Competitive Security Assessment

FBTC

Jun 20th, 2024

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3′s Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.
The comprehensive examination and auditing scope includes:
• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
• Verify the code base is compliant with the most up-to-date industry standards and security best practices.
• Comprehensive line-by-line manual code review of the entire codebase by industry experts.
The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

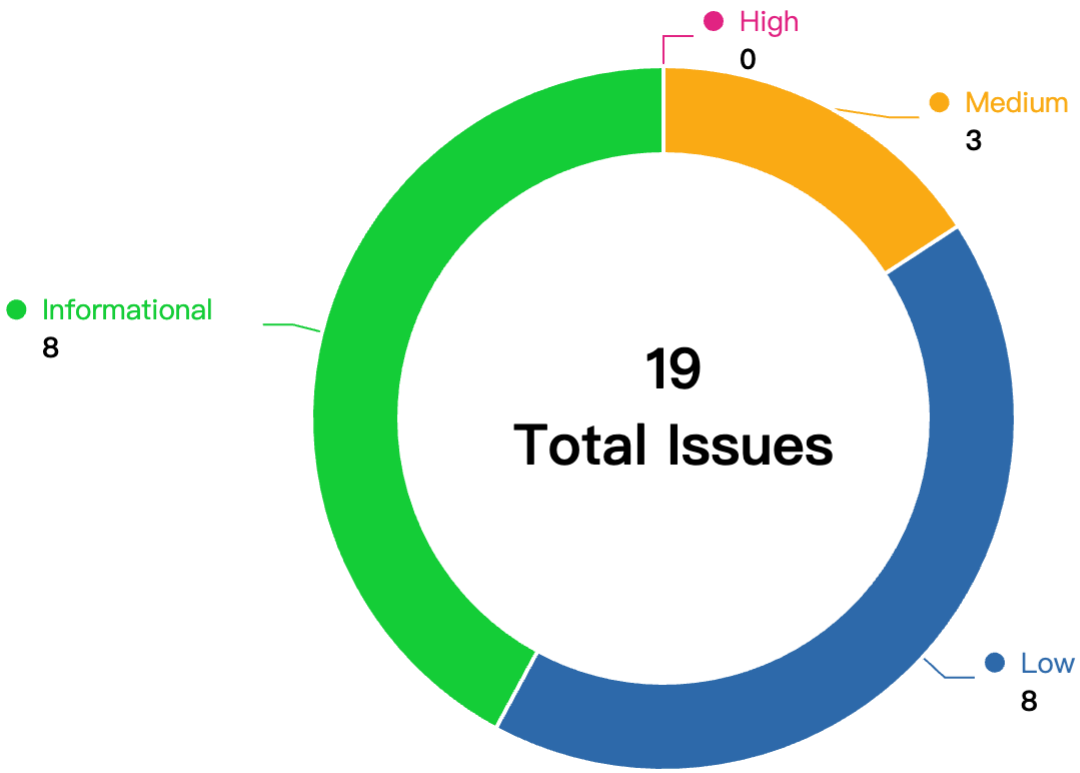
Overview

Project Name	FBTC
Language	solidity
Codebase	• https://github.com/fbtc-xyz/fbtc-contract • audit version - d557e5e0b73eda5af86de5bf4431653056844e64 • final version - cc0cadecd35b75b3a71a24017e03838ffde18e7f • contract in scope FBTCMinter 0x80b534D4bB3D809FbDA809DCB26D3f220634AED7 FBTC 0xC96dE26018A54D51c097160568752c4E3BD6C364 FeeModel 0xd12D39E682715a40dbC860fa07F02bF48841294e FireBridge 0xbee335BB44e75C4794a0b9B54E8027b111395943 FBTCGovernorModule 0x0d771823c72ccca4d74695934d5a346938914547
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
contracts/FireBridge.sol	d04baf3e953f10b040f1f56d0069549501ae699e48aea16bd71aa1c1f33560df
contracts/FBTCGovernorModule.sol	df80cb0402566fdb9f89b54b09f1fe58503f3b7196c5b0bf220be9d0b2f91c41
contracts/Common.sol	05fbb9b4e07a1ec48388e2421a501f40f580683fba979cb046c746b14d633024
contracts/FeeModel.sol	05816a5e0a64cdd654b13216c6927488db0e852656ab224a3adc9ee7496a1f12
contracts/FToken.sol	f137f82e145456d920f29efbe6140f05be1ed3e9ce132c64e0e7563fe604389d
contracts/RoleBasedAccessControl.sol	bab227a0d785019e3816b41411ef8962fca3a9eca7b141e17752c4fff9d9c421
contracts/FBTCMinter.sol	5e7ec9df2618fd6ab86182847a5c8961c3e036895eba7e69bf2fd4ecea3d7e7b
contracts/Governable.sol	35213285823e59d749dad199bb5c61047e44a3df73a0e182653c5da8303b5852
contracts/BridgeStorage.sol	8f4e868c1e2a9f5059285c21b79bf4afad236ce944f489afc51c218251ba1c71
contracts/FBTC.sol	3d5cde5c515b827af647c2395e0325214d2ad8b8550d3e9f46f148c498dd5854

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
FBC-1	Minting fee is not supposed to be charged to users and it is a gainst protocol whitepaper	Logical	Medium	Fixed	0xRizwan
FBC-2	Minter can mint before withdr awal finality for l2 -> l1 transa ctions.	Logical	Medium	Declined	0xffchain
FBC-3	Incorrect implementation of `confirmCrosschainRequest()` for destination chain validation	Logical	Medium	Fixed	0xRizwan
FBC-4	Have no refund mechanism for `addBurnRequest()` and `addCrosschainRequest()`	Logical	Medium	Declined	Cara, 0xhuy0512, 0xWeb3boy, 0xRizwan
FBC-5	Block Reorg Can Allow For Double Spending	Logical	Medium	Acknowledged	biakia

FBC-6	Ownership change should use two-step process	Logical	Low	Fixed	rajatbeladiya, Yaodao, Saaj, 0xffchain, 0x Rizwan, biaki a, 0xzoobi
FBC-7	No Upper Limit for fee	Logical	Low	Fixed	1nc0gn170, C ara, Yaodao
FBC-8	Lack of Target Chain Validity Verification	Logical	Low	Fixed	8olidity, Cara, biakia, 0xhuy 0512
FBC-9	Improper Initialization of `bytes32[]` Array Leading to Failur e in `getRequestsByHashes` Fu nction	Language Sp ecific	Low	Fixed	0xxm, BradM oonUESTC, C ara, Hupixion g3
FBC-10	Disable renounce of ownershi p in contracts	Logical	Low	Fixed	0xffchain, 0x Rizwan, Saaj
FBC-11	Centrazliation Risk	Logical	Low	Acknowledged	0xffchain, Ko ng7ych3, 0xh uy0512, ravikiran_web3, Sa aj, 0xRizwan, biakia, BradM oonUESTC
FBC-12	Centralization risk	Governance Manipulation	Low	Acknowledged	Cara
FBC-13	Centralization Risk	Privilege Rela ted	Low	Acknowledged	csanuragjain
FBC-14	Use `require` instead of `asse rt`	Language Sp ecific	Informational	Fixed	Saaj, Cara, 0x zoobi, Yaoda o
FBC-15	Unnecessary check in `fireBr idge.confirmCrosschainReque st()`	Logical	Informational	Fixed	0xhuy0512
FBC-16	Unhandle the result of `confir mCrosschainRequest()`	Logical	Informational	Fixed	Yaodao, 0xhu y0512, 0xRiz wan
FBC-17	Set the Constant to Private	Gas Optimiza tion	Informational	Acknowledged	Cara
FBC-18	Issues related to hardfork.	DOS	Informational	Acknowledged	0xzoobi, 8oli dity
FBC-19	For loop gas optimization(All optimizations included)	Gas Optimiza tion	Informational	Acknowledged	0xRizwan

FBC-20	Floating solidity pragma version should not be used in contracts	Language Specific	Informational	Fixed	0xRizwan
FBC-21	Consider using OpenZeppelin's SafeCast library to prevent unexpected overflows when downcasting	Logical	Informational	Fixed	rajatbeladiya

FBC-1:Minting fee is not supposed to be charged to users and it is against protocol whitepaper

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	0xRizwan

Code Reference

- code/contracts/FireBridge.sol#L216-L256
- code/contracts/FireBridge.sol#L397

```
216: function addMintRequest(
217:         uint256 _amount,
218:         bytes32 _depositTxid,
219:         uint256 _outputIndex
220:     )
221:         external
222:         onlyActiveQualifiedUser
223:         whenNotPaused
224:         returns (bytes32 _hash, Request memory _r)
225:     {
226:         // Check request.
227:         require(_amount > 0, "Invalid amount");
228:         require(uint256(_depositTxid) != 0, "Empty deposit txid");
229:         bytes memory _depositTxData = abi.encode(_depositTxid, _outputIndex);
230:
231:         bytes32 depositDataHash = keccak256(_depositTxData);
232:         require(
233:             usedDepositTxs[depositDataHash] == bytes32(uint256(0)),
234:             "Used BTC deposit tx"
235:         );
236:
237:         // Compose request. Main -> Self
238:         _r = Request({
239:             nonce: nonce(),
240:             op: Operation.Mint,
241:             srcChain: MAIN_CHAIN,
242:             srcAddress: bytes(userInfo[msg.sender].depositAddress),
243:             dstChain: chain(),
244:             dstAddress: abi.encode(msg.sender),
245:             amount: _amount,
246:             fee: 0, // To be set in `_splitFeeAndUpdate`
247:             extra: _depositTxData,
248:             status: Status.Pending
249:         });
250:
251:         // Split fee.
252:         _splitFeeAndUpdate(_r);
253:
254:         // Save request.
255:         _hash = _addRequest(_r);
256:     }
```

```
397: _payFee(r.fee, true);
```

Description

0xRizwan: As per FBTC whitepaper, For minting the FBTC, there is no minting fee has to be charged to users. It states:

> *"Minting Fees:*
> *Minting fees refer to the costs associated with creating FBTC on the destination chain by depositing BTC on the Bitcoin main chain. When qualified users mint FBTC, they may incur blockchain gas cost, which varies depending on network condition. There is no additional minting fee charged besides gas cost."*

However, in implementation of **`FireBridge.sol`**'s **`addMintRequest()`** which is used to initiate a FBTC minting request for the qualified User and here the fee is being splitted from the amount passed as argument to mint FBTC by users which can be seen as below:

```
    function addMintRequest(
        uint256 _amount,
        bytes32 _depositTxid,
        uint256 _outputIndex
    )
        external
        onlyActiveQualifiedUser
        whenNotPaused
        returns (bytes32 _hash, Request memory _r)
    {

        . . . some code

        // Split fee.
@>      _splitFeeAndUpdate(_r);          @audit // fee is being splitted from actual amount passed
by user

        // Save request.
        _hash = _addRequest(_r);
    }
```

This FBTC minting request by qualified user is actually confirmed by FBTC minter contract by calling **`confirmMintRequest()`** function which is implemented as below:

```
    function confirmMintRequest(
        bytes32 _hash
    ) external onlyMinter whenNotPaused {

        . . . some code

        // Mint tokens
        FToken(fbtc).mint(abi.decode(r.dstAddress, (address)), _amount);

        // Pay fee.
@>        _payFee(r.fee, true);       @audit // fee is being paid to feeRecipient by setting minti
ng as true
    }
```

When the minting request is confirmed, the computed fee via `**_splitFeeAndUpdate**` is paid at last in `**payFee()**` to fee recipient address which is already set by FBTC protocol.
This is violation of Whitepaper where the design of minting is actually deviated than its written.
Protocol Whitepaper is supreme and does not intents to be changed. On minting fee, there is also no mention in whitepaper that in future minting fee will be charged so there is no need of fee split and payment logic while adding and confirming minting requests.

Proof of concept

Consider a scenario,

1. Bob is a qualified user and wants to have FBTC so he sends the 10 BTC to custodial address.Bob calls `**addMintRequest()**` with BTC deposit address and FBTC minting amount. It should be noted that FBTC is 1:1 with BTC means, for every 1 BTC deposited, 1 FBTC can be minted. Additionally, per FBTC whitepaper there is no FBTC minting fee and bridge contract is deployed on Ethereum mainnet.
2. Bob knows the information as mentioned in Point 1 so when he calls addMintRequest with 10 FBTC as minting amount then this request is confirmed by FBTC minter contract by calling `**confirmMintRequest()**`.
3. When the mint request is confirmed then Bob immediatly receives his FBTC. Since while adding and confirming the minting request, there is split and payment of fee logic present so consider that, protocol intentionally or accidentally sets minting fee as 1 %.
4. Bob receives only 9.9 FBTC as 0.1 FBTC is deducted as minting fee with current implementation. Bob had expected to get 10 FBTC due to 1:1 raio of BTC to FBTC, However, due to current implementation its Bobs loss in receving less FBTC.

NOTE:
It could be argued that, the protocol wont set the minting fee and would set the fee as 0 then i still believe that its violating the FBTC whitepaper where there is no mention of charging minting fees now and in future. It should further be noted that, the above scenario considers only 1% fee , however the fee could be charged as any number as the design does not restrict in fee module.

Impact

Loss of users tokens due to forced charged of minting fee which is against FBTC whitepaper. This will impact in users getting less FBTC tokens due to forced minting fee. The issue violates the minting fee design as outlined in FBTC whitepaper.

Recommendation

0xRizwan: Per FBTC whitepaper, Do not charge minting fee. It should be noted that, the whitepaper does not discuss possible minting fee in future so the users should not be enforced minting fee now and in future, Therefore, remove the fee split and payment logic while adding and confirming FBTC minting request.
Consider below changes:
In **`addMintRequest()`**:

```
    function addMintRequest(
        uint256 _amount,
        bytes32 _depositTxid,
        uint256 _outputIndex
    )
        external
        onlyActiveQualifiedUser
        whenNotPaused
        returns (bytes32 _hash, Request memory _r)
    {
        // Check request.
        require(_amount > 0, "Invalid amount");
        require(uint256(_depositTxid) != 0, "Empty deposit txid");
        bytes memory _depositTxData = abi.encode(_depositTxid, _outputIndex);

        bytes32 depositDataHash = keccak256(_depositTxData);
        require(
            usedDepositTxs[depositDataHash] == bytes32(uint256(0)),
            "Used BTC deposit tx"
        );

        // Compose request. Main -> Self
        _r = Request({
            nonce: nonce(),
            op: Operation.Mint,
            srcChain: MAIN_CHAIN,
            srcAddress: bytes(userInfo[msg.sender].depositAddress),
            dstChain: chain(),
            dstAddress: abi.encode(msg.sender),
            amount: _amount,
            fee: 0, // To be set in `_splitFeeAndUpdate`
            extra: _depositTxData,
            status: Status.Pending
        });

-       // Split fee.
-       _splitFeeAndUpdate(_r);

        // Save request.
        _hash = _addRequest(_r);
    }
```

and In `confirmMintRequest()`:

```
    function confirmMintRequest(
        bytes32 _hash
    ) external onlyMinter whenNotPaused {
        // Check request.
        Request storage r = requests[_hash];
        require(r.op == Operation.Mint, "Not Mint request");

        uint256 _amount = r.amount;
        require(_amount > 0, "Invalid request amount");
        require(r.status == Status.Pending, "Invalid request status");

        // Check and update deposit data usage status.
        bytes32 depositDataHash = keccak256(r.extra);
        require(
            usedDepositTxs[depositDataHash] == bytes32(uint256(0)),
            "Used BTC deposit tx"
        );
        usedDepositTxs[depositDataHash] = _hash;

        // Update status.
        r.status = Status.Confirmed;
        emit RequestConfirmed(_hash);

        // Mint tokens
        FToken(fbtc).mint(abi.decode(r.dstAddress, (address)), _amount);

-       // Pay fee.
-       _payFee(r.fee, true);
    }
```

Client Response

client response for 0xRizwan: Fixed. Acknowledged. We will set mint fee to zero but keep the logic in the contract. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

FBC-2:Minter can mint before withdrawal finality for l2 -> l1 transactions.

Category	Severity	Client Response	Contributor
Logical	Medium	Declined	0xffchain

Code Reference

Description

0xffchain: Cross chain minting gives qualified users the ability to request for token mints across selected chain, this includes l1 chains Solana and Ethereum and L2 chain Mantle. L1 chains have finality ranging from 15 minutes (2 Epoch) on Ethereum and 12 seconds for Solana. The time to finality here is when the state of a block is considered final and no chain reorg or dispute can happen that alters the concerned block.
But with Mantle, all messages sent from l2 to l1 has to have a challange period of 7 days as is expected in Optimistic rollup, and challengers are allowed to challenge the submitted state as invalid. A transaction from l2 to l1 is not deemed complete until it has passed the challenge period, which means that the finality of l2 to l1 transactions can be deemed 7 days.
Vulnerability

```
function confirmCrosschainRequest(
    Request memory r
)
    external
    onlyMinter
    whenNotPaused
    returns (bytes32 _dsthash, Request memory _dstRequest)
{
    // Check request.
    require(r.amount > 0, "Invalid request amount");
    require(r.extra.length > 0, "Empty cross-chain data");
    require(r.dstChain == chain(), "Dst chain not match");
    require(
        r.op == Operation.CrosschainConfirm,
        "Not CrosschainConfirm request"
    );
    require(r.status == Status.Unused, "Status should not be used");
    require(r.extra.length == 32, "Invalid extra: not valid bytes32");
    require(
        r.dstAddress.length == 32,
        "Invalid dstAddress: not 32 bytes length"
    );
    require(
        abi.decode(r.dstAddress, (uint256)) <= type(uint160).max,
        "Invalid dstAddress: not address"
    );
    bytes32 srcHash = abi.decode(r.extra, (bytes32));
    //@audit encoding like in solana
    // Set to request to calc hash.
    require(
        r.getCrossSourceRequestHash() == srcHash,
        "Source request hash is incorrect"
    );
    require(
        crosschainRequestConfirmation[srcHash] == bytes32(0),
        "Source request already confirmed"
    );
    // Save request.
    r.nonce = nonce(); // Override src nonce to dst nonce.
    _dsthash = _addRequest(r);
    crosschainRequestConfirmation[srcHash] = _dsthash;
    _dstRequest = r;
    // Mint tokens.
    FToken(fbtc).mint(abi.decode(r.dstAddress, (address)), r.amount);
}
```

When confirming cross chain request, from l2 to l1, it does not take the finality of the request into view, This is important as dependence on a non finalized state of the l2 on the l1, could lead to loss of funds for the protocol, the admin could finalize a request on the l1 based on the request of a qualified user and the qualified user actually completing the mantle side of the operation, but since the challenge period is not taken into view, it means that if the current state of the l2 blockchain is disputed, a fork could occur which discards the erring blocks, and thus discarding the qualified user operation on the l2. It means that the bridge would have minted unbacked tokens to the qualified user, thus causing a loss for all actors with funds locked in the bridge.
POC

1. Qualified user locks 100BTC on the src chain while performing a crossChain bridge on Mantle.
2. Minters sees the transaction as already successfully performed on the mantle then calls confirmCrosschainRequest to confirm the request on Ethereum as its destination chain.
3. The mantle state which contains the qualified users burn is challenged on ethereum, with a valid state that does not contain the qualified users transaction/block.
4. The new state passes the challenge period and its thus finalized.
5. It then means that The admin has minted 100 unbacked BTC, and the qualified user now has 100 BTC on the Destination chain and on the Source chain.

Recommendation

0xffchain:
Funds sent from l2 to l1 should either have a 7 days countdown for finality or include the state hash on l1 that includes the transaction, so that the admin could check its finality before approval...

Client Response

client response for 0xffchain: Declined.
Secure3: . changed severity to Medium

FBC-3:Incorrect implementation of `confirmCrosschainRequest()` for destination chain validation

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	0xRizwan

Code Reference

- code/contracts/FireBridge.sol#L458

```
458: require(r.dstChain == chain(), "Dst chain not match");
```

Description

0xRizwan: **`FireBridge.sol`** contract has provided users or customers to use cross chain tranfer of FBTC tokens. The FBTC tokens can be transferred to other L1/L2 chains by calling **`addCrosschainRequest()`** which is implemented as below:

```
    function addCrosschainRequest(
        bytes32 _targetChain,
        bytes memory _targetAddress,
        uint256 _amount
    ) public whenNotPaused returns (bytes32 _hash, Request memory _r) {
        // Check request.
        require(_amount > 0, "Invalid amount");

        // Compose request. Self -> Target
@>      bytes32 _srcChain = chain();
@>      require(_targetChain != _srcChain, "Self-cross not allowed");

        _r = Request({
            nonce: nonce(),
            op: Operation.CrosschainRequest,
            srcChain: _srcChain,
            srcAddress: abi.encode(msg.sender),
            amount: _amount,
@>          dstChain: _targetChain,
@>          dstAddress: _targetAddress,
            fee: 0,
            extra: "", // Not include in hash.
            status: Status.Unused // Not used.
        });
```

As highlighted by (@>), the above function makes sure that target/destination chain must not be same as source chain. This is to prevent self cross transfer.
When the cross chain transfer request is successfully added then FBTC minter contract calls to confirm the request via **`confirmCrosschainRequest()`** function and it is implemented as:

```
    function confirmCrosschainRequest(
        Request memory r
    )
        external
        onlyMinter
        whenNotPaused
        returns (bytes32 _dsthash, Request memory _dstRequest)
    {
        // Check request.
        require(r.amount > 0, "Invalid request amount");
        require(r.extra.length > 0, "Empty cross-chain data");
@>      require(r.dstChain == chain(), "Dst chain not match");
```

The issue is that, the `**confirmCrosschainRequest()**` validates destination chain should be `**chain()**`(This is chainId of Ethereum mainnet or contract deployed on that chain).
Therfore, it works opposite than while the request added where it was ensured that source and destination chain are not same.
The current implementation ensuing source and destination chain are same while confirming the cross chain transfer requests.
It should further be noted that, `**confirmCrossChainRequest()**` Natspec mentions:

```
/// @dev Most fields of the request should be the same as the one on source chain.
```

Therefore, i believe the issue described is not the intended design.

Recommendation

0xRizwan: Ensure the correct implementation of `**confirmCrosschainRequest()**` function and validate the destination chain to be same as mentioned in cross chain request.

Client Response

client response for 0xRizwan: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

FBC-4:Have no refund mechanism for `addBurnRequest()` and `addCrosschainRequest()`

Category	Severity	Client Response	Contributor
Logical	Medium	Declined	Cara, 0xhuy0512, 0xWeb3boy, 0xRizwan

Code Reference

Description

Cara: The project's whitepaper and code do not appear to address scenarios of cross-chain execution failure caused by many reasons such as the transaction fee exceeding the amount of BTC deposited, the target chain address in the user's cross-chain request being invalid, insufficient gas and network failures.
For instance, a user successfully initiates an FBTC cross-chain request by calling the **`addCrosschainRequest`** function of the **`FireBridge`** contract. The user's FBTC has been burned on the source chain, but there might be a failure to mint FBTC for the user on the target chain. This could happen for one of two reasons:

1. The minter, when attempting to confirm the cross-chain request on the target chain by calling the **`confirmCrosschainRequest`** function, faces an execution failure due to insufficient gas or network issues.
2. When initiating the cross-chain request, the user accidentally inputs a target chain identifier that is not supported by the project, leading to a validation failure off-chain and thus the minter cannot proceed to confirm the cross-chain request.

When the aforementioned scenarios occur, it not only results in the user losing their FBTC but also disrupts the 1:1 ratio between FBTC and BTC, leading to an imbalance in the anchoring relationship.

```
function addCrosschainRequest(
    bytes32 _targetChain,
    bytes memory _targetAddress,
    uint256 _amount
) public whenNotPaused returns (bytes32 _hash, Request memory _r) {

    ......

    // Pay fee
    _payFee(_r.fee, false);

    // Burn tokens.
    FToken(fbtc).burn(msg.sender, _r.amount);
}
```

POC

To further illustrate this vulnerability, follow these steps.

1. The user initiates an FBTC cross-chain request by calling the `addCrosschainRequest` function in the `FireBridge` contract, accidentally inputting an incorrect or unsupported target chain identifier.
2. Within the `addCrosschainRequest` function, the user transfers the transaction fee to `feeRecipient`, and the FBTC intended for cross-chain transfer is burned on the source chain.
3. The Bridge Monitor detects this cross-chain request and forwards it to the TSS Gateway. The TSS Gateway validates the cross-chain request, and upon finding an error in the target chain identifier or that the project does not support the target chain, it does not initiate the confirm cross-chain request operation on the target chain.
4. The user loses the transaction fee and the intended FBTC for cross-chain transfer on the source chain, consequently, the 1:1 anchoring relationship between FBTC and BTC is no longer maintained.

0xhuy0512: When burning FBTC, user have to call to `addBurnRequest()` to firstly burn its FBTC, and then waiting for the protocol to transfer BTC back in BTC chain. The same process in transfering crosschain, user call to `addCrosschainRequest()` to firstly burn its FBTC in srcChain, and wait for the protocol to transfer BTC back in destination chain
In both case, if there're some error processes, that can't finish the state, the protocol doesn't have a refund mechanism, which mean the FBTC that user already burned can't be refunded back
0xWeb3boy: In contract `FireBridge.sol` function addBurnRequest and function addCrosschainRequest are used to Initiate a FBTC burning request for the qualifiedUser which is further approved in the function `confirmBurnRequest` and `confirmCrosschainRequest` consecutively, Now the problem in the above two functions is unlike in the addMintRequest function where the tokens are minted only in the `confirmMintRequest()`, but in the function BurnRequest and function addCrosschainRequest the feePayment and the burning of the tokens are done when the request is being made by the qualified user
Even if the confirmation is not done for these request the fee will be paid and the tokens will be burned already while adding the request only.
0xRizwan: FireBridge contract has functionalities for adding and confirming the FBTC minting/burning/transfer requests.
The contract does not have function to cancel the request if the user intends to cancel it after placing minting/burning/transfer request.
Consider a scenario to understand the issue better:

1. Alice has decided to burn her FBTC to get back the deposited BTC. She calls `addBurnRequest` with amount as argument and place the burning request.
2. After few moments, Alice has changed her decision. She do not want to get back her BTC and wants to keep the FBTC tokens due to some reasons.
3. Alice wants to cancel the burning request and she can see her FBTC burning request status is still pending and it is not confirmed by FBTC minter contract.
4. Alice can not cancel the burning request so she will get her BTC back to her account address as after few moments the buring request will be confirmed. This is due to lack of cancel request functionality in FireBridge contract which would affect the user experience to some extent.

Impact
Users can not cancel their minting/burning/transfer request if they change their decisions especially for burning the FBTC. This would affect users experience with FBTC protocol due to lack of cancel request functionality.

Recommendation

Cara: It is recommended to introduce a cross-chain failure refund feature, allowing users to set an expiration time for the cross-chain request. If the request remains unconfirmed after the expiration time, the TSS Gateway can initiate a transaction to refund the user whose cross-chain attempt has failed.

0xhuy0512: We should have a refund function that can only be called by owner in FireBridge contract, in case there's some error in burning and transferring process
0xWeb3boy: The recommended mitigation step would be to make these functions work like `addMintReqest` and `confirmMintRequest` function.
Where the fee payment and token minting is only done in the `confirmMintRequest` function.
0xRizwan: Allow users(OR qualified users) to cancel request before it gets confirmed by FBTC minter contract.
Implement `cancelRequest` functionality in FireBridge contract.

Client Response

client response for Cara: Declined. The refund mechanism is implemented by our off-chain system.
Secure3: . changed severity to Medium
client response for 0xhuy0512: Declined. The refund mechanism is implemented by our off-chain system.
client response for 0xWeb3boy: Declined. The refund mechanism is implemented by our off-chain system.
Secure3: . changed severity to Medium
client response for 0xRizwan: Declined. The refund mechanism is implemented by our off-chain system.

FBC-5:Block Reorg Can Allow For Double Spending

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	biakia

Code Reference

- code/contracts/FireBridge.sol#L300-L344
- code/contracts/FireBridge.sol#L438-L496

```
300: /// Customer methods.
301:
302:     /// @notice Initiate a FBTC cross-chain bridging request.
303:     /// @param _targetChain The target chain identifier.
304:     /// @param _targetAddress The encoded address on the target chain .
305:     /// @param _amount The amount of FBTC to cross-chain.
306:     /// @return _hash The hash of the new created request.
307:     /// @return _r The full new created request.
308:     function addCrosschainRequest(
309:         bytes32 _targetChain,
310:         bytes memory _targetAddress,
311:         uint256 _amount
312:     ) public whenNotPaused returns (bytes32 _hash, Request memory _r) {
313:         // Check request.
314:         require(_amount > 0, "Invalid amount");
315:
316:         // Compose request. Self -> Target
317:         bytes32 _srcChain = chain();
318:         require(_targetChain != _srcChain, "Self-cross not allowed");
319:
320:         _r = Request({
321:             nonce: nonce(),
322:             op: Operation.CrosschainRequest,
323:             srcChain: _srcChain,
324:             srcAddress: abi.encode(msg.sender),
325:             amount: _amount,
326:             dstChain: _targetChain,
327:             dstAddress: _targetAddress,
328:             fee: 0,
329:             extra: "", // Not include in hash.
330:             status: Status.Unused // Not used.
331:         });
332:
333:         // Split fee.
334:         _splitFeeAndUpdate(_r);
335:
336:         // Save request.
337:         _hash = _addRequest(_r);
338:
339:         // Pay fee
340:         _payFee(_r.fee, false);
341:
342:         // Burn tokens.
343:         FToken(fbtc).burn(msg.sender, _r.amount);
344:     }
```

```
438: /// @notice Confirm the cross-chain request.
439:     /// @dev Most fields of the request should be the same as the one on
440:     ///      source chain. Note:
441:     ///       1. The `op` should be `CrosschainConfirm`
442:     ///       2. The `nonce` is the source nonce, used to calc source request hash.
443:     ///       3. The `status` should be `Unused` (0).
444:     ///       4. The `extra` should contain the source request hash.
445:     /// @param r The full cross-chain request.
446:     /// @return _dsthash The hash of the confirmation request.
447:     function confirmCrosschainRequest(
448:         Request memory r
449:     )
450:         external
451:         onlyMinter
452:         whenNotPaused
453:         returns (bytes32 _dsthash, Request memory _dstRequest)
454:     {
455:         // Check request.
456:         require(r.amount > 0, "Invalid request amount");
457:         require(r.extra.length > 0, "Empty cross-chain data");
458:         require(r.dstChain == chain(), "Dst chain not match");
459:         require(
460:             r.op == Operation.CrosschainConfirm,
461:             "Not CrosschainConfirm request"
462:         );
463:         require(r.status == Status.Unused, "Status should not be used");
464:
465:         require(r.extra.length == 32, "Invalid extra: not valid bytes32");
466:         require(
467:             r.dstAddress.length == 32,
468:             "Invalid dstAddress: not 32 bytes length"
469:         );
470:         require(
471:             abi.decode(r.dstAddress, (uint256)) <= type(uint160).max,
472:             "Invalid dstAddress: not address"
473:         );
474:
475:         bytes32 srcHash = abi.decode(r.extra, (bytes32));
476:
477:         // Set to request to calc hash.
478:         require(
479:             r.getCrossSourceRequestHash() == srcHash,
480:             "Source request hash is incorrect"
481:         );
482:         require(
483:             crosschainRequestConfirmation[srcHash] == bytes32(0),
484:             "Source request already confirmed"
485:         );
486:
487:         // Save request.
488:         r.nonce = nonce(); // Override src nonce to dst nonce.
489:         _dsthash = _addRequest(r);
490:         crosschainRequestConfirmation[srcHash] = _dsthash;
491:
492:         _dstRequest = r;
493:
494:         // Mint tokens.
495:         FToken(fbtc).mint(abi.decode(r.dstAddress, (address)), r.amount);
496:     }
```

Description

biakia: In Web3 we often hear the idiom the blockchain is immutable but there are cases when it's not, or to be more precise it becomes practically immutable (reach finality) only after some time has passed since the block, which contains users' transactions, was minted. This is due to events called blockchain reorganizations, or reorgs for short. Reorgs happen because of the way blockchain validation client softer works. The algorithm that determines which is the canonical chain (fork choice rule), under certain circumstances, can conclude that the current chain history is not optimal and choose a different chain from those provided by competing miners (fork chains), to become the canonical chain.
Reorgs appear more often than people realize. Using on-chain explorers such as polygonscan.com/blocks_forked, for Polygon reorgs, you can see they appear often.
In `**FireBridge**`, a double-spending issue can happen when the source chain reorgs. Consider the following case:

1. Bob calls `**addEVMCrosschainRequest**` function to initiate a FBTC cross-chain bridging request from Polygon to Arbitrum
2. As per the whitepaper, the bridge monitor will monitor the events in real-time and detects the cross-chain request immediately.
3. TSS Gateway initiates a contract call to the Bridge contract on the destination chain to confirm the FBTC cross-chain operation. Multiple TSS Nodes with individual risk control co-sign the confirmation transaction.
4. The `**confirmCrosschainRequest**` function is called on Arbitrum and FBTC will be minted to Bob
5. The Polygon reorgs happens now and Bob's transaction on Polygon is discarded and his FBTC balance will recover
6. Finally, Bob does not burn FBTC on Polygon but gets FBTC on Arbitrum

Reference:
https://abarbatei.xyz/blockchain-reorgs-for-managers-and-auditors#heading-cross-chain-mechanisms

Recommendation

biakia: Consider waiting enough time before starting to process the cross-chain request.

Client Response

client response for biakia: Acknowledged. changed severity to Informational.
The off-chain system and risk-control system will monitor the events in real-time, however, will not handle the request immediately. All off-chain system will always wait enough confirmation (eg: BTC ~6 blocks, ETH ~64 block, etc) to ensure that we should never meet a reorg or fork.

FBC-6:Ownership change should use two-step process

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	rajatbeladiya, Yaodao, Saaj, 0xffchain, 0xRizwan, biakia, 0xzoobi

Code Reference

- code/contracts/FBTCGovernorModule.sol#L6

```
6: import {RoleBasedAccessControl, Ownable} from "./RoleBasedAccessControl.sol";
```

- code/contracts/FBTCMinter.sol#L17

```
17: constructor(address _owner, address _bridge) Ownable(_owner) {
```

- code/contracts/FeeModel.sol#L4
- code/contracts/FeeModel.sol#L8
- code/contracts/FeeModel.sol#L28

```
4: import "@openzeppelin/contracts/access/Ownable.sol";
```

```
8: contract FeeModel is Ownable {
```

```
28: constructor(address _owner) Ownable(_owner) {}
```

- code/contracts/FireBridge.sol#L169-L189

```
169: function setToken(address _token) external onlyOwner {
170:         fbtc = _token;
171:         emit TokenSet(_token);
172:     }
173:
174:     function setMinter(address _minter) external onlyOwner {
175:         minter = _minter;
176:         emit MinterSet(_minter);
177:     }
178:
179:     function setFeeModel(address _feeModel) external onlyOwner {
180:         require(_feeModel != address(0), "Invalid feeModel");
181:         feeModel = _feeModel;
182:         emit FeeModelSet(_feeModel);
183:     }
184:
185:     function setFeeRecipient(address _feeRecipient) external onlyOwner {
186:         require(_feeRecipient != address(0), "Invalid feeRecipient");
187:         feeRecipient = _feeRecipient;
188:         emit FeeRecipientSet(_feeRecipient);
189:     }
```

- code/contracts/RoleBasedAccessControl.sol#L5
- code/contracts/RoleBasedAccessControl.sol#L10

```
5: import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
```

```
10: abstract contract RoleBasedAccessControl is Ownable {
```

Description

rajatbeladiya: Use Ownable2Step.transferOwnership which is safer. Use it as it is more secure due to 2-stage ownership transfer.
Yaodao: It is possible that the `**onlyOnwer**` role mistakenly transfers ownership to the wrong address, resulting in the loss of the `**onlyOnwer**` role.
Saaj:

Vulnerability Details

`**FeeModel**` and `**FBTCMinter**` have inherited `**Ownable**` by OZ that have single-step ownership transfer mechanism, which adds the risk of setting an unwanted owner by accident (this includes address(0)); if the ownership transfer is not done with excessive care.
0xffchain: The Fire bridge contract requires an owner to perform setter functions like setToken() setMinter() setFeeReceipeint() setFeeModel() on the contract for it to function properly. IT also sets the owner of the contract at initialization

```
    constructor(address _owner, bytes32 _mainChain) {
        initialize(_owner);
        MAIN_CHAIN = _mainChain;
    }
```

It means it expect this initial owner to perform this actions, the governor contract does not have the ability to perform these actions, but the ownable2step contract allows for transfer of ownership,

```
function transferOwnership(address newOwner) public virtual override onlyOwner {
        _pendingOwner = newOwner;
        emit OwnershipTransferStarted(owner(), newOwner);
    }
```

So it means that the protocol intends to have an initial owner like the deployer which sets the initial values mentioned above then transfers ownership to the governor when done.
This conclusion is also backed up by the fact that the governor also requires the bridge contract address at initialization.

```
    constructor(
        address _owner,
        address _bridge,
        address _fbtc
    ) Ownable(_owner) {
        bridge = FireBridge(_bridge);
        fbtc = FBTC(_fbtc);
    }
```

Meaning the bridge should be deployed before the governor. Since this is the case, it means the governor cannot be the initial owner and would have to be transferred the ownership. But since the ownership contract is Ownable2Step, it means that any new owner which is transffered ownership must accept ownership for it to become the contract owner.

```
    function acceptOwnership() public virtual {
        address sender = _msgSender();
        if (pendingOwner() != sender) {
            revert OwnableUnauthorizedAccount(sender);
        }
        _transferOwnership(sender);
    }
```

And the governor does not have any function that calls the acceptOwnerShip function on the bridge contract. So in essence the governor cannot be the unitial owner and the governor cannot accept ownership if transferred to.
0xRizwan: Single-step process for critical ownership transfer is risky due to possible human error which could result in locking all the functions that use the onlyOwner modifier as used in contracts. Following contracts has used openzeppelin's single step **`ownable.sol`**.

1. **`FeeModel.sol`**
2. **`RoleBasedAccessControl.sol`** as an abstract which is used as base by **`FBTCGovernorModule.sol`** and **`FBTCMinter.sol`**

```
contract FeeModel is Ownable {
```

```
abstract contract RoleBasedAccessControl is Ownable {
```

The above implementation used in contract is not safe as the process is 1-step while transferring contract ownership which is risky due to a possible human error and such an error is unrecoverable.
For example:
An incorrect address, for which the private key is not known, could be passed accidentally. In **`FeeModel.sol`**, **`FBTCGovernorModule.sol`** and **`FBTCMinter.sol`** functions using **`onlyOwner`** modifier will be locked and can not be used if the owner address is set incorrectly and in worst case the whole **`FeeModel.sol`**, **`FBTCGovernorModule.sol`** and **`FBTCMinter.sol`** contracts will be of no use if such critical functions can not be accessed by real owner. The users funds and the setter functions are at direct risk, Therefore the issue is identified as Medium low severity.
Impact
functions using the onlyOwner modifier in **`FeeModel.sol`**, **`FBTCGovernorModule.sol`** and **`FBTCMinter.sol`** contracts will be locked and could not used if the issue happens while transferring contract ownership.

biakia: The contract `FeeModel` does not implement a two-step process for transferring ownership. So ownership of the contract can be easily lost when making a mistake when transferring ownership.
0xzoobi: The contracts `FeeModel.sol` and `RoleBasedAccessControl.sol` does not implement a 2-Step-Process for transferring ownership.
So ownership of the contract can easily be lost when making a mistake when transferring ownership.
Since the privileged roles have critical function roles assigned to them. Assigning the ownership to a wrong user can be disastrous.
So Consider using the Ownable2Step contract from OZ (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol) instead.
The way it works is there is a `transferOwnership` to transfer the ownership and `acceptOwnership` to accept the ownership. Refer the above `Ownable2Step.sol` for more details.

Recommendation

rajatbeladiya: Use Openzeppelin's Ownable2Step.sol

```
    function acceptOwnership() external {
        address sender = _msgSender();
        require(pendingOwner() == sender, "Ownable2Step: caller is not the new owner");
        _transferOwnership(sender);
    }
```

Yaodao: Recommend implementing a two-step process where the onwer nominates an account and the nominated account needs to call an `acceptOnwership()` function for the transfer of the ownership to fully succeed.
Saaj: Consider employing 2 step ownership transfer mechanisms by using Open Zeppelin's Ownable2Step.
0xffchain: Governor should implement acceptOwnerShip function.
0xRizwan: Use openzeppelin's ownable2step.sol for step ownership transfer.
Consider below changes in `FeeModel.sol` and `RoleBasedAccessControl.sol`:

```
- import "@openzeppelin/contracts/access/Ownable.sol";
+ import "@openzeppelin/contracts/access/Ownable2Step.sol";

- contract FeeModel is Ownable {
+ contract FeeModel is Ownable2Step {
```

```
- import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";
+ import {Ownable} from "@openzeppelin/contracts/access/Ownable2Step.sol";

- abstract contract RoleBasedAccessControl is Ownable {
+ abstract contract RoleBasedAccessControl is Ownable2Step {
```

biakia: Consider Ownable2StepUpgradeable(https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/master/contracts/access/Ownable2StepUpgradeable.sol) instead.
0xzoobi: Implement 2-Step-Process for transferring ownership via `Ownable2Step`.

Client Response

client response for rajatbeladiya: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for Yaodao: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for Saaj: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

client response for 0xffchain: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for 0xRizwan: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for biakia: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for 0xzoobi: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

FBC-7:No Upper Limit for fee

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	1nc0gn170, Cara, Yaodao

Code Reference

- code/contracts/FeeModel.sol#L39-L51
- code/contracts/FeeModel.sol#L45-L48
- code/contracts/FeeModel.sol#L53-L58
- code/contracts/FeeModel.sol#L53-58
- code/contracts/FeeModel.sol#L55-L56
- code/contracts/FeeModel.sol#L60-L68
- code/contracts/FeeModel.sol#L70-L79

```
39: function _getFee(
40:         uint256 _amount,
41:         FeeConfig memory _config
42:     ) internal pure returns (uint256 _fee) {
43:         _fee = ((uint256(_config.feeRate) * _amount) / FEE_RATE_BASE);
44:
45:         if (_fee < _config.minFee) {
46:             // Minimal fee
47:             _fee = _config.minFee;
48:         }
49:
50:         require(_fee < _amount, "amount lower than minimal fee");
51:     }
```

```
45: if (_fee < _config.minFee) {
46:             // Minimal fee
47:             _fee = _config.minFee;
48:         }
```

```
53: function _validateConfig(FeeConfig calldata _config) internal pure {
54:         require(
55:             _config.feeRate <= FEE_RATE_BASE / 100,
56:             "Fee rate too high, > 1%"
57:         );
58:     }
```

```
53: function _validateConfig(FeeConfig calldata _config) internal pure {
54:         require(
55:             _config.feeRate <= FEE_RATE_BASE / 100,
56:             "Fee rate too high, > 1%"
57:         );
58:     }
```

```
55: _config.feeRate <= FEE_RATE_BASE / 100,
56:             "Fee rate too high, > 1%"
```

```
60: function setDefaultFeeConfig(
61:         Operation _op,
62:         FeeConfig calldata _config
63:     ) external onlyOwner {
64:         _validateOp(_op);
65:         _validateConfig(_config);
66:         defaultFeeConfig[_op] = _config;
67:         emit DefaultFeeConfigSet(_op, _config);
68:     }
```

```
70: function setChainFeeConfig(
71:         Operation _op,
72:         bytes32 _dstChain,
73:         FeeConfig calldata _config
74:     ) external onlyOwner {
75:         _validateOp(_op);
76:         _validateConfig(_config);
77:         chainFeeConfig[_op][_dstChain] = _config;
78:         emit ChainFeeConfigSet(_op, _dstChain, _config);
79:     }
```

Description

1nc0gn170: ### Description
According to the Fee Model, Fee rate should not exceed 1% for a given amount.

```
    function _validateConfig(FeeConfig calldata _config) internal pure {
        require(
            _config.feeRate <= FEE_RATE_BASE / 100,
            "Fee rate too high, > 1%"
        );
    }
```

However due a missing check fee can exceed more than 1%, using the variable `**FeeConfig.minFee**` as this variable lacks any validation this can be used to bypass the above invariant.

```
    function _getFee(
        uint256 _amount,
        FeeConfig memory _config
    ) internal pure returns (uint256 _fee) {
        _fee = ((uint256(_config.feeRate) * _amount) / FEE_RATE_BASE);

        if (_fee < _config.minFee) { <-----
            // Minimal fee
            _fee = _config.minFee;   <-----
        }

        require(_fee < _amount, "amount lower than minimal fee");
    }
```

POC

```
        function testFeeBug() public {

            FeeModel.FeeConfig memory _config = FeeModel.FeeConfig(
                true,
                feeModel.FEE_RATE_BASE() / 100,
                2e17   // 10% of 2e18 <-------
            );
            feeModel.setDefaultFeeConfig(Operation.Mint, _config);

            bytes32 dummy = bytes32(0);
            bytes memory dummy_ = abi.encode(address(0));

            Request memory _r = Request({
                nonce: 1337,
                op: Operation.Mint,
                srcChain: dummy,
                srcAddress: dummy_,
                amount: 2e18,   /// Amount is 2e18 <------
                dstChain: dummy,
                dstAddress: dummy_,
                fee: 0,
                extra: "",
                status: Status.Unused
            });

            uint fee_ = feeModel.getFee(_r);
            assertEq(fee_, 2e17);   // Max fee should not exceed 1%, but it is 10%
        }
```

OUTPUT:

```
  [63069] FireBridgeTest::testFeeBug()
    ├─ [184] FeeModel::FEE_RATE_BASE() [staticcall]
    │   └─ ← [Return] 1000000 [1e6]
    ├─ [49089] FeeModel::setDefaultFeeConfig(1, FeeConfig({ active: true, feeRate: 10000 [1e4], mi
nFee: 200000000000000000 [2e17] }))
    │   ├─ emit DefaultFeeConfigSet(_op: 1, _config: FeeConfig({ active: true, feeRate: 10000 [1e
4], minFee: 200000000000000000 [2e17] }))
    │   └─ ← [Return]
    ├─ [5826] FeeModel::getFee(Request({ op: 1, status: 0, nonce: 1337, srcChain: 0x00000000000000
0000000000000000000000000000000000000000000000, srcAddress: 0x0000000000000000000000000000000000
0000000000000000000000000000, dstChain: 0x0000000000000000000000000000000000000000000000000000
000000000000, dstAddress: 0x0000000000000000000000000000000000000000000000000000000000000000, amou
nt: 200000000000000000 [2e18], fee: 0, extra: 0x })) [staticcall]
    │   └─ ← [Return] 200000000000000000 [2e17]
    └─ ← [Return]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.76ms (272.71µs CPU time)
```

Cara: In the **`FeeModel`** contract, the owner can set the default fee config and chain config by calling function **`setDefaultFeeConfig`** and function **`setChainFeeConfig`**. In these two functions, the **`_validateConfig`** function is called to limit the fee rate to be less than or equal to 1%, but there is no restriction on the **`minFee`** within the config. When calculating the actual fee, as seen from the **`_getFee`** function, if the fee calculated according to the fee rate is less than the **`minFee`** in the config, then the fee will be set to the **`minFee`**.
If **`minFee`** is set to a large value, it will cause a loss to the users. For example, when a qualified user initiates a FBTC burning request, if the **`minFee`** is sufficiently high, then the calculated **`fee`** will be very large, to the extent that the FBTC hold by the user is almost entirely transferred to the **`feeRecipient`**.

```
function _validateConfig(FeeConfig calldata _config) internal pure {
    require(
        _config.feeRate <= FEE_RATE_BASE / 100,
        "Fee rate too high, > 1%"
    );
}

function setDefaultFeeConfig(
    Operation _op,
    FeeConfig calldata _config
) external onlyOwner {
    _validateOp(_op);
    _validateConfig(_config);
    defaultFeeConfig[_op] = _config;
    emit DefaultFeeConfigSet(_op, _config);
}

function setChainFeeConfig(
    Operation _op,
    bytes32 _dstChain,
    FeeConfig calldata _config
) external onlyOwner {
    _validateOp(_op);
    _validateConfig(_config);
    chainFeeConfig[_op][_dstChain] = _config;
    emit ChainFeeConfigSet(_op, _dstChain, _config);
}
```

```
function _getFee(
    uint256 _amount,
    FeeConfig memory _config
) internal pure returns (uint256 _fee) {
    _fee = ((uint256(_config.feeRate) * _amount) / FEE_RATE_BASE);

    if (_fee < _config.minFee) {
        // Minimal fee
        _fee = _config.minFee;
    }

    require(_fee < _amount, "amount lower than minimal fee");
}
```

Yaodao: The function `setChainFeeConfig` is used to set the chainFee config and it will call `_validateConfig` to check the data of config.
However, only `config.feeRate` is checked but `config.minFee` is not checked.
As a result, the result of fee in the function `_getFee` may be very large when the `config.minFee` is set to be a large value.

Recommendation

1nc0gn170:

```
    function _validateConfig(FeeConfig calldata _config) internal pure {
        require(
-           _config.feeRate <= FEE_RATE_BASE / 100,
+           _config.feeRate <= FEE_RATE_BASE / 100 && _config.minFee <= FEE_RATE_BASE / 100,
            "Fee rate too high, > 1%"
        );
    }
```

Cara: It's suggested to constrain the setting of the **`minFee`** in fee config, to ensure that there is a check on its upper limit, or introduce multi sign or governance module to avoid a single point of key management failure.
Yaodao: Recommend checking the value of **`config.minFee`** in the function **`_validateConfig()`**.

Client Response

client response for 1nc0gn170: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for Cara: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for Yaodao: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

FBC-8:Lack of Target Chain Validity Verification

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	8olidity, Cara, biakia, 0xhuy0512

Code Reference

- code/contracts/FireBridge.sol#L302-L344
- code/contracts/FireBridge.sol#L308-L344
- code/contracts/FireBridge.sol#L308

```
302: /// @notice Initiate a FBTC cross-chain bridging request.
303:     /// @param _targetChain The target chain identifier.
304:     /// @param _targetAddress The encoded address on the target chain .
305:     /// @param _amount The amount of FBTC to cross-chain.
306:     /// @return _hash The hash of the new created request.
307:     /// @return _r The full new created request.
308:     function addCrosschainRequest(
309:         bytes32 _targetChain,
310:         bytes memory _targetAddress,
311:         uint256 _amount
312:     ) public whenNotPaused returns (bytes32 _hash, Request memory _r) {
313:         // Check request.
314:         require(_amount > 0, "Invalid amount");
315:
316:         // Compose request. Self -> Target
317:         bytes32 _srcChain = chain();
318:         require(_targetChain != _srcChain, "Self-cross not allowed");
319:
320:         _r = Request({
321:             nonce: nonce(),
322:             op: Operation.CrosschainRequest,
323:             srcChain: _srcChain,
324:             srcAddress: abi.encode(msg.sender),
325:             amount: _amount,
326:             dstChain: _targetChain,
327:             dstAddress: _targetAddress,
328:             fee: 0,
329:             extra: "", // Not include in hash.
330:             status: Status.Unused // Not used.
331:         });
332:
333:         // Split fee.
334:         _splitFeeAndUpdate(_r);
335:
336:         // Save request.
337:         _hash = _addRequest(_r);
338:
339:         // Pay fee
340:         _payFee(_r.fee, false);
341:
342:         // Burn tokens.
343:         FToken(fbtc).burn(msg.sender, _r.amount);
344:     }
```

```
308: function addCrosschainRequest(
309:         bytes32 _targetChain,
310:         bytes memory _targetAddress,
311:         uint256 _amount
312:     ) public whenNotPaused returns (bytes32 _hash, Request memory _r) {
313:         // Check request.
314:         require(_amount > 0, "Invalid amount");
315:
316:         // Compose request. Self -> Target
317:         bytes32 _srcChain = chain();
318:         require(_targetChain != _srcChain, "Self-cross not allowed");
319:
320:         _r = Request({
321:             nonce: nonce(),
322:             op: Operation.CrosschainRequest,
323:             srcChain: _srcChain,
324:             srcAddress: abi.encode(msg.sender),
325:             amount: _amount,
326:             dstChain: _targetChain,
327:             dstAddress: _targetAddress,
328:             fee: 0,
329:             extra: "", // Not include in hash.
330:             status: Status.Unused // Not used.
331:         });
332:
333:         // Split fee.
334:         _splitFeeAndUpdate(_r);
335:
336:         // Save request.
337:         _hash = _addRequest(_r);
338:
339:         // Pay fee
340:         _payFee(_r.fee, false);
341:
342:         // Burn tokens.
343:         FToken(fbtc).burn(msg.sender, _r.amount);
344:     }
```

```
308: function addCrosschainRequest(
```

Description

8olidity: In the **`addCrosschainRequest`** function, there is a lack of validation for the user-input target chain, meaning there is no check to ensure the target chain provided by the user is valid. If the user inputs a non-existent chain, it may lead to a security risk where the user's assets could be compromised or lost.
poc

```
function testdst() public {
    FeeModel.FeeConfig memory _config = FeeModel.FeeConfig(
        true,
        feeModel.FEE_RATE_BASE() / 100,
        0
    );
    feeModel.setDefaultFeeConfig(Operation.Mint, _config);

    feeModel.setDefaultFeeConfig(Operation.Burn, _config);

    feeModel.setDefaultFeeConfig(Operation.CrosschainRequest, _config);

    (bytes32 _hash, ) = bridge.addMintRequest(1000 ether, TX_DATA1, 1);
    minter.confirmMintRequest(_hash);
    assertEq(fbtc.balanceOf(OWNER), 990 ether);
    assertEq(fbtc.balanceOf(FEE), 10 ether);

    (_hash, ) = bridge.addBurnRequest(100 ether);

    assertEq(fbtc.balanceOf(OWNER), 890 ether);
    assertEq(fbtc.balanceOf(FEE), 11 ether);
    minter.confirmBurnRequest(_hash, TX_DATA2, 0);

    (_hash, ) = bridge.addCrosschainRequest(
        bytes32(uint256(0xddddd9)),
        abi.encode(ONE),
        100 ether
    );

    Request memory r = bridge.getRequestByHash(_hash);
    _confirmCrosschainRequest(r, _hash);

    assertEq(fbtc.balanceOf(OWNER), 790 ether);
    assertEq(fbtc.balanceOf(FEE), 12 ether);
}
```

Cara: When a user initiates an FBTC cross-chain request by calling the **`addCrosschainRequest`** function of the **`FireBridge`** contract, the legitimacy of the parameter **`_targetChain`** is not verified. If the target chain specified by the user does not exist, or if the project does not support that target chain, the cross-chain request will still be successfully created. The user is required to send fees to the **`feeRecipient`**, and the user's FBTC will also be burned.
However, because the target chain is not legitimate, the cross-chain request ultimately cannot be confirmed on the target chain. As a result, the user loses FBTC, and it also leads to an imbalance in the pegged relationship between FBTC and BTC within the project, which is no longer at a 1:1 ratio.
The specific steps leading to the vulnerability are as described in the poc of finding "No mechanism in place to handle cross-chain execution failures."

```
function addCrosschainRequest(
    bytes32 _targetChain,
    bytes memory _targetAddress,
    uint256 _amount
) public whenNotPaused returns (bytes32 _hash, Request memory _r) {
    // Check request.
    require(_amount > 0, "Invalid amount");

    // Compose request. Self -> Target
    bytes32 _srcChain = chain();
    require(_targetChain != _srcChain, "Self-cross not allowed");

    _r = Request({
        nonce: nonce(),
        op: Operation.CrosschainRequest,
        srcChain: _srcChain,
        srcAddress: abi.encode(msg.sender),
        amount: _amount,
        dstChain: _targetChain,
        dstAddress: _targetAddress,
        fee: 0,
        extra: "", // Not include in hash.
        status: Status.Unused // Not used.
    });

    // Split fee.
    _splitFeeAndUpdate(_r);

    // Save request.
    _hash = _addRequest(_r);

    // Pay fee
    _payFee(_r.fee, false);

    // Burn tokens.
    FToken(fbtc).burn(msg.sender, _r.amount);
}
```

POC

To further illustrate this vulnerability, follow these steps.

1. The user initiates an FBTC cross-chain request by calling the **`addCrosschainRequest`** function in the **`FireBridge`** contract, accidentally inputting an incorrect or unsupported target chain identifier.
2. Within the **`addCrosschainRequest`** function, the user transfers the transaction fee to **`feeRecipient`**, and the FBTC intended for cross-chain transfer is burned on the source chain.

3. The Bridge Monitor detects this cross-chain request and forwards it to the TSS Gateway. The TSS Gateway validates the cross-chain request, and upon finding an error in the target chain identifier or that the project does not support the target chain, it does not initiate the confirm cross-chain request operation on the target chain.
4. The user loses the transaction fee and the intended FBTC for cross-chain transfer on the source chain, consequently, the 1:1 anchoring relationship between FBTC and BTC is no longer maintained.

biakia: Any user can call **`addCrosschainRequest`** function to initiate a FBTC cross-chain bridging request. This function only checks whether the **`_targetChain`** is different from **`_srcChain`**:

```
// Compose request. Self -> Target
        bytes32 _srcChain = chain();
        require(_targetChain != _srcChain, "Self-cross not allowed");
```

It does not check whether the **`_targetChain`** is a legitimate chain. If the user mistakenly passes in an illegal chain ID, the user will lost his FBTC because the function **`addCrosschainRequest`** will burn the user's FBTC:

```
// Burn tokens.
FToken(fbtc).burn(msg.sender, _r.amount);
```

0xhuy0512: In function **`FireBridge.addCrosschainRequest()`**, we don't have any validation to check if the **`_targetChain`** is valid and supported by the protocol

```
    function addCrosschainRequest(
        bytes32 _targetChain,
        bytes memory _targetAddress,
        uint256 _amount
    ) public whenNotPaused returns (bytes32 _hash, Request memory _r) {
        require(_amount > 0, "Invalid amount");

        bytes32 _srcChain = chain();
        require(_targetChain != _srcChain, "Self-cross not allowed");

        ...
    }
```

This can lead to user transfering crosschain with the wrong chainId, which lead to losing transfering money

Recommendation

8olidity: Add a mechanism to validate the target chain in the addCrosschainRequest function to ensure that users can only input valid target chain identifiers. This can be addressed by implementing custom validation logic for chain validity, such as checking against a whitelist of valid chain identifiers or verifying if the input is within a reasonable range.
Cara: It is recommended that the project maintain a whitelist of supported chains within the **`FireBridge`** contract and verify in the **`addCrosschainRequest`** function whether the parameter **`_targetChain`** is on the whitelist. If it is not, the transaction should be rolled back.
biakia: Consider checking if **`_targetChain`** is a legitimate chain ID.
0xhuy0512: Must have a mapping variable to whitelist supported chain, and use it to validate **`dstChain`** in **`addCrosschainRequest()`** and **`srcChain`** in **`confirmCrosschainRequest()`**

Client Response

client response for 8olidity: Fixed.
client response for Cara: Fixed.
client response for biakia: Fixed.
client response for 0xhuy0512: Fixed.

FBC-9:Improper Initialization of `bytes32[]` Array Leading to Failure in `getRequestsByHashes` Function

Category	Severity	Client Response	Contributor
Language Specific	Low	Fixed	0xxm, BradMoonUEST C, Cara, Hupixiong3

Code Reference

- code/contracts/FireBridge.sol#L594
- code/contracts/FireBridge.sol#L594-L600

```
594: function getRequestsByHashes(
```

```
594: function getRequestsByHashes(
595:         bytes32[] calldata _hashes
596:     ) external view returns (Request[] memory rs) {
597:         for (uint i = 0; i < _hashes.length; i++) {
598:             rs[i] = getRequestByHash(_hashes[i]);
599:         }
600:     }
```

Description

0xxm: The return value `rs` of function `getRequestsByHashes` in `FireBridge` contract is not correctly initialized, which will lead to permanent revert when calling this function.
BradMoonUESTC: The Solidity function `getRequestsByHashes` is intended to retrieve a list of `Request` objects based on a provided array of hashes (`_hashes`). However, the array `rs` designed to store the results is not explicitly initialized with the correct size. In Solidity, dynamically-sized memory arrays must have a fixed size at creation to ensure that each index can hold a value. Without this, assigning directly to `rs[i]` leads to an error since the array has not been allocated the required space. This will result in the function failing and reverting during execution.
Here's the vulnerable code snippet:

```
function getRequestsByHashes(
    bytes32[] calldata _hashes
) external view returns (Request[] memory rs) {
    for (uint i = 0; i < _hashes.length; i++) {
        rs[i] = getRequestByHash(_hashes[i]);
    }
}
```

Since `rs` isn't pre-initialized with a proper size, the attempt to assign values to `rs[i]` directly is not permitted, leading to a
failure and making the `getRequestsByHashes` function non-operational.
Tested in remix, it will revert with:

```
call to Storage.getRequestsByHashes errored: Error occurred: revert.

revert
    The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value you send should be les
s than your current balance.
You may want to cautiously increase the gas limit if the transaction went out of gas.
```

Cara: In contract **`FireBridge`**, function **`getRequestsByHashes`** does not work, because it has incorrect array initialization, which will cause the transaction to revert. Specifically, a dynamic array **`rs`** with location memory is declared as a return variable, but this variable is never initialized before writing to it through the loop.
A dynamic array declared in memory needs to be initialized with length info.

```
function getRequestsByHashes(
    bytes32[] calldata _hashes
) external view returns (Request[] memory rs) {
    for (uint i = 0; i < _hashes.length; i++) {
        rs[i] = getRequestByHash(_hashes[i]);
    }
}
```

Hupixiong3: Array 'rs' is not initialized correctly in the getRequestsByHashes function. in Solidity language, memory arrays must be initialized to a fixed size before being indexed.

Recommendation

0xxm: Recomment to initialize **`rs`** as follows:

```
function getRequestsByHashes(
  bytes32[] calldata _hashes
) external view returns (Request[] memory rs) {
    uint256 len = _hashes.length;
    rs = new Request[](len);
    for (uint i = 0; i < len; i++) {
        rs[i] = getRequestByHash(_hashes[i]);
    }
}
```

BradMoonUESTC: To correct this issue, the **`rs`** array should be properly initialized to the length of the input **`_hashes`** array before attempting to assign values to specific indices. This will ensure each index is allocated memory, preventing execution errors.
The fixed code should look like this:

```
function getRequestsByHashes(
    bytes32[] calldata _hashes
) external view returns (Request[] memory rs) {
    // Initialize the memory array `rs` with the correct length
    rs = new Request[](_hashes.length);
    // Populate the initialized array
    for (uint i = 0; i < _hashes.length; i++) {
        rs[i] = getRequestByHash(_hashes[i]);
    }
}
```

Cara: It is suggested to initialize rs as shown below:

```
function getRequestsByHashes(
    bytes32[] calldata _hashes
) external view returns (Request[] memory rs) {
    rs = new Request[](_hashes.length);
    for (uint i = 0; i < _hashes.length; i++) {
        rs[i] = getRequestByHash(_hashes[i]);
    }
}
```

Hupixiong3: Initializes the size of the fixed 'rs'.

```
function getRequestsByHashes(
    bytes32[] calldata _hashes
) external view returns (Request[] memory rs) {
    rs = new Request[](_hashes.length);
    for (uint i = 0; i < _hashes.length; i++) {
        rs[i] = getRequestByHash(_hashes[i]);
    }
}
```

Client Response

client response for 0xxm: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for BradMoonUESTC: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for Cara: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for Hupixiong3: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

FBC-10:Disable renounce of ownership in contracts

Category	Severity	Client Response	Contributor
Logical	Low	Fixed	0xffchain, 0xRizwan, Saaj

Code Reference

- code/contracts/FBTC.sol#L4

```
4: import {FToken} from "./FToken.sol";
```

- code/contracts/FBTCMinter.sol#L17

```
17: constructor(address _owner, address _bridge) Ownable(_owner) {
```

- code/contracts/FeeModel.sol#L8
- code/contracts/FeeModel.sol#L28

```
8: contract FeeModel is Ownable {
```

```
28: constructor(address _owner) Ownable(_owner) {}
```

- code/contracts/FireBridge.sol#L9

```
9: import {Governable} from "./Governable.sol";
```

- code/contracts/Governable.sol#L10

```
10: Ownable2StepUpgradeable,
```

- code/contracts/RoleBasedAccessControl.sol#L10

```
10: abstract contract RoleBasedAccessControl is Ownable {
```

Description

0xffchain: Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.
The Ownable used in this project contract implements renounceOwnership . This can represent a certain risk if the ownership is renounced. Renouncing ownership will leave the contract without an owner, thereby removing any and all functionalities that is only available to the owner.
0xRizwan: Most of the inscope contracts have used openzeppelin's ownable.sol for ownership transfer related functionalities.
Transferring of ownership can happen periodically and can be understood as the decisions taken by protocol team. However, the current implementation in contracts also allows the renounce of ownership. This can represent a

certain risk if the ownership is renounced for any other reason than by design.
Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the contract owner. As they said, prevention is better than cure so disabling the renounce of ownership could be great in the best interest of users and protocol.
The follwing contracts use contract ownership are directly affected by this issue:

1. `**FireBridge.sol**`
2. `**FBTCGovernorModule.sol**`
3. `**FeeModel.sol**`
4. `**FToken.sol**`
5. `**FBTCMinter.sol**`
6. `**RoleBasedAccessControl.sol**`
7. `**Governable.sol**`

Impact
Renouncing contract ownership will leave all onlyOwner related functions useless in above contracts
Saaj: ## Vulnerability Details
` **FeeModel**` and ` **FBTCMinter**` contract inherits `**Ownable**`, while ` **FBTC**` and ` **FireBridge**` inherits ` **Ownable2StepUpgradeable**` indirectly from contracts they inherit.
` **Ownable2StepUpgradeable**` inherits ` **OwnableUpgradeable**` that have `**renounceOwnership**` function.
Owner of the contract can call `**renounceOwnership**` which can lead to transferring ownership to `**address(0)**`.

Impact

`**renounceOwnership**` function should be removed or modified to revert as it can lead to loss of crucial functionality if `**ownership**` is transferred to `**address(0)**`.

```
     * @dev Leaves the contract without owner. It will not be possible to call
     * `onlyOwner` functions. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * thereby disabling any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
    }
```

Thus, owner should not be allowed to revoke itself.

Recommendation

0xffchain: Disable renounceOwnerShip.
0xRizwan: Disable renounce of ownership in contracts.
For example:

```
    function renounceOwnership() public virtual override onlyOwner {
-       _transferOwnership(address(0));
+       revert ("disabled renounceOwnership");
    }
```

Saaj: The recommendation is made to remove the `**renounceOwnership**` or modify the function to revert like shown in below example.

```
+   error RevokingNotAllowed();

    function renounceOwnership() public virtual onlyOwner {
-       _transferOwnership(address(0));
+       revert RevokingNotAllowed();
    }
```

Client Response

client response for 0xffchain: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for 0xRizwan: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for Saaj: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

FBC-11:Centrazliation Risk

Category	Severity	Client Response	Contributor
Logical	Low	Acknowledged	0xffchain, Kong7ych3, 0xhuy0512, ravikiran_web3, Saaj, 0xRizwan, biakia, BradMoonUESTC

Code Reference

- code/contracts/FBTC.sol#L7

```
7: constructor(address _owner, address _bridge) {
```

- code/contracts/FBTCGovernorModule.sol#L37
- code/contracts/FBTCGovernorModule.sol#L66
- code/contracts/FBTCGovernorModule.sol#L71
- code/contracts/FBTCGovernorModule.sol#L71-L74

```
37: constructor(
```

```
66: function setFBTC(address _fbtc) external onlyOwner {
```

```
71: function setBridge(address _bridge) external onlyOwner {
```

```
71: function setBridge(address _bridge) external onlyOwner {
72:         bridge = FireBridge(_bridge);
73:         emit BridgeSet(_bridge);
74:     }
```

- code/contracts/FBTCMinter.sol#L17
- code/contracts/FBTCMinter.sol#L17-L25
- code/contracts/FBTCMinter.sol#L21
- code/contracts/FBTCMinter.sol#L29-L57
- code/contracts/FBTCMinter.sol#L45

```
17: constructor(address _owner, address _bridge) Ownable(_owner) {
```

```
17: constructor(address _owner, address _bridge) Ownable(_owner) {
18:         bridge = FireBridge(_bridge);
19:     }
20:
21:     function setBridge(address _bridge) external onlyOwner {
22:         address oldBridge = address(bridge);
23:         bridge = FireBridge(_bridge);
24:         emit BridgeUpdated(_bridge, oldBridge);
25:     }
```

```
21: function setBridge(address _bridge) external onlyOwner {
```

```
29: function confirmMintRequest(bytes32 _hash) external onlyRole(MINT_ROLE) {
30:         bridge.confirmMintRequest(_hash);
31:     }
32:
33:     function confirmBurnRequest(
34:         bytes32 _hash,
35:         bytes32 _withdrawalTxid,
36:         uint256 _outputIndex
37:     ) external onlyRole(BURN_ROLE) {
38:         bridge.confirmBurnRequest(_hash, _withdrawalTxid, _outputIndex);
39:     }
40:
41:     function confirmCrosschainRequest(
42:         Request calldata r
43:     )
44:         external
45:         onlyRole(CROSSCHAIN_ROLE)
46:         returns (bytes32 _hash, Request memory _r)
47:     {
48:         (_hash, _r) = bridge.confirmCrosschainRequest(r);
49:     }
50:
51:     function batchConfirmCrosschainRequest(
52:         Request[] calldata rs
53:     ) external onlyRole(CROSSCHAIN_ROLE) {
54:         for (uint256 i = 0; i < rs.length; i++) {
55:             bridge.confirmCrosschainRequest(rs[i]);
56:         }
57:     }
```

```
45: onlyRole(CROSSCHAIN_ROLE)
```

- code/contracts/FireBridge.sol#L80
- code/contracts/FireBridge.sol#L146-L152
- code/contracts/FireBridge.sol#L169-L177
- code/contracts/FireBridge.sol#L169-L172
- code/contracts/FireBridge.sol#L174-L177
- code/contracts/FireBridge.sol#L192-L206
- code/contracts/FireBridge.sol#L294
- code/contracts/FireBridge.sol#L340
- code/contracts/FireBridge.sol#L370-L496
- code/contracts/FireBridge.sol#L397

```
80: address _feeRecipient = feeRecipient;
```

```
146: function removeQualifiedUser(address _qualifiedUser) external onlyOwner {
147:         require(qualifiedUsers.remove(_qualifiedUser), "User not qualified");
148:         string memory _depositAddress = userInfo[_qualifiedUser].depositAddress;
149:         delete depositAddressToUser[_depositAddress];
150:         delete userInfo[_qualifiedUser];
151:         emit QualifiedUserRemoved(_qualifiedUser);
152:     }
```

```
169: function setToken(address _token) external onlyOwner {
170:         fbtc = _token;
171:         emit TokenSet(_token);
172:     }
173:
174:     function setMinter(address _minter) external onlyOwner {
175:         minter = _minter;
176:         emit MinterSet(_minter);
177:     }
```

```
169: function setToken(address _token) external onlyOwner {
170:         fbtc = _token;
171:         emit TokenSet(_token);
172:     }
```

```
174: function setMinter(address _minter) external onlyOwner {
175:         minter = _minter;
176:         emit MinterSet(_minter);
177:     }
```

```
192: function blockDepositTx(
193:         bytes32 _depositTxid,
194:         uint256 _outputIndex
195:     ) external onlyOwner {
196:         bytes32 REJECTED = bytes32(uint256(0xdead));
197:         bytes memory _depositTxData = abi.encode(_depositTxid, _outputIndex);
198:         bytes32 depositDataHash = keccak256(_depositTxData);
199:
200:         bytes32 requestHash = usedDepositTxs[depositDataHash];
201:         require(requestHash == bytes32(0), "Already confirmed or blocked");
202:
203:         // Mark it as rejected.
204:         usedDepositTxs[depositDataHash] = REJECTED;
205:         emit DepositTxBlocked(_depositTxid, _outputIndex);
206:     }
```

```
294: _payFee(_r.fee, false);
```

```
340: _payFee(_r.fee, false);
```

```
370: function confirmMintRequest(
371:         bytes32 _hash
372:     ) external onlyMinter whenNotPaused {
373:         // Check request.
374:         Request storage r = requests[_hash];
375:         require(r.op == Operation.Mint, "Not Mint request");
376:
377:         uint256 _amount = r.amount;
378:         require(_amount > 0, "Invalid request amount");
379:         require(r.status == Status.Pending, "Invalid request status");
380:
381:         // Check and update deposit data usage status.
382:         bytes32 depositDataHash = keccak256(r.extra);
383:         require(
384:             usedDepositTxs[depositDataHash] == bytes32(uint256(0)),
385:             "Used BTC deposit tx"
386:         );
387:         usedDepositTxs[depositDataHash] = _hash;
388:
389:         // Update status.
390:         r.status = Status.Confirmed;
391:         emit RequestConfirmed(_hash);
392:
393:         // Mint tokens
394:         FToken(fbtc).mint(abi.decode(r.dstAddress, (address)), _amount);
395:
396:         // Pay fee.
397:         _payFee(r.fee, true);
398:     }
399:
400:     /// @notice Confirm the burning request.
401:     /// @dev `_withdrawalTxData` packing format is defined by off-chain service.
402:     /// @param _hash The burning request id.
403:     /// @param _withdrawalTxid The BTC withdrawal txid
404:     /// @param _outputIndex The transaction output index to user's withdrawal address.
405:     function confirmBurnRequest(
406:         bytes32 _hash,
407:         bytes32 _withdrawalTxid,
408:         uint256 _outputIndex
409:     ) external onlyMinter whenNotPaused {
410:         // Check request.
411:         require(uint256(_withdrawalTxid) != 0, "Empty withdraw txid");
412:
413:         Request storage r = requests[_hash];
414:
415:         require(r.op == Operation.Burn, "Not Burn request");
416:         require(r.amount > 0, "Invalid request amount");
417:         require(r.status == Status.Pending, "Invalid request status");
418:
419:         bytes memory _withdrawalTxData = abi.encode(
420:             _withdrawalTxid,
421:             _outputIndex
422:         );
423:
424:         bytes32 _withdrawalDataHash = keccak256(_withdrawalTxData);
425:         require(
426:             usedWithdrawalTxs[_withdrawalDataHash] == bytes32(uint256(0)),
427:             "Used BTC withdrawal tx"
428:         );
429:         usedWithdrawalTxs[_withdrawalDataHash] = _hash;
430:
431:         // Update status.
```

```
397: _payFee(r.fee, true);
```

- code/contracts/FToken.sol#L33-L36
- code/contracts/FToken.sol#L43
- code/contracts/FToken.sol#L65-L73

```
33: function lockUser(address _user) external onlyOwner {
34:         userBlocked[_user] = true;
35:         emit UserLocked(_user);
36:     }
```

```
43: function setBridge(address _bridge) external onlyOwner {
```

```
65: function _update(
66:         address from,
67:         address to,
68:         uint256 value
69:     ) internal override whenNotPaused {
70:         require(!userBlocked[from], "from is blocked");
71:         require(!userBlocked[to], "to is blocked");
72:         super._update(from, to, value);
73:     }
```

Description

0xffchain: Owner can block a transaction that is yet to be made causing permanent locking of funds as there is no mechanism to unblock blocked deposits.

```
function blockDepositTx(
       bytes32 _depositTxid,
       uint256 _outputIndex
   ) external onlyOwner {
       bytes32 REJECTED = bytes32(uint256(0xdead));
       // @audit-issue owner can block fututre transactions as the parameters below are
       // not verified if available in storage
       bytes memory _depositTxData = abi.encode(_depositTxid, _outputIndex);
       bytes32 depositDataHash = keccak256(_depositTxData);
       bytes32 requestHash = usedDepositTxs[depositDataHash];
       require(requestHash == bytes32(0), "Already confirmed or blocked");
       // Mark it as rejected.
       usedDepositTxs[depositDataHash] = REJECTED;
       emit DepositTxBlocked(_depositTxid, _outputIndex);
   }
```

As specified in the parameters above, the owner of the contract can block any transaction inclusive of transactions that have not even been created. Meaning deposits that have not even been made on the src chain.
This should not be the case, the owner should check that the request exists then if it does it should block. And not block regardless of if the transaction exists or not.
Kong7ych3: In the protocol, CROSSCHAIN_ROLE can perform cross-chain minting of FBTC through the confirmCrosschainRequest function of the Minter contract. The minting operation is implemented through the

confirmCrosschainRequest function of FireBridge, which relies on the cross-chain minting request passed by CROSSCHAIN_ROLE. All checks rely on CROSSCHAIN_ROLE being honest and reliable. confirmCrosschainRequest does not accept any cross-chain proof to ensure that the confirmCrosschainRequest operation is trustworthy. This means that if the CROSSCHAIN_ROLE private key is leaked, FBTC will be over-issued (e.g., multichain event). This leads to the risk of excessive privileges.
0xhuy0512: In setFeeRecipient() and setFeeModel(), we checked **`address(0)`**

```
    function setFeeModel(address _feeModel) external onlyOwner {
        require(_feeModel != address(0), "Invalid feeModel");
        feeModel = _feeModel;
        emit FeeModelSet(_feeModel);
    }

    function setFeeRecipient(address _feeRecipient) external onlyOwner {
        require(_feeRecipient != address(0), "Invalid feeRecipient");
        feeRecipient = _feeRecipient;
        emit FeeRecipientSet(_feeRecipient);
    }
```

But in **`setToken()`** and **`setMinter()`**, we didn't check address(0):

```
    function setToken(address _token) external onlyOwner {
        fbtc = _token;
        emit TokenSet(_token);
    }

    function setMinter(address _minter) external onlyOwner {
        minter = _minter;
        emit MinterSet(_minter);
    }
```

ravikiran_web3: User FTokens can be locked by the owner at his discretion. The owner has full control on locking the user FTOkens and there by prevent moving of tokens to other accounts.
This issue is more a concern about the centralized control and also does not enforce any specific circumstances under which the owner can exercise this lock.
Saaj:

Vulnerability Details

` FBTCMinter` and **` FToken`** both contract have similar contract that **`setBridge`** function. The function does not have any check to prevent setting the old address as new one.

POC

Here is a test that demonstrate **`oldBridge`** can be passed and set again as new address for the **`bridge`**.

```
    FBTCMinter FBM; // pointer to FBTCMinter contract

    // forge t --mt test_setBridge -vv
    function test_setBridge() external {
        address old_Address = makeAddr("old_Address"); // old address generation
        // address new_Address = makeAddr('new_Address'); // new address generation

        vm.prank(FBM.owner()); // starting call with owner
        FBM.setBridge(old_Address); // setting old address bridge

        address old_bridge = address(FBM.bridge()); //caching old address bridge
        console.log('old bridge address:', old_bridge); // getting old address

        vm.prank(FBM.owner()); // starting another call with owner
        FBM.setBridge(old_Address); // setting old address bridge

        console.log('New bridge address:', address(FBM.bridge())); // getting new address
        assertEq(address(FBM.bridge()), old_Address); // validating old addres again set
    }
```

The result clearly shows both old and new address are same for `**bridge**`.

```
    [PASS] test_setBridge() (gas: 30889)
Logs:
  This is old bridge address: 0xd4FCB2f4947Ba1427ccA3cDDFe0c429829803522
  This is new bridge address: 0xd4FCB2f4947Ba1427ccA3cDDFe0c429829803522

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.03ms (377.78µs CPU time)

Ran 1 test suite in 31.97ms (2.03ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Saaj: ## Vulnerability Details
Contracts `**FBTCMinter**`, `**FBTC**` and `**FBTCGovernorModule**` lack address(0) in their constructor which can lead to address zero being set for bridge address.

Impact

`**bridge**` is a vital address as it is a pointer of `**FireBridge**` contract which is the main contract for the project that have vital functions to carry out cross chain transaction.
`**bridge**` address if sets to address(0) will leave no choice than redeploying again.

POC

Here is a test in which FBTCMinter contract is deployed and passed with a ` **address(0)**` in constructor for `**bridge** ` param.

```
    FBTCMinter public FBM;  // pointer to FBTCMinter contract

    address constant Zero = address(0); // zero address
    address immutable OWNER = address(this); // caller owner contract

    function setUp() public {
        btc = new FBTC(OWNER, Zero); // passing zero address to bridge param
        FBM = new FBTCMinter(OWNER, Zero);
    }

    // forge t --mt test_Bridge -vv
    function test_Bridge() external {
        console2.log('bridge address is:' ,address(FBM.bridge()));
        assertEq(address(FBM.bridge()), address(0)); // asserting bridge address to be address(0)
    }
```

The result clearly shows `**bridge**` is set to `**address(0)**` which will require redeployment of `**FBTCMinter**` contract.

```
[PASS] test_Bridge() (gas: 19544)
Logs:
  bridge address is: 0x0000000000000000000000000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.34ms (193.04µs CPU time)

Ran 1 test suite in 42.98ms (1.34ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

The same issue exist for `**FBTC**` and `**FBTCGovernorModule**` contract as they lack check for ` **address(0)**` in their constructor for `**bridge**`.
Saaj: ## Vulnerability Details
In `**FBTCGovernorModule**` contract, the functions `**setFBTC**` and `**setBridge**` does not have check for zero address which can lead to potential setting these crucial addresses to zero.

POC

This simple foundry test shows if we pass `**address(0)**` as argument in these functions the addresses `**bridge**` & `**fbtc**` can be set to zero.

```
    FBTCGovernorModule FGM; // pointer to FBTCGovernorModule contract
    address constant ZERO = address(0); // zero address

    // forge t --mt test_setter -vv
    function test_setter() external {
        vm.startPrank(FGM.owner()); // starting call with owner

        FGM.setBridge(ZERO);// passing zero address as param
        assertEq(address(FGM.bridge()), address(0)); // validating address set to zero

        FGM.setFBTC(ZERO); // passing zero address as param
        assertEq(address(FGM.fbtc()), address(0)); // validating address set to zero
    }
```

This may impact losing some core functionality for a temporary period until sets to a right address.
0xRizwan: Per the FBTC whitepaper, A qualified user can be an institution, an individual or a merchant. There are two things a qualified users can perform.

1. FBTC minting request via **`addMintRequest()`**
2. FBTC burning request via **`addBurnRequest()`**

A normal user can transfer FBTC to other L1/L2 chains via **`addCrosschainRequest()`**
If the current implementatation is seen carefully, then the following things can be noted and observed to understand this issue:

1. When the qualified users calls **`addMintRequest()`** then the fee is splitted from the actual amount passed and it can be seen at L-252. This does not posses that much impact for this issue.
2. When the qualified users call **`addBurnRequest()`** then the fee is splitted from actual amount and then paid to fee recipient address. After that, the FBTC tokens are burnt and transaction is successfully completed. This can be seen as below:

```
        // Split fee.
        _splitFeeAndUpdate(_r);

        // Save request.
        _hash = _addRequest(_r);

        // Pay fee
        _payFee(_r.fee, false);

        // Burn tokens.
        FToken(fbtc).burn(msg.sender, _r.amount);
```

It means that, the FBTC tokens are burnt first then the actual BTC will be received after the confirmation of burning request by FBTC minter contract. It should be noted that, **`addBurnRequest()`** can only be called when the contract is not paused.

3. Similarly, as explained in point 2 the **`addCrosschainRequest()`** burns the FBTC tokens and send the fee to recipient address before confirmation of cross chain request. This can be seen as below:

```
        // Split fee.
        _splitFeeAndUpdate(_r);

        // Save request.
        _hash = _addRequest(_r);

        // Pay fee
        _payFee(_r.fee, false);

        // Burn tokens.
        FToken(fbtc).burn(msg.sender, _r.amount);
```

It should be noted that, **`addCrosschainRequest()`** can only be called when the contract is not paused.
Now, consider a scenario to understand the issue better:

1. Alice wants to burn her FBTC to get back the BTC which she had sent to custodial address. Alice calls **`addBurnRequest()`** and according function logic, her FBTC burns as well as fee is splitted from amount and sent to fee recipient address.
2. Now, Alice FBTC burning request is sent and it should be confirmed by FBTC minter contract by calling **`confirmBurnRequest()`**. It should be noted that, **`confirmBurnRequest()`** can only be called when the contract is not paused.

```
    function confirmBurnRequest(
        bytes32 _hash,
        bytes32 _withdrawalTxid,
        uint256 _outputIndex
    ) external onlyMinter whenNotPaused {
```

3. Consider that, the contract owner paused the Fire bridge contract after the FBTC burning request sent by Alice. Now the contract is paused so FBTC minter contract can not call **`confirmBurnRequest()`** due to presence of **`whenNotPaused`** modifier which will revert if the funtion is called due to pausing of Fire bridge contract.
4. This pausing could be indefinitely as it depends on owner when to unpause it.
5. Alice's FBTC burning request wont be able to confirm now due to pausing of Fire bridge contract and her burnt FBTC tokens are already deducted from her account address.
6. Alice wont be able to receive her BTC to destination address until the contract is unpaused. This is due to presence of **`whenNotPaused`** modifier.

Impact
Minting, Burning or transfer of FBTC tokens can be indefinitely delayed under special condition of pausing of fire bridge contract. This would affect the users functionality and experience with FBTC protocol as the requests can not be confirmed by FBTC minter contract.
0xRizwan: **`FireBridge.sol`** is the core contract and heart of BTC protocol. This contract is responsible for minting, burning and cross chain transfer of FBTC tokens. FBTC tokens are pegged 1:1 with BTC as per FBTC

whitepaper.
The qualified user i.e indvidual or merchant deposits there BTC to custodial address and take this deposit transaction address with amount to request the equivalent FBTC in `FireBridge.sol`, similarly when the users need their BTC back to their address they can burn the FBTC and the BTC is transferred to their address. A normal user can transfer the FBTC to L1 and L2 chains where FBTC burns on source chain and mints on destination chain.
The issue is that, th `FireBridge` contract allows to change the FBTC token to any token address.

```
    function setToken(address _token) external onlyOwner {
        fbtc = _token;
        emit TokenSet(_token);
    }
```

It means that, the contract owner can change the address of FBTC token to any address like meme token or some invaluable tokens which does not have market value.
Consider below scenario to understand better:

1. Its morning time, Alice wants FBTC and she transfers her BTC to custodial address. She has noted the deposit transation id and amount of BTC deposited. Alice knows that, she will get the FBTC whenever she input the deposit transaction id which is unique to her deposit so she plans to mint the FBTC in evening.
2. After few hours, the owner mistakenly or maliciously or consider owner key leaked calls the `setToken()` function and change the address of FBTC address to some other address.
3. In evening, Alice calls the `addMintRequest()` with the required input arguments and the request is successfully placed. After few moments, the minter contract confirms her `mint` request so that the tokens are minted to her address.
4. Alice received the equivalent amount of tokens, when she checks the token details. The token is different and its not FBTC token by FBTC protocol due to this issue.
5. This issue has breached the trut of Alice due to open threat of FBTC setter function and probably malicious nature of owner but the issue risk probability is high with impact is high.

Impact
Risk of losing the value of deposited BTC tokens with some unknown/invaluable tokens. This impacts direct loss of user funds.
0xRizwan: In `FireBridge.sol`, the owner can removed any qualified users which would delete their user info and deposite address.

```
    function removeQualifiedUser(address _qualifiedUser) external onlyOwner {
        require(qualifiedUsers.remove(_qualifiedUser), "User not qualified");
        string memory _depositAddress = userInfo[_qualifiedUser].depositAddress;
        delete depositAddressToUser[_depositAddress];
        delete userInfo[_qualifiedUser];
        emit QualifiedUserRemoved(_qualifiedUser);
    }
```

When the qualified user is removed, the user can not burn his FBTC tokens to get back the deposited BTC from custodial address as the `addBurnRequest()` can only be called by qualified users.
Consider a scenario:

1. Alice has deposited BTC to FBTC custodial address and inreturn she got the equivalent FBTC tokens. Now, Alice is the qualified user in FBTC protocol. After some days, she decided to get back her BTC by burning FBTC.

2. The contract owner removes Alice address from qualified users by calling `**removeQualifiedUser()**` function and Alice' address is removed.
3. Now, When Alice tries to add FBTC burning request to get back the BTC by calling `**addBurnRequest()**` function, `**addBurnRequest()**` will revert with error `**"Caller not qualified"**` This is due to Alice is not a qualified user anymore.
4. This sudden removed of Alice has restricted her to burn the FBTC to get back BTC.

Note:
It should be noted that, Alice can transfer the FBTC to L1/L2 chains as a normal user, However, this issue is about not getting the BTC to Alice address.
0xRizwan: The FBTC protocol deducts fee in the form of FBTC tokens for minting, burning and cross chain FBTC transfers. FBTC is 1:1 with BTC.
The following functions have used `**_payFee()**` to tranfer the fee to feeRecipient address in the form of FBTC. These functions deduct fee at the time of request of minting/burning/cross chain transfers OR while confirming these requests.

1. `**addBurnRequest()**` at L-294
2. `**addCrosschainRequest()**` at L-340
3. `**confirmMintRequest()**` at L-397

The issue is that, the fee recipient address is not initialized in constructor. setting fee recipient address is not atomic. feeRecipient address is set by another setter function which can be only be accessed by contract owner.

```
    address public feeRecipient;
```

If the feeRecipient is not immediately set after the deployment as its not atomic then the initial transactions of FBTC minting/Burning/cross chain transfer will transfer fee to address(0) as feeRecipient is address variable whose default value would be address(0) if not set by owner.

```
    function _payFee(uint256 _fee, bool viaMint) internal {
        if (_fee == 0) return;

@>        address _feeRecipient = feeRecipient;      @audit // fee recipient as address(0) is not chec
ked, however fee amount as 0 is checked at start of this function
        if (viaMint) {
            FToken(fbtc).mint(_feeRecipient, _fee);
        } else {
            FToken(fbtc).payFee(msg.sender, _feeRecipient, _fee);
        }
        emit FeePaid(_feeRecipient, _fee);
    }
```

It can be seen that, the `**_payFee()**` function does not verify that fee recipient is not address(0) so fee shared by users will be lost permanently and wont be received by FBTC protocol.
Proof of Concept

1. FBTC protocol is successfully deployed and now users start minting/burning/transfer of FBTC tokens. It is assumed that, protocol has not set fee recipient address so by default fee recipient is address(0) as explained above.

2. Alice transfers her FBTC to other L1/L2 networks and the fee is deducted from her actual passed amount via `addCrosschainRequest()` and she successfull got the FBTC to her destination chain.
3. The FBTC protocol thinks, with the transaction of Alice they have received the fee(burning or transfer fee as mentioned in FBTC whitepaper) but in reality they have not received the fee as the fee is transferred to address(0) which is default value of feeRecipient address.
4. This is due to non-atomic setting of feeRecipient address or due to not intializing the value of feeRecipient in constructor or due to not verifying the feeRecipient address in `payFee()`

Impact
Loss of fee to FBTC protocol due to not verifying the feeRecipient address in `_payFee()` function.
0xRizwan: The setter functions and constructor does not check the address(0). It is good security practice to check for address variables are not set accidentally to zero address.
Check the recommendations to fix the issue instances.
biakia: In `FireBridge`, the owner can call `setMinter` to change the minter. If the private key of the owner is compromised, the hacker can change the minter to himself and then call `confirmCrosschainRequest` function to mint FBTC to any address because `confirmCrosschainRequest` function does not check whether the passed-in cross-chain request is legitimate.
BradMoonUESTC: In the smart contract under review, there exists a critical vulnerability within the `setBridge(address _bridge)` function. This function is responsible for updating the contract's `bridge` address, which is integral to the operation of the `pauseBridge()` function. The `pauseBridge()` function is designed to pause operations by invoking the `pause` method on the `bridge` contract.
However, the `setBridge` function sets a new address without validating whether the newly provided `_bridge` address points to a contract that implements the necessary `pause` function. If an incorrect or malicious address is set, any subsequent calls to `pauseBridge()` will fail, causing transaction reversion due to the absence of the `pause` method. This vulnerability exposes the contract to operational disruptions and potential security risks, especially in scenarios requiring the swift pausing of contract operations.

Recommendation

0xffchain:
Do not block future transactions, block only requested mints.
Kong7ych3: CROSSCHAIN_ROLE should be carefully managed. If possible, it is recommended to check whether the cross-chain operation has been signed by the consensus of the cross-chain program when executing cross-chain minting, or to use other methods for secondary verification to mitigate the above risks.
0xhuy0512:

```
    function setToken(address _token) external onlyOwner {
+       require(_token != address(0), "Invalid token");
        fbtc = _token;
        emit TokenSet(_token);
    }

    function setMinter(address _minter) external onlyOwner {
+       require(_minter != address(0), "Invalid minter");
        minter = _minter;
        emit MinterSet(_minter);
    }
```

ravikiran_web3: Locking user tokens at owner's discretion is not an acceptable approach. This implementation brings risk of being locked any time and that will indirectly effect the trust factor for this protocol.
having said that, there can be circumstances under which such exterme action needs to be taken. It will be better to involve more than one account while exercising that extreme action.
Also implement the circumstances which quality action of this exterme action.
Saaj: The recommendation is made to check new address differs from the new address being set for `bridge` in both ` FBTCMinter` and ` FToken` contract.

```
        require(_bridge != address(bridge), 'bridge address already set');
```

Saaj: The recommendation is made for implementing a address zaro check in constructor for `FBTCMinter` contract to avoid setting `bridge` at `address(0)` due to its vital importance.

```
    constructor(address _owner, address _bridge) Ownable(_owner) {
+         require(_bridge != address(0), 'Bridge cannot be zero address');
          bridge = FireBridge(_bridge);
    }
```

Same recommendation is made for `FBTC` and `FBTCGovernorModule` contracts .
Saaj: The recommendation is made to implement zero address check for both functions `setFBTC` and `setBridge` of `FBTCGovernorModule` contract.

```
        require(_fbtc != address(0), 'fbtc cannot be zero address');
        require(_bridge != address(0), 'Bridge cannot be zero address');
```

0xRizwan: Do not restrict the following functions from calling when the contract is paused since the `add request` functions are already burning the FBTC tokens along with burning/transfer fees.

2. Only restrict the `add request` functions with `whenNotPaused` modifier as this is the first step for transaction initiation by users so in emergency only `add request` functions should be restricted with `whenNotPaused` modifier.

Remove `whenNotPaused` modifier based on above detailed explanation.

```
    function confirmBurnRequest(
        bytes32 _hash,
        bytes32 _withdrawalTxid,
        uint256 _outputIndex
-     ) external onlyMinter whenNotPaused {
+      ) external onlyMinter  {
```

```
    function confirmCrosschainRequest(
        Request memory r
    )
        external
        onlyMinter
-        whenNotPaused
        returns (bytes32 _dsthash, Request memory _dstRequest)
    {
```

0xRizwan: Avoid/remove FBTC setter function in `**FireBridge.sol**` contract as FBTC is core part of FBTC protocol and make FBTC address `**immutable**` at contract construction. Initialize `**fbtc**` address in constructor.
Consider below changes:

```
+  address public immutable fbtc;

-    constructor(address _owner, bytes32 _mainChain) {
+        constructor(address _owner, bytes32 _mainChain, address _fbtc) {
        initialize(_owner);
        MAIN_CHAIN = _mainChain;
+        fbtc = _fbtc;
+        emit TokenSet(_fbtc);
    }


    . . . some code

-    function setToken(address _token) external onlyOwner {
-        fbtc = _token;
-        emit TokenSet(_token);
    }
```

0xRizwan: Acknowledge the issue OR Explicitely states the risk in documentation if the qualified user is removed by contract owner, he wont be able to use `**addBurnRequest()**` thereby restricting to get back the BTC.
0xRizwan: validate feeRecipient is not zero address in `**_payFee()**` function.
Consider below changes:

```
    function _payFee(uint256 _fee, bool viaMint) internal {
        if (_fee == 0) return;

        address _feeRecipient = feeRecipient;
+       require(_feeRecipient != address(0), "invalid feeRecipient");
        if (viaMint) {
            FToken(fbtc).mint(_feeRecipient, _fee);
        } else {
            FToken(fbtc).payFee(msg.sender, _feeRecipient, _fee);
        }
        emit FeePaid(_feeRecipient, _fee);
    }
```

0xRizwan: Add a validation check for address(0) in constructors and setter functions.
Consider below changes:
In `FBTCMinter.sol`:

```
    constructor(address _owner, address _bridge) Ownable(_owner) {
+     require(_bridge != address(0), "invalid address");
        bridge = FireBridge(_bridge);
    }

    function setBridge(address _bridge) external onlyOwner {
+     require(_bridge != address(0), "invalid address");
        address oldBridge = address(bridge);
        bridge = FireBridge(_bridge);
        emit BridgeUpdated(_bridge, oldBridge);
    }
```

In `FBTCGovernorModule.sol`:

```
    constructor(
        address _owner,
        address _bridge,
        address _fbtc
    ) Ownable(_owner) {
+       require(_bridge != address(0), "invalid address");
+       require(_fbtc != address(0), "invalid address");
        bridge = FireBridge(_bridge);
        fbtc = FBTC(_fbtc);
    }


        function setFBTC(address _fbtc) external onlyOwner {
+       require(_fbtc != address(0), "invalid address");
        fbtc = FBTC(_fbtc);
        emit FBTCSet(_fbtc);
    }

    function setBridge(address _bridge) external onlyOwner {
+       require(_bridge != address(0), "invalid address");
        bridge = FireBridge(_bridge);
        emit BridgeSet(_bridge);
    }
```

In `**FireBridge.sol**`:

```
    function setToken(address _token) external onlyOwner {
+       require(_token!= address(0), "invalid address");
        fbtc = _token;
        emit TokenSet(_token);
    }

    function setMinter(address _minter) external onlyOwner {
+       require(_minter != address(0), "invalid address");
        minter = _minter;
        emit MinterSet(_minter);
    }
```

```
    function __FToken_init(
        address _owner,
        address _bridge,
        string memory _name,
        string memory _symbol
    ) internal onlyInitializing {
+       require(_bridge != address(0), "invalid address");
        __ERC20_init(_name, _symbol);
        __Governable_init(_owner);
        bridge = _bridge;
    }


    function setBridge(address _bridge) external onlyOwner {
+       require(_bridge != address(0), "invalid address");
        address oldBridge = bridge;
        bridge = _bridge;
        emit BridgeUpdated(_bridge, oldBridge);
    }
```

biakia: Consider using a multi-signature address or a timelock to mitigate the centrazliation risk.
BradMoonUESTC: To mitigate this vulnerability, it is crucial to incorporate a validation mechanism within the `**setBridge**` function to ensure that the provided address points to a contract implementing a `**pause**` function. This can be achieved through a try/catch pattern that attempts to call the `**pause**` function on the proposed contract. If the function call results in an error or is non-existent, the address update should be rejected.
Here is the proposed code modification for the `**setBridge**` function:

```
interface IPausableBridge {
    function pause() external;
}

contract MyContract {
    IPausableBridge public bridge;

    event BridgeSet(address indexed newBridge);

    function setBridge(address _bridge) external onlyOwner {
        // Attempt to call 'pause' to check if the function exists in the provided contract
        try IPausableBridge(_bridge).pause() {
            revert("Provided address does not comply with the required interface");
        } catch {
            // If the call fails, it confirms the function does not exist and we can safely set th
e bridge
            bridge = IPausableBridge(_bridge);
            emit BridgeSet(_bridge);
        }
    }

    // Other functions...
}
```

Client Response

client response for 0xffchain: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for Kong7ych3: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for 0xhuy0512: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for ravikiran_web3: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for Saaj: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for Saaj: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for Saaj: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for 0xRizwan: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for 0xRizwan: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for 0xRizwan: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for 0xRizwan: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.

client response for 0xRizwan: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for biakia: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.
client response for BradMoonUESTC: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.

FBC-12:Centralization risk

Category	Severity	Client Response	Contributor
Governance Manipulation	Low	Acknowledged	Cara

Code Reference

- code/contracts/Governable.sol#L35-L47

```
35: function rescue(address token, address to) external onlyOwner {
36:         if (token == address(0)) {
37:             (bool success, ) = payable(to).call{value: address(this).balance}(
38:                 ""
39:             );
40:             require(success, "ETH transfer failed");
41:         } else {
42:             IERC20(token).safeTransfer(
43:                 to,
44:                 IERC20(token).balanceOf(address(this))
45:             );
46:         }
47:     }
```

Description

Cara: In the project, many interfaces can only be called by privileged accounts, such as the owner. For example, the owner can call the `**rescue**` function to extract assets locked in the contract, set fee configurations, etc. These interfaces are directly related to the circulation of assets. If the owner encounters single points of failure, such as private key leaks, both the project and its users will suffer significant losses.

```
function rescue(address token, address to) external onlyOwner {
    if (token == address(0)) {
        (bool success, ) = payable(to).call{value: address(this).balance}(
            ""
        );
        require(success, "ETH transfer failed");
    } else {
        IERC20(token).safeTransfer(
            to,
            IERC20(token).balanceOf(address(this))
        );
    }
}
```

Recommendation

Cara: It is recommended to use a multi-signature approach to protect privileged interfaces, to avoid single points of failure.

Client Response

client response for Cara: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.

FBC-13:Centralization Risk

Category	Severity	Client Response	Contributor
Privilege Related	Low	Acknowledged	csanuragjain

Code Reference

- code/contracts/FeeModel.sol#L70

```
70: function setChainFeeConfig(
```

- code/contracts/FireBridge.sol#L146

```
146: function removeQualifiedUser(address _qualifiedUser) external onlyOwner {
```

Description

csanuragjain: Seems like owner is allowed to remove a qualified user and then reassign the deposit address to someone else. This can lead to fund loss for initial qualified user

1. Owner adds Qualified User A
2. User A makes a BTC deposit on assigned deposit address
3. Before he can go for mint operation, Owner removes the user from qualified list and adds another Qualified User B with same deposit address
4. User B can now make mint request for the btc deposit made by User A causing loss to User A

Similarly Owner is allowed to change fee from 0-1% which might be unexpected from Users who may loss unexpected amounts as fees if fees changed in same block
Also owner can simply block fee recipient address which ensures that Users will not be able to burn and claim back there btc since fee transfer to fee recipient will fail

Recommendation

csanuragjain: Once a Deposit address has been allocated then it should not be allowed to be reallocated to any other address.
Add timelock to sensitive operations which can impact users

Client Response

client response for csanuragjain: Acknowledged. The owner is a Safe Multisig wallet of the DAO. We will consider to transfer the ownership to a time lock or DAO contract in the future.

FBC-14:Use `require` instead of `assert`

Category	Severity	Client Response	Contributor
Language Specific	Informational	Fixed	Saaj, Cara, 0xzoobi, Yaodao

Code Reference

- code/contracts/FireBridge.sol#L67
- code/contracts/FireBridge.sol#L71
- code/contracts/FireBridge.sol#L90

```
67: assert(nonce() == 1); // Force the request id starts from 1.
```

```
71: assert(r.fee == 0);
```

```
90: assert(r.nonce == requestHashes.length);
```

Description

Saaj:

Vulnerability Details

`assert()` should be avoided for solidity version 0.8.0 and above as Solidity docs clearly states that:
"The assert function creates an error of type Panic(uint256). ... Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix".
Assert should only be limited to testing and not in smart contract to avoid **`Panic`** error.

POC

The below test clearly shows if conditions are not met it will generate a **`Panic`** error instead of reverting.

```
function setUp() public {
    FireBridge BF = new FireBridge(OWNER, chain_ETH); // passing address to bridge param
   }

   // forge t --mt test_initializePanic -vv
   function test_initializePanic() external {
       BF.initialize(OWNER); // calling initialize function
   }
```

The test fails with **`Panic`** error when assert condition is not met.

```
    [FAIL. Reason: panic: assertion failed (0x01)] test_initializePanic() (gas: 29889)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 3.49ms (150.40µs CPU time)

Ran 1 test suite in 99.75ms (3.49ms CPU time): 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/tester.t.sol:tester
[FAIL. Reason: panic: assertion failed (0x01)] test_initializePanic() (gas: 29889)

Encountered a total of 1 failing tests, 0 tests succeeded
```

Cara: Using `require` or `revert` saves gas more than assert. Because when the conditions are not met, `assert` will consume the remaining gas, but `requir`e or `revert` will return it.
0xzoobi: The `assert()` and `require()` functions are a part of the error handling aspect in Solidity. Solidity makes use of state-reverting error handling exceptions. This means all changes made to the contract on that call or any sub-calls are undone if an error is thrown. It also flags an error.
They are quite similar as both check for conditions and if they are not met, would throw an error.
The big difference between the two is that the `assert()` function when false, uses up all the remaining gas and reverts all the changes made.
Meanwhile, a `require()` function when false, also reverts back all the changes made to the contract but does refund all the remaining gas fees we offered to pay. This is the most common Solidity function used by developers for debugging and error handling.
Yaodao: It is recommended to use `require` instead of `assert`. The `require` will return the remaining gas and can return the error message.
For example, the following check is used to check the nonce and will use all gas of the user.

```
    function initialize(address _owner) public initializer {
        __Governable_init(_owner);
        Request memory dummy;
        _addRequest(dummy);
        assert(nonce() == 1); // Force the request id starts from 1.
    }
```

Recommendation

Saaj: The recommendation is made to use `require` instead of `assert()` to avoid panic error.
Cara: It's recommended to use `require` or `revert` instead of `assert`.
0xzoobi: Use `require` instead of `assert`
Yaodao: Recommend using `require` and adding error messages in require statement

Client Response

client response for Saaj: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for Cara: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for 0xzoobi: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for Yaodao: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

FBC-15:Unnecessary check in `fireBridge.confirmCrosschainRequest()`

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	0xhuy0512

Code Reference

- code/contracts/FireBridge.sol#L457

```
457: require(r.extra.length > 0, "Empty cross-chain data");
```

Description

0xhuy0512: In **`fireBridge.confirmCrosschainRequest()`**, we have this line:
`require(r.extra.length > 0, "Empty cross-chain data");`
But later on, we also have this:
`require(r.extra.length == 32, "Invalid extra: not valid bytes32");`
The later line will make the sooner line become unnneccesary

```
    function confirmCrosschainRequest(
        Request memory r
    )
        external
        onlyMinter
        whenNotPaused
        returns (bytes32 _dsthash, Request memory _dstRequest)
    {
        ...
        require(r.extra.length > 0, "Empty cross-chain data");
        ...
        require(r.extra.length == 32, "Invalid extra: not valid bytes32");


        ...
    }
```

Recommendation

0xhuy0512:

```
    function confirmCrosschainRequest(
        Request memory r
    )
        external
        onlyMinter
        whenNotPaused
        returns (bytes32 _dsthash, Request memory _dstRequest)
    {
        ...
-       require(r.extra.length > 0, "Empty cross-chain data");
        ...
        require(r.extra.length == 32, "Invalid extra: not valid bytes32");


        ...
    }
```

Client Response

client response for 0xhuy0512: Fixed.

FBC-16:Unhandle the result of `confirmCrosschainRequest()`

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	Yaodao, 0xhuy0512, 0xRizwan

Code Reference

- code/contracts/FBTCMinter.sol#L51-57
- code/contracts/FBTCMinter.sol#L51-L57
- code/contracts/FBTCMinter.sol#L55

```
51: function batchConfirmCrosschainRequest(
52:         Request[] calldata rs
53:     ) external onlyRole(CROSSCHAIN_ROLE) {
54:         for (uint256 i = 0; i < rs.length; i++) {
55:             bridge.confirmCrosschainRequest(rs[i]);
56:         }
57:     }
```

```
51: function batchConfirmCrosschainRequest(
52:         Request[] calldata rs
53:     ) external onlyRole(CROSSCHAIN_ROLE) {
54:         for (uint256 i = 0; i < rs.length; i++) {
55:             bridge.confirmCrosschainRequest(rs[i]);
56:         }
57:     }
```

```
55: bridge.confirmCrosschainRequest(rs[i]);
```

Description

Yaodao: The function **`batchConfirmCrosschainRequest`** is used to batch confirm the crosschain requests. It will call the function **`bridge.confirmCrosschainRequest`** in loop. The function **`bridge.confirmCrosschainRequest`** will return the **`hash`** and **`request`**. The return values of **`bridge.confirmCrosschainRequest`** are not handled.
0xhuy0512: In **`FBTCMinter.confirmCrosschainRequest()`**, it returning **`bytes32 _hash, Request memory _r`** :

```
    function confirmCrosschainRequest(
        Request calldata r
    )
        external
        onlyRole(CROSSCHAIN_ROLE)
        returns (bytes32 _hash, Request memory _r)
    {
        (_hash, _r) = bridge.confirmCrosschainRequest(r);
    }
```

But in **`batchConfirmCrosschainRequest()`**, it will not return anything:

```
    function batchConfirmCrosschainRequest(
        Request[] calldata rs
    ) external onlyRole(CROSSCHAIN_ROLE) {
        for (uint256 i = 0; i < rs.length; i++) {
            bridge.confirmCrosschainRequest(rs[i]);
        }
    }
```

OxRizwan: In **`FBTCMinter.sol`**, **`batchConfirmCrosschainRequest()`** is used as a batch function used to perform multiple calls simultaneously. Its implemented as below:

```
    function batchConfirmCrosschainRequest(
        Request[] calldata rs
    ) external onlyRole(CROSSCHAIN_ROLE) {
        for (uint256 i = 0; i < rs.length; i++) {
@>          bridge.confirmCrosschainRequest(rs[i]);
        }
    }
```

This function calls **`confirmCrosschainRequest()`** from Fire bridge contract so that the cross chain transfer requests by users can be confirmed and their FBTC can be transferred to destination chains.
The issue is that, the return values of **`confirmCrosschainRequest()`** are not returned by **`batchConfirmCrosschainRequest()`** and the return values are important as it returns **`destination hash`** and **`destination request`** and its explicitely part of **`confirmCrosschainRequest()`** function which is implemented as below:

```
    function confirmCrosschainRequest(
        Request memory r
    )
        external
        onlyMinter
        whenNotPaused
        returns (bytes32 _dsthash, Request memory _dstRequest)
    {
```

Therefore, in batch function , the return values must be returned.
It must be noted that, for single cross chain request confirmation, **`FBTCMinter.confirmCrosschainRequest()`** is used and it returns the return value of **`bridge.confirmCrosschainRequest()`** which can be seen as below:

```
    function confirmCrosschainRequest(
        Request calldata r
    )
        external
        onlyRole(CROSSCHAIN_ROLE)
        returns (bytes32 _hash, Request memory _r)
    {
        (_hash, _r) = bridge.confirmCrosschainRequest(r);
    }
```

Impact
The return values `**_dsthash**` and `**_dstRequest**` are important to be explicitely return along with `**confirmCrosschainRequest()**` function call, otherwise this information wont be returned thereby violating intended design of cross chain request confirmations.

Recommendation

Yaodao: Recommend handling the results of `**bridge.confirmCrosschainRequest**`.
0xhuy0512: Returning values in `**batchConfirmCrosschainRequest()**`, like in `**confirmCrosschainRequest**`
0xRizwan: consider below changes:

```
-    function batchConfirmCrosschainRequest(Request[] calldata rs) external onlyRole(CROSSCHAIN_RO
LE) {
+     function batchConfirmCrosschainRequest(Request[] calldata rs) external onlyRole(CROSSCHAIN_R
OLE) returns(bytes[] _hashes, Request[] memory _r){
        for (uint256 i = 0; i < rs.length; i++) {
-            bridge.confirmCrosschainRequest(rs[i]);
+            (_hashes[i],_r[i]) = bridge.confirmCrosschainRequest(rs[i]);
        }
    }
```

Client Response

client response for Yaodao: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for 0xhuy0512: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c
client response for 0xRizwan: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

FBC-17:Set the Constant to Private

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	Cara

Code Reference

- code/contracts/FBTCGovernorModule.sol#L26-L29

```
26: bytes32 public constant FBTC_PAUSER_ROLE = "1_fbtc_pauser";
27:     bytes32 public constant LOCKER_ROLE = "2_fbtc_locker";
28:     bytes32 public constant BRIDGE_PAUSER_ROLE = "3_bridge_pauser";
29:     bytes32 public constant USER_MANAGER_ROLE = "4_bridge_user_manager";
```

- code/contracts/FBTCMinter.sol#L11-L13

```
11: bytes32 public constant MINT_ROLE = "1_mint";
12:     bytes32 public constant BURN_ROLE = "2_burn";
13:     bytes32 public constant CROSSCHAIN_ROLE = "3_crosschain";
```

- code/contracts/FeeModel.sol#L9

```
9: uint32 public constant FEE_RATE_BASE = 1_000_000;
```

Description

Cara: For constants, if the visibility is set to **`public`**, the compiler will automatically generate a getter function for it, which will consume more gas during deployment. After testing, setting the visibility of the following constants to **`private`** can save approximately 116,000 gas in total.

```
contract FBTCGovernorModule is RoleBasedAccessControl {
    bytes32 public constant FBTC_PAUSER_ROLE = "1_fbtc_pauser";
    bytes32 public constant LOCKER_ROLE = "2_fbtc_locker";
    bytes32 public constant BRIDGE_PAUSER_ROLE = "3_bridge_pauser";
    bytes32 public constant USER_MANAGER_ROLE = "4_bridge_user_manager";
```

```
contract FeeModel is Ownable {
    uint32 public constant FEE_RATE_BASE = 1_000_000;
```

```
contract FBTCMinter is RoleBasedAccessControl {
    ......

    bytes32 public constant MINT_ROLE = "1_mint";
    bytes32 public constant BURN_ROLE = "2_burn";
    bytes32 public constant CROSSCHAIN_ROLE = "3_crosschain";
```

Run the following script using the command **`forge test --match-test testDeployGas --gas-report`**.

```
contract GasTest is Test {
    FBTCMinter public minter;
    FireBridge public bridge;
    FBTC public fbtc;
    FeeModel public feeModel;
    FBTCGovernorModule public fbtcGovernorModule;

    address constant ONE = address(1);
    address immutable OWNER = address(this);

    function testDeployGas() public {
        bridge = new FireBridge(OWNER, ChainCode.BTC);

        feeModel = new FeeModel(OWNER);
        bridge.setFeeModel(address(feeModel));
        FeeModel.FeeConfig memory feeConfig = FeeModel.FeeConfig({active : true, feeRate : 9000, m
inFee : 10000});
        feeModel.setDefaultFeeConfig(Operation.Mint, feeConfig);

        fbtc = new FBTC(OWNER, address(bridge));

        bridge.setToken(address(fbtc));
        minter = new FBTCMinter(OWNER, address(bridge));
        bridge.setMinter(address(minter));
        minter.setBridge(ONE);

        fbtcGovernorModule = new FBTCGovernorModule(OWNER, address(bridge), address(fbtc));
        fbtcGovernorModule.setFBTC(address(fbtc));
    }
}
```

The gas usage for deploying the contract was as follows:

contract	set public	set private	diff
FBTCGovernorModule	1053170	993111	60059
FBTCMinter	1071410	1029364	42046
FeeModel	558852	544639	14213

Recommendation

Cara: It is recommended to set the visibility of constants to `**private**` instead of `**public**`.

Client Response

client response for Cara: Acknowledged. It's marked public intentionally thus users can easily read the ROLE id on the block explorer.

FBC-18:Issues related to hardfork.

Category	Severity	Client Response	Contributor
DOS	Informational	Acknowledged	0xzoobi, 8olidity

Code Reference

- code/contracts/FireBridge.sol#L56
- code/contracts/FireBridge.sol#L243

```
56: bytes32 public immutable MAIN_CHAIN;
```

```
243: dstChain: chain(),
```

Description

0xzoobi: There have been scenarios wherein the chainID of EVM networks tends to change due to hard forks or shifts in consensus mechanisms. Consider recent events like the **PoW vs PoS** Ethereum chain split. If a protocol has hardcoded the chain ID and a hard fork occurs, leading people to migrate to the new chain with a different ID, the protocol ceases to function properly as it relies on outdated information. While the probability of this occurrence is low, one should be prepared.
Regarding **`MAIN_CHAIN`**, which represents the BTC chain, if this serves as a unique identifier that could change during hard forks, it should be configured for compatibility.
8olidity: In the **`addMintRequest`** function, the **`dstChain`** is set to the current chain identifier returned by **`chain()`**. However, if the chain undergoes a hard fork, the chain identifier returned by **`chain()`** may change, leading to incorrect cross-chain operations.

Recommendation

0xzoobi: Add a setter function for **`MAIN_CHAIN`** for future compatibility.
8olidity: In the **`addMintRequest`** function, avoid dynamically fetching the chain identifier using the **`chain()`** function, and instead use a variable that can be dynamically updated to store the chain identifier.

Client Response

client response for 0xzoobi: Acknowledged. . changed severity to Informational
client response for 8olidity: Acknowledged. . changed severity to Informational

FBC-19:For loop gas optimization(All optimizations included)

Category	Severity	Client Response	Contributor
Gas Optimization	Informational	Acknowledged	0xRizwan

Code Reference

- code/contracts/FBTCMinter.sol#L54

```
54: for (uint256 i = 0; i < rs.length; i++) {
```

- code/contracts/FireBridge.sol#L528-L537
- code/contracts/FireBridge.sol#L597

```
528: for (uint256 i = 0; i < qualifiedUsers.length(); ++i) {
529:             UserInfo storage info = userInfo[qualifiedUsers.at(i)];
530:             if (!info.locked) {
531:                 activeCount += 1;
532:             }
533:         }
534:
535:         _users = new address[](activeCount);
536:         uint256 j = 0;
537:         for (uint256 i = 0; i < qualifiedUsers.length(); ++i) {
```

```
597: for (uint i = 0; i < _hashes.length; i++) {
```

Description

0xRizwan: The FBTC contracts have used **`For loop`** particularly in **`FireBridge.sol`** and **`FBTCMinter.sol`** contracts.
For For loop, following gas optimizations can be implemented:

1. The **`unchecked`** keyword is new in solidity version 0.8.0, so this only applies to that version or higher. The FBTC contracts have used solidity version ^0.8.20. This change in For loop will saves 30-40 gas per loop. A detailed informative reference can be checked here.
2. Use pre increment i.e **`++i`** instead of **`i++`**
3. Do not intitialize variables with default value.
4. Catch the array length in local variable instead of repeatedly reading it.

Recommendation

0xRizwan: Consider the above 4 changes for **`For loop`** gas optimizations.
For example:

```
-         for (uint256 i = 0; i < qualifiedUsers.length(); ++i) {
+         for (uint256 i; i < qualifiedUsers.length(); ) {
        . . . some code . . .
+        unchecked{
+            ++i;
+          }
           }
```

Client Response

client response for 0xRizwan: Acknowledged.

FBC-20:Floating solidity pragma version should not be used in contracts

Category	Severity	Client Response	Contributor
Language Specific	Informational	Fixed	0xRizwan

Code Reference

- code/contracts/FireBridge.sol#L2

```
2: pragma solidity ^0.8.20;
```

Description

0xRizwan: 1) An outdated compiler version might introduce the vulnerabilities which can affect the contracts negatively or recently released pragma versions may have unknown security vulnerabilities.

2. Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues mentioned here can affect the current compiler version. Such known compiler bugs can be checked at below link- https://github.com/ethereum/solidity/releases

The following contracts have used floating pragma and may affect with this issue: **`FireBridge.sol`**, **`FBTCGovernorModule.sol`**, **`Common.sol`**, **`FeeModel.sol`**, **`FToken.sol`**, **`RoleBasedAccessControl.sol`**, **`FBTCMinter.sol`**, **`Governable.sol`**, **`BridgeStorage.sol`** and **`FBTC.sol`** contracts.

Recommendation

0xRizwan: Lock the solidity version in contracts and avoid using floating pragma version, Preferably use the latest version of solidity.
For example:

```
- pragma solidity ^0.8.20;
+ pragma solidity 0.8.20;
```

Client Response

client response for 0xRizwan: Fixed.commit id - 33737b455568434bbdee51b93a3badcb0d52575c

FBC-21:Consider using OpenZeppelin's SafeCast library to prevent unexpected overflows when downcasting

Category	Severity	Client Response	Contributor
Logical	Informational	Fixed	rajatbeladiya

Code Reference

- code/contracts/FireBridge.sol#L507

```
507: return uint128(requestHashes.length);
```

Description

rajatbeladiya: Downcasting from `uint256`/`int256` in Solidity does not revert on overflow. This can result in undesired exploitation or bugs, since developers usually assume that overflows raise errors. OpenZeppelin's SafeCast library restores this intuition by reverting the transaction when such an operation overflows. Using this library eliminates an entire class of bugs, so it's recommended to use it always. Some exceptions are acceptable like with the classic `uint256(uint160(address(variable)))`

Recommendation

rajatbeladiya: Use Openzeppelin's SafeCast library

Client Response

client response for rajatbeladiya: Fixed. commit id - 33737b455568434bbdee51b93a3badcb0d52575c

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.