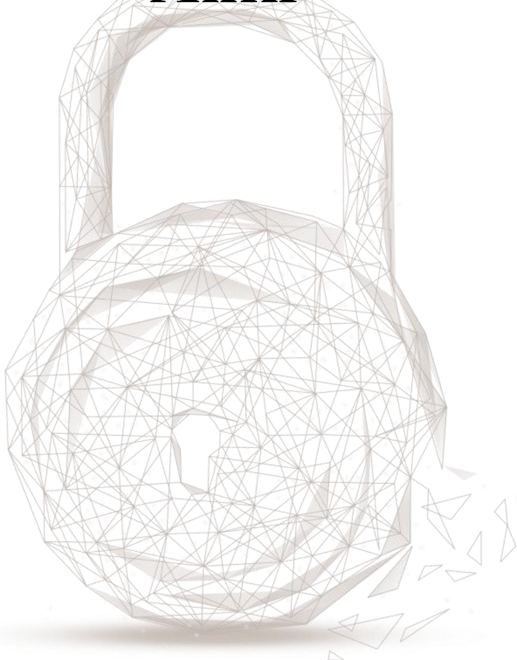
Smart Contract Audit Report

for

Ankr



Audit Number: 202203091900

Project Name: Ankr

Deployment Platform: Ethereum, BNB Chain, Polygon Chain etc.

Project Contract Address:

Contract Name	Hash(SHA256)
BridgeRouter.sol	600814254edc5a7eb20b4bd3e0e3d2aeb145990e32350a49f5e221f872124ec8
CrossChainBridge.sol	06aa7f7fff23b21758e3e7ac67f6c3f6d7f9f3370661e6ca7138766446f86755
InternetBond.sol	42678df1203c0723ffe246b3cc036dd794d2e0b78f960eafc1e3c3b0cb04e240
InternetBondProxy.sol	112649c106b937657227191a4fc7b559364721f410908a14b6d2c23af31f61a5
InternetBondRatioFeed.sol	2582e148155d6cb5196fa6f578c804964e2bccc91da2b97f3b9040d198983bd5
SimpleToken.sol	b21a2614caeae2eea534a4aca87df5127b7b057d194993d01564ddd3b71b24d1
SimpleTokenProxy.sol	d495e84b6f6c409a2dac3a587c8c514395209228954555b73a0109003e0cb381
CallDataRLPReader.sol	8c29613da2ac1f843fa58c105e24a4da13c0e62b973a87412c74577d4493d4d6
EthereumVerifier.sol	4571177b308a052116086d778eb7e1e7a49a3ebd08c2cf85e3b4e1676f1dca1e
ProofParser.sol	dfb87255aa6b3f40d11a09f1d9aa9969f52e22b01cf3b83f9279264b9207b5d9
Utils.sol	eb02eb8146b1f15581f24648594995a484266c8f878467e131d8498a2b42f7ab

Audit Start Date: 2022.02.16

Audit Completion Date: 2022.03.09

Audit Team: Beosin Technology Co. Ltd.

Audit Results Overview

Beosin Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of Ankr project, including Coding Conventions, General Vulnerability and Business Security. **After auditing, the Ankr project was found to have 2 Critical-risks, 2 Medium-risks, 1 Low-risk and 6 Info items.** The following is the detailed audit information for this project.

Index	Risk items	Risk level	Status
CrossChainBridge-1	The *deposit* function lacks a judgment on the fromToken address	Critical	Fixed
CrossChainBridge-2	The *withdraw* function is improperly designed	Critical	Fixed
CrossChainBridge-3	The *_depositErc20* function is improperly designed	Medium	Fixed
CrossChainBridge-4	Missing function of *_nativeMetaData* function	Low	Fixed
CrossChainBridge-5	Event triggering and zero address checking are not performed in some functions in the CrossChainBridge contract	Info	Fixed
CrossChainBridge-6	Error message exception of require in *_peggedDestinationErc20Token* and *_peggedDestinationErc20Bond* functions	Info	Fixed
CrossChainBridge-7	Unused pause and unpause functions	Info	Fixed
CrossChainBridge-8	Redundant code	Info	Fixed
InternetBond-1	Design flaws of *increaseAllowance* and *decreaseAllowance* functions	Medium	Fixed
InternetBond-2	The algorithms used in the *_sharesToBonds* and *_bondsToShares* functions are different	Info	Acknowledged
InternetBondRatioFeed-1	The *addOperator* function does not perform event triggering and zero address checking	Info	Fixed

Table 1 – Key Audit Findings

Risk description:

- Item InternetBond-2 is not fixed and may cause event trigger errors.

Findings

[CrossChainBridge-1 Critical] The *deposit* function lacks a judgment on the fromToken address

Description: The contract provides the *deposit* function for deposit tokens. In the *deposit* function, the validity of the fromToken input by the caller is not judged, when cross-chain is performed from the Peg token on the A chain to the Peg token on the B chain through the CrossChainBridge contract, the attacker can exploit malicious fromToken to forge, so as to mint the correct Peg token on the B chain, and then use the Peg token on the B chain to cross-chain into the origin token on the C chain.

```
    function deposit(address fromToken, uint256 toChain, address toAddress, uint256 amount) public nonReentrant override {
        (uint256 chain, address origin) = getOrigin(fromToken);
        if (chain != 0) {
            /* if we have pegged contract then its pegged token */
            _depositPegged(fromToken, toChain, toAddress, amount, chain, origin);
        } else {
            /* otherwise its erc20 token, since we can't detect is it erc20 token it can only return insufficient balance in case of any errors */
            _depositErc20(fromToken, toChain, toAddress, amount);
        }
    }
```

Figure 1 source code of *deposit* function

```
    function _depositPegged(address fromToken, uint256 toChain, address toAddress, uint256 totalAmount, uint256 chain, address origin) in
        /* sender is our from address because he is locking funds */
        address fromAddress = address(msg.sender);
        /* check allowance and transfer tokens */
        require(IERC20Upgradeable(fromToken).balanceOf(fromAddress) >= totalAmount, "insufficient balance");
        InternetBondType bondType = getBondType(fromToken);
        uint256 amt;
        if (bondType == InternetBondType.REBASING_BOND) {
            amt = _peggedAmountToShares(totalAmount, getRatio(origin));
        } else {
            amt = totalAmount;
        }
        address toToken;
        if (bondType == InternetBondType.NOT_BOND) {
            toToken = _peggedDestinationErc20Token(fromToken, origin, toChain, chain);
        } else {
            toToken = _peggedDestinationErc20Bond(fromToken, origin, toChain, chain);
        }
        IERC20Mintable(fromToken).burn(fromAddress, amt);
        Metadata memory metaData = Metadata(
            Utils.stringToBytes32(IERC20Extra(fromToken).symbol()),
            Utils.stringToBytes32(IERC20Extra(fromToken).name()),
            chain,
            origin,
            createBondMetadata(0, bondType)
        );
        /* emit event with all these params */
        emit DepositBurned(
            Utils.currentChain(), // just to mix hash of deposit event
            fromAddress, // who send these funds
            toAddress, // who can claim these funds in "toChain" network
            fromToken, // this is our current native token (can be ETH, CLV, DOT, BNB or something else)
            toToken, // this is an address of our target pegged token
            amt, // how much funds was locked in this contract
            metaData,
            origin
        );
    }
```

Figure 2 source code of *_depositPegged* function

```
function _peggedDestinationErc20Token(address fromToken, address origin, uint256 toChain, uint originChain) internal view returns (address) {
    /* lets determine target bridge contract */
    address toBridge = _bridgeAddressByChainId[toChain];
    require(toBridge != address(0x00), "bad chain");
    if (toChain == originChain) {
        require(_peggedTokens[fromToken] != address(0x00), "pegged contract not supported");
        return _peggedTokens[fromToken];
    } else {
        return _bridgeRouter.peggedTokenAddress(address(toBridge), origin);
    }
}

function _peggedDestinationErc20Bond(address fromToken, address origin, uint256 toChain, uint originChain) internal view returns (address) {
    /* lets determine target bridge contract */
    address toBridge = _bridgeAddressByChainId[toChain];
    require(toBridge != address(0x00), "bad chain");
    if (toChain == originChain) {
        require(_peggedTokens[fromToken] != address(0x00), "pegged contract not supported");
        return _peggedTokens[fromToken];
    } else {
        return _bridgeRouter.peggedBondAddress(address(toBridge), origin);
    }
}
```

Figure 3 source code of *_peggedDestinationErc20Token&_peggedDestinationErc20Bond* functions (Unfixed)

Fix recommendations: It is recommended to judge the legitimacy of the fromToken input by the user.

Status: Fixed.

```
function _peggedDestinationErc20Token(address fromToken, address origin, uint256 toChain, uint originChain) internal view returns (address) {
    /* lets determine target bridge contract */
    address toBridge = _bridgeAddressByChainId[toChain];
    require(toBridge != address(0x00), "bad chain");
    require(_peggedTokenOrigin[fromToken] == origin, "non-pegged contract not supported");
    if (toChain == originChain) {
        return _peggedTokenOrigin[fromToken];
    } else {
        return _bridgeRouter.peggedTokenAddress(address(toBridge), origin);
    }
}

function _peggedDestinationErc20Bond(address fromToken, address origin, uint256 toChain, uint originChain) internal view returns (address) {
    /* lets determine target bridge contract */
    address toBridge = _bridgeAddressByChainId[toChain];
    require(toBridge != address(0x00), "bad chain");
    require(_peggedTokenOrigin[fromToken] == origin, "non-pegged contract not supported");
    if (toChain == originChain) {
        return _peggedTokenOrigin[fromToken];
    } else {
        return _bridgeRouter.peggedBondAddress(address(toBridge), origin);
    }
}
```

Figure 4 source code of *_peggedDestinationErc20Token&_peggedDestinationErc20Bond* functions (Fixed)

[CrossChainBridge-2 Critical] The *withdraw* function is improperly designed

Description: The *withdraw* function in the CrossChainBridge contract lacks a determination of the state.chainId in the event, which will lead to a double-spending attack.

```solidity
function withdraw(
    bytes calldata /* encodedProof */,
    bytes calldata rawReceipt,
    bytes memory proofSignature
) external nonReentrant whenNotPaused override {
    uint256 proofOffset;
    uint256 receiptOffset;
    assembly {
        proofOffset := add(0x4, calldataload(4))
        receiptOffset := add(0x4, calldataload(36))
    }
    /* we must parse and verify that tx and receipt matches */
    (EthereumVerifier.State memory state, EthereumVerifier.PegInType pegInType) = EthereumVerifier.parseTransactionReceipt(receiptOffset);
    ProofParser.Proof memory proof = ProofParser.parseProof(proofOffset);
    require(_bridgeAddressByChainId[proof.chainId] == state.contractAddress, "crosschain event from not allowed contract");
    state.receiptHash = keccak256(rawReceipt);
    proof.status = 0x01; // execution must be successful
    proof.receiptHash = state.receiptHash; // ensure that rawReceipt is preimage of receiptHash
    bytes32 hash;
    assembly {
        hash := keccak256(proof, 0x100)
    }
    // we can trust receipt only if proof is signed by consensus
    require(ECDSAUpgradeable.recover(hash, proofSignature) == _consensusAddress, "bad signature");
    // withdraw funds to recipient
    _withdraw(state, pegInType, hash);
}
```

Figure 5 source code of *withdraw* function (Unfixed)

Fix recommendations: It is recommended to judge state.chainId.

Status: Fixed.

```solidity
function withdraw(
    bytes calldata /* encodedProof */,
    bytes calldata rawReceipt,
    bytes memory proofSignature
) external nonReentrant whenNotPaused override {
    uint256 proofOffset;
    uint256 receiptOffset;
    assembly {
        proofOffset := add(0x4, calldataload(4))
        receiptOffset := add(0x4, calldataload(36))
    }
    /* we must parse and verify that tx and receipt matches */
    (EthereumVerifier.State memory state, EthereumVerifier.PegInType pegInType) = EthereumVerifier.parseTransactionReceipt(receiptOffset);
    require(state.chainId == Utils.currentChain(), "receipt points to another chain");
    ProofParser.Proof memory proof = ProofParser.parseProof(proofOffset);
    require(_bridgeAddressByChainId[proof.chainId] == state.contractAddress, "crosschain event from not allowed contract");
    state.receiptHash = keccak256(rawReceipt);
    proof.status = 0x01; // execution must be successful
    proof.receiptHash = state.receiptHash; // ensure that rawReceipt is preimage of receiptHash
    bytes32 hash;
    assembly {
        hash := keccak256(proof, 0x100)
    }
    // we can trust receipt only if proof is signed by consensus
    require(ECDSAUpgradeable.recover(hash, proofSignature) == _consensusAddress, "bad signature");
    // withdraw funds to recipient
    _withdraw(state, pegInType, hash);
}
```

Figure 6 source code of *withdraw* function (Fixed)

[CrossChainBridge-3 Medium] The *_depositErc20* function is improperly designed

Description: In order to avoid adding fee-on-transfer tokens in the *_depositErc20* function, the sender address is incorrectly used to judge whether the fromToken is a fee-on-transfer token based on the balance before and after the transfer. Here, it should be judged by the balance before and after the transfer of the receiving address to determine whether fromToken is fee-on-transfer token.

```
function _depositErc20(address fromToken, uint256 toChain, address toAddress, uint256 totalAmount) internal {
    /* sender is our from address because he is locking funds */
    address fromAddress = address(msg.sender);
    InternetBondType bondType = getBondType(fromToken);
    /* check allowance and transfer tokens */
    {
        uint256 balanceBefore = IERC20(fromToken).balanceOf(fromAddress);
        require(totalAmount <= balanceBefore, "insufficient balance");
        uint256 allowance = IERC20(fromToken).allowance(fromAddress, address(this));
        require(totalAmount <= allowance, "insufficient allowance");
        require(IERC20(fromToken).transferFrom(fromAddress, address(this), totalAmount), "can't transfer");
        uint256 balanceAfter = IERC20(fromToken).balanceOf(fromAddress);
        if (bondType != InternetBondType.REBASING_BOND) {
            require(balanceBefore - totalAmount == balanceAfter, "incorrect behaviour");
        } else {
            // For rebasing internet bonds we can't assert that exactly totalAmount will be transferred
            require(balanceBefore >= balanceAfter, "incorrect behaviour");
        }
    }
```

Figure 7 source code of *_depositErc20* function (Unfixed)

Fix recommendations: It is recommended to replace fromAddress with address(this) and modify the corresponding logic.

Status: Fixed.

```
function _depositErc20(address fromToken, uint256 toChain, address toAddress, uint256 totalAmount) internal {
    /* sender is our from address because he is locking funds */
    address fromAddress = address(msg.sender);
    InternetBondType bondType = getBondType(fromToken);
    /* check allowance and transfer tokens */
    {
        uint256 balanceBefore = IERC20(fromToken).balanceOf(address(this));
        uint256 allowance = IERC20(fromToken).allowance(fromAddress, address(this));
        require(totalAmount <= allowance, "insufficient allowance");
        require(IERC20(fromToken).transferFrom(fromAddress, address(this), totalAmount), "can't transfer");
        uint256 balanceAfter = IERC20(fromToken).balanceOf(address(this));
        if (bondType != InternetBondType.REBASING_BOND) {
            // Assert that enough coins were transferred to bridge
            require(balanceAfter >= balanceBefore + totalAmount, "incorrect behaviour");
        } else {
            // For rebasing internet bonds we can't assert that exactly totalAmount will be transferred
            require(balanceAfter >= balanceBefore, "incorrect behaviour");
        }
    }
```

Figure 8 source code of *_depositErc20* function (Fixed)

[CrossChainBridge-4 Low] Missing function of *_nativeMetaData* function

Description: The *__CrossChainBridge_init* function in the CrossChainBridge contract determines "nativeAddress == ADDRESS_MATIC", while in the *_nativeMetaData* function there is no "fromToken == ADDRESS_MATIC" selection.

```
function __CrossChainBridge_init(address consensusAddress, SimpleTokenFactory tokenFactory, InternetBondFactory bondFactory, address nativeAddress,
    require(
        nativeAddress == ADDRESS_ETHEREUM ||
        nativeAddress == ADDRESS_BSC ||
        nativeAddress == ADDRESS_CLOVER ||
        nativeAddress == ADDRESS_MATIC,
        "bad address"
    );
    _consensusAddress = consensusAddress;
    _tokenImplementation = tokenFactory.getImplementation();
    _bondImplementation = bondFactory.getImplementation();
    _nativeAddress = nativeAddress;
    _internetBondRatioFeed = bondFeed;
    _bridgeRouter = router;
}
```

Figure 9 source code of *__CrossChainBridge_init* function

```
function _nativeMetaData(address fromToken) internal view returns (Metadata memory) {
    uint256 chain = Utils.currentChain();
    if (fromToken == ADDRESS_ETHEREUM) {
        return Metadata(Utils.stringToBytes32("ETH"), Utils.stringToBytes32("Ethereum"), chain, fromToken, 0x0);
    } else if (fromToken == ADDRESS_BSC) {
        return Metadata(Utils.stringToBytes32("BNB"), Utils.stringToBytes32("Binance"), chain, fromToken, 0x0);
    } else if (fromToken == ADDRESS_CLOVER) {
        return Metadata(Utils.stringToBytes32("CLV"), Utils.stringToBytes32("Clover"), chain, fromToken, 0x0);
    }
    revert("bad token");
}
```

Figure 10 source code of *_nativeMetaData* function

Fix recommendations: It is recommended to increase the option of "fromToken == ADDRESS_MATIC" in the *_nativeMetaData* function.

Status: Fixed. Project party's description: In previous revision of smart contract they used to track allowed "native addresses" in crosschain bridge contract. They made an observation that they do not need to register "native addresses" for all chains in each bridge contract. It's sufficient for each bridge to only know it's own "native address". They made changes to calculate this native address (as well as all the other fileds of native asset Metadata structure) right in the contract initializer.

```
    function __CrossChainBridge_init(
        address consensusAddress,
        SimpleTokenFactory tokenFactory,
        InternetBondFactory bondFactory,
        string memory nativeTokenSymbol,
        string memory nativeTokenName,
        InternetBondRatioFeed bondFeed,
        BridgeRouter router
    ) internal {
        _consensusAddress = consensusAddress;
        _tokenImplementation = tokenFactory.getImplementation();
        _bondImplementation = bondFactory.getImplementation();
        _nativeTokenMetadata = Metadata(
            Utils.stringToBytes32(nativeTokenSymbol),
            Utils.stringToBytes32(nativeTokenName),
            Utils.currentChain(),
            // generate unique address that will not collide with any contract address
            address(bytes20(keccak256(abi.encodePacked("CrossChainBridge:", nativeTokenSymbol)))),
            0x0
        );
        _internetBondRatioFeed = bondFeed;
        _bridgeRouter = router;
    }
```

Figure 11 source code of *__CrossChainBridge_init* function

[CrossChainBridge-5 Info] Event triggering and zero address checking are not performed in some functions in the CrossChainBridge contract

Description: The *changeConsensus*, *changeRouter*, *setTokenFactory* and *setBondFactory* functions in the CrossChainBridge contract lack zero address checking and event triggering.

```
    function changeConsensus(address consensus) public onlyOwner {
        _consensusAddress = consensus;
    }

    function changeRouter(address router) public onlyOwner {
        _bridgeRouter = BridgeRouter(router);
    }
}
```

Figure 12 source code of *changeConsensus&changeRouter* functions (Unfixed)

```
    function setTokenFactory(SimpleTokenFactory factory) public onlyOwner {
        _tokenImplementation = factory.getImplementation();
    }

    function getBondImplementation() public view override returns (address) {
        return _bondImplementation;
    }

    function setBondFactory(InternetBondFactory factory) public onlyOwner {
        _bondImplementation = factory.getImplementation();
    }
```

Figure 13 source code of *setTokenFactory&setBondFactory* functions (Unfixed)

Fix recommendations: It is recommended to add event triggering and zero address checking.

Status: Fixed.

```
    function changeConsensus(address consensus) public onlyOwner {
        require(consensus != address(0x0), "zero address disallowed");
        _consensusAddress = consensus;
        emit ConsensusChanged(_consensusAddress);
    }

    function changeRouter(address router) public onlyOwner {
        require(router != address(0x0), "zero address disallowed");
        _bridgeRouter = BridgeRouter(router);
        // We don't have special event for router change since it's very special technical c
        // In future changing router will be disallowed
    }
}
```

Figure 14 source code of *changeConsensus&changeRouter* functions (Fixed)

```
    function setTokenFactory(SimpleTokenFactory factory) public onlyOwner {
        _tokenImplementation = factory.getImplementation();
        require(_tokenImplementation != address(0x0));
        emit TokenImplementationChanged(_tokenImplementation);
    }

    function getBondImplementation() public view override returns (address) {
        return _bondImplementation;
    }

    function setBondFactory(InternetBondFactory factory) public onlyOwner {
        _bondImplementation = factory.getImplementation();
        require(_bondImplementation != address(0x0));
        emit BondImplementationChanged(_tokenImplementation);
    }
```

Figure 15 source code of *setTokenFactory&setBondFactory* functions (Fixed)

[CrossChainBridge-6 Info] Error message exception of require in *_peggedDestinationErc20Token* and *_peggedDestinationErc20Bond* functions

Description: There is an error message exception of require in *_peggedDestinationErc20Token* and *_peggedDestinationErc20Bond* functions of CrossChainBridge contract. The require checks if the fromToken address is a pegged Tokens contract or not. When it is not a pegged Tokens contract, the error message is incorrect.

```
function _peggedDestinationErc20Token(address fromToken, address origin, uint256 toChain, uint originChain) internal view returns (address) {
    /* lets determine target bridge contract */
    address toBridge = _bridgeAddressByChainId[toChain];
    require(toBridge != address(0x00), "bad chain");
    if (toChain == originChain) {
        require(_peggedTokens[fromToken] != address(0x00), "pegged contract not supported");
        return _peggedTokens[fromToken];
    } else {
        return _bridgeRouter.peggedTokenAddress(address(toBridge), origin);
    }
}

function _peggedDestinationErc20Bond(address fromToken, address origin, uint256 toChain, uint originChain) internal view returns (address) {
    /* lets determine target bridge contract */
    address toBridge = _bridgeAddressByChainId[toChain];
    require(toBridge != address(0x00), "bad chain");
    if (toChain == originChain) {
        require(_peggedTokens[fromToken] != address(0x00), "pegged contract not supported");
        return _peggedTokens[fromToken];
    } else {
        return _bridgeRouter.peggedBondAddress(address(toBridge), origin);
    }
}
```

Figure 16 source code of *_peggedDestinationErc20Token&_peggedDestinationErc20Bond* functions (Unfixed)

Fix recommendations: It is recommended to change the error message to 'non-pegged contract not supported'.

Status: Fixed.

```
function _peggedDestinationErc20Token(address fromToken, address origin, uint256 toChain, uint originChain) internal view returns (address) {
    /* lets determine target bridge contract */
    address toBridge = _bridgeAddressByChainId[toChain];
    require(toBridge != address(0x00), "bad chain");
    require(_peggedTokenOrigin[fromToken] == origin, "non-pegged contract not supported");
    if (toChain == originChain) {
        return _peggedTokenOrigin[fromToken];
    } else {
        return _bridgeRouter.peggedTokenAddress(address(toBridge), origin);
    }
}

function _peggedDestinationErc20Bond(address fromToken, address origin, uint256 toChain, uint originChain) internal view returns (address) {
    /* lets determine target bridge contract */
    address toBridge = _bridgeAddressByChainId[toChain];
    require(toBridge != address(0x00), "bad chain");
    require(_peggedTokenOrigin[fromToken] == origin, "non-pegged contract not supported");
    if (toChain == originChain) {
        return _peggedTokenOrigin[fromToken];
    } else {
        return _bridgeRouter.peggedBondAddress(address(toBridge), origin);
    }
}
```

Figure 17 source code of *_peggedDestinationErc20Token&_peggedDestinationErc20Bond* functions (Fixed)

[CrossChainBridge-7 Info] Unused *pause* and *unpause* functions

Description: *Pause* and *unpause* functions are not used in CrossChainBridge contracts.

```
    function pause() public onlyOwner {
        _pause();
    }

    function unpause() public onlyOwner {
        _unpause();
    }
```

Figure 18 source code of *pause* and *unpause* functions

Fix recommendations: It is recommended to add whenNotPaused modifier to the relevant function.

Status: Fixed.

```
    function deposit(uint256 toChain, address toAddress) public payable nonReentrant whenNotPaused override {
        _depositNative(toChain, toAddress, msg.value);
    }

    function deposit(address fromToken, uint256 toChain, address toAddress, uint256 amount) public nonReentrant whenNotPaused override {
        (uint256 chain, address origin) = getOrigin(fromToken);
        if (chain != 0) {
            /* if we have pegged contract then its pegged token */
            _depositPegged(fromToken, toChain, toAddress, amount, chain, origin);
        } else {
            /* otherwise its erc20 token, since we can't detect is it erc20 token it can only return insufficient balance in case of any errors */
            _depositErc20(fromToken, toChain, toAddress, amount);
        }
    }
```

Figure 19 source code of related function

[CrossChainBridge-8 Info] Redundant code

Description: The *withdrawNotarized* function in the CrossChainBridge contract has no practical significance and is redundant code.

```
    function withdrawNotarized(bytes calldata /*rawReceipt*/, bytes calldata /*signature*/) public nonReentrant override {
        revert("not implemented");
        //        // we must parse and verify that tx and receipt matches
        //        (EthereumVerifier.State memory state, EthereumVerifier.PegInType pegInType) = EthereumVerifier.parseTransactionReceipt(rawReceipt);
        //        // make sure this receipt is notarized by consensus
        //        require(ECDSAUpgradeable.recover(state.receiptHash, signature) == _consensusAddress, "bad signature");
        //        // withdraw funds to recipient
        //        _withdraw(state, pegInType);
    }
```

Figure 20 source code of *withdrawNotarized* function

Fix recommendations: It is recommended to remove redundant code.

Status: Fixed.

[InternetBond-1 Medium] Design flaws of *increaseAllowance* and *decreaseAllowance* functions

Description: In the *increaseAllowance* and *decreaseAllowance* functions of the InternetBond contract, the value of amount is not converted to shares, which will lead to an error in the caller authorization than the actual authorization.

```solidity
function approve(address spender, uint256 amount) public override returns (bool) {
    uint256 shares = _bondsToShares(amount);
    _approve(_msgSender(), spender, shares, false);
    emit Approval(_msgSender(), spender, allowance(_msgSender(), spender));
    return true;
}

function increaseAllowance(address spender, uint256 amount) public override returns (bool) {
    _increaseAllowance(_msgSender(), spender, amount, false);
    emit Approval(_msgSender(), spender, allowance(_msgSender(), spender));
    return true;
}

function decreaseAllowance(address spender, uint256 amount) public override returns (bool) {
    _decreaseAllowance(_msgSender(), spender, amount, false);
    emit Approval(_msgSender(), spender, allowance(_msgSender(), spender));
    return true;
}
```

Figure 21 source code of *increaseAllowance and decreaseAllowance* functions (Unfixed)

Fix recommendations: It is recommended to convert the value of amount to shares.

Status: Fixed.

```solidity
function increaseAllowance(address spender, uint256 amount) public override returns (bool) {
    uint256 shares = _bondsToShares(amount);
    _increaseAllowance(_msgSender(), spender, shares, false);
    emit Approval(_msgSender(), spender, allowance(_msgSender(), spender));
    return true;
}

function decreaseAllowance(address spender, uint256 amount) public override returns (bool) {
    uint256 shares = _bondsToShares(amount);
    _decreaseAllowance(_msgSender(), spender, shares, false);
    emit Approval(_msgSender(), spender, allowance(_msgSender(), spender));
    return true;
}
```

Figure 22 source code of *increaseAllowance and decreaseAllowance* functions (Fixed)

[InternetBond-2 Info] The algorithms used in the *_sharesToBonds* and *_bondsToShares* functions are different

Description: In the *_sharesToBonds* and *_bondsToShares* functions of the InternetBond contract, different algorithms are used. After testing, the error of the two algorithms is not more than 1, and the event may trigger an error when the *transferFrom* function is called to transfer funds.

```
function _sharesToBonds(uint256 amount) internal view returns (uint256) {
    if (_rebasing) {
        uint256 currentRatio = ratio();
        require(currentRatio > 0, "ratio not available");
        return Utils.multiplyAndDivideCeil(amount, 10 ** decimals(), currentRatio);
    } else {
        return amount;
    }
}

function _bondsToShares(uint256 amount) internal view returns (uint256) {
    if (_rebasing) {
        uint256 currentRatio = ratio();
        require(currentRatio > 0, "ratio not available");
        return Utils.multiplyAndDivideFloor(amount, currentRatio, 10 ** decimals());
    } else {
        return amount;
    }
}
```

Figure 23 source code of *_sharesToBonds and _bondsToShares* functions

```
function transferFrom(address sender, address recipient, uint256 amount) public override returns (bool) {
    uint256 shares = _bondsToShares(amount);
    _transfer(sender, recipient, shares, false);
    emit Transfer(sender, recipient, _sharesToBonds(shares));
    _decreaseAllowance(sender, _msgSender(), shares, false);
    emit Approval(sender, _msgSender(), allowance(sender, _msgSender()));
    return true;
}
```

Figure 24 source code of *transferFrom* function

Fix recommendations: It is recommended to use the same algorithm.

Status: Acknowledged.

[InternetBondRatioFeed-1 Info] The *addOperator* function does not perform event triggering and zero address checking

Description: The *addOperator* function in the InternetBondRatioFeed contract does not perform event triggering and zero address checking.

```
function addOperator(address operator) public onlyOwner {
    _isOperator[operator] = true;
}
```

Figure 25 source code of *addOperator* function (Unfixed)

Fix recommendations: It is recommended to add event triggering and zero address checking to the *addOperator* function.

Status: Fixed.

```
function addOperator(address operator) public onlyOwner {
    require(operator != address(0x0), "operator must be non-zero");
    require(!_isOperator[operator], "already operator");
    _isOperator[operator] = true;
    emit OperatorAdded(operator);
}
```

Figure 26 source code of *addOperator* function (Fixed)

Other Audit Items Descriptions

1. Other audit recommendations

- In the SimpleToken and InternetBond contracts, beware that when the user calls the *approve* function to modify the authorization value, it may cause multiple authorizations. Using function *'increaseAllowance'* and *'decreaseAllowance'* to alter allowance is recommended.
- Note that offline cross-chain scripts are not included in the scope of this audit, so the security of the entire cross-chain project cannot be guaranteed.

2. Description from the project party

- Proof notarization is done by Ankr protocol which uses threshold ECDSA signatures with private key shares distributed among protocol members. That's why from blockchain perspective it looks like single address. However private key from this address does not exist anywhere in one party hands so essentially consensusAddress address is already multisig.
- Following roadmap is suggested for bridge owner. First deploy: owner is private key held in cold storage, after some time ownership will be renounced in favor of Ankr governance contract, when supported ownership might be passed to Ankr protocol key.
- Don't care about ratio at all and sync it when one of cross chain event happen,It's very, very cheap but for tokens like aETHb balance won't update automatically every block or every day, some lags might happen (or we can trigger dummy cross chain operation just to sync ratio).Use their backend services to distribute ratio to all chains using ratio feeds with batch send. It won't be very expensive to distribute ratio for all chains since they do it in a batch transaction, but they still need to send all ratios to all chains (it requires almost N^2 data to be sent). Since ratio feeds can be injected directly in Internet Bond token template and cross chain contract its not very complicated to integrate it.

Appendix 1 Vulnerability Severity Level and Status Description

- Vulnerability Severity Level

Vulnerability Level	Description	Example
Critical	Vulnerabilities that lead to the complete destruction of the project and cannot be recovered. It is strongly recommended to fix.	Malicious tampering of core contract privileges and theft of contract assets.
High	Vulnerabilities that lead to major abnormalities in the operation of the contract due to contract operation errors. It is strongly recommended to fix.	Unstandardized docking of the USDT interface, causing the user's assets to be unable to withdraw.
Medium	Vulnerabilities that cause the contract operation result to be inconsistent with the design but will not harm the core business. It is recommended to fix.	The rewards that users received do not match expectations.
Low	Vulnerabilities that have no impact on the operation of the contract, but there are potential security risks, which may affect other functions. The project party needs to confirm and determine whether the fix is needed according to the business scenario as appropriate.	Inaccurate annual interest rate data queries.
Info	There is no impact on the normal operation of the contract, but improvements are still recommended to comply with widely accepted common project specifications.	It is needed to trigger corresponding events after modifying the core configuration.

- Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

Appendix 2 Description of Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
3	Business Security	Business Logics
		Business Implementations

1. Coding Conventions

1.1. Compiler Version Security

The old version of the compiler may cause various known security issues. Developers are advised to specify the contract code to use the latest compiler version and eliminate the compiler alerts.

1.2. Deprecated Items

The Solidity smart contract development language is in rapid iteration. Some keywords have been deprecated by newer versions of the compiler, such as throw, years, etc. To eliminate the potential pitfalls they may cause, contract developers should not use the keywords that have been deprecated by the current compiler version.

1.3. Redundant Code

Redundant code in smart contracts can reduce code readability and may require more gas consumption for contract deployment. It is recommended to eliminate redundant code.

1.4. SafeMath Features

Check whether the functions within the SafeMath library are correctly used in the contract to perform mathematical operations, or perform other overflow prevention checks.

1.5. require/assert Usage

Solidity uses state recovery exceptions to handle errors. This mechanism will undo all changes made to the state in the current call (and all its subcalls) and flag the errors to the caller. The functions assert and require can be used to check conditions and throw exceptions when the conditions are not met. The assert function can only be used to test for internal errors and check non-variables. The require function is used to confirm the validity of conditions, such as whether the input variables or contract state variables meet the conditions, or to verify the return value of external contract calls.

1.6. Gas Consumption

The smart contract virtual machine needs gas to execute the contract code. When the gas is insufficient, the code execution will throw an out of gas exception and cancel all state changes. Contract developers are required to control the gas consumption of the code to avoid function execution failures due to insufficient gas.

1.7. Visibility Specifiers

Check whether the visibility conforms to design requirement.

1.8. Fallback Usage

Check whether the Fallback function has been used correctly in the current contract.

2. General Vulnerability

2.1. Integer overflow

Integer overflow is a security problem in many languages, and they are especially dangerous in smart contracts. Solidity can handle up to 256-bit numbers (2**256-1). If the maximum number is increased by 1, it will overflow to 0. Similarly, when the number is a uint type, 0 minus 1 will underflow to get the maximum number value. Overflow conditions can lead to incorrect results, especially if its possible results are not expected, which may affect the reliability and safety of the program. For the compiler version after Solidity 0.8.0, smart contracts will perform overflow checking on mathematical operations by default. In the previous compiler versions, developers need to add their own overflow checking code, and SafeMath library is recommended to use.

2.2. Reentrancy

The reentrancy vulnerability is the most typical Ethereum smart contract vulnerability, which has caused the DAO to be attacked. The risk of reentry attack exists when there is an error in the logical order of calling the call.value() function to send assets.

2.3 Pseudo-random Number Generator (PRNG)

Random numbers may be used in smart contracts. In solidity, it is common to use block information as a random factor to generate, but such use is insecure. Block information can be controlled by miners or obtained by attackers during transactions, and such random numbers are to some extent predictable or collidable.

2.4. Transaction-Ordering Dependence

In the process of transaction packing and execution, when faced with transactions of the same difficulty, miners tend to choose the one with higher gas cost to be packed first, so users can specify a higher gas cost to have their transactions packed and executed first.

2.5. DoS(Denial of Service)

DoS, or Denial of Service, can prevent the target from providing normal services. Due to the immutability of smart contracts, this type of attack can make it impossible to ever restore the contract to its normal working state. There are various reasons for the denial of service of a smart contract, including malicious revert when acting as the recipient of a transaction, gas exhaustion caused by code design flaws, etc.

2.6. Function Call Permissions

If smart contracts have high-privilege functions, such as coin minting, self-destruction, change owner, etc., permission restrictions on function calls are required to avoid security problems caused by permission leakage.

2.7. call/delegatecall Security

Solidity provides the call/delegatecall function for function calls, which can cause call injection vulnerability if not used properly. For example, the parameters of the call, if controllable, can control this contract to perform unauthorized operations or call dangerous functions of other contracts.

2.8. Returned Value Security

In Solidity, there are transfer(), send(), call.value() and other methods. The transaction will be rolled back if the transfer fails, while send and call.value will return false if the transfer fails. If the return is not correctly judged, the unanticipated logic may be executed. In addition, in the implementation of the transfer/transferFrom function of the token contract, it is also necessary to avoid the transfer failure and return false, so as not to create fake recharge loopholes.

2.9. tx.origin Usage

The tx.origin represents the address of the initial creator of the transaction. If tx.origin is used for permission judgment, errors may occur; in addition, if the contract needs to determine whether the caller is the contract address, then tx.origin should be used instead of extcodesize.

2.10. Replay Attack

A replay attack means that if two contracts use the same code implementation, and the identity authentication is in the transmission of parameters, the transaction information can be replayed to the other contract to execute the transaction when the user executes a transaction to one contract.

2.11. Overriding Variables

There are complex variable types in Solidity, such as structures, dynamic arrays, etc. When using a lower version of the compiler, improperly assigning values to it may result in overwriting the values of existing state variables, causing logical exceptions during contract execution.

3. Business Security

3.1 Business Logic

Whether the business logic is designed clearly and flawlessly.

3.2 Business Implementations

Whether the code implementation conforms to comments, project whitepaper, etc.

Appendix 3 Disclaimer

This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin. Due to the technical limitations of any organization, this report conducted by Beosin still has the possibility that the entire risk cannot be completely detected. Beosin disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin Technology.

Appendix 4 About Beosin

BEOSIN is a leading global blockchain security company dedicated to the construction of blockchain security ecology, with team members coming from professors, post-docs, PhDs from renowned universities and elites from head Internet enterprises who have been engaged in information security industry for many years. BEOSIN has established in-depth cooperation with more than 100 global blockchain head enterprises; and has provided security audit and defense deployment services for more than 1,000 smart contracts, more than 50 blockchain platforms and landing application systems, and nearly 100 digital financial enterprises worldwide. Relying on technical advantages, BEOSIN has applied for nearly 50 software invention patents and copyrights.

Official Website

https://beosin.com

E-mail

contact@beosin.com

Twitter

https://twitter.com/Beosin_com