Introduction

A smart contract security audit review of the init captial protocol was done by sparkware's auditor, 0xladboy233, with a focus on the security aspects of the application's implementation.

About Sparkware security

Sparkware security offers cutting edge and affordable smart contract auditing solution. The auditors get reputatoin and skill by consistently get top place in bug bounty and audit competition.

Our past audit prject including optimism and notional finance and graph protocol. Below is a list of comprehensive security review and research: https://github.com/JeffCX/Sparkware-audit-portfolio

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

About **Protocol Name**

Security Assessment Summary

***review commit hash* - [fe708f3531ccf052bdc0d8f20db5dda9ef53af0c]**

and

[f0b314dc3470b6a2ca12ca5d91390949acbbd9ee]

and

Scope

The following smart contracts were in scope of the audit:

Scope1 (https://github.com/init-capital/init-blast-private-contract-audit/tree/main):
MarginTradingHook.sol
ThrusterSwapHelper.sol

Scope2 (https://github.com/init-capital/init-private-audit/tree/main):
MoeLBSwapHelper.sol
UniversalSwapHelper.sol

Findings

Here's a similar summary table based on the new list of findings you provided:

ID	Title	Severity
M-1	unwrap swapInfo.tokenIn leads to transaction revert in the callback	Medium
L-1	Transaction that attaches ETH does not count as fund	Low
L-2	Need works to claim the thruster points	Low
L-3	Loss of gas spent for contract ThrusterSwapHelper	Low
L-4	Lack of integration with Thruster Uniswap V2 Fork	Low
R-1	Thorough testing for MarginTradingHook.sol	Recommendation
R-2	Consider use of multicall to simplify the token conversion flow	Recommendation

This table maintains a format similar to your initial example, providing an organized view of the findings, their descriptions, and their severity levels.

M - unwrap swapInfo.tokenIn leads to transaction revert in the callback

Description

When increase the position, the collToken and borrToken is encoded in swap data:

```
    address collToken = ILendingPool(_param.collPool).underlyingToken();
    address borrToken = ILendingPool(_param.borrPool).underlyingToken();

    // @audit
    _require(
        _param.tokenIn == _getRealUnderlyingToken(collToken) || _param.tokenIn
    == _getRealUnderlyingToken(borrToken),
        Errors.INVALID_INPUT
    );
```

```
    /**
    struct SwapInfo {
        uint256 initPosId; // nft id
        SwapType swapType; // swap type
        address tokenIn; // token to swap
        address tokenOut; // token to receive from swap
        uint256 amtOut; // token amount out info for the swap
        bytes data; // swap data
    }
    */
    {
        SwapInfo memory swapInfo =
            SwapInfo(_param.initPosId, SwapType.OpenExactIn, borrToken,
    collToken, _param.minAmtOut, _param.data);
        multicallData[1] =
            abi.encodeWithSelector(IInitCore.callback.selector, address(this),
    0, abi.encode(swapInfo));
    }
```

note, the tokenId -> borrToken, tokenOut -> collToken

When the callback is triggered:

```
    function coreCallback(address _sender, bytes calldata _data) external
    payable returns (bytes memory result) {
        _require(msg.sender == CORE, Errors.NOT_INIT_CORE);
        _require(_sender == address(this), Errors.NOT_AUTHORIZED);
        SwapInfo memory swapInfo = abi.decode(_data, (SwapInfo));
        MarginPos memory marginPos = __marginPositions[swapInfo.initPosId];
        // @audit
        // even the swapInfo.tokenIn is the collateral, the token that is
    transferred in is the underlying token
        //
        if (_isRebasingWrapper(swapInfo.tokenIn)) {
            // unwrap amtIn to real token amount for swap

     IERC20RebasingWrapper(swapInfo.tokenIn).unwrap(IERC20(swapInfo.tokenIn).ba
    lanceOf(address(this)));
            swapInfo.tokenIn = _getRealUnderlyingToken(swapInfo.tokenIn);
        }
```

if the swapInfo.token is the wrapped WETH or wrapped USDB, the code will try to unwrap it.

but the problem is that when _increasePosition before trigger the callback

we enforce that the params.tokenIn is the WNATIVE or USDB.

```
    _transmitTokenIn(_param.tokenIn, _param.amtIn);

    // perform multicall to INIT core:
    // 1. borrow tokens
    // 2. callback (perform swap from borr -> coll)
    // 3. deposit collateral tokens
    // 4. collateralize tokens
    bytes[] memory multicallData = new bytes[](4);
    multicallData[0] = abi.encodeWithSelector(
```

```
        IInitCore.borrow.selector, _param.borrPool, _param.borrAmt,
    _param.initPosId, address(this)
    );
    address collToken = ILendingPool(_param.collPool).underlyingToken();
    address borrToken = ILendingPool(_param.borrPool).underlyingToken();

    // @audit
    _require(
        _param.tokenIn == _getRealUnderlyingToken(collToken) || _param.tokenIn
    == _getRealUnderlyingToken(borrToken),
        Errors.INVALID_INPUT
    );
```

this means that the token that is transferred in has to be WNATIVE or USDB,

then because no wrapped WEtH and wrapped USDB is transfer in,

```
     IERC20RebasingWrapper(swapInfo.tokenIn).unwrap(IERC20(swapInfo.tokenIn).ba
    lanceOf(address(this)));
```

unwrapping during the callback will revert.

Recommendation

change the check to

```
    _require(
        _param.tokenIn == collToken || _param.tokenIn == borrToken,
        Errors.INVALID_INPUT
    );
```

or remove the unwrap code during the callback.

L – Transaction that attach ETH does not count as fund

Description

note the modifier depositNative, and refundNative

```
    function addCollateral(uint256 _posId, uint256 _amtIn) external payable
depositNative refundNative {
        uint256 initPosId = initPosIds[msg.sender][_posId];
        _require(initPosId != 0, Errors.POSITION_NOT_FOUND);
        MarginPos storage marginPos = __marginPositions[initPosId];
        address collToken =
ILendingPool(marginPos.collPool).underlyingToken();
        // transfer in tokens
        _transmitTokenIn(_getRealUnderlyingToken(collToken), _amtIn);
        // wrap rebasing tokens to wrap token if necessary
        if (_isRebasingWrapper(collToken)) _amtIn =
IERC20RebasingWrapper(collToken).wrap(_amtIn);
        // transfer tokens to collPool
        IERC20(collToken).safeTransfer(marginPos.collPool, _amtIn);
        // mint collateral pool tokens
        IInitCore(CORE).mintTo(marginPos.collPool, POS_MANAGER);
        // collateralize to INIT position
        IInitCore(CORE).collateralize(initPosId, marginPos.collPool);
    }
```

deposit native convert native ETh to WETH

```
    modifier depositNative() {
        if (msg.value != 0) IWNative(WNATIVE).deposit{value: msg.value}();
        // note: no need to wrap blast rebasing token here since it will be
handled by callback function
        _;
    }
```

while refund native convert the left over WETH to ETH and sent back to user.

```
    modifier refundNative() {
        _;
        // refund native token
        uint256 wNativeBal = IERC20(WNATIVE).balanceOf(address(this));
        // NOTE: no need receive function since we will use
TransparentUpgradeableProxyBlastReceiveETH
        if (wNativeBal != 0) IWNative(WNATIVE).withdraw(wNativeBal);
        uint256 nativeBal = address(this).balance;
        if (nativeBal != 0) {
            (bool success,) = payable(msg.sender).call{value: nativeBal}
("");
            _require(success, Errors.CALL_FAILED);
        }
    }
```

But in the function addCollateral, the code always transfer the token in:

```
_transmitTokenIn(_getRealUnderlyingToken(collToken), _amtIn)
```

the WETH or USDB is transferred in,

if the underlying token is WETH, only the _amtIn that is transferred count,

even user attach 1 ETH when add collateral, the 1 will be just convert to WETH and then convert back to 1 ETH without count as collateral.

Note, this issue should apply to any function that use depositNative refundNative modifier together if the code intention is to count the attached ETH (msg.value) as fund.

Recommendation

```
    function addCollateral(uint256 _posId, uint256 _amtIn) external payable
depositNative refundNative {
        uint256 initPosId = initPosIds[msg.sender][_posId];
        _require(initPosId != 0, Errors.POSITION_NOT_FOUND);
        MarginPos storage marginPos = __marginPositions[initPosId];
        address collToken =
ILendingPool(marginPos.collPool).underlyingToken();

        // transfer in tokens
        // added code here;
        address token = _getRealUnderlyingToken(collToken);
        if (token == WNATIVE) {
            uint256 WETH_balance = WNATIVE.balanceOf(this);
            uint256 needed_amount = _amtIn - WETH_balance;
            _transmitTokenIn(token, needed_amount);
        } else {
            _transmitTokenIn(collToken, _amtIn);
        }

        // wrap rebasing tokens to wrap token if necessary
        if (_isRebasingWrapper(collToken)) _amtIn =
IERC20RebasingWrapper(collToken).wrap(_amtIn);
        // transfer tokens to collPool
        IERC20(collToken).safeTransfer(marginPos.collPool, _amtIn);
        // mint collateral pool tokens
        IInitCore(CORE).mintTo(marginPos.collPool, POS_MANAGER);
```

```
            // collateralize to INIT position
            IInitCore(CORE).collateralize(initPosId, marginPos.collPool);
        }
```

L - Need works to claim the thruster points

Description

the protocol integrate with the thruster helper,

then the protocol may be eligible for thruster points and golds

but there may some operation work to claim such points

https://docs.thruster.finance/incentives/blast-points-and-gold

Recommendation

Recommendate contact thrsuter team for integration and point claim.

L - Lose of gas spent for contract ThrusterSwapHelper

Description

Blast network distribute gas spent to contract,

but the claim mode for gas for ThrusterSwapHelper is not claimable

and the contract is not inherti from ClaimableGas.sol

Recommendation

recommendate ThrusterSwapHelper inherit from the contract ClaimableGas

Lack of integration with Thruster Uniswap V2 Fork

Description

The ThrusterSwap integrate wtih Thruster Uniswap V3 fork

https://docs.thruster.finance/informational/contracts,

but for contract

ThrusterRouter (0.30%)

0x98994a9A7a2570367554589189dC9772241650f6

https://blastscan.io/address/0x98994a9A7a2570367554589189dC9772241650f6

which is a Uniswap V2 Fork, there are still active user using this router.

Recommendation

recommend implement another swapper to integrate with thruster Uniswap V2 Fork router.

R1 – Thorough testing for MarginTradingHook.sol

A thorough integration testing should be performed espeically the reduce position and order filling flow

R2 – Consider use multicall to simplify the token conversion flow.

_getRealUnderlyingToken is used to convert wrapped WETH / USDB to WETH / USDB

_isRebasingWrapper is check if the token is lending collateral

when charge user transfer, token should be converted to WETH / USDB by calling _getRealUnderlyingToken

when transfer fund to user, the token should be converted to WETH / USDB by calling _getRealUnderlyingToken

this conversion flow is very brain power consuming and even leads to error

I highly recommand that the margin trading hook use

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Multicall.sol

then implement a function to let wrap the WNATIVE to Wrapped WETH

and wrap the USDB to wrapped USDB in the margin trading hook

in this way in multicall, user can do the following in single transaction.

1. wrap USDB to wrapped USDB
2. add collateral / remove collateral

or

1. wrapped USDB to wrapped USDB
2. fill order

In this way, we do not need to worry about the conversion.