Final Commit:
https://github.com/etherfi-protocol/cash-v3/tree/a9abc1838458339eff8689eea4b57cab41c0060b

High severity findings

H-01. Bridging funds can get locked inside the module that requested them

Description: The requestBridge() function is used by a module to request a withdrawal from the CashModule. Once the withdrawal timelock expires executeBridge() is called which transfers the assets to the module, which then bridges them.

However there is a serious vulnerability here that occurs in case the CashModule is configured with no withdrawalDelay - meaning that requested withdrawals get executed immediately. Here is a breakdown of the issue:

1. Module_1 requestBridge() gets called to bridge 100 tokens, which internally calls CashModule.requestWithdrawalByModule() function.
2. Since withdrawalDelay is set to 0 _processWithdrawal is called in the same transaction, sending 100 tokens from Safe -> Module_1
3. The withdrawal request is deleted in the CashModule, but remains in Module_1
4. Module_1 cannot call executeBridge() since the record does not exist anymore on CashModule. If it calls cancelBridge() it would delete the record, but the 100 tokens would still remain locked in the contract

In the end the tokens remain permanently locked in the Module contract.

Recommendations: When calling requestBridge() make sure to check if withdrawalDelay is set to 0 and also validate the before and after balances received match the requested amounts and if that is the case, immediately bridge the funds, else execute the current logic that assumes delay

Customer's response: Fixed in commit 670384

Fix Review: Fixed

H-02. Modules do not account for pending withdraws, which allows malicious Safes to prevent being liquidated and cause a DOS to multiple other functions

Description: Most modules execute actions on behalf of a safe through the execTransactionFromModule() function.

Those actions include transferring funds out of the Safe to an external entity for which the specific module handles the interaction with. The problem is that those fund transfers do not consider the already pending withdrawals in the CashModule, which creates a loophole that can be exploited by a malicious safe owner.

In order to demonstrate the problem, this issue would focus on the deposit flow inside EtherFiLiquidModule. Here is the flow of the exploit:

1. Safe_1 has pending withdrawal in the CashModule - 100 units of TokenA & 100 units of TokenB, which are also the total balances sitting in the contract for those tokens
2. Safe_1 admin issues a signature to call deposit() on EtherFiLiquidModule for 100 units of tokenA and executes it. Here is how the call trace would look:
    a. Safe_1 execTransactionFromModule() is called and 100 TokenA tokens are deposited into the liquidAsset contract. This reduces the TokenA balances of Safe_1 to 0, but it also still has a pending withdrawal for 100 TokenA
    b. At the end of the execTransactionFromModule() the postOpHook runs to validate the health status of the safe (ensureHealth())
    c. ensureHealth() -> getMaxBorrowAmount() -> CashLens.getUserTotalCollateral() - this is where the important detail lies. Since the balance of TokenA is already 0 the code after that will not run, allowing a state where balance for a token is 0, but still having a pendingWithdrawal for 100 tokens. In contrast if the module transfer was not for the full 100 tokens, let's say 99, then the next line would be reached and we would have 99-100, which would underflow and revert, preventing the Safe admin to create a state where pending withdrawals exist without balances to cover them.

3. At the end the Safe has been put into a state where the pending withdrawal for 100 TokenA remains, while the actual balances remain at 0
4. Imagine at the same time that Safe_1 had borrowed 100 USD worth of debt and that Safe_1 would have enough collateral of other tokens to remain healthy even if all balances of TokenA are withdrawn. Meaning that the above call flow would still execute successfully
5. A week later collateral prices drop and Safe_1 becomes unhealthy and ready to be liquidated().
6. And here the Safe Admin takes advantage of the inconsistent state created earlier - he donates 1 wei of TokenA to Safe_1 so that balances are not 0
7. Someone calls the liquidate() function - the function internally calls liquidatable() which validates if the Safe can be liquidated. liquidatable() uses getMaxBorrowAmount() which again uses CashLens.getUserTotalCollateral(). BUT since balance for TokenA is NOT 0 anymore, the following line would run trying to subtract 100 from 1, causing a revert.
8. As result the liquidate function would always fail and prevent the safe from getting liquidated

The big issue here is that it is possible to create a state where pending withdrawals can exist, without having available balances to cover them. The above exploit (causing a DOS to liquidations) is the most severe our team managed to uncover, however there are multiple areas that would be affected by this, like:

- All functions in CashLens that rely on substrating pending withdraws from balances (getUserCollateralForToken(), getUserTotalCollateral(), canSpend())
- Functions that rely on the above functions - ensureHealth() & liquidatable() which are frequently used throughout the codebase - borrow(), spend(), the postOpHook() which runs after all execTransactionFromModule() calls

The problem stems from the fact that modules directly transfer funds without considering the pending amounts, which allows the inconsistent state to be created

Recommendations: Inside all modules consider implementing a similar approach to spending, where the pending withdrawals are reduced or cancelled to create

enough available balances that can be taken out of a safe. This would ensure consistent accounting and prevent broken states

Customer's response: Fixed in commit 35992fb

Fix Review: Fixed

Medium severity findings

M-01. Canceling bridge transactions can be front-runned and temporarily block pending withdrawals from being resolved

Description: The newly added cancelBridge() function in the EtherFiLiquid and Stargate modules is used to cancel bridging transactions that were requested through requestBridge().

The function requires signatures which are verified against the safe and if they're valid a call to CashModule.cancelWithdrawalByModule() is done before the withdraw record is delete:

```
JavaScript
 try cashModule.cancelWithdrawalByModule(safe) {}
        catch {
            // If the cancellation fails, we still want to emit the event and
delete the withdrawal
            // This allows to cancel a bridge tx even if the withdrawal was
overridden on cash module
        }
emit LiquidBridgeCancelled(safe, withdrawal.asset, withdrawal.destEid,
withdrawal.destRecipient, withdrawal.amount);

delete withdrawals[safe];
```

The specific thing about the flow is that even if the call to cashModule.cancelWithdrawalByModule() fails, the module would suppress the error and delete the withdrawal record.

This creates a niche case that allows a malicious user to exploit the flow and cause the pending withdrawal to be temporarily stuck by leveraging the 63/64 rule (EIP-150)

Exploit scenario:

1. Safe1 has a pending bridging withdrawal

2. Safe1 multisig issues a signature to call cancelBridge() and sends a transaction
3. Bob monitors and front-runs it, taking the signatures and calling cancelBridge(). Bob has precalculated the appropriate amount of gas to send so that upon reaching the try block, the external call to cancelWithdrawalByModule() would consume 63/64 of the gas that is left - meaning 63/64 parts would be provided for execution to cancelWithdrawalByModule() and 1/64 part would remain in cancelBridge(). The 63/64 part, which Bob carefully precalculated, would not be sufficient for cancelWithdrawalByModule() and it would fail, however the 1/64 that is left would still be enough to delete the storage variable (which actually refunds gas) and emit an event.
4. As result the withdrawal record is deleted in the module, but still exists in the CashModule. Due to the following check it can be cancelled only by the module (which already deleted the record)

The issue is of medium severity since a Safe can still remediate the situation by requesting a new bridge transaction which would cancel the latest withdrawal on the CashModule and create a new one which is in sync with the record in the module.

Recommendations: Probability of the exploit is low, but it is still important to make the team aware of it. One approach to solve the above issue is to require that the caller of the function is an expected address - the safe multisig or another entity.

Customer's response: Fixed in commit 9292a76

Fix Review: Fixed

Low severity findings

L-01. Module pending withdrawals can be cancelled outside of the module

Description: Currently the cancelBridge() function requires a signature which creates the assumption that pending withdrawals should only be cancelled after a safe multisig has issued a signature to approve it. This is further reinforced by the following check. However there are currently a few ways to bypass that and still cancel the withdrawals of modules externally without using the cancelBridge() function :

1. Calling requestWithdrawal() invokes _cancelOldWithdrawal() which cancels the module bridge withdrawal

2. When spending credit & debit _cancelOldWithdrawal() is invoked again

3. Also upon repayments

4. Module A calling requestBridge() will cancel Module B pending withdrawal since new requests always overwrite the old ones

No serious impact has been detected while analyzing the above scenarios, other than breaking the assumption that cancellations of module withdrawals can happen only through the module that created them

Recommendations: Given that all of the above flows originate from the Safe multisig this might be the expected behavior, but it is important that the team is aware of it.

Customer's response: Acknowledged

Fix Review: Acknowledged

Informational findings

I-01. Unnecessary pending withdrawal cancelation

Description: Latest changes in the code introduce and modify the behaviour during debit spending and repayment. Up until now, the code called _updateWithdrawalRequestIfNecessary() which reduced the amount of the pending withdrawal in order to free enough funds necessary for the respective operation:

```
JavaScript
function _updateWithdrawalRequestIfNecessary(address safe, address token,
uint256 amount) internal {
.....

 if (amount + safeCashConfig.pendingWithdrawalRequest.amounts[tokenIndex] >
balance) {
       safeCashConfig.pendingWithdrawalRequest.amounts[tokenIndex] = balance -
       amount;
       eventEmitter.emitWithdrawalAmountUpdated(safe, token, balance - amount);
        }

}
```

In the updated code _updateWithdrawalRequestIfNecessary() is replaced with _cancelWithdrawalRequestIfNecessary() which as the name suggest directly cancels the whole withdrawal for all tokens:

```
JavaScript
function _cancelWithdrawalRequestIfNecessary(address safe, address token,
uint256 amount) internal {
.....

if (amount + safeCashConfig.pendingWithdrawalRequest.amounts[tokenIndex] >
balance) {
         _cancelOldWithdrawal(safe);
  }
```

```
}
```

Here is an example of how the flow would work with the new logic:

There are pending withdrawals for 5 tokens and there is a debit spend for token A that requires 100 tokens, but only 90 are available and 20 pending. Instead of reducing the pending withdrawal for A with 10 tokens, all pending withdrawals for all other 4 tokens would be cancelled as well, although they do not actually need to be cancelled since only A requires more funds

Recommendations: It looks like the switch from reducing pending amounts to fully cancelling them is a deliberate change. Still the cancelling mechanism could be made more efficient. Instead of cancelling the whole pending withdrawals for all tokens, consider cancelling only the one for the token that requires it. This way it won't be necessary to request an entirely new withdrawal for the other tokens every time

Customer's response: Acknowledged

Fix Review: Acknowledged

I-02. Duplicate operation execution

Description: Inside the function cancelBridge() at the end it always emits an event and deletes the withdrawal. However most of the time this would already be done through the cancelBridgeByCashModule() callback, so there is no need to emit a second event if it was already done.

Recommendations: At the bottom of cancelBridge() first check if the withdrawal mapping was not already deleted in the previous step and if not, only then execute the emit+delete operation.

Customer's response: Fixed in commit 6bc07164a

Fix Review: Fixed

I-03. Missing non-reentrant modifiers

Description: The Stargate module has a reentrancy guard to its executeBridge() function. However the EtherFiLiquidModule lacks that guard for the same function. Also consider introducing the guard for the requestBridge() & cancelBridge() functions which are not expected to be called during the execution of other flows

Recommendations: Consider adding the re-entrancy guards

Customer's response: Fixed in commit ed49c66a

Fix Review: Fixed

I-04. Typo in StargateModule.sol

Description: The Stargate module contract defines hundred percent in bps in the variable HUNDRES_PERCENT_IN_BPS. There is a typo here, since it is supposed to be hundred, not hundres.

Recommendations: Consider fixing the typo

Customer's response: Fixed in commit d8a337db

Fix Review: Fixed