macro

Fuji A-1

Security Audit

Jun 21, 2023
Version 1.0.0

Presented by 0xMacro

Table of Contents

Introduction

This document includes the results of the security audit for Fuji's smart contract code as found in the section titled ‘Source Code’. The security audit was performed by the Macro security team from Feb 13, 2023 to Mar 8, 2023.

It is important to note that Fuji is comparatively complex since it involves a lot of moving parts (lending, cross-chain, permits), hence we were able to find a significant number of issues. Most of these issues were on the surface level, hence there is the possibility of more issues once you go deep. Fixes resolution is also not ideal for these many issues, since all fixes increase the possibilities and do not allow reviewing them in isolation. Considering all of this, we suggest a reaudit to Fuji team.

Note: Fuji has chosen to get this audit with another audit provider for a fresh perspective, which we support and consider a sensible decision. This report includes fixes for critical and high issues only.

The purpose of this audit is to review the source code of certain Fuji Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro’s review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	5	-	-	5
High	7	-	-	7
Medium	8	8	-	-
Low	9	9	-	-
Code Quality	10	10	-	-
Informational	6	-	-	-
Gas Optimization	4	4	-	-

Fuji was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions on Telegram with the Fuji team.
- Available documentation in the repository.

Source Code

The following source code was reviewed during the audit:

- **Repository:** fuji-v2
- **Commit Hash:** `50fd0b74ccee1a73a459118e50e044a2bcfacd10`

Specifically, we audited the following contracts within this repository:

Contract	SHA256
BaseRouter.sol	`baa296b414300783226365b71e45c63ca14ec1701a23963f6cf4f729bc23e710`
BaseVault.sol	`a1b926a697a7b5643a6e8a93c10225aa2630c9826bcfe96c8a8aeb33adbc32a2`
EIP712.sol	`02f7fbd7caea343a7fd1942dc4cae37adae86748b80475b8b3abd7ebceb93583`
PausableVault.sol	`4a92ddfc1159c2cd8943a07aa5b7e9a69aefcc962dfa8b34e256cf65ee7d23c8`
ConnextRouter.sol	`05953cfd4e31ced27fe9a00dca235043055de6a711872e24cb29185ccffcdec1`
VaultPermissions.sol	`8308fb22001ad69573a297b68809a1bd0eb325de157f21a67cbc655f17a9460b`
BorrowingVault.sol	`df8abf2fb3cddada0443638f03cd1e4ffb301208e334a7bc6baa5fc304ad97fc`

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves, and does not pertain to any other programs or scripts, including deployment scripts.

Issue Descriptions and Recommendations

Click on an issue to jump to it, or scroll down to see them all.

M-2 Incorrect Rounding for Shares/Assets/Debt Calculation

M-3 Vaults Cannot Reach Their Deposit Cap

M-4 Vaults max__ functions fail to comply with EIP-4626

M-5 `Rebalance` allows breaking defined conservative `maxLTV` and liquidation ratio

M-6 For some assets, `_setProviders` will revert if new providers overlap with previous ones.

M-7 The slippage check on the destination is only done if the first action is deposit/withdrawal

M-8 If liquidity conditions don't improve on the destination, the hardcoded slippage protections of `_crossTransferWithCalldata` won't allow adapting.

L-1 Initializing the deposit cap to `type(uint128).max` with the option of changing it later through timelock defies its purpose.

L-2 Lack of validation regarding the previous active `providers` inside `setProviders`

L-3 Unnecessary `receive()` declaration for `BorrowingVault.sol`

L-4 No check to ensure `maxLTV < liqRatio` inside `setMaxLtv()`

L-5 Incomplete check inside `setLiqRatio()` allows defining `liqRatio` = `maxLTV`

L-6 Lack of option for borrowers to `payback` their complete loan

L-7 None of vault actions from the router use slippage protections provided by the vault

L-8 `_getBeneficiaryFromCalldata` of ConnextRouter doesn't consider nested `XTransfer`, `XTransferWithCall` and `DepositETH`

L-9 `_addTokenToList` is missing for `XTransfer` and `XTransferWithCall` actions

Q-1 Different versioning systems inside Fuji Vault

Q-2 Unnecessary use of `_msgSender()`

Q-3 Double conversion of **`Shares <> Assets`** in withdraw

Q-4 Redundant code on `L597` of `BaseVault.sol`

Q-5 Unnecessary use of counters library for `nonces`

Q-6 Unused Import inside `BorrowingVault.sol`

Q-7 Unused mapping in `BorrowingVault.sol` : `_borrowAllowances`

Q-8 Unnecessary state variable and logic in `BaseRouter.sol` : `isAllowedCaller`

Q-9 Unsafe Transfer of ERC20

Q-10 Incorrect naming for beneficiary inside payback action of the router.

G-1 Consider defining `_asset` as immutable inside `BaseVault.sol`

G-2 Consider defining `_debtAsset` as immutable inside `**BorrowingVault.sol**`

G-3 Consider adding a break in the loop once the validity of the provider is proved inside `_isValidProvider`

G-4 Consider adding a break in the loop of `_isInTokenList`

I-1 Implicit Limit on User Withdrawal Amounts

I-2 Lack of Support for Certain ERC20 Token Types

I-3 Providers should not have a state of their own, and any storage writes.

I-4 Fuji borrowing vault doesn't handle the case of liquidation of its debt position on actual lending providers.

I-5 Anyone can sweep the unused funds from the router.

I-6 Beneficiaries could be changed in one bundle by initially passing `address(0)` as beneficiary.

Security Level Reference

We quantify issues in three parts:

1. The high/medium/low/spec-breaking **impact** of the issue:
    - How bad things can get (for a vulnerability)
    - The significance of an improvement (for a code quality issue)
    - The amount of gas saved (for a gas optimization)
2. The high/medium/low **likelihood** of the issue:
    - How likely is the issue to occur (for a vulnerability)
3. The overall critical/high/medium/low **severity** of the issue.

This third part – the severity level – is a summary of how much consideration the client should give to fixing the issue. We assign severity according to the table of guidelines below:

Severity	Description
(C-x) Critical	We recommend the client **must** fix the issue, no matter what, because not fixing would mean **significant funds/assets WILL be lost.**
(H-x) High	We recommend the client **must** address the issue, no matter what, because not fixing would be very bad, *or* some funds/assets will be lost, *or* the code's behavior is against the provided spec.
(M-x) Medium	We recommend the client to **seriously consider** fixing the issue, as the implications of not fixing the issue are severe enough to impact the project significantly, albiet not in an existential manner.
(L-x) Low	The risk is small, unlikely, or may not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
(Q-x) Code Quality	The issue identified does not pose any obvious risk, but fixing could improve overall code quality, on-chain composability, developer ergonomics, or even certain aspects of protocol design.
(I-x) Informational	Warnings and things to keep in mind when operating the protocol. No immediate action required.
(G-x) Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Issue Details

~~C-1~~ Incorrect overloading of `_spendAllowance` allows the owner to use their allowance indefinitely.

TOPIC	STATUS	IMPACT	LIKELIHOOD
Trust Model	Fixed	High	High

`BaseVault.sol` intends to overload `_spendAllowance` so that it takes withdrawal allowance into consideration for `transferFrom` calls.

However, defined function signature of `spendAllowance` in `BaseVault` :

`_spendAllowance(address, address, address, uint256)` (link)

is different from the function signature ERC20 expects or calls:

`_spendAllowance(address, address, uint256)` (link)

Therefore, if someone executes a `transferFrom`, it will call the vanilla ERC20's `spendAllowance`, without taking withdrawal allowances into consideration. This allows owners to transfer shares to a new address using `transferFrom`, while still retaining their withdrawal balance.

POC

```
    // VaultPermissions.t.sol
    function test_spend_allowance_issue() public {

        do_deposit(1 ether, vault, owner);
```

```
        vm.startPrank(owner);
        vault.approve(receiver, 1 ether);
        uint256 state1 = vault.allowance(owner, receiver);
        vm.stopPrank();

        vm.startPrank(receiver);
        vault.transferFrom(owner, receiver, 1 ether);
        uint256 state2 = vault.allowance(owner, receiver);

        assertEq(state1, state2);
    }
```

Remediation to consider:

Consider overloading `spendAllowance` with the right function signature.

~~C-2~~ Using `SWAP` as an action, anyone can steal user's funds from router

TOPIC	STATUS	IMPACT	LIKELIHOOD
Theft of User Assets	Fixed	High	High

Fuji allows users to do multiple actions through the router. There is a beneficiary check defined on multiple actions like withdraw, borrow, to ensure that the actions benefit the actual owner of the bundle.

However, no such check is being made for SWAP action, allowing someone to steal those funds.

For example.

```
The user gives the router an allowance to withdraw, off-chain through sign
```

```
Someone can now use this signature for their benefit using swap.

1. Permit Withdraw
Allowance
operator = router, owner = user, receiver = router

2. Withdraw from the vault to the router.
Set the beneficiary to be the owner.

3. Call swap from collateral token to different token,
with the receiver being the attacker.

Exit
```

Remediation to consider:

Consider adding a beneficiary check for swap action and consider whitelisting swappers.

Whitelisting swappers is necessary because the swapper is user input. Without whitelisting, a user could pass a dummy swapper and trick your contract into thinking that the beneficiary is what the contract expects. However, the user could then do whatever they want inside, allowing them to get away with tokens.

In general, it is recommended to whitelist all external addresses that are being called, Vaults, Swappers, and Flashers. Not doing so has created multiple issues, as seen in this report. For example, check C-3.

~~C-3~~ Using the user's router allowance, anyone can steal user's funds

TOPIC	STATUS	IMPACT	LIKELIHOOD
Theft of User Assets	Fixed	High	High

A vault is a user input for all actions of the router bundle.

Suppose Alice gives the router an allowance to spend 1000 tokens and intends to deposit these tokens into vault X. An attacker who sees Alice's approval in the mempool can front-run Alice's deposit with their own bundle.

To pass the checks of _safePullTokenFrom, they pass sender and receiver both as Alice only. However, please note that the vault is user-input, so anyone can pass anything there and make you believe Alice is the receiver when she is not. This way, anyone can use anyone's allowance and run away with their funds.

```
// DEPOSIT
// sender = receiver = user
(IVault vault, uint256 amount, address receiver, address sender) =
          abi.decode(args[i], (IVault, uint256, address, address));
____
_safePullTokenFrom(token, sender, receiver, amount);
____
vault.deposit(amount, receiver);

_____________________________
function _safePullTokenFrom(
    address token,
    address sender,
    address owner,
    uint256 amount
)
internal
{
    if (sender != address(this) && (sender == owner || sender == msg.sende
      ERC20(token).safeTransferFrom(sender, address(this), amount);
    }
}
```

Remediation to consider:

Consider whitelisting vaults and verifying that the sender is equal to `msg.sender` inside `_safePullTokenFrom`. Implementing both changes would reduce the possible exposure to the contract due to user input and ensure that the sender is the actual

user performing the transaction

~~C-4~~ Frontrunning `Permit` allows the user to do a different set of actions than the beneficiary intended and, in the worst case run away with funds

TOPIC	STATUS	IMPACT	LIKELIHOOD
Theft of User Assets	Fixed	High	High

The router contract tries to make sure that beneficiary should remain the same, but there is no validation to check if the actions being performed are the ones user intended.

Consider Alice sets withdraw allowance for router through permit and intends to send it to the vault of Chain B.

Anyone can use this Alice permit and do a totally different set of actions. They can `xCall` and bridge this asset to some random chain.

```
Alice's Intended Bundle
1. PermitWithdraw(Owner=Alice, Operator= Router, Receiver= Router)
2. Withdraw (Beneficiary = Alice)
3. xCallWithCalldata(ChainB, Deposit(receiver=alice)) (Beneficiary = Alice

It is possible to create a different bundle using the same permit.
1. PermitWithdraw(Owner=Alice, Operator= Router, Receiver= Router)
2. Withdraw (Beneficiary = Alice)
3. xCall(any random chain, receiver = Alice) (Beneficiary = Alice)
```

Fuji Team responded to this lead with following :

Response:

We have referred to these situations as front-running a permit. We believe we have put some measures in place, perhaps not enough. We ensure that the beneficiary remains the same. When calling the withdraw action, the beneficiary is set to the owner. Then when attempting to bridge the _beneficiary will be checked and `_internalBundle` will revert.

It is true that the beneficiary, Alice, in this case, remains the same. However, it is not guaranteed that only Alice's intended actions will be done using her permit. This significantly impacts user experience, which directly interferes with the core intention of the Fuji protocol: bundling user actions through debt permits to enable cross-chain functionality.

Not only is this a user experience issue, but there are also ways attackers can steal from Alice, keeping `Beneficiary=Alice` only

1. An attacker can manipulate the swap parameters of a swap action since they are user input. They can either add a swap action on top of Alice's bundle or edit it if it's already there.

   Currently, there is no check on the beneficiary as explained in [C-2]. However, even if you add check, the exposure to this issue remains.

```
(
    ISwapper swapper,
    address assetIn,
    address assetOut,
    uint256 amountIn,
    uint256 amountOut,
    address receiver,
    address sweeper,
    uint256 minSweepOut
) = abi.decode(
    args[i],
    (
        ISwapper,
        address,
```

```
        address,
        uint256,
        uint256,
        address,
        address,
        uint256
    )
); // @audit one can basically tweak any parameter of this as of now
```

a. Sandwich Attack

An attacker can allow very high slippage for Alice's action and sandwich their own action in between. They can even change the `assetOut` address to select the pool with the lowest liquidity.

b. Sweeper Attack

An attacker can set the `amountOut` to a very low value, set themselves as the sweeper, and run away with leftover `amountIn` (user funds).

There is no check on who the sweeper could be. With the right parameters, attackers can steal nearly everything.

Please note that in both cases, the beneficiary or receiver can remain the same.

2. Another way is not as easy as the method described above, but the attacker can basically pass any amount of slippage in `xCalls` and sandwich them on the destination.

To sum up, simply checking the beneficiary is not enough. Many other parameters can impact the amount received by the beneficiary and benefit the attacker.

Remediation to consider: Consider embedding the hash of actions the user intends to perform with their arguments inside the signature. This would resolve many of the issues mentioned in the report.

For example, there would no longer be a need to whitelist swapper, flasher, or vault.
C-2 & C-3

~~C-5~~ `Reentrancy` allows anyone to modify beneficiary between the bundle and steal assets

TOPIC	STATUS	IMPACT	LIKELIHOOD
Theft of User Assets	Fixed	High	High

Currently, all the external calls from the router are unprotected. This means that the executor can pass `swapper`, `flasher`, and `vault` as anything they like and re-enter.

Even if Fuji whitelist `swapper`, `vault`, and `flasher` as solution to other critical issues of this report, there are two legitimate ways users can still re-enter: the flasher callback and withdrawing ETH.

While the flasher reentrancy is intended, the contract currently doesn't handle it correctly. It allows the executor to change beneficiary once it's set between the bundle.

For example:

```
beneficiary and tokensToCheck are state variables

bundle #PARENT
some action: beneficiary: X
flasher:
    reenter with child bundle  #CHILD
    need to do some action for beneficiary X only,
    since checkBeneficiary would only allow for X
    this child bundle is done with execution
    so it would set beneficiary ,tokenToCheck to 0 in the end
    https://github.com/Fujicracy/fuji-v2/blob/50fd0b74ccee1a73a459118e50e0

#PARENT
return to the parent bundle
```

```
and there you go;
since beneficiary is set to address(0).
one can can set beneficiary to any address
```

The cause of the issue is that beneficiary and "tokensToCheck" are state variables. Since they are state variables, the state cleared in the child is also considered cleared for the parent.

Remediation to consider: To resolve this, consider making `beneficiary` and `tokensToCheck` memory variables instead.

Memory for each call is different, unlike storage. Hence, both the parent and child will have their own isolated `beneficiary` and `tokensToCheck`. This not only solves the problem but also significantly reduces the gas costs involved due to the use of memory instead of storage.

If you decide to go this route, please add a check inside the "flashloan" action that confirms the first action in `requestorCalldata` is the same as the one in the parent bundle, similar to how it's done in `_crossTransferWithCalldata`.

~~H-1~~ Someone can execute a Denial of Service on Fuji's Borrowing vault.

TOPIC	STATUS	IMPACT	LIKELIHOOD
3rd Party Behavior	Fixed	High	Low

Multiple lending providers offer the option to repay a loan on someone else's behalf. For example, Aave V2 allows this feature.

```
contracts/protocol/lendingpool/LendingPool.sol#L236
```

```
function repay(
    address asset,
    uint256 amount,
    uint256 rateMode,
    address onBehalfOf
  ) external override whenNotPaused returns (uint256) {
    DataTypes.ReserveData storage reserve = _reserves[asset];
```

It's possible for someone to leverage this feature and pay off vault debts. The Fuji vault currently does not handle this case, halting both borrowing and payback, locking collateral for borrowers.

The reason is `convertDebtToShares` would revert due to division by 0

```
function _convertDebtToShares(
    uint256 debt,
    Math.Rounding rounding
  )
    internal
    view
    returns (uint256 shares)
  {
    uint256 supply = debtSharesSupply;
    return (debt == 0 || supply == 0) ? debt : debt.mulDiv(supply, totalDe
  }
```

While paying a vault's debt is theoretically possible, one may question why someone would do so.

We acknowledge that it is less probable for individuals to pay off significant debts, but in the early stages of the vault, when the debt is minimal or even worth 1 wei, one can pay it on the vault's behalf and execute a denial of service on further debt actions on the vault for a lifetime.

The only option for Fuji in this scenario would be to redeploy the vault and hope that attackers do not grief it again.

Remediation to consider:

We couldn't come up with an ideal isolated solution to this problem. Please let us know if you have any ideas.

There are some possible ways, like considering shares to mint equal to debt only in `_convertDebtToShares`, like it's done if the supply of shares is zero. However, this approach would distribute the new debt position over all previous borrowers, which is not ideal and could create additional complexity and attack vectors.

Instead, as mitigation after deployment, consider borrowing a significant amount yourself to reduce the likelihood of someone repaying the vault's debt on their own. Even in the extreme case where someone pays off the entire vault's debt, users should still be able to withdraw their funds. Consider adding a test for same.

~~H-2~~ Partial Liquidations won't be possible for vaults with the collateral asset of decimals < 18

TOPIC	STATUS	IMPACT	LIKELIHOOD
Accounting	Fixed	High	High

The `liquidate` function of Fuji's borrowing vault has the following code:

```
uint256 price = oracle.getPriceOf(debtAsset(), asset(), _debtDecimals);
uint256 discountedPrice = Math.mulDiv(price, LIQUIDATION_PENALTY, 1e18);
gainedShares = convertToShares(Math.mulDiv(debt, liquidationFactor, discou
```

The purpose of `gainedShares` is to calculate the number of shares a liquidator should receive for paying an owner's debt. Ideally, it should be done like this:

```
debtToCover = debt * liquidationFactor / 1e18

If 1 * 10 ** assetDecimals is worth this much price,
then how many assets are equivalent to debtToCover?

1 * 10 ** assetDecimals = price
? = debtToCover

gainedAssets = debtToCover * 10 ** assetDecimals / price
                          = debt * liquidationFactor * 10 ** assetDecimals
```

However, Fuji's current implementation needs to be corrected since it ignores both offsets. `Math.mulDiv(debt, liquidationFactor, discountedPrice);`

```
gainedAssets = debt * liquidationFactor / price
```

Nearly all widely used ERC20s have decimals ≤ 18. There won't be any issue for tokens with decimals = 18 since the numerator and denominator would cancel each other out. However, for all tokens with decimals < 18, Fuji would overestimate the gained shares and hence would assign `gainedShares` equal to the total balance of the owner due to the following check:

```
if (gainedShares > existingShares) {
      gainedShares = existingShares;
}
```

If the concerned liquidation is whole, liquidation will go untroubled.

However, if the liquidation is partial, then it would revert while burning owner shares since you cannot burn all owner shares until all the owner's debt is paid.

Remediations to consider:

Consider applying proper offsets `10 ** 18 / 10 ** assetDecimals` for gained asset calculation.

~~H-3~~ Setting fixed values for `maxLTV` and `liqRatio` initially exposes the vault to liquidation.

TOPIC	STATUS	IMPACT	LIKELIHOOD
Design	Fixed	High	High

Fuji aims to support a diverse range of pairs for its borrowing vaults. Ideally, Fuji intends to be more conservative than its providers by setting `maxLTV` and `liqRatio` to comparatively lower values to avoid liquidations.

However, in the current implementation of the borrowing vault, both `maxLTV` and `liqRatio` are defined to a fixed value with the option of changing them later through timelock.

```
constructor(
    address asset_,
    address debtAsset_,
    address oracle_,
    address chief_,
    string memory name_,
    string memory symbol_,
    ILendingProvider[] memory providers_
  )
    BaseVault(asset_, chief_, name_, symbol_)
  {
   ____
    maxLtv = 75 * 1e16; // @audit
    liqRatio = 80 * 1e16; // @audit
 _______
  }
```

There are two cases to consider:

1. **If the initial fixed ratios (Fuji) are less than the planned and provider ratios:** This

won't create any issues and will only limit borrowers until timelock acts.

2. **If the initial fixed ratios (Fuji) are greater than the planned and provider ratios:** Since the provider sees only one debt position from their point of view, borrowers can borrow more than what the provider would itself have allowed for their collateral. This is because they can use someone else's collateral, potentially exceeding the planned limit.

   At least one or both of the following would happen

   - The market moves, and the vault gets liquidated. The current implementation does not have a way to penalize borrowers if the vault is liquidated. The loss is borne by all holders, including users who did not borrow anything. This makes the situation worse.
   - Once the timelock is activated, the borrower would be liquidated. Some may argue that this is unfair.

Remediation to consider:

Consider defining both `maxLTV` and `liqRatio` during construction depending on the token pair.

~~H-4~~ Partial Liquidations may not be possible in some cases due to check in `beforeTokenTransfer`

TOPIC	STATUS	IMPACT	LIKELIHOOD
Design	Fixed	High	High

In certain cases, partial liquidations may not be possible, and the vault will remain vulnerable until the health factor drops below `FULL_LIQUIDATION_THRESHOLD`.

For example, consider the following scenario:

```
1.
Alice's collateral is worth $4000, and she has borrowed 2 assets of $1500
With a maximum loan-to-value ratio of 75% and liquidation ratio of 80%.

Collateral: $4000
Debt: 2 * ($1500)

2.
Debt asset's price increases to $1650
Collateral: $4000
Debt: 2 ($1650)
The health factor: 4000 * 0.80/ 3300 = 0.96969697 > 0.95
Vault intends to liquidate half of the debt position.

The assets liquidator would get
1 (half debt) * 1650 / 0.9 (liq discount) = 1833.333333333
Alice still has a debt of 1 * 1650,
Which by 0.75 LTV, utilizes 2200 of Alice's collateral.

maxRedeem for Alice = 4000 - 2200 = 1800
What Alice need to pay to the liquidator = 1833.33

1833 > 1800
Code Reverts !! On this [line](https://github.com/Fujicracy/fuji-v2/blob/1

**BaseVault.sol#L594**
function _beforeTokenTransfer(address from, address to, uint256 amount) in
    to;
    if (from != address(0)) {
    #L594 require(amount <= maxRedeem(from), "Transfer more than max");
    }
}
```

Remediation to consider:

Consider adding a pass for this case inside `_beforeTokenTransfer`. Please note that it cannot be skipped for all burns, as the withdraw relies on the burning of tokens to determine the amount of collateral a user can withdraw.

~~H-5~~ Anyone can override the handler records using the same `trasnferId` and failing bundle

TOPIC	STATUS	IMPACT	LIKELIHOOD
Unrestricted Access	Fixed	High	High

If `try this.xBundle(actions, args)` fails inside `xReceive` of the router, and the contract has dormant funds; a record is made inside the handler in the name of `transferId`, and funds are transferred there.

However, since anyone can call `xReceive`, one can call it with the same `transferId` later, make the bundle fail on purpose, and override the content inside the handler.

Remediation to consider:

Consider quering `connext` to check if the `transferId` passed is already processed.

~~H-6~~ Executor of a bundle containing `_crossTransfer` can execute a sandwich attack on the destination

TOPIC	STATUS	IMPACT	LIKELIHOOD
MEV	Fixed	High	Medium

The `_crossTransfer` and `_crossTransferWithCalldata` functions pass `msg.sender` as `delegate` in `xCall`.

This means that `msg.sender` would have complete control over slippage on the

destination; they can increase or decrease it to any number they like. For more information, see:
https://docs.connext.network/developers/reference/contracts/calls#xcall

```
function _crossTransfer(bytes memory params) internal override {
  ____
    ____
    bytes32 transferId = connext.xcall(
      // _destination: Domain ID of the destination chain
      uint32(destDomain),
      // _to: address of the target contract
      receiver,
      // _asset: address of the token contract
      asset,
      // _delegate: address that has rights to update the original slippag
      // by calling Connext's forceUpdateSlippage function
      **msg.sender,**
      // _amount: amount of tokens to transfer
      amount,
      // _slippage: can be anything between 0-10000 becaus
      // the maximum amount of slippage the user will accept in BPS, 30 ==
      slippage,
      // _callData: empty because we're only sending funds
      ""
    );
    emit XCalled(transferId, msg.sender, receiver, destDomain, asset, amou
  }
```

This is okay in the case of `_crossTransferWithCalldata` since it has its own slippage protections in calldata.

However, for `_crossTransfer`, this allows executor of the bundle to set slippage to the maximum and sandwich the call.

```
function forceUpdateSlippage(TransferInfo calldata _params, uint256 _slipp
external onlyDelegate(_params) {

https://github.com/connext/monorepo/blob/1f1aa5f845581d5b1050a0f693f19b45c
```

Remediations to consider: Consider passing a delegate as the beneficiary instead of `msg.sender`. However, please note that if the beneficiary is a contract, it may not have the ability to update this slippage when necessary.

If liquidity conditions do not improve, the funds may remain stuck in the Connext bridge.

If your use case involves contract addresses as beneficiaries in `_crossTransfer`, consider passing `tx.origin` as a delegate if the beneficiary is contract.

Or

Consider passing Fuji's timelock address as a delegate.

~~H-7~~ Incorrect handling of `requesterCallData` for Flashloan action inside `_getBeneficiaryFromCalldataof` the router

TOPIC	STATUS	IMPACT	LIKELIHOOD
Spec Break	Fixed	Medium	High

When a Flashloan action is executed, the `requesterCallData` parameter is extracted from the calldata and passed to the `_getBeneficiaryFromCalldata` function again. (Decode B in the following code)

```
function _getBeneficiaryFromCalldata(bytes memory callData)
    internal
    pure
    returns (address beneficiary)
{
    **// Decode A**
  (Action[] memory actions, bytes[] memory args,) =
      abi.decode(callData, (Action[], bytes[], uint256));
```

```
  ------
    } else if (actions[0] == Action.Flashloan) {
            **// Decode B**
      (,,,, bytes memory requestorCalldata) =
        abi.decode(args[0], (IFlasher, address, uint256, address, bytes));
      beneficiary = _getBeneficiaryFromCalldata(requestorCalldata);
    }
  --------
}
```

To decode the `requesterCallData` into actions and arguments inside this second `_getBeneficiaryFromCalldata` call, the following encoding would be used at Decoding A.

```
(Action[] memory actions, bytes[] memory args,) =
abi.decode(callData, (Action[], bytes[], uint256));
```

However, the `requesterCallData` of the Flasher has a different format than `(Action[], bytes[], uint256)` , it is:

```
**(BaseRouter.xBundle.selector, Actions[], bytes[])**
```

This means that the decoding process would fail at point A, and the router would not be able to bridge calldata with Flashloan as its first position.

This is problematic because having Flashloan as the first action is necessary for some of Fuji's use cases, such as closing a position with Flashloan.

Remediation to consider:

Consider refactoring `_getBeneficiaryFromCalldata` to accommodate the Flashloan action.

One way of doing so is

```
**_crossTransferWithCalldata**
         decode calldata into actions and args
         _getBeneficiaryFromActionsAndArgs

**_getBeneficiaryFromActionsAndArgs**
(Action[] memory actions, bytes[] memory args)
{
    -- FlashloanCase
      (,,,, bytes memory requestorCalldata) =
        abi.decode(args[0], (IFlasher, address, uint256, address, bytes));

            (, Action[] memory actions, bytes[] memory args) =
      abi.decode(requestorCalldata, (BaseRouter.xBundle.selector, Actions[
      beneficiary = _getBeneficiaryFromCalldata(actions, args);
    }
 --------
}
```

M-1 Fuji's vault would remain vulnerable to an inflation attack despite the explicit measures taken

TOPIC	STATUS	IMPACT	LIKELIHOOD
Share Value Manipulation	Acknowledged	Medium	Medium

There is one widely known issue regarding the front-running of the first deposit. Fuji team is already aware of this and mentions the same on L85 of the base vault, and has added a `minAmount` protection against it. However, it's not enough since one can deposit `minAmount`, withdraw it such that only 1 wei of assets remain in the contract, and then sandwich the first deposit.

Remediation to consider: Contract Changes:

1. Consider defining state variables to track balances instead of `.balanceOf`

2. Represent shares with more precision/decimals than assets (refer openzeppelin's discussion)

Or Consider depositing some amount into the vaults on your own when they are deployed and leave it there. This will ensure that the total supply of shares will always remain at a certain level.

Other solutions, like minting dead shares to zero address, are suboptimal, as explained in this MixBytes Blog Post.

M-2 Incorrect Rounding for Shares/Assets/Debt Calculation

TOPIC	STATUS	IMPACT	LIKELIHOOD
Accounting	Acknowledged	High	Medium

All rounding should be done considering the best result for the vault and the worst result for users so that someone cannot take advantage of the vault with consecutive actions without losing any value. However, Fuji vaults perform incorrect rounding in multiple instances (all instances listed here), allowing users to extract more than their contribution.

For example, consider Fuji's `previewWithdraw` function, which rounds down instead of rounding up:

```
Shares (Alice) = 50
TotalSupplyShares = 100
TotalAssets = 200
```

Alice calls withdraw for one asset,

`previewWithdraw` (round down): the number of shares to be burned is `(1 * 100) / 200 = 0`.

Hence, Alice can theoretically keep withdrawing one worth of assets without burning any shares. Although the associated gas costs won't make this trade feasible, it breaks the invariant. Hence we classified this issue as a medium rather than high.

Remediation to consider:

Consider reversing rounding in the following cases:

1. `previewWithdraw`: Round up. You want to burn the maximum number of shares for given assets.
2. `previewMint`: Round up. You want to pull the maximum number of assets for given shares.
3. `borrow`:

fuji-v2/BorrowingVault.sol at main · Fujicracy/fuji-v2

```
uint256 shares = convertDebtToShares(debt);
```

Round up. You don't want users to add debt to the vault without getting shares for it.

1. `_computeMaxBorrow` :

fuji-v2/BorrowingVault.sol at 6231dd1161602cc4b081fd2bab621fa6a3cb9394 · Fujicracy/fuji-v2

```
uint256 debt = convertToDebt(debtShares);
```

Round up. You want the user to borrow the least possible.

1. `_computeFreeAssets`:

fuji-v2/BorrowingVault.sol at main · Fujicracy/fuji-v2

`uint256 debt = convertToDebt(debtShares);`

Round up. You want the user to be responsible for the maximum debt.

1. `getHealthFactor` :

fuji-v2/BorrowingVault.sol at main · Fujicracy/fuji-v2

`uint256 debt = convertToDebt(debtShares)`

Round up. You want to factor in the maximum possible debt.

1. `liquidate` :

fuji-v2/BorrowingVault.sol at main · Fujicracy/fuji-v2

`uint256 debt = convertToDebt(_debtShares[owner]);`

Round up. You want the liquidator to cover the maximum possible debt for given shares.

M-3 Vaults Cannot Reach Their Deposit Cap

TOPIC	STATUS	IMPACT	LIKELIHOOD
Spec Break	Acknowledged	Low	High

Both methods of depositing assets into the vault (deposit and mint) have the following check to revert if the deposit exceeds the deposit cap:

```
if (shares + totalSupply() > maxMint(receiver)) { // @audit issue
```

```
        revert BaseVault__deposit_moreThanMax();
    }
```

However, due to double accounting of `totalSupply`, the vault can never reach its deposit cap.

`maxMint` is derived from `maxDeposit`, and `maxDeposit` is calculated as `depositCap - totalAssets`.

Therefore, `maxMint` calculates the number of shares the vault can mint from that point onwards, not the maximum number of shares it can mint during its lifetime. Hence, the addition of total supply in the following line is redundant and problematic: `shares + totalSupply() > maxMint(receiver)`.

Consider a deposit cap of 100 tokens. Someone makes the first deposit of 10 tokens, resulting in a total supply of 10 tokens. Now, if someone wants to deposit 89 tokens, it should ideally be allowed, but it cannot be done due to this issue.

```
Second deposit: 89
To mint: 89
Shares + totalSupply() > maxMint(receiver)
89 + 10 > 100 - 10
99 > 90
And it reverts!
```

This makes the resultant deposit cap half of the defined one. To mitigate this issue, one can still increase the deposit cap. Therefore, we classify it as a medium severity rather than high.

Remediation to consider:

Consider removing `totalSupply` from `shares + totalSupply() > maxMint(receiver)` check.

M-4 Vaults max__ functions fail to comply with EIP-4626

TOPIC	STATUS	IMPACT	LIKELIHOOD
Spec Break	Acknowledged	Medium	High

Case 1:

According to EIP-4626, the `maxDeposit`, `maxWithdraw`, `maxMint`, `maxRedeem` functions must return 0 even when the corresponding actions are temporarily disabled.

However, in Fuji vaults, this is not considered and all functions would still return actual value even if vaults are paused.

Although this is not an issue for Fuji vaults, since the pause is later enforced in the code, developers building on top of Fuji vaults may expect the EIP behavior and not handle it explicitly, leading their code down an unexpected path.

Remediation to consider:

Consider returning 0 if vaults are paused for `maxDeposit`, `maxWithdraw`, `maxMint`, and `maxRedeem`. Please note that `maxBorrow` is not part of EIP-4626, but as it takes inspiration from EIP, consider returning 0 for it as well.

Case 2:

According to EIP-4626 , the `maxDeposit` must not revert. However, it will eventually revert in the case of Fuji Vaults.

```
function maxDeposit(address) public view virtual override returns (uint256
    return depositCap - totalAssets();
}
```

Total assets in vaults can increase through two means: deposits and yield. Therefore, once yield begins to accumulate, total assets could exceed the deposit cap, even if no new deposits are made.

This presents a problem for `maxDeposit` as it would underflow and revert. This could have an impact on protocols built on top of Fuji vaults, as their code assumes that `maxDeposit` will never revert.

Remediation to consider:

Consider adding this check before performing the subtraction.

```
if (totalAssets >= depositCap) return 0 ;
```

M-5 `Rebalance` allows breaking defined conservative `maxLTV` and liquidation ratio

TOPIC	STATUS	IMPACT	LIKELIHOOD
Lack of Checks	Acknowledged	High	Low

Fuji aims to maintain a conservative `maxLTV` and liquidation ratio compared to other lending providers to avoid liquidation of the vault's debt position. However, the current `Rebalance` function allows for rebalancing with any number of collateral and debt, which breaks this invariant.

Currently, the `RebalancerManager` performs checks for assets and debts on `L57` and `L62`. Both checks verify that the vault holds the number of assets and debts being moved, but there is no check on the ratio of assets and debts being moved.

For example, suppose the vault intends to maintain `maxLTV` at 50%. In one provider, the vault's debt position is 100 worth of assets as collateral and 50 worth

of assets as debt. During rebalancing, the rebalancer role holder can move 30 value of collateral assets to another provider and keep the previous provider at a ratio of `70(collateral):50(debt)`, breaking the assumption of Fuji that they will always have conservative ratios compared to actual providers.

This increases the likelihood of vault liquidation.

Currently, using rebalance, one can move the debt positions in any way until actual providers allow.

Remediation to consider:

Consider adding a check to ensure that the ratio of assets to debt remains the same during rebalance.

M-6 For some assets, `_setProviders` will revert if new providers overlap with previous ones.

TOPIC	STATUS	IMPACT	LIKELIHOOD
ERC20 Variations	Acknowledged	Medium	High

The `_setProviders` function has two purposes: setting the providers array with a new set of providers and giving maximum approval for vault assets to all of the new providers.

```
function _setProviders(ILendingProvider[] memory providers) internal overr
    uint256 len = providers.length;
    for (uint256 i = 0; i < len;) {
     ____
      IERC20(asset()).approve(
        providers[i].approvedOperator(asset(), asset(), debtAsset()), type
      );
```

```
        IERC20(debtAsset()).approve(
          providers[i].approvedOperator(debtAsset(), asset(), debtAsset()),
        );
      ----
      }
      _providers = providers;
          -------
  }
```

This code works for normal tokens, but for tokens that revert on `approve` if they already have an allowance, it will revert.

For example, `USDT`. If a new set of providers have even one provider from the previous, it will revert.

Remediation to consider:

Consider resetting `allowance` to 0 before setting it to the max. or Consider using `[forceApprove](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/4fb6833e325658946c2185862b8e57e32f3683bc/contracts/token/ERC20/utils/SafeERC20.sol#L82)` from the OZ SafeERC20 library.

M-7 The slippage check on the destination is only done if the first action is deposit/withdrawal

TOPIC	STATUS	IMPACT	LIKELIHOOD
Design	Acknowledged	Medium	Medium

The slippage check is only performed if the first action is a deposit or withdrawal, not for all actions.

```
ConnextRouter.sol
```

```
_accountForSlippage()
if (action == Action.Deposit || action == Action.Payback)
```

This covers all cases where a user bridges an asset, except in cases where the received funds are used in a nested action. For example:

1. `xReceive` with X amount of funds. 2 .Bundle : `1. permitWithdraw() 2. withdraw(Y) 3. xCallWithCallData(Deposit (X + Y))`

Fuji Team responded to this lead with following : **Response:**

In this case after the 3rd action xCallwithCallData, on the destination chain the slippage will be checked on a deposit.

While it is true that slippage would be checked at the final destination, that check would occur for X+ Y in terms of nested `xCall` slippage parameters, not on X in terms of the initial `xCAll`. Therefore, the contract would accept any X slippage and execute an `xCall` to the final destination, missing a required check on the first level.

If X + Y does not fall within the slippage range, the call would revert at the final destination. However, the user's position on the middle chain would already be withdrawn, and they would be unable to take any action to rectify the situation.

Remediation to consider: Consider checking for slippage in `xReceive` whenever the contract receives any funds.

To avoid looping through all actions to get the amount, consider sending `AmountMinOut` instead of `slippage` in calldata.

Please note if you decide to solve [M-8] If liquidity conditions don't improve on the destination, the hardcoded slippage protections of `_crossTransferWithCalldata` won't allow adapting. by removing the explicit slippage protection, this issue won't be applicable anymore.

M-8 If liquidity conditions don't improve on the destination, the hardcoded slippage protections of `_crossTransferWithCalldata` won't allow adapting.

TOPIC	STATUS	IMPACT	LIKELIHOOD
Design	Acknowledged	High	Low

If the liquidity currently available on the destination is causing slippage more than what the user specified while doing `xCall`, the call will revert.

Connext currently exposes two options for users to mitigate it : https://docs.connext.network/developers/guides/handling-failures#high-slippage

- Wait it out until slippage conditions improve (relayers will continuously re-attempt the transfer execution).
- Increase the slippage tolerance.

The first option is still possible in the case of `_crossTransferWithCalldata` of Fuji's router.

However, **the second is not,** since slippage is hardcoded inside the calldata, and checked using `_accountForSlippage`.

Hence in the worst case, if slippage conditions don't improve or if the user bundles action with very low slippage, the `xCall` would keep failing.

Remediations to consider:

Please check with the Connext team to see if any attack vectors require you to have your own slippage protection. If none exist, consider relying on Connext's slippage protection instead of doing your calculations.

If explicit slippage protection for `_crossTransferWithCalldata` is removed, apply M-7 to `_crossTransferWithCalldata` as well.

L-1 Initializing the deposit cap to `type(uint128).max` with the option of changing it later through timelock defies its purpose.

TOPIC	STATUS	IMPACT	LIKELIHOOD
Spec Break	Acknowledged	Low	High

The deposit cap is intended to cap the exposure initially. However, Fuji vaults define the deposit cap as `type(uint128).max` during construction with the option of reducing it through the timelock.

```
BaseVault.sol

constructor(
    address asset_,
    address chief_,
    string memory name_,
    string memory symbol_
  )
    ERC20(name_, symbol_)
    SystemAccessControl(chief_)
    VaultPermissions(name_)
  {
        _____
    depositCap = type(uint128).max;
    ____
  }
```

This defeats the purpose of the deposit cap, since it allows exceeding the planned deposit cap before the timelock takes effect. Therefore, we classify this issue as

medium severity.

Remediation to consider: Consider making a deposit cap a user input during construction.

L-2 Lack of validation regarding the previous active `providers` inside `setProviders`

TOPIC	STATUS	IMPACT	LIKELIHOOD
Input Validation	Acknowledged	High	Low

There is no validation inside `setProviders` to check if the new `providers` array contains all of the current active debt positions of the vault.

If any active debt positions are missed, the vault will underestimate the total assets exposing the vault until it is corrected.

Remediation to consider:

Since access to this function is limited to admins through timelock, we can expect them to behave always in protocol interest; however, consider adding a check to ensure none of the current active debt position providers are missed on the smart contract level to remove this exposure completely.

L-3 Unnecessary `receive()` declaration for `BorrowingVault.sol`

TOPIC	STATUS	IMPACT	LIKELIHOOD
Asset Transfers	Acknowledged	Low	Low

```
L147 receive() external payable {}
```

Fuji vaults don't have any use case for accepting native assets directly to the vault.

Remediation to consider:

Consider removing this entry point for native assets from the vault so someone doesn't send native assets incorrectly.

L-4 No check to ensure `maxLTV < liqRatio` inside `setMaxLtv()`

TOPIC	STATUS	IMPACT	LIKELIHOOD
Spec Break	Acknowledged	High	Low

The `setLiqRatio` function checks that the passed `liqRatio` is greater than or equal to `maxLTV`, but no such check is performed inside the `setMaxLTV` function. Hence admins can unintentionally break this invariant.

Remediation to consider: Consider adding an explicit check to ensure that `maxLTV` is less than `liqRatio`.

L-5 Incomplete check inside `setLiqRatio()` allows defining `liqRatio` = `maxLTV`

TOPIC	STATUS	IMPACT	LIKELIHOOD
Spec Break	Acknowledged	High	Low

The comment on `L122` of the borrowing vault mentions

```
* - Must check `maxLTV` Must < `liqRatio`.
```

However, one can set `liqRatio = maxLTV` using `setLiqRatio`

```
function setLiqRatio(uint256 liqRatio_) external onlyTimelock {
    if (liqRatio_ < maxLtv || liqRatio_ == 0) {
      revert BaseVault__setter_invalidInput();
    }
    liqRatio = liqRatio_;
    emit LiqRatioChanged(liqRatio);
}
```

Remediation to consider: Consider adding equality to the concerned check.

From `if (liqRatio_ < maxLtv || liqRatio_ == 0)`

To `if (liqRatio_ <= maxLtv || liqRatio_ == 0)`

L-6 Lack of option for borrowers to `payback` their complete loan

TOPIC	STATUS	IMPACT	LIKELIHOOD
User Experience	Acknowledged	Low	High

Fuji's Borrowing Vault accepts absolute debt amount only in `payback`.

Lending providers charge interest per block, so between the time you submit a transaction and the time it gets processed, the debt will have increased by some small margin, especially on L2s. On L1s, it is also not guaranteed that you will land in the same block; there is always some latency. This prevents someone from paying off their total debt, and there will always be some leftover dust.

Remediation to Consider:

1. Ask users to pass the unit max as the amount in `payback` if they want to repay their total debt, then define the amount as the user's current debt inside `payback`. AAVE does the same.
2. Define a different `payback` in terms of shares.

L-7 None of vault actions from the router use slippage protections provided by the vault

TOPIC	STATUS	IMPACT	LIKELIHOOD
User Experience	Acknowledged	Low	High

Fuji's vault provides various slippage protections for users, but none of them are being used inside the router, for instance, the `minShares` of deposit.

Remediation to Consider:

Consider using provided slippage protections

L-8 `_getBeneficiaryFromCalldata` of ConnextRouter doesn't consider nested `XTransfer`, `XTransferWithCall` and `DepositETH`

TOPIC	STATUS	IMPACT	LIKELIHOOD
Edge Case	Acknowledged	Low	Medium

When checking beneficiary for `_crossTransferWithCalldata` using `_getBeneficiaryFromCalldata`, there is no consideration if the first action is `XTransfer` or `XTransferWithCall` or `DepositETH`.

```
function _getBeneficiaryFromCalldata(bytes memory callData) // @audit come
    internal
    pure
    returns (address beneficiary)
  {
    (Action[] memory actions, bytes[] memory args,) =
      abi.decode(callData, (Action[], bytes[], uint256));
    if (actions[0] == Action.Deposit || actions[0] == Action.Payback) {
       --
    } else if (actions[0] == Action.Withdraw || actions[0] == Action.Borrc
     --
    } else if (actions[0] == Action.WithdrawETH) {
       ---
    } else if (actions[0] == Action.PermitBorrow || actions[0] == Action.P
     --
    } else if (actions[0] == Action.Flashloan) {
     --
    } else if (actions[0] == Action.Swap) {
      revert BaseRouter__bundleInternal_swapNotFirstAction(); // @audit is
    }
  }
```

One cannot misuse it since `_getBeneficiaryFromCalldata` returns `address(0)` as the beneficiary, which won't match the actual beneficiary inside `_checkBeneficiary`.

However, it is still possible to sweep the assets using this method, although there are many other ways of doing so, as explained in the following section [I-5] Anyone can sweep the unused funds from the router.

Remediations to consider: Although we couldn't think of a way to exploit this, it doesn't mean that none will exist. The best practice is to reduce exposure at the source. Consider adding a revert inside `_getBeneficiaryFromCalldata` if the first action is `XTransfer`, `XTransferWithCall`, or `DepositETH`, similar to swap.

L-9 `_addTokenToList` is missing for `XTransfer` and `XTransferWithCall` actions

TOPIC	STATUS	IMPACT	LIKELIHOOD
Input Validation	Acknowledged	Low	Medium

Currently, `_addTokenToList` is not done for both `XTransfer` and `XTransferWithCall` actions. This can result in dormant contract funds being used or unintentionally leaving funds in the contract.

Remediations to consider:

While this issue is not considered severe since there are other ways of sweeping the contract, [I-5] Anyone can sweep the unused funds from the router.

consider adding tokens to the `tokensToCheck` list as is done for all other actions to remove this exposure completely.

Q-1 Different versioning systems inside Fuji Vault

TOPIC	STATUS	QUALITY IMPACT
Design	Acknowledged	Medium

Currently, two versioning systems are being used: "1,2,3..n" for the `EIP712` domain separator and "0.0.1, x.y.z" for the immutable `VERSION`. This can lead to confusion for users and requires a conscious effort from Fuji developers to update both version numbers instead of just one.

To simplify this process, consider combining both version systems into one and using a single versioning system across the board.

Q-2 Unnecessary use of `_msgSender()`

TOPIC	STATUS	QUALITY IMPACT
Redundancy	Acknowledged	Medium

```
For example :

BaseVault.sol
function approve(address receiver, uint256 shares) public override(ERC20,
    address owner = _msgSender();
        .....
}

BorrowingVault.sol
function borrow(uint256 debt, address receiver, address owner) public over
    address caller = _msgSender();
```

Currently, Fuji's code uses both `_msgSender()` from the OZ context library and standard `msg.sender`. However, since Fuji has no intention of enabling meta transactions, using standard `msg.sender` should work everywhere. Consider using `msg.sender` in all places to remove this confusion and save a small amount of gas.

Q-3 Double conversion of `Shares <> Assets` in withdraw

TOPIC	STATUS	QUALITY IMPACT
Precision Loss	Acknowledged	Medium

```
Withdraw
address caller = _msgSender();
if (caller != owner) {
      _spendAllowance(owner, caller, receiver, **convertToShares(assets)**
}

function _spendAllowance(
    address owner,
    address operator,
    address receiver,
    uint256 shares // @audit inherited incorrectly
) internal
{
    _spendWithdrawAllowance(owner, operator, receiver, **convertToAssets(s
}
```

In the `Withdraw` function of `BaseVault` on line 420, assets are converted into shares and passed to `spendAllowance()`. However, `spendAllowance()` again converts these shares into assets.

This double conversion is redundant and causes a slight loss of precision.

Consider defining a different internal method to spend allowance in terms of assets.

Q-4 Redundant code on `L597` of `BaseVault.sol`

TOPIC	STATUS	QUALITY IMPACT
Redundancy	Acknowledged	Low

```
BaseVault.sol
L596: function _beforeTokenTransfer(address from, address to, uint256 amou
L597: to;
    ....
```

Consider removing this `to;` since it doesn't do anything.

Q-5 Unnecessary use of counters library for `nonces`

TOPIC	STATUS	QUALITY IMPACT
Redundancy	Acknowledged	Medium

```
VaultPermissions.sol
L372
function _useNonce(address owner) internal returns (uint256 current) {
    Counters.Counter storage nonce = _nonces[owner];
    current = nonce.current();
    nonce.increment();
}
```

Consider defaulting back to standard variable increment and a read instead of depending on the counters library.

Q-6 Unused Import inside `BorrowingVault.sol`

TOPIC	STATUS	QUALITY IMPACT
Redundancy	Acknowledged	Low

```
BorrowingVault.sol
import {IFlasher} from "../../interfaces/IFlasher.sol";
```

Consider removing this unused import.

Q-7 Unused mapping in `BorrowingVault.sol` : `_borrowAllowances`

TOPIC	STATUS	QUALITY IMPACT
Redundancy	Acknowledged	Medium

```
BorrowingVault.sol
L93 mapping(address => mapping(address => uint256)) private _borrowAllowan
```

Consider removing this unused state variable.

Q-8 Unnecessary state variable and logic in `BaseRouter.sol` : `isAllowedCaller`

TOPIC	STATUS	QUALITY IMPACT
Best Practices	Acknowledged	Medium

`BaseRouter` defines the `isAllowedCaller` mapping on `L61`. It allows timelock to mutate it using `allowCaller()`. However, `isAllowedCaller` is not used anywhere inside routers. Instead, it is used in `ConnextHandler`.

To avoid unnecessary external calls and improve code readability, consider moving this mapping and its features to `ConnextHandler` only.

Q-9 Unsafe Transfer of ERC20

TOPIC	STATUS	QUALITY IMPACT
ERC20 Variations	Acknowledged	Medium

```
function sweepToken(ERC20 token, address receiver) external onlyHouseKeepe
    uint256 balance = token.balanceOf(address(this));
    token.transfer(receiver, balance);
}
```

Inside `sweepToken` of `BaseRouter.sol`, the safe transfer is not added as it's done elsewhere.

There is no security exposure since there is no state update based on token transfers. However, consider adding this feature to prevent housekeepers from assuming failed transfers as successful.

Q-10 Incorrect naming for beneficiary inside payback action of the router.

TOPIC	STATUS	QUALITY IMPACT
Naming	Acknowledged	Low

```
BaseRouter.sol
L168

(IVault vault, uint256 amount, **address receiver**, address sender) =
          abi.decode(args[i], (IVault, uint256, address, address));
________
vault.payback(amount, receiver);
```

Since its owner whose debt position is being paid, consider renaming `receiver` to the `owner`.

G-1 Consider defining `_asset` as immutable inside `BaseVault.sol`

TOPIC	STATUS	GAS SAVINGS
SSTORE/SLOAD	Acknowledged	High

```
BaseVault.sol

L65: IERC20Metadata internal _asset;
```

Since there is no option to change the collateral asset of the vault once it is defined, consider defining `_asset` as immutable. This will save gas since gas-

intensive `SLOADs` will no longer be required.

G-2 Consider defining `_debtAsset` as immutable inside `**BorrowingVault.sol**`

TOPIC	STATUS	GAS SAVINGS
SSTORE/SLOAD	Acknowledged	High

```
BorrowingVault.sol

L87: IERC20Metadata internal _debtAsset;
```

Since there is no option to change the debt asset of the vault once it is defined, consider defining `_debtAsset` as immutable. This will save gas since gas-intensive `SLOADs` will no longer be required.

G-3 Consider adding a break in the loop once the validity of the provider is proved inside `_isValidProvider`

TOPIC	STATUS	GAS SAVINGS
Loop Break	Acknowledged	High

```
function _isValidProvider(address provider) internal view returns (bool ch
    uint256 len = _providers.length;
    for (uint256 i = 0; i < len;) {
        if (provider == address(_providers[i])) {
            check = true;
```

```
                        <<break here="true">>
        }
        unchecked {
            ++i;
        }
    }
}
```

Once the match is found inside the `providers` array, you can break the loop and return. Doing so would save significant gas costs since no further `SLOAD` operations would be performed for the remaining `providers`.

G-4 Consider adding a break in the loop of `_isInTokenList`

TOPIC	STATUS	GAS SAVINGS
Loop Break	Acknowledged	High

```
function _isInTokenList(address token) private view returns (bool value) {
    uint256 len = _tokensToCheck.length;
    for (uint256 i = 0; i < len;) {
      if (token == _tokensToCheck[i].token) {
        value = true;
                <<break here="true">>
      }
      unchecked {
        ++i;
      }
    }
  }
```

Once the token is found inside the `_tokensToCheck` array, there is no need to continue iterating; consider adding a `break` statement as shown in the code above to save gas costs of further interactions in the loop.

I-1 Implicit Limit on User Withdrawal Amounts

TOPIC	IMPACT
Insolvency	Informational ✳

While the Vault allows the distribution of its debt position to multiple providers, withdrawals can only be made from the active provider. If the current active provider cannot fulfill the order, there is no way to withdraw from other providers. This results in a limit on the amount that users can withdraw.

Response From Fuji Team: We acknowledge the problem, and in fact distributing funds across providers of a vault is intentional. The reason for this is if the vault have enough pooled assets, they will be incurring interest rate slippage when rebalancing funds. To avoid the situation you are highlighting, we were thinking on handling this aspect offchain. That means, that we only rebalance at least the position size (collateral+debt) of the major depositor. Let me know if that clarifies. Though a more robust solution will be handling rebalancing restrictions to that of the "major-position" size. That will avoid the situation your described at smart contract level.

I-2 Lack of Support for Certain ERC20 Token Types

TOPIC	IMPACT
ERC20 Variations	Informational ✳

The following ERC20 token types are not supported by fuji vaults and routers:

Token Type	Description
Reentrant	ERC777 Tokens
Fee on Transfer	Some tokens have a transfer fee (e.g. STA, PAXG), while others may introduce one in the future (e.g. USDT, USDC).
Rebasing	Some tokens may modify balances arbitrarily outside of transfers (e.g. Ampleforth-style rebasing tokens, Compound-style airdrops of governance tokens, mintable/burnable tokens).
Multiple Addresses	Some proxied tokens have multiple addresses.
Low Decimals	Some tokens, like GUSD, only have 2 decimals.

I-3 Providers should not have a state of their own, and any storage writes.

TOPIC	IMPACT
Delegate Call	Informational ✳

Fuji does delegates call to their provider contracts so that `BaseVault` could act as a counterparty to actual lending providers like Aave.

Delegate calls should be handled carefully. Please ensure that:

1. Providers do not have any state of their own.
2. They do not write any state of vaults.

They should purely act as logical contracts and nothing else.

Please note that having immutable variables inside providers should remain safe since they are directly embedded in bytecode.

I-4 Fuji borrowing vault doesn't handle the case of liquidation of its debt position on actual lending providers.

TOPIC	IMPACT
Insolvency	Informational ✳

If any actual lending provider liquidates Fuji's borrowing vault's debt position, the penalty is applied to all shareholders and not only the borrowers, which is unfair.

Fuji's counter is they will always have more conservative ratios than the actual providers. Therefore, users' positions would be liquidated on their platform first. However, this assumption may not hold true during flash crashes. With conservative debt ratios

- The liquidator role holders of Fuji need to be very vigilant to handle liquidations before actual ones in the event of flash crashes.
- The asset/debt asset pair need to be relatively prone to flash crashes.

Please consider this moving forward.

I-5 Anyone can sweep the unused funds from the router.

TOPIC	IMPACT
Theft of Assets	Informational ✳

There are sweep methods explicitly defined with access to only timelock. However, there are several ways to anyone can execute a sweep. One way is through `xReceiver`,

Please check the following code snippet from `xRecieve` of the router.

```
function xReceive(
    bytes32 transferId,
    uint256 amount,
    address asset,
    address originSender,
    uint32 originDomain,
    bytes memory callData
  )
_____
            balance = asset_.balanceOf(address(this));
        if (balance < amount) {
        revert ConnextRouter__xReceive_notReceivedAssetBalance();
      } else {
        _tokensToCheck.push(Snapshot(asset, balance - amount));
      }
```

It is possible to call `xReceive` with `amount` equal to the balance of the contract without sending any tokens. The above code snippet treats those funds as incoming and allows the sender to spend them as they please.

Another way is through `_crossTransfer`. Since there are no checks regarding the token list there, one can easily bridge any dormant assets of the contract to any receiver.

I-6 Beneficiaries could be changed in one bundle by initially passing `address(0)` as beneficiary.

TOPIC	IMPACT
Spec Break	Informational ✳

By continuously passing `address(0)` as the beneficiary, it is theoretically possible to have `address(0)` as the beneficiary for multiple actions and change it later, as shown in the code below.

```
function _checkBeneficiary(address user) internal {
    if (_beneficiary == address(0)) {
      _beneficiary = user;
    } else {
      if (_beneficiary != user) {
        revert BaseRouter__bundleInternal_notBeneficiary();
      }
    }
  }
```

We couldn't find any incentive for someone to do this, but we wanted to make you aware of this case in case you can think of any ways one can exploit this.

Disclaimer

Macro makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Macro specifically disclaims all implied warranties of merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Macro will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand by any other party. In no event will Macro be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Macro has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Emergent team and only the source code Macro notes as being within the scope of Macro's review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites operated by persons other than Macro. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Macro is not responsible for the content or operation of such websites, and that Macro shall have no liability to your or any other person or entity for the use of third party websites. Macro assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.