BIFROST

BiFi-Bifrost-Extension Security Audit

revision 1.2

Prepared for
Bifrost

Prepared by
Theori

September 9th, 2021

Theori

Table of Contents

Executive Summary

Starting on August 11, 2021, Theori assessed BiFi-Bifrost-Extension, a protocol to bridge BTC from the Bitcoin network to the Ethereum network for use in BiFi. The fundamental features of the protocol are swapping in BTC for BiBTC and swapping out BiBTC for BTC. BiBTC is an internal representation of BTC used within BiFi. We focused on identifying issues that result is a loss of locked BTC or improper minting of BiBTC. We reviewed the Ethereum smart contracts and the off-chain scripts. Three critical issues were identified and promptly fixed by Bifrost.

Scope

BiFi-Bifrost-Extension, a protocol to bridge BTC from Bitcoin to Ethereum, is audited.

The code was retrieved on August 11, 2021, from:

- bifrost-platform/Bifrost-BiFi-extension-Contract
- bifrost-platform/Bifrost-BiFi-Extension-Bifrost
- bifrost-platform/Bifrost-BiFi-Extension-Relayer
- bifrost-platform/Bifrost-BiFi-Extension-Python-Package

Overview

The assessment of the smart contracts and off-chain scripts focused on risks to user funds. While the scope of our audit is limited to vulnerabilities in the software, in this section we discuss risks to the protocol that are outside of our scope. We assumed that the Bifrost team is trusted as they retain control of the smart contracts and the Bifrost server. We also assumed that the Bifrost server is properly secured. These assumptions are fundamental to the security of the protocol.

Centralization

BiFi-Bifrost-Extension relies on the Bifrost server for proper operation and as such is a centralized bridge. The Bifrost server is responsible for issuing BTC deposit addresses and retains sole control of the private key for all locked assets on the Bitcoin blockchain. If the Bifrost server becomes permanently unavailable, the locked assets will be lost and it will be impossible for users to swap out their BiBTC for BTC. Additionally, if an attacker can gain access to the Bifrost server, they could steal all the BTC currently locked in BiFi-Bifrost-Extension.

Unlike the Bifrost server, the relayers are decentralized and use staking to reward good behavior. Anyone can become a relayer by staking BFC and then submitting Bitcoin blocks to the BiFi-Bifrost-Extension smart contracts. While anyone can become a relayer, there are currently no incentives to be a relayer. The user fees go to the protocol, not the relayers, the only exception is when challenging a block submitted by another relayer.

Given that the protocol cannot operate without the Bifrost server and there are no incentives for relayers, Bifrost should consider allowing only trusted relayers. This would reduce the attack surface of the protocol.

Risks

The risks to user funds are similar to other bridged tokens:

- Vulnerability in protocol
- Attacks on Bifrost server infrastructure
- Bifrost server stops operating

There are two competing thoughts on software security. One idea is to open source the code so that security researchers can look for vulnerabilities and report them. The alternative is to keep the source code hidden to prevent attackers from easily analyzing the software. We recommend that the off-chain software, e.g. Bifrost server and relayer scripts, is not released

publicly so that it cannot be easily analyzed. In the case of the smart contracts, an attacker can download the smart contract bytecode and attempt to decompile it, so the benefits of hiding the source code are less pronounced.

The Bifrost server infrastructure should be secured according to best practices. Limit remote access to the minimum number of employees, utilize two factor authentication, and log all admin activities to another server. For additional security, run multiple instances of the Bifrost server and use multisig wallets so that a majority of the servers must approve and sign a transaction. These should run on diverse cloud platforms and admin access to the platforms should be distinct sets of people. Lastly, a cold wallet could be used to store funds to prevent online attacks, though this has some UX difficulties because large swap outs may require moving funds to the hot wallet.

Even if the protocol and servers are not attacked, BiFi-Bifrost-Extension relies on Bifrost continuing to operate the servers. If these servers became unavailable, then BTC locked in BiFi-Bifrost-Extension will be lost. While users of DeFi platforms already rely on admin keys to not be abused, in the case of centralized bridges like BiFi-Bifrost-Extension or WBTC, users also rely on the companies backing these protocols to continue operating the bridges.

Findings

These are the potential issues that may have correctness and/or security impacts. We advise Bifrost team to remediate found issues quickly to ensure the safety of the contract.

Summary

#	ID	Title	Severity
1	THE-BIFIBTC-001	Merkle tree verification may return a partial result	Fixed (Critical)
2	THE-BIFIBTC-002	BTC output script parsing is incomplete	Fixed (Critical)
3	THE-BIFIBTC-003	Bifrost server does not validate event address	Fixed (Critical)
4	THE-BIFIBTC-004	Do not use an AMM naively as a price oracle	Medium

Issue #1: Merkle tree verification may return a partial result

ID	Summary	Severity
THE-BIFIBTC-001	Smart contract uses the wrong Merkle tree hash when provided with an invalid combination of inputs	Fixed (Critical)

In BTCPureLibs.sol, the *computeMultiMerkleRoot* function is supposed to return a Merkle tree root given a set of leaf hashes and a Merkle proof. However, it fails to ensure that it has calculated the root and it may instead return a hash from a non-root node. This allows an attacker to inject arbitrary transactions into a BTC block that the smart contract will accept because these hashes are not included in the "root" hash returned by *computeMultiMerkleRoot*.

```
function computeMultiMerkleRoot(
    bytes32[] calldata leaves,
    bytes32[] calldata merkleProof,
    bytes calldata proofSwitches
) external pure returns (bytes32) {
    bytes32 left; bytes32 right;
    uint256 leavespos; uint256 hashpos; uint256 proofpos;

    bytes32[] memory hashes = new bytes32[](proofSwitches.length);

    for(uint256 i; i < proofSwitches.length; i++) {
        if(proofSwitches[i] == 0x01) left = merkleProof[proofpos++];
        else left = (leavespos < leaves.length ? leaves[leavespos++] :
hashes[hashpos++]);

        if(proofSwitches[i] == 0x02) right = merkleProof[proofpos++];
        else if(proofSwitches[i] == 0x00) right = left;
        else right = (leavespos < leaves.length ? leaves[leavespos++] :
hashes[hashpos++]);

        hashes[ i ] = left.hash256_concat(right);
    }
    return hashes[ hashes.length-1 ];
}
```

Let's consider two examples, one valid and one malicious:

```
Transactions to verify:
```

```
 -
0x01000000017e293b848be2062e90cc97190300b66a4201c34c20cc6b0ad1660079c53f7f370100000023
22002075120ac5c23a6ac16d1621fa202830c80e126e78f4fa00f52c316a56ce13b768ffffffff025e1879
010000000017a9143c194d9b0a4889c9d9ca033b70bbcabf22f933988772bf0500000000001976a9149b80
9872c8af1a67388370ee892bd8ddef825f6188ac00000000

Leaves (e.g. relayed transaction hashes):
 - 0x00174814ce0157e1bd2b2ef822d1c77441e7fe1f176f77f2d67378d29900f3d1

Merkle proof
 - 0xbe9fed0d6ae9079b956c08fe27b96a36c0d00dd9b23492db8dab8e693041a93b
 - 0x95cfa93a96fe98c3dcc356da2a168949ff15bf3833b9123364bbd62346959552
 - 0x055ab134b11471bd199eebb6b97b8421a243e6a7dc277827a6717163526e19d9
 - 0xacf24d0d619fa43c3cf32d25b41b2d4e0d9ee335a11bb86ea5104312b2755475
 - 0x2b10ae8f1b41580208df35e6406e682e06e8dd5912138e34c9d6cfeb5e38f178
 - 0x75ce9b6ffd7a044c31f207feaf12b6054c6a6052286ff7b5ae3263904bb3f719
 - 0x92b2c89971c2a8d95b2b9f9fb4c3b8e3ddda540d34a050c8d1819dd5a705d0e3
 - 0x45cb58a0223b22fcf30999470b4f6fd59b491925e0e5bb04567a80cf9a19b4a5
 - 0xbd500670afd7cc5eaa90dc15961d97171c8ce0ad34dc7cf430cd3cf10a7bc7cd
 - 0x253d375eeee3865399367000d23f6b47a6a6c7fbe2e2581fc0664355c400d993
 - 0x259226740106a71da64640ffc0863ee07228de099d7b8126eded765e332e878b
 - 0xe002ee189294d842fe91f966936d1bf7b356e61da053038e0b1342ab95afc58b

Proof switches
 - 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2

Intermediate hashes
 - 0x7c53386ed00d3f7e4bde64111dd280e137a8df88e3cc6aaa79ecf4a55a43e932
 - 0x1ca789ee451941c113986d865fc453acacd1f10c0bb31fc5385039c07d9df3a4
 - 0x863e5943b4c979109a8e180dac7798a96108070d6afb755a3762c21df1b31ba6
 - 0x8e84205cd1d1c388b2c19c8c49fc1374f8b710dab7c8625430907720b76fe9aa
 - 0x839b34488e10f952dff047d43b9e6fc338aa2cc82d8d9dee06b56bf9b7a54588
 - 0x770ec5068679ccebaa4bc0983783e8f30544265d273f572f91cd6d8b81e44e56
 - 0xd35bd87df640ebf938a18a4db6af1a8992ab5b04f6844ee7bab1d151a0e12e1c
 - 0x7905b92dbba83cea1dc54e2ba8011c9a24294145464fff6926abc7545e5c4b46
 - 0xb8680943524e738bbd9e0774fde568fe703899b95d14806991ed0b1eb7030261
 - 0xabce43d8bb57f5af280e37a54b9938f6945b37cd16381650e287868957c3c2b7
 - 0x101f112e9fba6ede550321ca411a99d4cc8eb172cb3f46c53b2bc0a4ef1857e0
 - 0x6e3b2f1c647567cee10fe459ee67938e942f8de4bab535b721a3c7d740b05f47 (result)
```

In this example, every proof switch is 0x1 or 0x2, so every iteration a leaf hash or intermediate hash is consumed and a Merkle proof hash is consumed. Also, every iteration a new intermediate hash is generated. There are 12 proof switches, 12 Merkle proofs, and 1 leaf hash, so after 12 iterations, every input leaf hash and every intermediate hash (except the last one) has been consumed.

```
Transactions to verify:
 -
0x01000000017e293b848be2062e90cc97190300b66a4201c34c20cc6b0ad1660079c53f7f370100000023
22002075120ac5c23a6ac16d1621fa202830c80e126e78f4fa00f52c316a56ce13b768ffffffff025e1879
010000000017a9143c194d9b0a4889c9d9ca033b70bbcabf22f933988772bf0500000000001976a9149b80
9872c8af1a67388370ee892bd8ddef825f6188ac00000000
 - 0xdeadbeef
```

```
Leaves (e.g. relayed transaction hashes):
 - 0x00174814ce0157e1bd2b2ef822d1c77441e7fe1f176f77f2d67378d29900f3d1
 - 0x281dd50f6f56bc6e867fe73dd614a73c55a647a479704f64804b574cafb0f5c5

Merkle proof
 - 0xbe9fed0d6ae9079b956c08fe27b96a36c0d00dd9b23492db8dab8e693041a93b
 - 0x0
 - 0x95cfa93a96fe98c3dcc356da2a168949ff15bf3833b9123364bbd62346959552
 - 0x0
 - 0x055ab134b11471bd199eebb6b97b8421a243e6a7dc277827a6717163526e19d9
 - 0x0
 - 0xacf24d0d619fa43c3cf32d25b41b2d4e0d9ee335a11bb86ea5104312b2755475
 - 0x0
 - 0x2b10ae8f1b41580208df35e6406e682e06e8dd5912138e34c9d6cfeb5e38f178
 - 0x0
 - 0x75ce9b6ffd7a044c31f207feaf12b6054c6a6052286ff7b5ae3263904bb3f719
 - 0x0
 - 0x92b2c89971c2a8d95b2b9f9fb4c3b8e3ddda540d34a050c8d1819dd5a705d0e3
 - 0x0
 - 0x45cb58a0223b22fcf30999470b4f6fd59b491925e0e5bb04567a80cf9a19b4a5
 - 0x0
 - 0xbd500670afd7cc5eaa90dc15961d97171c8ce0ad34dc7cf430cd3cf10a7bc7cd
 - 0x0
 - 0x253d375eeee3865399367000d23f6b47a6a6c7fbe2e2581fc0664355c400d993
 - 0x0
 - 0x259226740106a71da64640ffc0863ee07228de099d7b8126eded765e332e878b
 - 0x0
 - 0xe002ee189294d842fe91f966936d1bf7b356e61da053038e0b1342ab95afc58b
 - 0x0

Proof switches
 - 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2

Intermediate hashes
 - 0x7c53386ed00d3f7e4bde64111dd280e137a8df88e3cc6aaa79ecf4a55a43e932
 - 0x094323846674918286ca8693256e45419ab9be71bc54e1af72b969f206fc23b5
 - 0x1ca789ee451941c113986d865fc453acacd1f10c0bb31fc5385039c07d9df3a4
 - 0xa931cd7945a438ce10f3cb6deb43ac3d5660093771eccdb24068d41709723c4a
 - 0x863e5943b4c979109a8e180dac7798a96108070d6afb755a3762c21df1b31ba6
 - 0x0a4dc8a7bbd1daaa7b456320de503c69911af02a56fcbf0bc58d02e473e701cb
 - 0x8e84205cd1d1c388b2c19c8c49fc1374f8b710dab7c8625430907720b76fe9aa
 - 0x66de0c62381e37d7075bcbc91ca042140357b6b26621bf820d52829991e4d654
 - 0x839b34488e10f952dff047d43b9e6fc338aa2cc82d8d9dee06b56bf9b7a54588
 - 0x9804614f56e5b0c69d4191839790696b62f511a72e64d62d3692ad6cc2621e40
 - 0x770ec5068679ccebaa4bc0983783e8f30544265d273f572f91cd6d8b81e44e56
 - 0xe50f370c348bdf39e7dc1c83102d3ffd81f04b7759fe28b20df46df4fd9fd431
 - 0xd35bd87df640ebf938a18a4db6af1a8992ab5b04f6844ee7bab1d151a0e12e1c
 - 0xc5562c3f2d61b311bfbd8e859f1fcbb5c3788101beef926b653310ad2ff7d8f3
 - 0x7905b92dbba83cea1dc54e2ba8011c9a24294145464fff6926abc7545e5c4b46
 - 0x3bde8d791e88010c824f4b7e7d7129e521a4aaf4f144332bb4592d3f516d994b
 - 0xb8680943524e738bbd9e0774fde568fe703899b95d14806991ed0b1eb7030261
 - 0x4381f8b02a8a6b0a2753452e9961e9c7c5e52179548c1ff54b421683f46cbc4b
 - 0xabce43d8bb57f5af280e37a54b9938f6945b37cd16381650e287868957c3c2b7
 - 0xb01861c2c92cfbc891f0cd49ae49f89d724c7e7899451c98443a647738119502
 - 0x101f112e9fba6ede550321ca411a99d4cc8eb172cb3f46c53b2bc0a4ef1857e0
 - 0x65cfbfca3b93d6cedb73256680320e4a1f53b9cfd8eb6bd92d9f3c27f0249d4b (not consumed)
 - 0x6e3b2f1c647567cee10fe459ee67938e942f8de4bab535b721a3c7d740b05f47 (result)
```

This example produces the same result as the previous example even though it includes the fake transaction *0xdeadbeef*. Essentially, we construct a Merkle tree where the left half is the original Merkle tree and the right half is a fake Merkle tree. Then, by not specifying enough switches, we force *computeMultiMerkleRoot* to return a hash of only the left half of the Merkle tree. The hash of the right half of the Merkle tree is never used.

Fix

We suggested that *computeMultiMerkleRoot* enforce that it has consumed all of the leaf hashes and intermediate hashes, except for the final root hash. Bifrost applied this fix:

```
bytes32[] memory hashes = new bytes32[](proofSwitches.length);

for(uint256 i; i < proofSwitches.length; i++) {
    if(proofSwitches[i] == 0x01) left = merkleProof[proofpos++];
    else left = (leavespos < leaves.length ? leaves[leavespos++] : hashes[hashpos++]);

    if(proofSwitches[i] == 0x02) right = merkleProof[proofpos++];
    else if(proofSwitches[i] == 0x00) right = left;
    else right = (leavespos < leaves.length ? leaves[leavespos++] :
hashes[hashpos++]);

    hashes[ i ] = left.hash256_concat(right);
}
// to ensure that the merkle root is calculated from all the given txs and merkle
proofs.
require(leavespos == leaves.length && hashpos == hashes.length-1,
"computeMultiMerkleRoot");
return hashes[ hashes.length-1 ];
```

Bifrost may want to consider using a standard Merkle tree proof with one transaction at a time. Merkle multiproofs are less standard and have pitfalls. Given that there are a relatively small number of BiFi-Bifrost-Extension transactions, the overhead of one Merkle proof per transaction will be small.

Issue #2: BTC output script parsing is incomplete

ID	Summary	Severity
THE-BIFIBTC-002	Smart contract fails to validate opcodes in the output scripts when parsing addresses from Bitcoin transactions	Fixed (Critical)

Bitcoin is a ledger where every transaction has a set of inputs and outputs and each of these has a script. An input references a previous output, and when the previous output's script is combined with the input script, it must return a "true" value. As such, the Bitcoin blockchain does not have the concept of an account or an address, unlike Ethereum. Instead, the output script encodes the logic to accept a particular public key, for example.

In BTCPureLibs.sol, *parse_unit_from_rawTX* parses the output addresses and amounts from a raw transaction. However, the logic is too lax and can be fooled by non-standard output scripts:

```
// BTC amount
units[j].amount = rawTX.parseUint64_bigend(amount_pos);
// output PubKeyHash parsing
if(output_script_len == 0x19) units[j++].BTCaddr = rawTX.parseAddress(pos+3);
else if(output_script_len == 0x16) units[j++].BTCaddr = rawTX.parseAddress(pos+2);
else if(output_script_len == 0x17) units[j++].BTCaddr = rawTX.parseAddress(pos+2);
else revert("unknown pubkey script");
```

For instance, consider an output script looks exactly like a standard P2PKH output script (pay-to-pubkey-hash) except it always returns true. An attacker could then immediately spend this output, even though *parse_unit_from_rawTX* thinks the funds were sent to a BiFi-Bifrost-Extension deposit address.

Fix

We suggested that all non-address bytes of the output script are verified. There are a limited number of supported, standard scripts, so validation is easy. Bifrost applied this fix:

```
// BTC amount
units[j].amount = rawTX.parseUint64_bigend(amount_pos);
// output PubKeyHash parsing

// pay-to-pubkey-hash
if(output_script_len == 0x19) {
    require(
```

```
        rawTX[pos   ] == 0x76 &&    // OP_DUP
        rawTX[pos+1 ] == 0xa9 &&    // OP_HASH160
        rawTX[pos+2 ] == 0x14 &&    // addr len(varint: 20)
        rawTX[pos+23] == 0x88 &&    // OP_EQUALVERIFY
        rawTX[pos+24] == 0xac,      // OP_CHECKSIG
        "invalid P2PKH"
    );
    units[j++].BTCaddr = rawTX.parseAddress(pos+3);
}
// pay-to-witness-pubkey-hash
else if(output_script_len == 0x16) {
    require(
        rawTX[pos   ] == 0x00 &&    // OP_RETURN
        rawTX[pos+1 ] == 0x14,      // addr len(varint: 20)
        "invalid P2WPKH"
    );
    units[j++].BTCaddr = rawTX.parseAddress(pos+2);
}
// pay-to-script-hash
else if(output_script_len == 0x17) {
    require(
        rawTX[pos   ] == 0xa9 &&    // OP_HASH160
        rawTX[pos+1 ] == 0x14 &&    // addr len(varint: 20)
        rawTX[pos+22] == 0x87,      // OP_EQUAL
        "invalid P2SH"
    );
    units[j++].BTCaddr = rawTX.parseAddress(pos+2);
}
else revert("invalid output script");
```

Issue #3: Bifrost server does not validate event address

ID	Summary	Severity
THE-BIFIBTC-003	Bifrost server does not validate that event logs are emitted by the correct smart contract	Fixed (Critical)

A user swaps out BiBTC on Ethereum for BTC on Bitcoin by calling the *swapOut* function in BTCEntryLogicExternal.sol. This on-chain logic will burn the BiBTC and emit an *OutFlow* event if the swap out is valid:

```
// emit ouflow event, Bifrost will send bitcoin transaction
emit OutFlow(recipient, refundPubkeyHash, addressFormatType, actionType,
memOutflow.pendingAmount, memOutflow.timeLimit);
```

The Bifrost server monitors the Ethereum blockchain for *OutFlow* events and then sends BTC according to the event. It also monitors for other events, such as the *Registration* event. If we compare the logic for the *OutFlow* event and the *Registration* event, we can see that the *OutFlow* event is missing a check of the event address:

```
for transaction in block.transactions:
    for log in self.registration_decoder.find_logs(transaction=transaction):
        if log.address != HexBytes(self.resolver_contract.resolver.address):
            continue
...
```

```
for transaction in block.transactions:
    for log in self.outflow_decoder.find_logs(transaction=transaction):
        if Outflow.objects(
                eth_user_address=Web3.toChecksumAddress(transaction.from_.hex()),
                eth_tx_hash=transaction.hash.hex(),
                log_index=log['logIndex'],
        ).count() > 0:
            self.logger.warning("Duplicate outflow detected. %s",
transaction.hash.hex())
            continue

        event_data = self.outflow_decoder.decode(transaction=transaction, log=log)
...
```

Fix

We suggested that a check of the event address was added. Bifrost refactored the code slightly and moved the check into *find_logs*:

```
def find_logs(self, transaction: EthereumTransaction) -> List[dict]:
    logs = list()
    for log in transaction.logs:
        if log.address != HexBytes(self.contract.address):
            continue
        for event in self.interested_events:
            try:
              logs.append(self.contract.events[event]().processLog(log=log.to_dict()))
            except MismatchedABI:
              continue
    return logs
```

Issue #4: Do not use AMM naively as a price oracle

ID	Summary	Severity
THE-BIFIBTC-004	An automated market maker smart contract should never be used as a price oracle without mitigations against price manipulation	Medium

In FundInternal.sol, `_getPrice_btc_bfc18` gets the price of BTC in terms of BFC by getting the price of BTC and BFC in terms of ETH. The price of BTC is from a Chainlink price feed, which is considered to be a safe price oracle. The price of BFC, however, is from the AMM pair for BFC and ETH. It is well known that the price of assets on an AMM can be manipulated with flash loans.

The vulnerable function, `_getPrice_btc_bfc18`, is used in two places: `calcOutflowFee` and `penaltyTransfer`. Manipulating the price of BFC in `calcOutflowFee` has limited benefit, the attacker can only use it to reduce the fee they pay to swap out BiBTC for BTC. Manipulating `penaltyTransfer` is more interesting because an attacker could potentially steal BFC. The risk of exploitation is low because the attacker must be a relayer and the attacker must trigger the penalty condition (i.e., an outflow that is not relayed fast enough).

Recommendations

Uniswap has an example of a moving average to avoid price manipulation when using an AMM as an oracle[1]. Alternatively, Bifrost could collect prices off-chain from various exchanges and have a trusted account store the latest price in Ethereum once per interval (e.g., every hour).

Fix

Bifrost will use various safe price oracles for the price of BFC depending on the AMM pool situation in the future.

[1] https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/examples/ExampleSlidingWindowOracle.sol

Recommendations

These are the recommendations to improve the code quality for better readability, optimization, and security. They do not impose any immediate security impacts.

Summary

#	Title	Type	Importance
1	Validate that length of BTC transaction is not 64 bytes	Security	Medium
2	Unused variable in _popRelayer	Code quality	Minor
3	Avoid using temporary variables with generic names	Code quality	Minor
4	Refactor "garbage number" in _checkHeaderChain	Code quality	Minor
5	_checkHeaderChain should explicitly revert if challenge is not yet resolved	Code quality	Medium
6	Use only approved relayers	Security	Minor
7	Use multisig for Bifrost server accounts	Security	Minor

Recommendation #1: Validate that length of BTC transaction is not 64 bytes

A known issue with Bitcoin's implementation of a Merkle tree is there is no distinction between a leaf node and an internal node. An attacker could submit an internal node (e.g., *leftChild || rightChild*) as a transaction and it would be accepted because it is in the Merkle tree.

While this behavior is unfortunate, it should not be security critical since an attacker cannot control enough of the value *leftChild || rightChild* without breaking the security assumptions of SHA-256.

One way to mitigate this risk is by ensuring that the relayed raw transactions are not exactly 64 bytes. This should not break normal behavior because the length of a transaction with at least one input and at least one supported output is longer than 64 bytes.

Fix

Bifrost added a check that the raw transaction is not exactly 64 bytes:

```
function parse_unit_from_rawTX(
    bytes calldata rawTX,
    uint256[] calldata units_indices
) external pure returns (S_Unit[] memory units) {
    require(rawTX.length != 64, "btc tx length 64");
    ...
}
```

Recommendation #2: Unused variable in _popRelayer

In FundExternal.sol, the *_popRelayer* function copies the *relayers* storage variable in to the *_relayers* memory variable but then continues to reference the storage variable. This could waste significant gas depending on the number of relayers. Either remove the *_relayers* variable or use *_relayers* instead of *relayers*.

Fix

Bifrost removed the *_relayers* variable.

Recommendation #3: Avoid using temporary variables with generic names

In BTCEntryLogicInternal.sol, the *_checkHeaderChain* function uses fields in a memory variable to avoid having too many stack variables. The problem is that one of these fields, *tmpUint*, is reused and has multiple meanings. This makes the code harder to read and harder to analyze. If there is no way to eliminate the use of *tmpUint*, then the code is probably poorly organized and should be refactored.

Recommendation #4: Fix "garbage number" in _checkHeaderChain

In BTCEntryLogicInternal.sol, the *_checkHeaderChain* function sets the temporary variable *tmpUint* to a "garbage number" without any thorough documentation. It is difficult for a reader to understand why it is a garbage number and what the code author is trying to accomplish.

```
if( vars.tmpUint + RetargetBlockInterval-1 <= vars.endHeight &&
    RetargetBlockInterval-1 != _modRetargetBlockInterval(vars.anchorHeight) //avoid
last tx did submit at 2015
) {
    // get rawHeadersIndex for nextTarget
    // vars.tmpUint = vars.tmpUint + RetargetBlockInterval-1 -vars.anchorHeight -1;
    vars.tmpUint = vars.tmpUint + RetargetBlockInterval -vars.anchorHeight -2;
    vars.tmpUint *= Header_Length;
} else {
    // submit range not matched to target update(set garbage number)
    vars.tmpUint = RetargetBlockInterval;
}
```

The code author is trying to avoid a later conditional that tests tmpUint: *if(i == vars.tmpUint)*. Since the iterator *i* is a multiple of *Header_Length* and *RetargetBlockInterval % Header_Length != 0*, the condition will not be satisfied with the "garbage number". A much better choice of a garbage number would be max unsigned int. This would make the code's behavior more obvious to the reader.

Fix

Bifrost changed the value to uint256 max and improved the source code comment:

```
// There is no difficultyTarget update within this 'submit range'
// Move the update point to outside of the range (as int max) to avoid difficulty
recalculation
```

```
vars.tmpUint = type(uint256).max;
```

Recommendation #5: _checkHeaderChain should explicitly handle challenge timeout

In BTCEntryLogicInternal.sol, the *_checkHeaderChain* function has logic to resolve a challenge. The relayer must send enough headers (*Confirm_Guarantee*) and the challenge must not have timed out. However, if the challenge has timed out, *vars.anchorHeight* and *vars.endHeight* will be uninitialized. This will likely cause the function to eventually revert.

```
if(challengeResolveFlag) {
    if(ctx.challengeETHheight + challengeTimeout >= block.number) {
        if( vars.anchorHash == ctx.challengeHash) {
        ...
        } else if (vars.anchorHash == _db_get_hashByHeight(cons.db,
ctx.challengeBTCheight)) {
        ...
        } else {
            revert("undefined resolve");
        }
    } else {
        // HANDLE CHALLENGE TIMEOUT
    }
    ...
} else {
    ...
}
```

The challenge time out case should be explicitly handled. If the explicit behavior is to revert, then there needs to be a way for an admin or relayer to clear the challenge state. Otherwise, no new blocks can be submitted.

Another edge case in *_checkHeaderChain* is if there are multiple competing forks. The current code assumes that there will only ever be one fork at a time and one of the two forks will win with enough headers. If there is a third fork that becomes the canonical chain, then it may be possible that the challenge cannot be satisfied and the relayers must wait until the challenge timeout.

Recommendation #6: Use only approved relayers

BiFi-Bifrost-Extension relies on the Bifrost server because it has the private keys for the BTC deposit addresses. As such, there is almost no current benefit to allowing non-Bifrost

relayers. The attack surface of the protocol and its smart contracts could be reduced by having a trusted set of relayers that are operated by Bifrost.

Fix

Bifrost will add an approved list of relayers as a temporary measure. This will be removed in the future to support decentralization.

Recommendation #7: Use multisig for Bifrost server accounts

Since BiFi-Bifrost-Extension is a centralized protocol, the Bifrost server's BTC private key and its ETH private key are single points of failure. If an attacker can gain access to these, then the security of the system completely breaks because the attacker could steal BTC or mint unlimited BiBTC. Likewise, if the BTC private key is lost, the locked BTC would be lost forever.

One way to reduce the risk of private key leakage is to use multisig wallets and diversify the servers. For instance, if a 2-of-3 multisig is used, then Bifrost could operate one server in the Amazon cloud, one server in the Google cloud, and one server on-premises. Then, if an attacker gains access to Bifrost's accounts on one of the cloud platforms, the private key is not leaked. Access to each server should also be diversified so that no one person becomes a single point of failure.

Another way to reduce risk is to use a cold wallet for most of the BTC funds. While this may delay swap outs from BiBTC to BTC, it mitigates some risk of vulnerabilities in the BiFi-Bifrost-Extension protocol as well as risk of private key leakage from the Bifrost server. This is considered best practice for centralized exchanges, and it would be worth exploring how to integrate it into BiFi-Bifrost-Extension.

Fix

Bifrost is working on improvements to the protocol to support multisig and threshold signatures schemes.