hexens × STAKEWISE

SEPT.24

SECURITY REVIEW REPORT FOR STAKEWISE

CONTENTS

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered all smart contracts of StakeWise V3 Periphery. These contracts can be used to more easily setup leveraged positions in StakeWise V3 Core and corresponding Aave pools.

Our security assessment was a full review of the smart contracts, spanning a total of 1 week.

During our audit, we have identified 1 critical severity vulnerability. The vulnerability could have allowed an attacker to perform actions on other users' positions, eventually leading to principal asset loss.

Finally, all of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/stakewise/v3-periphery/tree/53f84db29e430a0a78e4e0051b89e6fd08095c1c

The issues described in this report were fixed in the following commit:

https://github.com/stakewise/v3-periphery/tree/3a8325e956ead1f1156b4d4c9099e0d98c70ac0c

AUDITING DETAILS

STARTED
09.09.2024

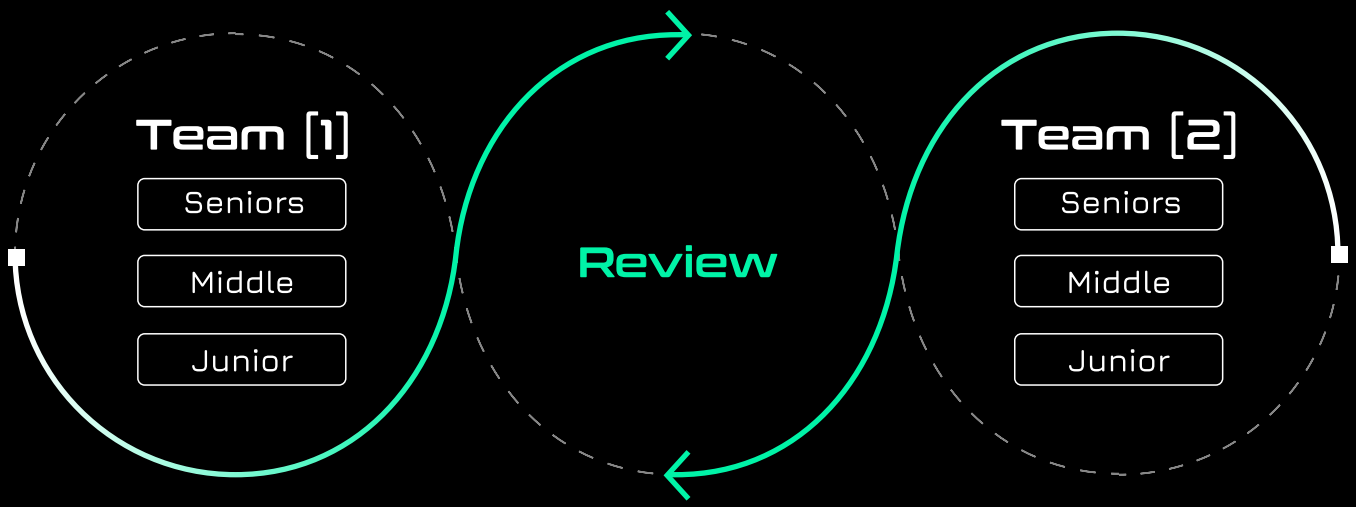
DELIVERED
19.09.2024

Review Led by

KASPER ZWIJSEN
Head of Audits | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

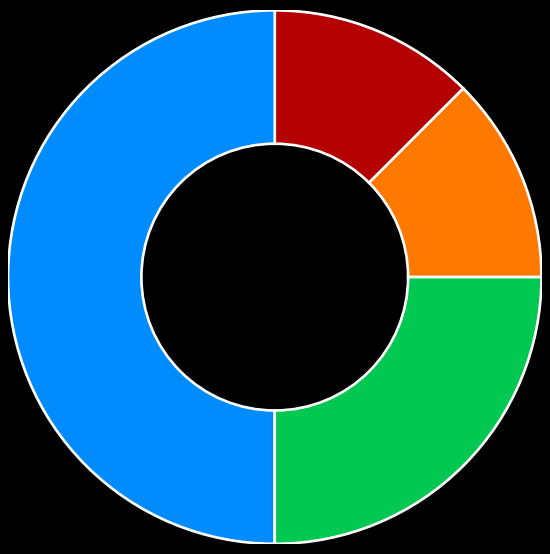
ISSUE SYMBOLIC CODES

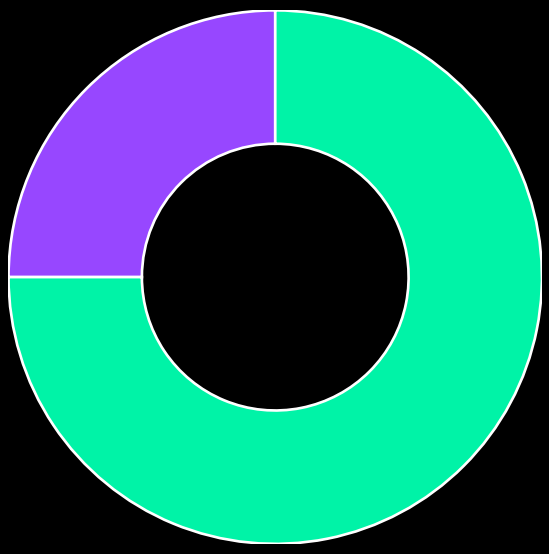
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	0
Medium	1
Low	2
Informational	4

Total: 8





WEAKNESSES

This section contains the list of discovered weaknesses.

STAKE-6

UNPROTECTED FLASH LOAN CALLBACK CAN BE ABUSED TO MANIPULATE/CLAIM OTHER USERS' POSITIONS

SEVERITY: Critical

PATH:

LeverageStrategy.sol:receiveFlashLoan#L304-L333

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The Leverage Strategy contract uses flash loans from the osToken Flash Loan module to flash loan osTokens and perform actions, such as deposit, claim and rescue. Each of these functions performs the flash loan with specific parameters in the **userData** field.

The callback function **receiveFlashLoan** will be called by the Flash Loan module and while the function correctly checks that the caller is the module, it does not check whether the flash loan was initiated by the contract itself.

So it becomes possible for an attacker to initiate a flash loan with the Leverage Strategy as receiver. The **userData** can then be arbitrarily chosen by the attacker, which depending on the action can be the proxy address, vault address or exit position ID.

The attacker can make use of the specific Flash Loan callback functions to manipulate any arbitrary proxy and change their leveraged positions or make a claim and cause a deadlock in exiting because **isStrategyProxyExiting** was never cleared.

```
    function receiveFlashLoan(uint256 osTokenShares, bytes memory userData)
external {
        // validate sender
        if (msg.sender != address(_osTokenFlashLoans)) {
            revert Errors.AccessDenied();
        }

        // decode userData action
        (FlashloanAction flashloanType) = abi.decode(userData,
(FlashloanAction));
        if (flashloanType == FlashloanAction.Deposit) {
            // process deposit flashloan
            (, address vault, address proxy) = abi.decode(userData,
(FlashloanAction, address, address));
            _processDepositFlashloan(vault, proxy, osTokenShares);
        } else if (flashloanType == FlashloanAction.ClaimExitedAssets) {
            // process claim exited assets flashloan
            (, address vault, address proxy, uint256 exitPositionTicket) =
                abi.decode(userData, (FlashloanAction, address, address,
uint256));
            _processClaimFlashloan(vault, proxy, exitPositionTicket,
osTokenShares);
        } else if (flashloanType == FlashloanAction.RescueVaultAssets) {
            // process vault assets rescue flashloan
            (, address vault, address proxy, uint256 exitPositionTicket) =
                abi.decode(userData, (FlashloanAction, address, address,
uint256));
            _processVaultAssetsRescueFlashloan(vault, proxy,
exitPositionTicket, osTokenShares);
        } else if (flashloanType == FlashloanAction.RescueLendingAssets) {
```

```
            // process lending assets rescue flashloan
            (, address proxy, uint256 assets) = abi.decode(userData,
(FlashloanAction, address, uint256));
            _processLendingAssetsRescueFlashloan(proxy, assets,
osTokenShares);
        } else {
            revert InvalidFlashloanAction();
        }
    }
```

The **receiveFlashLoan** callback function should check whether the contract itself initiated the flash loan. This can be done using a storage variable that is set to true before the initiating function calls **flashLoan** and set to false after.

For example:

```
function deposit(address vault, uint256 osTokenShares) external {
  [..]
  if (performingFlashLoan) {
    revert Errors.Reentrant();
  }
  performingFlashLoan = true;
  // execute flashloan
  _osTokenFlashLoans.flashLoan(
    address(this), flashloanOsTokenShares,
abi.encode(FlashloanAction.Deposit, vault, proxy)
  );
  performingFlashLoan = false;
  [..]
}

function receiveFlashLoan(uint256 osTokenShares, bytes memory userData)
external {
  if (!performingFlashLoan) {
    revert Errors.AccessDenied();
  }
  [..]
}
```

LACK OF SLIPPAGE PROTECTION IN DEPOSIT FUNCTION

SEVERITY: Medium

PATH:

src/leverage/LeverageStrategy.sol::deposit()#L125-L182

REMEDIATION:

Allow users to specify a minimum number of osTokenShares they are willing to accept after leverage calculations.

STATUS: Acknowledged, see commentary

DESCRIPTION:

The **deposit()** function in the **LeverageStrategy.sol** contract executes several key operations involving token transfers, asset borrowing, and minting, including the potential use of flash loans. However, it lacks slippage protection, meaning that if token prices fluctuate between the time the transaction is submitted and executed, users could receive fewer tokens or face adverse outcomes without the ability to revert or mitigate the loss.

In particular, the function lacks checks to ensure that the actual number of **osTokenShares** minted matches expectations after asset borrowing.

Example:

A user submits a transaction to deposit 100 **osTokenShares**, expecting to leverage them to borrow additional assets. Between the time the transaction is broadcasted and mined, market conditions change (e.g., a large price movement), leading to less favorable borrowing conditions. Since there is no slippage check, the user could end up with fewer minted **osTokenShares** than expected, negatively impacting their strategy.

```
    function deposit(address vault, uint256 osTokenShares) external {
        if (osTokenShares == 0) revert Errors.InvalidShares();

        // fetch strategy proxy
        (address proxy,) = _getOrCreateStrategyProxy(vault, msg.sender);
        if (isStrategyProxyExiting[proxy]) revert
Errors.ExitRequestNotProcessed();

        // transfer osToken shares from user to the proxy
        IStrategyProxy(proxy).execute(
            address(_osToken),
abi.encodeWithSelector(_osToken.transferFrom.selector, msg.sender, proxy,
osTokenShares)
        );

        // fetch vault state and lending protocol state
        (uint256 stakedAssets, uint256 mintedOsTokenShares) =
_getVaultState(vault, proxy);
        (uint256 borrowedAssets, uint256 suppliedOsTokenShares) =
_getBorrowState(proxy);

        // check whether any of the positions exist
        uint256 leverageOsTokenShares = osTokenShares;
        if (stakedAssets != 0 || mintedOsTokenShares != 0 || borrowedAssets
!= 0 || suppliedOsTokenShares != 0) {
            // supply osToken shares to the lending protocol
            _supplyOsTokenShares(proxy, osTokenShares);
            suppliedOsTokenShares += osTokenShares;

            // borrow max amount of assets from the lending protocol
            uint256 maxBorrowAssets =

Math.mulDiv(_osTokenVaultController.convertToAssets(suppliedOsTokenShares),
_getBorrowLtv(), _wad);
            if (borrowedAssets >= maxBorrowAssets) {
                // nothing to borrow
                emit Deposited(vault, msg.sender, osTokenShares, 0);
                return;
            }
```

```
            uint256 assetsToBorrow;
            unchecked {
                // cannot underflow because maxBorrowAssets > borrowedAssets
                assetsToBorrow = maxBorrowAssets - borrowedAssets;
            }
            _borrowAssets(proxy, assetsToBorrow);

            // mint max possible osToken shares
            leverageOsTokenShares = _mintOsTokenShares(vault, proxy,
assetsToBorrow, type(uint256).max);
            if (leverageOsTokenShares == 0) {
                // no osToken shares to leverage
                emit Deposited(vault, msg.sender, osTokenShares, 0);
                return;
            }
        }

        // calculate flash loaned osToken shares
        uint256 flashloanOsTokenShares = _getFlashloanOsTokenShares(vault,
leverageOsTokenShares);

        // execute flashloan
        _osTokenFlashLoans.flashLoan(
            address(this), flashloanOsTokenShares,
abi.encode(FlashloanAction.Deposit, vault, proxy)
        );

        // emit event
        emit Deposited(vault, msg.sender, osTokenShares,
flashloanOsTokenShares);
    }
```

Commentary from the client:

“ - Both Aave's osETH/ETH oracles and StakeWise vaults don't rely on the secondary market, so price fluctuations won't affect the amount you can borrow. Also, calculating the exact amount you can leverage is tricky, as it depends heavily on your current position's LTVs.”

PERMIT FUNCTION CAN BE FRONTRUNNED

SEVERITY: Low

PATH:

src/leverage/LeverageStrategy.sol::permit()#L114-L122

REMEDIATION:

Consider adding try/cath, or if block, so if the permit() is already called, just check the allowance of msg.sender and skip the call to pemit().

STATUS: Fixed

DESCRIPTION:

The **permit()** function within the **LeverageStrategy.sol** contract is designed to allow token approvals through signatures, leveraging the ERC-2612 standard. However, the function is vulnerable to front-running attacks. A malicious actor could observe e.g. Alice's signature in the mempool and use it to execute a transaction on her behalf before Alice's original transaction is mined. Since signatures are single-use and tied to nonces, once the signature is exploited, Alice's original transaction would fail, resulting in wasted gas and the function DOS.

```
    function permit(address vault, uint256 osTokenShares, uint256 deadline, uint8 v,
bytes32 r, bytes32 s) external {
        (address proxy,) = _getOrCreateStrategyProxy(vault, msg.sender);
        IStrategyProxy(proxy).execute(
            address(_osToken),
            abi.encodeWithSelector(
                  IERC20Permit(address(_osToken)).permit.selector, msg.sender, proxy,
osTokenShares, deadline, v, r, s
            )
        );
    }
```

INCORRECT ROUNDING IN MINTSHARES AND CUMULATIVEFEEPERSHARE CALCULATIONS

SEVERITY: Low

PATH:

lib/v3-core/contracts/tokens/OsTokenVaultController.sol::mintShares()#L103-L129

lib/v3-core/contracts/tokens/OsTokenVaultController.sol::cumulativeFeePerShare()#L200-L228

REMEDIATION:

In the mentioned functions, use Math.Rounding.Ceil for the convertToShares calculations.

STATUS: Acknowledged, see commentary

DESCRIPTION:

In the **OsTokenVaultController.sol** contract, the functions **mintShares()** and **cumulativeFeePerShare()** use rounding mode **Math.Rounding.Floor**, which can lead to incorrect asset-to-share conversions. Specifically, this rounding method allows the user to underpay during share minting and fee calculations, which results in values that consistently favor the user over the protocol.

The **mintShares()** function calculates the number of assets needed to mint a certain number of shares by using the **convertToAssets()** method, which in turn calls **_convertToAssets()** with **Math.Rounding.Floor**:

```
  function convertToAssets(uint256 shares) public view override returns
(uint256 assets) {
    return _convertToAssets(shares, _totalShares, totalAssets(),
Math.Rounding.Floor);
  }
```

allowing the user to pay less than the required amount of assets. This discrepancy might accumulate over multiple transactions, resulting in a protocol loss.
The fee calculation also uses the **Math.Rounding.Floor** method, potentially under-crediting the treasury with fewer assets than it is entitled to. Over time, this rounding method results in the protocol losing part of its fee revenue.

Example:

If a user mints 100 shares, and the actual cost should be 91.5 assets, the rounding down behavior causes the user to pay only 91 assets. Over time, this discrepancy can accumulate, leading to a significant shortfall in the protocol's asset pool. Similarly, the protocol's treasury fee is rounded down, reducing the fees collected by the protocol.

```
  function mintShares(address receiver, uint256 shares) external override
returns (uint256 assets) {
    if (
      !IVaultsRegistry(_registry).vaults(msg.sender) ||
      !
IVaultsRegistry(_registry).vaultImpls(IVaultVersion(msg.sender).implementati
on())
    ) {
      revert Errors.AccessDenied();
    }
    if (receiver == address(0)) revert Errors.ZeroAddress();
    if (shares == 0) revert Errors.InvalidShares();

    // pull accumulated rewards
    updateState();

    // calculate amount of assets to mint
    assets = convertToAssets(shares);

    uint256 totalAssetsAfter = _totalAssets + assets;
    if (totalAssetsAfter > capacity) revert Errors.CapacityExceeded();

    // update counters
    _totalShares += SafeCast.toUint128(shares);
    _totalAssets = SafeCast.toUint128(totalAssetsAfter);

    // mint shares
    IOsToken(_osToken).mint(receiver, shares);
    emit Mint(msg.sender, receiver, assets, shares);
  }
```

```
  function cumulativeFeePerShare() external view override returns (uint256)
{
    // SLOAD to memory
    uint256 currCumulativeFeePerShare = _cumulativeFeePerShare;

    // calculate rewards
    uint256 profitAccrued = _unclaimedAssets();
    if (profitAccrued == 0) return currCumulativeFeePerShare;

    // calculate treasury assets
    uint256 treasuryAssets = Math.mulDiv(profitAccrued, feePercent,
_maxFeePercent);
    if (treasuryAssets == 0) return currCumulativeFeePerShare;

    // SLOAD to memory
    uint256 totalShares_ = _totalShares;

    // calculate treasury shares
    uint256 treasuryShares;
    unchecked {
      treasuryShares = _convertToShares(
        treasuryAssets,
        totalShares_,
        // cannot underflow because profitAccrued >= treasuryAssets
        _totalAssets + profitAccrued - treasuryAssets,
        Math.Rounding.Floor
      );
    }

    return currCumulativeFeePerShare + Math.mulDiv(treasuryShares, _wad,
totalShares_);
  }
```

Commentary from the client:

“ - In OsTokenVaultController, rounding up could cause the user's LTV in the vault to exceed the maximum LTV if they attempt to mint the maximum amount of osTokens. That's why we always round down.”

UNUSED IMPORT

SEVERITY: Informational

PATH:

src/StrategiesRegistry.sol#L8

REMEDIATION:

Remove the redundant import.

STATUS: Fixed

DESCRIPTION:

The **StrategiesRegistry.sol** contract imported the **IStrategy.sol** on line 8 but has never used it.

```
import {IStrategy} from './interfaces/IStrategy.sol';
```

REDUNDANT VARIABLES

SEVERITY: Informational

PATH:

src/StrategiesRegistry.sol#L16
src/leverage/AaveLeverageStrategy.sol#L20

REMEDIATION:

Remove unused variables.

STATUS: Fixed

DESCRIPTION:

The **_maxPercent** variable is declared in the **StrategiesRegistry.sol** contract but never used.
The **_wad** constant in the **AaveLeverageStrategy.sol** is redundant because it is never used in any of the functions or calculations within that contract.

```
    uint256 private constant _maxPercent = 1e18;
```

```
    uint256 private constant _wad = 1e18;
```

UNUSED ERROR

SEVERITY: Informational

PATH:

libraries/Errors.sol#L53

REMEDIATION:

Remove unused error.

STATUS: Fixed

DESCRIPTION:

In the **Errors.sol** contract the error **EigenInvalidWithdrawal()** is declared on line 53 but never used.

```
error EigenInvalidWithdrawal();
```

FLASHLOANOSTOKENSHARES SHOULD BE VALIDATED AS NON-ZERO BEFORE THE FLASHLOAN

SEVERITY: Informational

PATH:

src/leverage/LeverageStrategy.sol#L165-L178

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In the **deposit()** function, there's a check that returns without executing the flashloan if **leverageOsTokenShares == 0**. However, the **flashloanOsTokenShares** variable, calculated by **_getFlashloanOsTokenShares()**, may still be 0 even when **leverageOsTokenShares** > 0

```
uint256 flashloanOsTokenShares = _getFlashloanOsTokenShares(vault,
leverageOsTokenShares);
```

This can cause reverts because of borrowing zero during deposit when **leverageOsTokenShares** is very small.
Therefore, to avoid the zero flashloan, it should check **flashloanOsTokenShares** instead of **leverageOsTokenShares**.

```
function deposit(address vault, uint256 osTokenShares) external {
    if (osTokenShares == 0) revert Errors.InvalidShares();

    // fetch strategy proxy
    (address proxy,) = _getOrCreateStrategyProxy(vault, msg.sender);
    if (isStrategyProxyExiting[proxy]) revert
Errors.ExitRequestNotProcessed();

    // transfer osToken shares from user to the proxy
    IStrategyProxy(proxy).execute(
        address(_osToken),
abi.encodeWithSelector(_osToken.transferFrom.selector, msg.sender, proxy,
osTokenShares)
    );

    // fetch vault state and lending protocol state
    (uint256 stakedAssets, uint256 mintedOsTokenShares) =
_getVaultState(vault, proxy);
    (uint256 borrowedAssets, uint256 suppliedOsTokenShares) =
_getBorrowState(proxy);

    // check whether any of the positions exist
    uint256 leverageOsTokenShares = osTokenShares;
    if (stakedAssets != 0 || mintedOsTokenShares != 0 || borrowedAssets != 0
|| suppliedOsTokenShares != 0) {
        // supply osToken shares to the lending protocol
        _supplyOsTokenShares(proxy, osTokenShares);
        suppliedOsTokenShares += osTokenShares;

        // borrow max amount of assets from the lending protocol
        uint256 maxBorrowAssets =

Math.mulDiv(_osTokenVaultController.convertToAssets(suppliedOsTokenShares),
_getBorrowLtv(), _wad);
        if (borrowedAssets >= maxBorrowAssets) {
            // nothing to borrow
            emit Deposited(vault, msg.sender, osTokenShares, 0);
            return;
        }
        uint256 assetsToBorrow;
        unchecked {
```

```
            // cannot underflow because maxBorrowAssets > borrowedAssets
            assetsToBorrow = maxBorrowAssets - borrowedAssets;
        }
        _borrowAssets(proxy, assetsToBorrow);

        // mint max possible osToken shares
        leverageOsTokenShares = _mintOsTokenShares(vault, proxy,
assetsToBorrow, type(uint256).max);
        if (leverageOsTokenShares == 0) {
            // no osToken shares to leverage
            emit Deposited(vault, msg.sender, osTokenShares, 0);
            return;
        }
    }

    // calculate flash loaned osToken shares
    uint256 flashloanOsTokenShares = _getFlashloanOsTokenShares(vault,
leverageOsTokenShares);

    // execute flashloan
    _osTokenFlashLoans.flashLoan(
        address(this), flashloanOsTokenShares,
abi.encode(FlashloanAction.Deposit, vault, proxy)
    );

    // emit event
    emit Deposited(vault, msg.sender, osTokenShares,
flashloanOsTokenShares);
}
```

Replace the check for **leverageOsTokenShares == 0** with **flashloanOsTokenShares**, as follows:

```
        // mint max possible osToken shares
        leverageOsTokenShares = _mintOsTokenShares(vault, proxy,
assetsToBorrow, type(uint256).max);
    }

    // calculate flash loaned osToken shares
    uint256 flashloanOsTokenShares = _getFlashloanOsTokenShares(vault,
leverageOsTokenShares);

    if (flashloanOsTokenShares == 0) {
        // no osToken shares to leverage
        emit Deposited(vault, msg.sender, osTokenShares, 0);
        return;
    }
    // execute flashloan
    _osTokenFlashLoans.flashLoan(
        address(this), flashloanOsTokenShares,
abi.encode(FlashloanAction.Deposit, vault, proxy)
    );
```


hexens
STAKEWISE