SHERLOCK

SHERLOCK SECURITY REVIEW FOR

SHERLOCK

Prepared for:	Sherlock
Prepared by:	Sherlock
Lead Security Expert:	hyh
Dates Audited:	September 27 - October 4, 2022
Prepared on:	October 17, 2022

Introduction

Sherlock works to protect Decentralized Finance (DeFi) users from smart contract exploits with security reviews from top auditors backed by smart contract coverage on the audited contracts.

Scope

The focus of this audit is on any potential losses in Sherlock V2 related to the integration with TrueFi or Euler.

- `contracts/EulerStrategy.sol`
- `contracts/TrueFiStrategy.sol`

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Total Issues

Medium	High
3	1

Security Experts

hyh
csanuragjain
sirhashalot
defsec
Chom

Issue H-1: An attacker can gain an inflated amount of Sherlock shares by staking when TrueFiPool2 pool has its value temporary reduced during loan closure

Source: https://github.com/sherlock-audit/2022-09-sherlock-judging/tree/main/018-H

Found by

hyh

Summary

Sherlock's initial staking can be performed while TrueFiStrategy balance is reduced with TrueFi's FixedTermLoanAgency loan token reclaim call. This is possible as the 1inch fee swapping call with user-supplied parameters is performed when the system is in the reduced pool value state, while FixedTermLoanAgency's reclaim() has no access controls, and the functions involved allow reentrancy. The net result is an artificially increased number of Sherlock shares being issued to the attacker.

Vulnerability Detail

TrueFiStrategy balance is based on TrueFiPool2 poolValue(), which drops on FixedTermLoanAgency loan reclaiming as the loan token is being removed from the active tokens array. Right after that and before reclaimed tokens end up on the TrueFiPool2 balance, the 1inch call is performed aimed to swap the corresponding fees.

FixedTermLoanAgency's reclaim() is permissionless, 1inch call uses user supplied parameters, so an attacker can supply a pre-cooked contract instead of DEX pool, which does the swap, so all the corresponding checks pass, i.e. proper amount of tokens needed are placed on the target balance, but also performs initial Sherlock staking, which issues inflated number of the shares for the attacker as TrueFiPool2 poolValue() is reduced during this call by the value of the loan being closed. Right after that the reclaimed token less fees are transferred to the TrueFiPool2 balance, so TrueFiStrategy _balanceOf() display the full value, and the attacker's shares are priced with a premium compared to her investment. This premium is proportional to the value of the closed loan, and is effectively stolen from all other stakers.

Impact

As a result of increased issuance of the shares, the attacker will steal from the all other stakers proportionally to the size of a loan she be able to run reclaim for.

Code Snippet

The call sequence is: `FixedTermLoanAgency.reclaim()->_redeemAndRepay()->_swapFee()->OneInchExchange.exchange()->AggregationRouterV4.swap()`.

FixedTermLoanAgency's reclaim() removes the loan token and calls _redeemAndRepay():

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/FixedTermLoanAgency.sol#L384-L407

```
function reclaim(IFixedTermLoan loanToken, bytes calldata data) external {
    IFixedTermLoan.Status status = loanToken.status();
    require(
        status == IFixedTermLoan.Status.Settled || status ==
↪   IFixedTermLoan.Status.Defaulted,
        "FixedTermLoanAgency: LoanToken is not closed yet"
    );
    if (status == IFixedTermLoan.Status.Defaulted) {
        require(msg.sender == owner(), "FixedTermLoanAgency: Only owner can
↪   reclaim from defaulted loan");
    }

    // find the token, repay loan and remove loan from loan array
    ITrueFiPool2 pool = loanToken.pool();
    IFixedTermLoan[] storage _loans = poolLoans[pool];
    uint256 loansLength = _loans.length;
    for (uint256 index = 0; index < loansLength; index++) {
        if (_loans[index] == loanToken) {
            _loans[index] = _loans[loansLength - 1];
            _loans.pop();

            uint256 fundsReclaimed = _redeemAndRepay(loanToken, pool, data);
            emit Reclaimed(address(pool), address(loanToken), fundsReclaimed);
            return;
        }
    }
```

At that moment FixedTermLoanAgency's value() drops by the currentValue() of the removed loan token, let's denote it `loan_value=loanToken.currentValue(address(ftlAgency))`:

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/FixedTermLoanAgency.sol#L371-L378

```
function value(ITrueFiPool2 pool) external view override returns (uint256) {
    IFixedTermLoan[] memory _loans = poolLoans[pool];
    uint256 totalValue;
    for (uint256 index = 0; index < _loans.length; index++) {
        totalValue = totalValue.add(_loans[index].currentValue(address(this)));
```

```
    }
    return totalValue;
}
```

Which means TrueFiPool2's loansValue() and poolValue() drop by the same `loan_value`:

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/TrueFiPool2.sol#L419-L428

```
function loansValue() public view returns (uint256) {
    if (inSync) {
        return loansValueCache;
    }
    uint256 lenderLoansValue = 0;
    if (address(lender) != address(0)) {
        lenderLoansValue = lender.value(this);
    }
    return lenderLoansValue.add(ftlAgency.value(this));
}
```

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/TrueFiPool2.sol#L382-L390

```
/**
 * @dev Calculate pool value in underlying token
 * "virtual price" of entire pool - LoanTokens, UnderlyingTokens, strategy value
 * @return pool value denominated in underlying token
 */
function poolValue() public view override returns (uint256) {
    // this assumes defaulted loans are worth their full value
    return
↪   liquidValue().add(loansValue()).add(deficitValue()).add(creditValue()).add(debtValue);
}
```

_redeemAndRepay() first calls _swapFee(), and only then repays the `fundsReclaimed.sub(feeAmount)` to the pool, reimbursing the token removal skew:

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/FixedTermLoanAgency.sol#L455-L475

```
function _redeemAndRepay(
    IFixedTermLoan loanToken,
    ITrueFiPool2 pool,
    bytes calldata data
) internal returns (uint256) {
    // call redeem function on LoanToken
```

```
    uint256 balanceBefore = pool.token().balanceOf(address(this));
    loanToken.redeem();
    uint256 balanceAfter = pool.token().balanceOf(address(this));

    // gets reclaimed amount and pays back to pool
    uint256 fundsReclaimed = balanceAfter.sub(balanceBefore);

    uint256 feeAmount;
    if (address(feeToken) != address(0)) {
        // swap fee for feeToken
        feeAmount = _swapFee(pool, loanToken, data);
    }

    pool.token().safeApprove(address(pool), fundsReclaimed.sub(feeAmount));
    pool.repay(fundsReclaimed.sub(feeAmount));
```

This way when _swapFee() is called the TrueFiPool2's poolValue() is still reduced by `loan_value`.

This allows Megan the attacker to enter Sherlock's pool at a depressed valuation.

Notice, that _swapFee() and all the previous function calls just redirect user-supplied `data`, that is finally passed to the 1inch's exchange():

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/FixedTermLoanAgency.sol#L484-L511

```
/// @dev Swap `token` for `feeToken` on 1inch
function _swapFee(
    ITrueFiPool2 pool,
    IFixedTermLoan loanToken,
    bytes calldata data
) internal returns (uint256) {
    uint256 feeAmount = loanToken.interest().mul(fee).div(BASIS_RATIO);
    IERC20WithDecimals token = IERC20WithDecimals(address(pool.token()));
    if (token == feeToken) {
        return feeAmount;
    }
    if (feeAmount == 0) {
        return 0;
    }
    (I1Inch3.SwapDescription memory swap, uint256 balanceDiff) =
↪   _1inch.exchange(data);
    uint256 expectedDiff =
↪   pool.oracle().tokenToUsd(feeAmount).mul(uint256(10)**feeToken.decimals()).div(1
↪   ether);

    require(swap.srcToken == address(token), "FixedTermLoanAgency: Source token
↪   is not same as pool's token");
```

```
    require(swap.dstToken == address(feeToken), "FixedTermLoanAgency: Destination
↪   token is not fee token");
    require(swap.dstReceiver == address(this), "FixedTermLoanAgency: Receiver is
↪   not agency");
    require(swap.amount == feeAmount, "FixedTermLoanAgency: Incorrect fee swap
↪   amount");
    require(swap.flags & ONE_INCH_PARTIAL_FILL_FLAG == 0, "FixedTermLoanAgency:
↪   Partial fill is not allowed");
    require(
        balanceDiff >=
↪   expectedDiff.mul(BASIS_RATIO.sub(swapFeeSlippage)).div(BASIS_RATIO),
        "FixedTermLoanAgency: Fee returned from swap is too small"
    );

    return feeAmount;
```

OneInchExchange's exchange() uses user supplied `data`, filling the basic swap description from it.

_swapFee() then verify tokens, amounts and receiving address from the filled swap structure.

However, `data` also contains the route, whom to call with what parameters to perform the swap, which is passed to `_1inchExchange`:

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/libraries/OneInchExchange.sol#L34-L70

```
function exchange(I1Inch3 _1inchExchange, bytes calldata data)
    internal
    returns (I1Inch3.SwapDescription memory description, uint256 returnedAmount)
{
    if (data[0] == 0x7c) {
        // call `swap()`
        (, description, ) = abi.decode(data[4:], (address,
↪   I1Inch3.SwapDescription, bytes));
    } else {
        // call `unoswap()`
        (address srcToken, uint256 amount, uint256 minReturn, bytes32[] memory
↪   pathData) = abi.decode(
            data[4:],
            (address, uint256, uint256, bytes32[])
        );
        description.srcToken = srcToken;
        description.amount = amount;
        description.minReturnAmount = minReturn;
        description.flags = 0;
        uint256 lastPath = uint256(pathData[pathData.length - 1]);
```

```
        IUniRouter uniRouter = IUniRouter(address(lastPath & ADDRESS_MASK));
        bool isReverse = lastPath & REVERSE_MASK > 0;
        description.dstToken = isReverse ? uniRouter.token0() :
↪   uniRouter.token1();
        description.dstReceiver = address(this);
    }

    IERC20(description.srcToken).safeApprove(address(_1inchExchange),
↪   description.amount);
    uint256 balanceBefore =
↪   IERC20(description.dstToken).balanceOf(description.dstReceiver);
    // solhint-disable-next-line avoid-low-level-calls
    (bool success, ) = address(_1inchExchange).call(data);
    if (!success) {
        // Revert with original error message
        assembly {
            let ptr := mload(0x40)
            let size := returndatasize()
            returndatacopy(ptr, 0, size)
            revert(ptr, size)
        }
    }
```

For 1inch the `data` above contains not only swap description, but `calls` array (also named `data` in the docs), which is used by 1inch as is to perform the swapping, i.e. 1inch calls the supplied addresses with the supplied parameters, only checking the destination address has received the required number of output tokens:

https://docs.1inch.io/docs/aggregation-protocol/smart-contract/AggregationRouterV4#swap

```
function swap(
  contract IAggregationExecutor caller,
  struct AggregationRouterV4.SwapDescription desc,
  bytes data
) external returns (uint256 returnAmount, uint256 gasLeft)
```

https://github.com/1inch/1inch-v2-contracts/blob/master/contracts/OneInchExchange.sol#L92-L127

```
function swap(
    IOneInchCaller caller,
    SwapDescription calldata desc,
    IOneInchCaller.CallDescription[] calldata calls
)
    external
    payable
```

```
    whenNotPaused
    returns (uint256 returnAmount)
{
    require(desc.minReturnAmount > 0, "Min return should not be 0");
    require(calls.length > 0, "Call data should exist");

    uint256 flags = desc.flags;
    IERC20 srcToken = desc.srcToken;
    IERC20 dstToken = desc.dstToken;

    if (flags & _REQUIRES_EXTRA_ETH != 0) {
        require(msg.value > (srcToken.isETH() ? desc.amount : 0), "Invalid
↪   msg.value");
    } else {
        require(msg.value == (srcToken.isETH() ? desc.amount : 0), "Invalid
↪   msg.value");
    }

    if (flags & _SHOULD_CLAIM != 0) {
        require(!srcToken.isETH(), "Claim token is ETH");
        _claim(srcToken, desc.srcReceiver, desc.amount, desc.permit);
    }

    address dstReceiver = (desc.dstReceiver == address(0)) ? msg.sender :
↪   desc.dstReceiver;
    uint256 initialSrcBalance = (flags & _PARTIAL_FILL != 0) ?
↪   srcToken.uniBalanceOf(msg.sender) : 0;
    uint256 initialDstBalance = dstToken.uniBalanceOf(dstReceiver);

    caller.makeCalls{value: msg.value}(calls);

    uint256 spentAmount = desc.amount;
    returnAmount = dstToken.uniBalanceOf(dstReceiver).sub(initialDstBalance);
```

This way Megan can supply pre-cooked contract that will do the swap (so all the conditions above be satisfied) and call Sherlock to enter the deposit in the same time.

I.e. only passing the control is needed here so an additional action, Sherlock deposit, can be performed while TrueFiPool2 poolValue() and TrueFiStrategy's _balanceOf() are depressed.

As Sherlock's shares are constant, this will mean that USDC amount supplied by Megan to Sherlock's initialStake() will be credited with an inflated number fo shares:

https://github.com/sherlock-audit/2022-09-sherlock/blob/main/sherlock-v2-core/contracts/Sherlock.sol#L523-L553

I.e. inflated `stakeShares_=(_amount*totalStakeShares_)/(totalTokenBalanceStakers()-_amount)` to be issued as part of totalTokenBalanceStakers() is strategy's balanceOf():

https://github.com/sherlock-audit/2022-09-sherlock/blob/main/sherlock-v2-core/contracts/Sherlock.sol#L159-L164

Which uses current TrueFiPool2 poolValue():

https://github.com/sherlock-audit/2022-09-sherlock/blob/main/sherlock-v2-core/contracts/strategy/TrueFiStrategy.sol#L131-L147

Tool used

Manual Review

Recommendation

Consider adding reentrancy guarding modifiers to TrueFi's reclaim() and liquidValue().

Issue M-1: Unregulated joining fees

Source: https://github.com/sherlock-audit/2022-09-sherlock-judging/tree/main/001-M

Found by

csanuragjain, defsec

Summary

tfUSDC.join deducts fees from deposited balance. Now this fees is unregulated and could be 100% of amount passed as can be seen at setJoiningFee function at https://github.com/trusttoken/contracts-pre22/blob/main/contracts/truefi2/TrueFiPool2.sol#L449. Since minting will not fail even with zero amount, our contract will end up everything given as fees

Vulnerability Detail

1. Observe the _deposit function

```
function _deposit() internal override whenNotPaused {
    //
    ↪  https://github.com/trusttoken/contracts-pre22/blob/main/contracts/truefi2/TrueFiPool2.s
    tfUSDC.join(want.balanceOf(address(this)));
...
}
```

2. This makes call to join function

```
function join(uint256 amount) external override joiningNotPaused {
        uint256 fee = amount.mul(joiningFee).div(BASIS_PRECISION);
        uint256 mintedAmount = mint(amount.sub(fee));
        claimableFees = claimableFees.add(fee);

        // TODO: tx.origin will be deprecated in a future ethereum upgrade
        latestJoinBlock[tx.origin] = block.number;
        token.safeTransferFrom(msg.sender, address(this), amount);

        emit Joined(msg.sender, amount, mintedAmount);
    }
```

3. As we can see this join function deducts a fees from the deposited amount before minting. Lets see this joining fees
4. The joining fees is introduced using setJoiningFee function

```
function setJoiningFee(uint256 fee) external onlyOwner {
        require(fee <= BASIS_PRECISION, "TrueFiPool: Fee cannot exceed
        ↪  transaction value");
        joiningFee = fee;
        emit JoiningFeeChanged(fee);
    }
```

5. This means the joiningFee will always be in between 0 to BASIS_PRECISION. This BASIS_PRECISION can be 100% as shown

```
uint256 private constant BASIS_PRECISION = 10000;
```

6. This means if joiningFee is set to BASIS_PRECISION then all user deposit will goto joining fees with user getting nothing

Impact

Contract will lose all deposited funds

Code Snippet

https://github.com/sherlock-audit/2022-09-sherlock/blob/main/sherlock-v2-core/contracts/strategy/TrueFiStrategy.sol#L90

Tool used

Manual Review

Recommendation

Post calling join, check amount of shares minted for this user (use balanceOF on TrueFiPool2.sol) and if it is below minimum expected revert the transaction

```
uint256 tfUsdcBalance = tfUSDC.balanceOf(address(this));
require(tfUsdcBalance>=minSharesExpected, "Too high fees");
```

Issue M-2: Loss of funds with high liquidExitPenalty

Source: https://github.com/sherlock-audit/2022-09-sherlock-judging/tree/main/012-M

Found by

defsec, Chom, sirhashalot

Summary

A high `liquidExitPenalty` fee percentage in the TrueFi protocol can result in a loss of user funds. The issue is that only the Sherlock contract owner can set the `yieldStrategy` in use and call `liquidExit` to withdraw funds from TrueFi. A withdrawal from the TrueFi yield strategy with unfavorable withdrawal fees can occur in the event that `yieldStrategy` is set to TrueFi and `liquidExit` is called with high withdrawal fees. This could be done intentionally by a rogue admin (centralization risk) but it can also happen in a benign manner when many users want to withdraw funds at the same time and USDC must be made available to them, regardless of the exit penalty fee.

Vulnerability Detail

TrueFi has a variable withdrawal fee. The specific numbers are in this spreadsheet, with a maximum fee of 10%. There is a large comment block in the TrueFi strategy explaining how this strategy will keep the withdrawal fee small. The problem underlying this strategy as it is coded is that a withdrawal at the wrong time can result in substantial loss of user funds. In a worse case scenario, it could lead to a bank run where some users are unable to get their deposits back because so much is lost to fee.

There is another substantial risk in lending to the TrueFi protocol. The risk is that TrueFi borrowers do not pay back their undercollateralized loans. This risk is clearly identified in the TrueFi documentation. While this risk may be unlikely with the current list of TrueFi borrowers, Sherlock must monitor new borrowers in the TrueFi ecosystem over time to assure that the risk level remains low.

Impact

The TrueFi strategy can result in loss of user deposits in certain circumstances.

Code Snippet

The function that the owner can use to set the yield strategy https://github.com/sherlock-protocol/sherlock-v2-core/blob/355c70df23aa9aa7d46567c9540a6d15be93fcab/contracts/Sherlock.sol#L272-L280

The function that the owner can use to withdraw funds from TrueFi https://github.com/sherlock-protocol/sherlock-v2-core/blob/355c70df23aa9aa7d46567c9540a6d15be93fcab/contracts/strategy/TrueFiStrategy.sol#L155-L185

Tool used

Manual Review

Recommendation

The TrueFi strategy is higher risk than other strategies for Sherlock staking. This level of risk may not be acceptable to some users because of the risk of loss of funds. If this strategy is to be used, Sherlock must insure its deposits in TrueFi to avoid losing user's deposits because Sherlock's reputation is on the line.

Issue M-3: Griefing attack is possible via TrueFi borrowing

Source: https://github.com/sherlock-audit/2022-09-sherlock-judging/tree/main/017-M

Found by

hyh

Summary

As TrueFiStrategy net asset value reported with _balanceOf() depends on the TrueFi liquidity situation, which can be modified by any user eligible for TrueFi loans, a griefing attack of borrowing right ahead of Sherlock requesting NAV figure from the strategy is possible.

Vulnerability Detail

On observing a Sherlock's redeemNFT() or arbRestake() from Bob the target user, Alice the attacker will front run his transaction with borrow() call, reducing the amount of liquid funds a TrueFiPool2 has, immediately repaying her debt right afterwards. I.e. Alice will become a short term debtor for sandwiching Bob's redeemNFT(). Alice needs to be whitelisted as a borrower, which is achievable, and her status will not be affected by this attack and examined alone her actions are legitimate, i.e. she only takes are repays short-term loan, which is valid usage of TrueFi.

As loan term is small the overall cost of the attack (that consists of the loan interest along with gas costs for borrow and repay transactions) isn't substantial for Alice, while Bob can incur big enough slippage as TrueFiStrategy's _balanceOf() will observe low `liquidValue()` and thus high `liquidExitPenalty(totalAmount)` of the TrueFiPool2, diminishing the value of the Bob's withdrawal. Alice can repeat this many times over with various Sherlock users.

Impact

Sherlock stakers who unstaked will incur higher fees proportionally to the current TrueFi liquidity situation and the budget Alice have for the attack (with increased budget she can do it more frequently and take bigger loans, moving liquidity more substantially).

Code Snippet

There are two agencies that can be used for borrowing, FixedTermLoanAgency and LineOfCreditAgency, and Alice needs to become an allowed borrower in either of

them:

FixedTermLoanAgency:

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/FixedTermLoanAgency.sol#L288-L291

```
function allowBorrower(address who) external onlyOwner {
    isBorrowerAllowed[who] = true;
    emit BorrowerAllowed(who);
}
```

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/FixedTermLoanAgency.sol#L327-L351

```
function borrow(
    ITrueFiPool2 pool,
    uint256 amount,
    uint256 term,
    uint256 _maxApy
) external onlyAllowedBorrowers {
    require(poolFactory.isSupportedPool(pool), "FixedTermLoanAgency: Pool not
↪   supported by the factory");
    require(poolLoans[pool].length < maxLoans, "FixedTermLoanAgency: Loans number
↪   has reached the limit");

    address borrower = msg.sender;
    require(borrowingMutex.isUnlocked(borrower), "FixedTermLoanAgency: There is
↪   an ongoing loan or credit line");
    require(
        creditOracle.status(borrower) == ITrueFiCreditOracle.Status.Eligible,
        "FixedTermLoanAgency: Sender is not eligible for loan"
    );

    require(amount > 0, "FixedTermLoanAgency: Loans of amount 0, will not be
↪   approved");
    require(
        pool.oracle().tokenToUsd(amount) <= borrowLimit(pool, borrower),
        "FixedTermLoanAgency: Loan amount cannot exceed borrow limit"
    );

    require(term > 0, "FixedTermLoanAgency: Loans cannot have instantaneous term
↪   of repay");
    require(isTermBelowMax(term), "FixedTermLoanAgency: Loan's term is too
↪   long");
    require(isCredibleForTerm(term), "FixedTermLoanAgency: Credit score is too
↪   low for loan's term");
```

LineOfCreditAgency:

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/LineOfCreditAgency.sol#L236-L240

```
/// @dev set borrower `who` to whitelist status `isAllowed`
function allowBorrower(address who, bool isAllowed) external onlyOwner {
    isBorrowerAllowed[who] = isAllowed;
    emit BorrowerAllowed(who, isAllowed);
}
```

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/LineOfCreditAgency.sol#L380-L395

```
/**
 * @dev Borrow from `pool` for `amount` using lines of credit
 * Only whitelisted borrowers that meet all requirements can borrow
 * @param pool Pool to borrow from
 * @param amount Amount of tokens to borrow
 */
function borrow(ITrueFiPool2 pool, uint256 amount) external onlyAllowedBorrowers
↪  {
    require(poolFactory.isSupportedPool(pool), "LineOfCreditAgency: The pool is
↪  not supported for borrowing");
    require(amount > 0, "LineOfCreditAgency: Borrowed amount has to be greater
↪  than 0");
    require(
        creditOracle.status(msg.sender) == ITrueFiCreditOracle.Status.Eligible,
        "LineOfCreditAgency: Sender not eligible to borrow"
    );
    require(!_hasOverdueInterest(pool, msg.sender), "LineOfCreditAgency: Sender
↪  has overdue interest in this pool");
    uint8 rawScore = creditOracle.score(msg.sender);
    require(rawScore >= minCreditScore, "LineOfCreditAgency: Borrower has credit
↪  score below minimum");
```

However, this is one time requirement and the attack itself doesn't look malicious, i.e. Alice uses ultra short term borrowing in a flash loan manner, paying the interest, there is no violation in Truefi2 use cases, so becoming an allowed borrower once Alice can perform the attack many times over an extended period of time.

As Alice will seek the easiest path, the FixedTermLoanAgency that doesn't require credit scores for short term borrowing looks the most suitable:

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/FixedTermLoanAgency.sol#L525-L527

```
function isCredibleForTerm(uint256 term) internal view returns (bool) {
```


```
    return term <= longTermLoanThreshold || creditOracle.score(msg.sender) >=
↪   longTermLoanScoreThreshold;
}
```

https://github.com/trusttoken/contracts-pre22/blob/986fefb91fb0619217d17ecf0b4c5b7b921130ed/contracts/truefi2/FixedTermLoanAgency.sol#L206

```
longTermLoanThreshold = 90 days;
```

Tool used

Manual Review

Recommendation

As this is design specifics only partial remediation looks viable. As an example, some delay can be introduced in the TrueFi between loan request and funds transfer. It will not impact valid use cases, but close this attack surface.