (using the current rate) which will be further used in the operation of adding liquidity:

```
if (balanceEUR <= amountEUR) {
        amountEUR = balanceEUR;
        exchangeAmount = amountEUR.mul(tokenRatio).div(eurRatio);
        emit EmptyEURxbBalance();
        }
```

But further, the unclaimed amount of the deposited user's tokens is not returned to the user's balance, and instead, this amount is transferred to the balance of the team pool:

```
uint256 routerTokenBalance = IERC20(token).balanceOf(address(this));
TransferHelper.safeTransfer(token, _teamAddress, routerTokenBalance);
```

This is a violation of the business logic implementation.
The unclaimed amount of deposited user's tokens must be returned to the user.

The possible implementation instead of the above:

```
TransferHelper.safeTransfer(token, _teamAddress, exchangeAmount);
uint256 routerTokenBalance = IERC20(token).balanceOf(address(this));
if (routerTokenBalance > 0) {
TransferHelper.safeTransfer(token, sender, routerTokenBalance);
}
```

It is worth noting here that the current and the proposed above implementations are always processed during the "_addLiquidityUniswap" function execution (without any conditions).
Thus, the proposed logic is applicable for any case, and if after processing of the user's funds there is an unclaimed amount left, then it will be returned to the user.

Using a misleading error message in the event

Severity - Informational

During the "_addLiquidityUniswap" function call, it is checked that the corresponding Uniswap pair has been set up in the "Router" contract for the token address passed in the argument to the function:

```
address pairAddress = _uniswapPairs[token];
require(pairAddress != address(0), "Unsupported token");
```

But the error message used in the check is misleading.
In this case, the token address passed in the argument, can be valid.
But at the same time, the corresponding Uniswap pair for this token might not be set in the "Router" contract, - this must be done by the contract administrator using the "setUniswapPair" function.
It is recommended to change the error message, for example, to "Pool address is invalid".

Features: "_addLiquidityBalancer" function

As mentioned above, the "_addLiquidityBalancer" function is used to process an operation to add liquidity to one of the supported Balancer pools (USDC-EURxb, DAI-EURxb).

The "TransferHelper" library from Uniswap is used to perform operations with tokens ("transfer", "transferFrom", "approve"):
Thus, a more secure interaction with token contracts is implemented.

The additional internal function "_getBalancerReservesRatio" is used to determine the current exchange rate between the tokens in the corresponding Balancer pool.
This function returns the current values of the balances in the Balancer pool, or returns the initializing values of the balances if the current balances values are zero. The initializing values of the balances takes into account the decimals of the tokens.

The function implements:

- a calculation of the EURxb tokens amount, which at current rate is equals to the half of the tokens amount deposited by the user;
- if the EURxb tokens amount is sufficient on the contract balance, then execution of “approve” operations for the required EURxb tokens amount and for the required tokens amount deposited by the user. Thus the corresponding Balancer pool contract is allowed to execute “transferFrom” operations to claim these tokens.
After that, performing the operation of adding 99% of liquidity to the Balancer pool contract through the "joinPool" function (in the form of both tokens in appropriate proportions), as well as making a transfer to the pool address of the team deposited token in the amount received as a result of exchange for EURxb token;
The remaining 1% of liquidity is returned to the caller in the form of EURxb tokens and deposited tokens.
- if the EURxb tokens amount is insufficient on the contract balance, then only the number of tokens sufficient to exchange for the remaining EURxb tokens on the balance of the contract is transferred to the balance of the Router contract.
After that, the operation of adding 99% of liquidity to the Balancer pool contract through the "joinPool" function (in the form of both tokens in appropriate proportions), as well as the transfer to the pool address of the team token contributed in the amount received as a result of exchange for EURxb tokens;
The remaining 1% of liquidity is returned to the caller in the form of EURxb tokens and deposited tokens.
- if the staking period is active, then performing the staking operation through the “addStake” function in the “StakingManager” contract on behalf of the user. During the staking operation, an amount of LP-tokens obtained as a result of adding liquidity to the corresponding Balancer pool is used;
- if the staking period is over, then transferring the amount of LP-tokens obtained as a result of adding liquidity to the corresponding Balancer pool to the user's balance.

Performing the required check not at the beginning of the logic execution

Severity - Informational

During the “_addLiquidityBalancer” function call, a check should be done that the corresponding Balancer pool has been set up in the “Router” contract for the token address passed in the argument to the function:

```
address poolAddress = _balancerPools[token];
require(poolAddress != address(0), "Invalid pool address");
```

But this check executes only when the additional internal function “_getBalancerReservesRatio” is called.
This is not entirely logical, since the following interaction with the Balancer pool occurs before the above check:

```
address poolAddress = _balancerPools[token];
IBalancerPool pool = IBalancerPool(poolAddress);
uint256 totalSupply = pool.totalSupply();
```

Implementation violation in the logic of working with user funds

Severity - High

In order to perform the operation of adding liquidity through the “joinPool” function to the Balancer pool (using the appropriate proportions of both tokens), the amount of LP-tokens which would be requested from the Balancer pool during the liquidity addition should be calculated:

```
amountBPT = totalSupply.mul(userEurAmount).div(balance);
amountBPT = amountBPT.mul(99).div(100);
```

During such a calculation, an “insurance” against possible slippage of the current rate is used (by the recommendation of the Balancer).
But using such an "insurance", the unclaimed tokens amount could be up to 2% of the total amount of the tokens deposited by the user.
This could be considered as a violation of the business logic implementation.

The Balancer recommendation refers to the case when there is a time delay between the operation of current rate calculation, the operation of LP-tokens amount calculation (which would be requested from the Balancer pool during the liquidity addition), and the operation of adding liquidity to the Balancer pool. So, the tokens exchange rate could be changed in the Balancer pool by other participants between these operations.
In this case, the Balancer's recommendation regarding the “insurance” against possible slippage of the current rate is excessive, because all operations in the “_addLiquidityBalancer” function occur within a single transaction.
Thus, it is recommended to disable the following code in the “_addLiquidityBalancer” function:

```
amountBPT = amountBPT.mul(99).div(100);
```

Also, in the current implementation, the unclaimed amount of the deposited user's tokens is not returned to the user's balance, and instead, this amount is transferred to the balance of the team pool:

```
uint256 routerTokenBalance = IERC20(token).balanceOf(address(this));
TransferHelper.safeTransfer(token, _teamAddress, routerTokenBalance);
```

This is a violation of the business logic implementation.
The unclaimed amount of deposited user's tokens must be returned to the user.
The possible implementation instead of the above:

```
 TransferHelper.safeTransfer(token, _teamAddress, exchangeAmount);
uint256 routerTokenBalance = IERC20(token).balanceOf(address(this));
if (routerTokenBalance > 0) {
TransferHelper.safeTransfer(token, sender, routerTokenBalance);
}
```

It is worth noting here that the current and the proposed above implementations are always processed during the “_addLiquidityBalancer” function execution (without any conditions).
Thus, the proposed logic is applicable for any case, and if after processing of the user's funds there is an unclaimed amount left, then it will be returned to the user.

Features: “_getTokenDecimals” function

Since tokens can have decimals other than 18, this must be taken into account when interacting with Balancer pools / Uniswap pairs.
The additional internal function “_getTokenDecimals” returns the decimals for the token passed in the argument. This function is used in the logic of “_addLiquidityUniswap” and “_addLiquidityBalancer” functions.

The approach used by Uniswap in the “TransferHelper” library was applied here for the required logic implementation.
This approach was chosen also because the public function “decimals” is not a part of the ERC20 standard and therefore is not defined in the corresponding interface “IERC20” from OpenZeppelin:

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

No bugs were found in the implementation of this feature.

Features: used variables and data structures

The “Router” contract uses the following variables and data structures:

```
address private _teamAddress;
        address private _stakingManager;
        uint256 private _startTime;
        address private _tUSDT;
        address private _tUSDC;
        address private _tBUSD;
        address private _tDAI;
        IERC20 private _tEURxb;

        IUniswapV2Router02 private _uniswapRouter;

        bool _isClosedContract = false;

        mapping(address => address) private _balancerPools;

        mapping(address => address) private _uniswapPairs;
```

The contract contains getter-functions for the above variables (“_teamAddress”, “_stakingManager”, “_startTime”, “_isClosedContract”), as well as for the data structures “_balancerPools” и “_uniswapPairs”.

Missing getter-functions

Severity - Informational

The corresponding getter-functions are not implemented for the “_tUSDT”, “_tUSDC”, “_tBUSD”, “_tDAI”, “_tEURxb”, “_uniswapRouter” variables.