Zellic

Nukem Loans

Smart Contract Security Assessment

October 30, 2023

Prepared for:

Kashaf Bashir

Nukem Loans

Prepared by:

Ayaz Mammadov and Vlad Toie

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Nukem Loans from September 25th to October 12th, 2023. During this engagement, Zellic reviewed Nukem Loans's code for security vulnerabilities, design issues, and general weaknesses in security posture.

The codebase showcases an extensive level of customization of libraries like SafeERC20 and ERC4626, which is quite unique in the ecosystem. While this reflects innovative thinking, it's important to consider the aspects of long-term maintainability and scalability, as even minor alterations could potentially impact the protocol's security. Diverging from standard industry practices might also lead to unforeseen behaviors and security challenges, which could be complex to identify and address. We suggest a thorough review of these customizations with a view to align more closely with industry standards, as this could enhance the safety and scalability of the project.

Regarding the design choice to use DEX AMMs directly for valuation, rather than relying on oracles or integrating smoothing mechanisms like geometric/time-based algorithms, it's crucial to approach any future code enhancements with caution. This is especially important in order to prevent potential severe vulnerabilities, like flashloan or price manipulation attacks. Careful consideration and strategic planning in code expansion will be key to maintaining the integrity and overall security of the system.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- How does Nukem Loans issue debt tokens? Is there any way to mint debt tokens to someone else? Is debt transferable?
- What are the collateral margins for borrowers and lenders? How are they calculated? Are they high enough to prevent liquidations or protect lenders?
- What are the main components of the Nukem Loans protocol? How do they interact with each other? What is the flow of funds?
- How are the signatures validated? Is there any way to replay signatures across multiple transactions? What if the signature is replayed on a different chain?
- Is there any way to borrow without collateral? Is there any way to borrow more than the collateral value? What if the prices are manipulated?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

A thorough assessment of the heavily customized libraries could not be conducted as our focus was directed towards other integral parts of the code. The unconventional modifications to these libraries, which significantly deviate from industry standards, warrant a deeper examination to understand the implications fully. Moreover, upgradability was not examined in detail, as no Hardhat or Foundry deployments were provided. We recommend further analysis of these areas to ensure project safety and scalability.

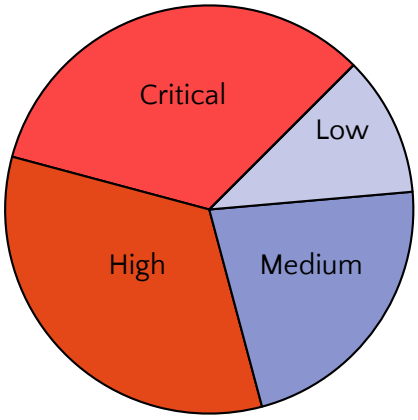
1.3 Results

During our assessment on the scoped Nukem Loans contracts, we discovered nine findings. Three critical issues were found. Three were of high impact, two were of medium impact, and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Nukem Loans's benefit in the Discussion section (4).

Breakdown of Finding Impacts

Impact Level	Count
Critical	3
High	3
Medium	2
Low	1
Informational	0



2 Introduction

2.1 About Nukem Loans

Nukem Loans is a protocol revolving around a set of smart contracts that enhance meme communities by creating their own token economy through establishment of permissionless lending markets created by lenders. At first, the lending market will be deployed manually by the Nukem team, eventually allowing for permissionless market deployment when protocol will be stable enough.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

Nukem Loans Contracts

Repository	https://github.com/K3mistry/nukem
Version	nukem: `331b0f08364b7c2fb6c091cea948bb1505ff93da`

Programs	AttributesLibrary.sol ERC20Base.sol ProxyManager.sol SafeERC20.sol SaferERC20.sol Extensible.sol Auctions.sol Collateral.sol Credit.sol Debt.sol LendingStrategy.sol Market.sol MarketComponent.sol Vault.sol vsCredit.sol vsToken.sol AbstractSwapper.sol UniswapV2Swapper.sol EIP712.sol ERC20Permit.sol ERC20Upgradable.sol EnFi20.sol EnFi4626.sol Integrity.sol Ownable2.sol ProxyOwnable.sol Roles.sol Upgradable.sol
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of four person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Ayaz Mammadov, Engineer
ayaz@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 25, 2023	Kick-off call
September 25, 2023	Start of primary review period
October 12, 2023	End of primary review period

3 Detailed Findings

3.1 [FIXED] Collateral inflation attack allows for theft of funds

- **Target**: Collateral
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The maximum withdraw amount for a user is calculated as the swappable value of their collateral minus their debt.

```
  function maxWithdraw(
      address account
  ) public view virtual override returns (uint256) {
...
      IDebt debt = _market.debt();
      uint256 account_debt_assets = debt.assetsOf(account);
      if (account_debt_assets == 0) return asset_balance;

      ILendingStrategy strategy = _market.strategy();
      uint256 debt_value = _market.swapper().valueOf(
          debt.asset(),
          account_debt_assets
      );
      uint256 borrowable_collateral = (asset_balance *
          strategy.maxCollateralizationRatio()) / strategy.precision();
      return
          (borrowable_collateral > debt_value)
              ? borrowable_collateral - debt_value
              : 0;
  }
```

Impact

It is profitable for a malicious attacker to manipulate the underlying pool in order to inflate the value of their collateral. This allows them to withdraw most of the collateral while keeping the debt that is worth more than the original collateral. As a result of

this, protocol funds can be stolen.

Recommendations

Only allow withdrawal of collateral once all the debt has been paid off.

Remediation

The Nukem team has fixed this issue by removing the partial withdraw mechanism in commit 2e3ecbe0.

3.2 [FIXED] Free liquidation

- **Target**: Auctions
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

A check that ensures the price of the bid is less than the current price (including the discounts) was omitted, allowing any user to submit a bid used to determine whether the liquidation was hard or soft.

Impact

In the code below, the `currentPrice` check happens only in the soft liquidation branch. However, it should be checked before the branch to decide the type of liquidation taken.

```
function _execute(
     address account,
     Auction memory auction,
     uint256 bid
 ) internal {
     require(
         isAuctionable(auction.market, auction.debtor),
         "not.auctionable"
     );
     IMarket market = IMarket(auction.market);

     uint256 swappable
    = market.collateral().swappableValue(auction.debtor);

     if (swappable > bid) {
         /* hard liquidation */
...
     } else {
         /* soft liquidation */
         require(
             currentPrice(
                 auction.market,
                 auction.debtor,
                 auction.starts_at,
```

```
                auction.ends_at
            ) ≤ bid,
            "bid.price"
        );
...
    }
```

Impact

Users can supply a bid of 1 as soon as the auction is open to get the debtor's collateral for free.

Recommendations

Add the `currentPrice` check before the branch.

Remediation

The Nukem team has fixed this issue by moving the `currentPrice` check before the branch in commit 7f86bbba.

3.3 [FIXED] Interest can be stolen by staking for one block before rebases

- **Target**: Credit, Debt
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

By being a creditor for the last possible block before a rebase, and then redeeming their stake in the next block, a user can accrue the vast majority of interest without contributing or exposing themselves to market risk.

Impact

A user can stake a two-block credit with a very large amount of capital to get almost all the interest without participating in the protocol mechanic.

Recommendations

Add a minimum stake period.

Remediation

The Nukem team has fixed this issue by introducing a `CreditLockPayment` mechanism, default configured to 1 day, which was added in commit bec528a9. This ensures that creditors stay locked in their positions.

3.4 Centralized pricing arbitrage

- **Target**: Market
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

As the protocol uses a combination of automated market maker (AMM) oracles and centralized prices provided in signatures, an arbitrage opportunity may exist that will result in the drainage of pools.

```
    function borrow( ... ) public virtual returns (uint256 debt_shares) {
        require(
            checkSignature(authorizer, market, price, deadline, v, r, s),
            "invalid.signature"
        );

        require(authorizer == _owner, "authorizer");
        require(market == address(this), "market");
        require(deadline ≥ block.timestamp, "expired");

        // protect against price manipulation
        require(validatePrice(price, amount), "invalid.price");
        debt_shares = _debt.withdraw(amount, receiver, owner);
    }
}
```

The signatures have a deadline of five minutes and can be constantly polled from off-chain components. A malicious user could continuously poll these signatures, waiting for the increase in price of those tokens, and exploit the price difference between the signed price five minutes ago and the current market price.

If the price goes up substantially relative to the price in the signature five minutes ago, there may be an edge case where a user can actually borrow more capital than the price of their collateral.

For markets that have high collateralization ratio, this can be especially risky because in addition to that, AMM oracles can be manipulated to a certain extent (limited by `validatePrice`) in the protocol.

Impact

For some market configurations that have have a 90% collateralization rate and 95% liquidation rate, the delta allowed for the underlying AMM is ~5%. A malicious user constantly polling for signatures would have to wait for one instance of such a price movement in the market in order to execute such an arbitrage.

Recommendations

With the current architecture, such attacks will always be possible in case of drastic price movements in the five-minute period. However, the goal here is to make this unlikely, by reducing either maximum collateralization rates or reducing the deadline time period such that the required price movement is almost impossible in the time period.

Remediation

The Nukem team agreed with this finding and decided to lower collateralization rates in existing market configurations. The team plans to add additional mitigations in a future update.

3.5 [FIXED] Slippage is set to zero during swap

- **Target**: Credit, Collateral
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

Multiple slippage checks are set to zero when performing a token swap.

Impact

This is hazardous because it could allow users to trade at 100% slippage rates.

```
swapper.swap(asset_, address(this), receiver, amount, 0);
```

Recommendations

We recommend passing a nonzero slippage parameter for the `swap` function and making sure that the user is aware of the slippage rate.

Remediation

This issue was fixed by enforcing a 2% maximum slippage from the reference value of the swap provided by the signed reserves, added in commit 571dbc66.

3.6 [FIXED] EIP-712 replayable signature in case of fork

- **Target**: EIP712
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: High

Description

EIP-712 is a standard for the hashing and signing of typed, structured data. The standard code does not allow replaying signatures in case of a fork by default, as it rebuilds the domain separator in case the cached address of the contract and the cached chain ID differ to current values.

However, in the case of the project's implementation, the aforementioned checks are removed. The domain separator will not be updated in case of a fork, and the signature can be replayed.

```
function _domainSeparatorV4() internal view returns (bytes32) {
    if (address(this) == _cachedThis && block.chainid == _cachedChainId)
        {
        return _cachedDomainSeparator;
    } else {
        return _buildDomainSeparator();
    }
}
```

Impact

Even though the signatures can be replayed, the impact of this issue is relatively limited due to time constraints, mainly affecting the ERC20Permit implementation, which has direct access to user funds. The other contracts that use EIP-712 for verifying signatures do not allow performing actions on behalf of other users, so the impact there is limited to a user's own actions.

Recommendations

We recommend using the default implementation of the EIP-712 standard to remove the possibility of replaying signatures in case of a fork.

Remediation

The Nukem team remediated this issue in commit 46abe2cd by always rebuilding the domain separator.

3.7 [FIXED] Assure debtors are auctionable

- **Target**: Auctions
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

The Auctions contract handles the liquidation of debtors. The `groupedLiquidation` function is used by the owner of the contract to liquidate multiple debtors at once. However, it does not check that the debtors are auctionable, which means that if the owner of the contract by mistake passes a nonauctionable debtor, the liquidation will still be performed, and the debtors will be left in an inconsistent state.

```
function groupedLiquidation(
    address market,
    address[] memory debtors
) external returns (uint256 liquidated, uint256 tip) {

    require(
        (_msgSender() == owner()) || // owner
            attributes[_msgSender()].has(Role.EXECUTE_AUCTION),
        "authorizer"
    );
    (liquidated, tip) = IMarket(market).credit().liquidate(
        debtors,
        _msgSender()
    );
    emit GroupedLiquidations(
        market,
        block.timestamp,
        debtors,
        liquidated,
        tip,
        _msgSender()
    );
}
```

Impact

Since this function is a privileged function, the impact is limited to the owner of the contract. However, it can still lead to an inconsistent state of the debtors, which can lead to further unexpected behavior.

Recommendations

We recommend individually checking that each debtor is auctionable before performing the grouped liquidation. For example,

```
function groupedLiquidation(
    address market,
    address[] memory debtors
) external returns (uint256 liquidated, uint256 tip) {

    for(uint256 i = 0; i < debtors.length; i++) {
        require(isAuctionable(market, debtors[i]), "not auctionable");
    }
    // ...
}
```

Remediation

The Nukem team fixed this finding in commit 571dbc66 by ensuring every debtor is auctionable.

3.8 [FIXED] User's max collateralization is limited by the size of the market

- **Target**: Collateral
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Liquidations in the protocol affect the underlying AMM, which may cause more liquidations. This is a cascading effect. To counteract this, the worth of user collateral is calculated conservatively (the swap price is calculated as if one liquidation would set off all the liquidations). This works, however it devalues the user collateral, and this effect becomes worse as the market grows relative to the underlying pool.

Impact

Users risk more collateral for smaller loans and may be able to borrow less than expected.

Recommendations

Re-architect the conservative value calculations to only account for positions that a liquidation would actually put at risk.

Remediation

The Nukem team has acknowledged this issue and will put it as an object in their road map once markets start becoming large relative to their underlying pools.

3.9 ERC-4626 vault inflation

- **Target**: EnFi4626
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

Contracts inheriting the EnFi4626 contract are vulnerable to the ERC-4626 inflation attack.

In accordance with ERC-4626, EnFi4626 is a vault that holds assets on behalf of its users. Whenever a user deposits assets, it issues to the user a number of shares such that the proportion of the user's shares over the total issued shares is equal to the user's assets over the total withdrawable assets. This allows assets gained by EnFi4626 to increase the value of every user's shares in a proportional way.

ERC-4626 vaults are susceptible to inflation attacks; an attacker can "donate" funds to the vault without depositing them, increasing the value of a share unexpectedly. In some circumstances, including when an unsuspecting user is the first depositor, an attacker can make back more than they donated, stealing value from the first depositor.

Impact

The attack works as follows:

1. The benign user submits a deposit transaction to the vault, depositing 1,000 coins.

2. Before the deposit transaction is mined, an attacker front-runs it with an earlier transaction, which deposits 0.000001 coins and then donates 1,000 coins to the vault. After this, the attacker has one share and the vault has 1,000.000001 coins.

3. Then, the user's deposit transaction is mined. After the user's deposit, the vault has 2,000.000001 coins, of which 1,000 was just deposited by the user. Since shares are now worth 1,000.0000005 coins after the attacker's front-run transactions, the user is given less than one share, which the vault rounds to zero.

4. Finally, the attacker, with their one share that represents all the issued shares, withdraws all of the assets, stealing the benign user's coins.

Recommendations

Please see Github issue #3706 in OpenZeppelin for discussion about how to mitigate this vulnerability.

In short, the first deposit to a new vault could be made by a trusted admin during vault construction to ensure that `totalSupply` remains greater than zero. However, this remediation has the drawback that this deposit is essentially locked, and it needs to be high enough relative to the first few legitimate deposits such that front-running them is unprofitable. Even if this prevents the attack from being profitable, an attacker can still grief legitimate deposits with donations, making the user gain less shares than they should have gained.

Another solution is to track `totalAssets` internally by recording the assets gained through its Market positions and not increasing it when donations occur. This makes the attack significantly harder, since the attacker would have to donate funds by affecting price feeds for the underlying assets rather than just sending tokens to the vault.

Alternatively, an ERC-4626 router can be used. This, however, will consume more gas for any of the performed operations. Additionally, it introduces a new potential attack vector, as the router will have to be trusted to perform the operations correctly.

Remediation

The Nukem Loans team acknowledged this issue and have decided to supply seed liquidity necessary to prevent the attack.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Initializer called in constructor

The initializer function is typically called in an upgradable contract as a way of initializing the storage variables, as the constructor itself will not be called. However, in the case of Ownable2.sol, the initializer is called in the constructor, which does not follow the intended pattern for upgradable contracts.

```
constructor() {
    Ownable_init();
}

// @audit-issue this is not really an initializer
function Ownable_init() internal {
    _transferOwnership(_msgSender());
}
```

We recommend removing the call to the initializer in the constructor and instead calling it in an external `initialize` function, as is the standard for upgradable contracts. This should be applied project wide, as the same erroneous pattern is used in other contracts as well.

4.2 Potential storage-collision issue

The ProxyManager contract manages deploying and upgrading clones of master contracts. In its `setImplementation` function, it handles the actual upgrade of the clones. However, it does not check if the new implementation is a valid upgrade of the clones. This means that if by mistake the owner of the contract sets a new implementation that is not compatible with the clones, the clones will be left in an inconsistent state, possibly having their storage corrupted by the newer, totally different implementation:

```
function setImplementation(
    bytes32 implementation_id,
    address implementation,
    bytes calldata upgrade_data,
    bool callUpdate
) public onlyOwner {
    require(implementation.code.length > 0, "#075051C6");
    implementations[implementation_id] = implementation;

    // for each item in instances, change the implementation
    for (uint i = 0; i < instances[implementation_id].length; i++) {

        // @audit no check on whether the new implementation is a direct
    upgrade of the old one, or a different contract altogether.


    IUpgradable(instances[implementation_id][i]).updateImplementation(
            implementation
        );
        if (callUpdate)
            IUpgradable(instances[implementation_id][i]).upgrade(
                upgrade_data
            );
    }
}
```

Although this issue cannot be remediated in code, we recommend simulating the contract upgrade by using the OpenZeppelin plug-ins, which check and warn in case of storage collisions. More information can be found here.

4.3 SaferERC20 additional checks

The SaferERC20 contract handles sending both ERC-20 and native tokens. It does that through an if-else statement, where it checks if the token that is being sent is the native token, and if so, it checks that the value sent is enough.

However, it does not check that `msg.value` is zero in case the token is not the native token. This means that if a user intends to perform an operation where they are using a non-native token and by mistake send some native tokens (i.e., `msg.value > 0`), the operation will still be performed and the native tokens will be lost.

An example is the `safeTransferToSelf` function:

```
function safeTransferToSelf(IERC20 token, uint256 value)
    internal returns (uint256) {
    if(address(token)
    == address(0xFFfFfFffFFfffFFfFFFFFFffFFFffffFfFFFfF)) {
        require(msg.value ≥ value, "insuff.eth");
        uint256 dust = msg.value - value;
        if(dust > 0) {
            payable(msg.sender).transfer(dust);
        }
        return value;
    }
    else {
        require(msg.value == 0, "SaferERC20: cannot transfer value with
            ether");
        require(token.allowance(msg.sender, address(this)) ≥ value,
    "insuff.approval");
        return safeTransferFromWithAmount(token, msg.sender,
    address(this), value);
    }
}
```

Similarly, all the other functions that handle sending tokens should check that `msg.value` is zero in case the token is not the native token.

4.4 EIP-712 implementation does not follow latest standard

The EIP-712 implementation in the contracts does not follow the latest standard, which was updated after EIP-5267. The EIP-5267 complements EIP-712 by standardizing how contracts should publish the fields and values that describe their domain. This enables applications to retrieve this description and generate appropriate domain separators in a general way and thus integrate EIP-712 signatures securely and scalably.

It does that through the addition of the `eip712Domain` function:

```
function eip712Domain()
    public
    view
    virtual
```

```
    returns (
        bytes1 fields,
        string memory name,
        string memory version,
        uint256 chainId,
        address verifyingContract,
        bytes32 salt,
        uint256[] memory extensions
    )
{
    return (
        hex"0f", // 01111
        _EIP712Name(),
        _EIP712Version(),
        block.chainid,
        address(this),
        bytes32(0),
        new uint256[](0)
    );
}
```

For more information, see the latest EIP-712 implementation example here.

4.5 Market setters should only be called once

For consistency, all setters should only be called once. This is not the case for the `registerDebt`, `registerCollateral`, and more functions, which can be called multiple times. This can lead to unexpected behavior if the credit or collateral for a market are changed in the meanwhile, after accounting already existed for the previous credit and collateral.

```
function registerDebt(address debt_) external originIs(Role.ADMIN) {
    _debt = IDebt(payable(debt_));
}

function registerCollateral(
    address collateral_
) external originIs(Role.ADMIN) {
    _collateral = ICollateral(payable(collateral_));
}
```

```
// ... rest of the setters
```

Moreover, we recommend that an additional check is performed so that all contracts are in sync. For example, ensure that the new debt's market address points to the same market as the collateral's market address and that they both point to `address(this)`, the address of the Market contract.

```
function registerDebt(
    address debt_
) external originIs(Role.ADMIN) {

    require(_collateral.market() == IMarket(debt_).market(), "not same
        market");
    require(IMarket(debt_).market() == address(this), "not same market");
    _debt = IDebt(payable(debt_));
}

function registerCollateral(
    address collateral_
) external originIs(Role.ADMIN) {

    require(_debt.market() == ICollateral(collateral_).market(), "not
        same market");
    require(ICollateral(collateral_).market() == address(this), "not
        same market");
    _collateral = ICollateral(payable(collateral_));
}


// ... likewise for the rest of the setters
```

4.6 Utilizing eth-brownie for testing

We encountered a significant bottleneck while utilizing eth-brownie for testing the smart contracts. For that reason, a disproportionate amount of time was spent on debugging issues of the framework itself rather than focusing on testing the actual smart contract functionalities. The intuitive error handling, speed, flexibility, and advanced

testing capabilities of Hardhat and Foundry could have expedited our testing phase, delivering more precise insights in a shorter time span.

We recommend opting for either of the two frameworks for future development, as it would drastically lower the time spent on testing and compiling the contracts and would allow for a more efficient development process.

4.7 Centralization

There are many instances of centralization, from signatures that determine the prices of assets to liquidations that can only be authorized with a centralized signature. Therefore, the protocol has the responsibility of starting liquidation auctions before at-risk positions accrue bad debt.

4.8 AMM oracle pricing

A large part of the protocol requires the use of swappers that emulate swaps in existing AMMs to estimate the worth of collateral and more. Nukem's protocol design is extremely susceptible for manipulation due to flash loans imbalancing pools, affecting the value of certain positions. A possible alternative to reduce price volatility would be to use TWAPs or, ideally, oracles; however, depending on the market size, TWAPs may lag too far behind to provide competitive prices for smaller market cap coins.

4.9 Borrowing on behalf of the Market contract

The Market contract allows borrowing on behalf of itself if prior deposits are also performed on behalf of the Market. Although this does not come with any immediate security implications, it does not seem to follow the intended design of the contract, as debt will be issued for the Market contract, which is theoretically not a user. We recommend addressing or explicitly documenting this behavior.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: AbstractSwapper.sol

Function: `swap(address token, address owner, address receiver, uint256 amount, uint256 slippage)`

This acts like a middleware for swapping on a specific DEX.

Inputs

- `token`
  - **Control**: Fully controlled by the user.
  - **Constraints**: Must be a valid address, not whitelisted.
  - **Impact**: The token to be swapped.
- `owner`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: Not used.
- `receiver`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A.
  - **Impact**: The address to receive the swapped tokens.
- `amount`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A; transfer logic handled in `safeTransferToSelf`.
  - **Impact**: The amount of tokens to be swapped.
- `slippage`
  - **Control**: Fully controlled by the user.
  - **Constraints**: N/A.

- **Impact**: The maximum amount of slippage allowed.

Branches and code coverage (including function calls)

Intended branches

- Decrease the depositing token balance of the `msg.sender`.
  - ☐ Test coverage
- Increase the swapped token balance of the `receiver`.
  - ☐ Test coverage
- Assure that the resulting tokens are within the slippage bounds.
  - ☑ Test coverage

Negative behavior

- Should not allow swapping same token for same token.
  - ☐ Negative test

5.2 Module: Auctions.sol

Function: `checkSignature(address authorizer, address market, address debtor, uint256 block_number, uint256 starts_at, uint256 ends_at, uint256 deadline, uint8 v, byte[32] r, byte[32] s)`

This checks the centralized signature.

Inputs

- `authorizer`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The signer.
- `market`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The relevant market where the liquidation is happening.
- `debtor`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The debtor.
- `block_number`
  - **Control**: Full.

    - **Constraints**: None.
    - **Impact**: The `block_number` of the signature.
- `starts_at`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: When the auction starts.
- `ends_at`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: When the auction ends.
- `deadline`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The deadline.
- `v`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: Signature.
- `r`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: Signature.
- `s`
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: Signature.

Branches and code coverage (including function calls)

Intended branches

- Verify signature against authorizer.
  ☐ Test coverage

Negative behavior

- Works through blockchain fork.
  ☐ Negative test

Function: `execute(address authorizer, address market, address debtor, uint256 block_number, uint256 starts_at, uint256 ends_at, uint256 deadline, uint256 bid, uint8 v, byte[32] r, byte[32] s)`

This executes the auction.

Inputs

- `authorizer`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The signer.
- `market`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The relevant market where the liquidation is happening.
- `debtor`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The debtor.
- `block_number`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The `block_number` of the signature.
- `starts_at`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: When the auction starts.
- `ends_at`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: When the auction ends.
- `deadline`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The deadline.
- `bid`
  - **Control**: Full.
  - **Constraints**: None.

  - **Impact**: The current bid.
- `v`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Signature.
- `r`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Signature.
- `s`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Signature.

Branches and code coverage (including function calls)

Intended branches

- Timestamp checks work.
  ☑ Test coverage
- Cannot liquidate user without enough stable.
  ☑ Test coverage

Negative behavior

- Cannot auction healthy positions.
  ☑ Negative test
- Soft/hard liquidations verified against bids.
  ☐ Negative test

Function call analysis

- `execute → _exexute(msg.sender, Auction( ... ), bid)`
  - **What is controllable?** Nothing (must pass sig checks).
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `execute → _exexute(msg.sender, Auction( ... ), bid) → isAuctionable(auction.marker, auction.debtor)`
  - **What is controllable?** Nothing (must pass sig checks).
  - **If return value controllable, how is it used and how can it go wrong?** Could

stop liquidations even in unhealthy positions.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `execute → _exexute(msg.sender, Auction(...), bid) → currentPrice(...)`
  - **What is controllable?** Nothing (must pass sig checks).
  - **If return value controllable, how is it used and how can it go wrong?** Make liquidations unprofitable, resulting in unhealthy positions.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `execute → _exexute(msg.sender, Auction(...), bid) → market.collateral().swappableValue(auction.debtor)`
  - **What is controllable?** Nothing (must pass sig checks).
  - **If return value controllable, how is it used and how can it go wrong?** The return value is controllable and affects what type of liquidation it is — controllable by manipulating the underlying pool.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `execute → _exexute(msg.sender, Auction(...), bid) → market.credit().liquidate(debtors, account)`
  - **What is controllable?** Nothing (must pass sig checks).
  - **If return value controllable, how is it used and how can it go wrong?** Amounts liquidated.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `execute → _exexute(msg.sender, Auction(...), bid) → market.credit().auction(account, bid, auction.debtor)`
  - **What is controllable?** Only bid, everything else is checked against signature.
  - **If return value controllable, how is it used and how can it go wrong?** Amounts liquidated.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `groupedLiquidation(address market, address[] debtors)`

This groups liquidations.

Inputs

- `market`
  - **Control**: Full.

  - **Constraints**: None.
  - **Impact**: The market.
- `debtors`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The debtors to be liquidated.

Branches and code coverage (including function calls)

Intended branches

- Grouped liquidations function as expected.
  ☐ Test coverage

Negative behavior

- Cannot be called by anyone other than `Role.EXECUTE_AUCTION`.
  ☐ Negative test

Function call analysis

- `groupedLiquidation(market, debtors)` → `market.credit().liquidate(debtors, msg.sender)`
  - **What is controllable?** Debtors.
  - **If return value controllable, how is it used and how can it go wrong?** Used for event info.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `isAuctionable(address market, address debtor)`

This checks if a position is auctionable (are they unhealthy?).

Inputs

- `market`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The market.
- `debtor`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The debtor.

Branches and code coverage (including function calls)

Intended branches

- Unhealthy user is auctionable.
  ☑ Test coverage

Negative behavior

- Healthy user is not auctionable.
  ☐ Negative test

Function call analysis

- `isAuctionable(market, debtor)` → `market.collateral.collateralizationRatio()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Collateralization ratio, if controllable, could be used to unfairly liquidate users.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `isAuctionable(market, debtor)` → `market.strategy.liquidationRatio()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Liquidation ratio, if controllable, could be used to unfairly liquidate users.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.3 Module: Collateral.sol

Function: `amountValue(uint256 amount_of_assets)`

This returns the value of the collateral as stable, using the same precision as the debt collateral.

Inputs

- `amount_of_assets`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of assets.

Branches and code coverage (including function calls)

Intended branches

- `amountValue` works as intended and is a conservative value.
  ☑ Test coverage

Negative behavior

- Reverts against zero.
  ☐ Negative test

Function call analysis

- `amountValue → debt.totalAssets()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Amount of debt.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `amountValue → debt.totalAssets()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Amount of debt.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `amountValue → debt.precision()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Debt precision.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `amountValue → swapper.valueOf(debt.asset(), debt_precision)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Value of one debt in stable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `amountValue → debt.asset()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Debt asset.

  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `amountValue → swapper.conservativeValueOf(asset(), amount_of_assets, total_assets)`
  - **What is controllable?** `amount_of_assets` only.
  - **If return value controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `computeCollateralizationRatio(address account, uint256 assets)`

This calculates an account's collateralization ratio.

Inputs

- `account`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The account.
- `assets`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Number of assets.

Branches and code coverage (including function calls)

Intended branches

- Computes correctly (tested partially through other indirect calls).
  ☑ Test coverage

Negative behavior

- Works with different precision tokens.
  ☐ Negative test

Function call analysis

- `computeCollateralizationRatio(account, assets) → market.strategy().precision()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Pre-

cision.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `computeCollateralizationRatio(account, assets)` → `market.debt().value(account)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Amount of debt of the account.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `maxWithdraw(address account)`

This calculates how much a user can withdraw.

Inputs

- `account`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The account.

Branches and code coverage (including function calls)

Intended branches

- Works as intended and allows user to withdraw as long as position is healthy.
  ☑ Test coverage

Negative behavior

- Zero edge cases.
  ☐ Negative test

Function call analysis

- `maxWithdraw(account)` → `assetsOf(account)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Account assets (collateral).
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `maxWithdraw(account)` → `debt.assetsOf(account)`

  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Amount of debt.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `maxWithdraw(account) → market.swapper().valueOf(debt.asset(), account_debt_assets)`
  - **What is controllable?** Full.
  - **If return value controllable, how is it used and how can it go wrong?** Manipulated to return a small number, allowing a user to remove their collateral for their debt.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `maxWithdraw(account) → (asset_balance * strategy.maxCollateralizationRatio()) / strategy.precision()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Used in calculations to calculate how much collateral can be withdrawn.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `swap(address[] accounts, address receiver)`

Performs a swap.

Inputs

- `accounts`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Account to burn from.
- `receiver`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Receiver of the swapped tokens.

Branches and code coverage (including function calls)

Intended branches

- Swap works as intended.

☑ Test coverage

Function call analysis

- `swap → assetsOf(accounts[i])`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** How much to swap and burn.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `swap → _swap(total_assets, receiver)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Returned.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `_leverage(address sender, uint256 _amount)`

This leverages a position.

Inputs

- `sender`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The `msg.sender` from market.
- `_amount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The amount to leverage.

Branches and code coverage (including function calls)

Intended branches

- Leverages the user's position.
  ☑ Test coverage

Negative behavior

- Zero/boundary checks.
  ☐ Negative test

Function call analysis

- `_leverage` → `asset_.balanceOf`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Initial amount of asset.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `_leverage` → `market.credit().borrowForCollateral(amount, sender)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `_leverage` → `asset_.balanceOf`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Post amount of asset.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `_leverage` → `_mint(nothing)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.4 Module: Credit.sol

Function: `auction(address liquidator, uint256 liquidated, address debtor)`

This auctions.

Inputs

- `liquidator`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Liquidator.

- `liquidated`
  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: Amount liquidated.
- `debtor`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Account to be liquidated.

Branches and code coverage (including function calls)

Intended branches

- Auction pays off and transfers tokens.
  ☑ Test coverage

Negative behavior

- Zero check.
  ☐ Negative test
- Cannot auction with insufficient shares.
  ☐ Negative test

Function call analysis

- `auction( ... ) → assetsPerShare()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Assets per share.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `auction( ... ) → collateral.balanceOf(debtor)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Amount of collateral of debtor.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `auction( ... ) → converToShares(liquidated)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Share conversion.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

N/A.

- `auction( ... ) → collateral.balanceOf(liquidator)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Amount of collateral of liquidator.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `auction( ... ) → collateral.auctionTransfer(debtor, liquidator, collateral_balance)`
  - **What is controllable?** Everything (`onlyAuction`).
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `auction( ... ) → debt.erase(debtor)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `auction( ... ) → _mint(this, profit_shares)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `borrow(address account, uint256 amount, uint256 fee)`

This borrows `onlyDebt`.

Inputs

- `account`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The account.
- `amount`
  - **Control**: Full.
  - **Constraints**: None.

  - **Impact**: The amount to borrow.
- `fee`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The fee.

Branches and code coverage (including function calls)

Intended branches

- Limits are respected.
  ☑ Test coverage
- Sends the user the borrowed assets.
  ☑ Test coverage

Negative behavior

- Slippage enforced.
  ☐ Negative test

Function call analysis

- `borrow(...)` → `assetsPerShare()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Assets per share.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `borrow(...)` → `market.debt().mintDebt(account, amount + fee)`
  - **What is controllable?** Full.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `borrow(...)` → `asset.safeTransfer(account, amount)`
  - **What is controllable?** Full.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `borrow(...)` → `_mint(market, (fee * _base_unit) / assets_per_share)`
  - **What is controllable?** Full.

  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `liquidate(address[] debtors, address liquidator)`

This liquidates (already confirmed; only auction).

Inputs

- `debtors`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The debtors.
- `liquidator`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The liquidator.

Branches and code coverage (including function calls)

Intended branches

- Tips are paid.
  ☑ Test coverage
- Sends the user the borrowed assets.
  ☑ Test coverage

Negative behavior

- Cannot liquidate zero.
  ☐ Negative test
- Tip cannot be below liquidated.
  ☐ Negative test

Function call analysis

- `liquidate( ... )` → `debt.rebase()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

N/A.

- `liquidate( ... ) → assetsPerShare()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Assets per share.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `liquidate( ... ) → market.collateral().swap(debtors, this)`
  - **What is controllable?** Debtors.
  - **If return value controllable, how is it used and how can it go wrong?** Amount received for the unhealthy position.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `liquidate( ... ) → (liquidated * strat_.liquidationTip()) / strat_.precision()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Used to calculate the tip for the liquidator (incentive).
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `liquidate( ... ) → debt_.erase(debtors[i])`
  - **What is controllable?** Debtors.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `liquidate( ... ) → _mint(this, profit_shares)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `_borrowForCollateral(uint256 amount, address sender)`

This borrows collateral (`OnlyCollateral`).

Inputs

- `amount`

  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount to borrow.
- `sender`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Sender from collateral.

Branches and code coverage (including function calls)

Intended branches

- Limits are respected.
  ☑ Test coverage
- Sends the user the borrowed assets.
  ☑ Test coverage

Negative behavior

- Slippage enforced.
  ☐ Negative test

Function call analysis

- `_borrowForCollateral → assetsPerShare()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Assets per share.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `_borrowForCollateral → market.strategy().borrowFeeFor()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Add fee onto borrow.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `_borrowForCollateral → _swap(amount, msg.sender)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

- `_borrowForCollateral → _mint( ... )`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.5 Module: Debt.sol

Function: `erase(address account)`

This erases debt.

Inputs

- `account`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The account whose debt will be erased.

Branches and code coverage (including function calls)

Intended branches

- Removes debt.
  ☑ Test coverage

Negative behavior

- Cannot erase debt from an empty account.
  ☑ Negative test

Function: `totalSupply()`

This gets the total supply going through the rounds.

Branches and code coverage (including function calls)

Intended branches

- Tests as functioned.
  ☐ Test coverage

Function call analysis

- `totalSupply → strat.numberOfRounds()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Number of rounds to iterate.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `totalSupply → strat.peekRaw()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Last computed rate.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `totalSupply → strat.precision()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Precision.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `_deposit(uint256 assets, address receiver)`

This pays off debt.

Inputs

- `assets`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of assets.
- `receiver`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Whose debt to payoff.

Branches and code coverage (including function calls)

Intended branches

- Debt is paid off.
  ☑ Test coverage

- User can borrow again.
  ☑ Test coverage

Negative behavior

- Zero check.
  ☐ Negative test

Function call analysis

- `_deposit` → `asset.safeTransferFromWithAmount(msg.sender, credit, assets)`
  - **What is controllable?** `assets` (checks).
  - **If return value controllable, how is it used and how can it go wrong?** Amount transferred.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `_withdraw(uint256 assets, address receiver, address owner)`

This borrows.

Inputs

- `assets`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount to borrow.
- `receiver`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Receiver of borrowed coins.
- `owner`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The owner of the collateral.

Branches and code coverage (including function calls)

Intended branches

- Respects `maxWithdraw`.
  ☑ Test coverage

Negative behavior

- Fee cannot be larger than borrow.
  - ☐ Negative test

Function call analysis

- `_withdraw(assets, receiver, owner)` → `borrowFeeFor(assets)`
  - **What is controllable?** Full.
  - **If return value controllable, how is it used and how can it go wrong?** The borrow fee.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `_withdraw(assets, receiver, owner)` → `_market.credit().borrow(receiver, assets, borrow_fee)`
  - **What is controllable?** Full.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.6 Module: EIP712.sol

Function: `_domainSeparatorV4()`

This should build the domain separator for the current chain.

Branches and code coverage (including function calls)

Intended branches

- Return the cached domain separator if the `block.chainId` is the same as the cached chain ID and the cached address. This is the same as the current address. Otherwise, build the domain separator. This is currently not performed and allows for replaying the signatures in case of a fork.
  - ☐ Test coverage

5.7 Module: ERC20Base.sol

Function: `approve(address spender, uint256 amount)`

This allows `msg.sender` to approve another address to spend tokens on their behalf.

Inputs

- `spender`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to approve to spend tokens on behalf of `msg.sender`.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to approve to spend on behalf of `msg.sender`.

Branches and code coverage (including function calls)

Intended branches

- Increase the allowance of `msg.sender` for spender by amount.
  ☑ Test coverage

Negative behavior

- Should not allow someone else to edit the allowance of others.
  ☑ Negative test

Function: `mint(address account, uint256 amount)`

This allows the minter to mint tokens.

Inputs

- `account`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to mint tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to mint.

Branches and code coverage (including function calls)

Intended branches

- Increase balance of account by amount.
  - ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the minter or the owner.
  - ☑ Negative test

Function: `transferFrom(address from, address to, uint256 amount)`

This allows users to transfer tokens from one address to another, provided that the sender has been approved to spend tokens on behalf of the owner.

Inputs

- `from`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to transfer tokens from.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to transfer tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to transfer.

Branches and code coverage (including function calls)

Intended branches

- Assumed `from` posesses enough tokens to transfer.
  - ☑ Test coverage
- Assumed `from` has approved `msg.sender` to spend tokens on their behalf.
  - ☑ Test coverage
- Decrease balance of `from` by amount.
  - ☑ Test coverage
- Increase balance of `to` by amount.

☑ Test coverage

Negative behavior

- Should not be callable if `from` does not posess enough tokens to transfer.
  ☑ Negative test
- Should not be callable if `from` has not approved `msg.sender` to spend tokens on their behalf.
  ☑ Negative test

Function: `transfer(address to, uint256 amount)`

This allows users to transfer tokens to another address.

Inputs

- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to transfer tokens to.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to transfer.

Branches and code coverage (including function calls)

Intended branches

- Assumed `msg.sender` posesses enough tokens to transfer.
  ☐ Test coverage
- Decrease balance of `msg.sender` by amount.
  ☐ Test coverage
- Increase balance of `to` by amount.
  ☐ Test coverage

Negative behavior

- Should not be callable if `msg.sender` does not posess enough tokens to transfer.
  ☐ Negative test

Function: `withdrawEth()`

This allows the owner to withdraw ETH from the contract.

Branches and code coverage (including function calls)

Intended branches

- Transfer ETH from this contract to the owner.
  ☑ Test coverage

Negative behavior

- Should only be callable by the owner or a role with `WITHDRAW_ETH` permission.
  ☑ Negative test

Function: `xapprove(address _token, address _spender, uint256 _value)`

This is a function to be used by XApprove role. Approves a spender to spend tokens from the contract.

Inputs

- `_token`
  - **Control**: Fully controlled by XApprove role.
  - **Constraints**: None.
  - **Impact**: The token to be approved.
- `_spender`
  - **Control**: Fully controlled by XApprove role.
  - **Constraints**: None.
  - **Impact**: The address of the spender.
- `_value`
  - **Control**: Fully controlled by XApprove role.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to be approved.

Branches and code coverage (including function calls)

Intended branches

- Approve the spender to spend the tokens.
  ☑ Test coverage

Negative behavior

- Should only be callable by XApprove role or owner.
  ☑ Negative test
- Should not allow approving `address(this)` tokens. Currently not checked.

☐ Negative test

Function: `xtransfer(address _token, address _creditor, uint256 _value)`

This is a function to be used by XTransfer role. Transfers tokens from the contract to a creditor.

Inputs

- `_token`
  - **Control**: Fully controlled by XTransfer role.
  - **Constraints**: None.
  - **Impact**: The token to be transferred.
- `_creditor`
  - **Control**: Fully controlled by XTransfer role.
  - **Constraints**: None.
  - **Impact**: The address of the creditor.
- `_value`
  - **Control**: Fully controlled by XTransfer role.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Transfer tokens from this contract to the creditor.
  ☑ Test coverage

Negative behavior

- Should only be callable by XTransfer role or owner.
  ☑ Negative test
- Should not allow transferring `address(this)` tokens. Currently not checked.
  ☐ Negative test

5.8 Module: ERC20Permit.sol

Function: `_permit(address owner, address spender, uint256 value, uint256 deadline, uint8 v, bytes32 r, bytes32 s)`

This validates a signature for a permit and calls `_approve`.

Inputs

- `owner`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is the recovered address from the signature in the `isValidSignatureNow` call.
  - **Impact**: The owner of the funds to be approved.
- `spender`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The spender of the funds to be approved.
- `value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of funds to be approved.
- `deadline`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is in the future.
  - **Impact**: The deadline for the signature to be valid.
- `v`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must recover in a valid signature.
  - **Impact**: The v part of the signature.
- `r`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must recover in a valid signature.
  - **Impact**: The r part of the signature.
- `s`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must recover in a valid signature.
  - **Impact**: The s part of the signature.

Branches and code coverage (including function calls)

Intended branches

- Check that the owner is the recovered address from the signature. That is ensured in the `isValidSignatureNow` call.
  ☑ Test coverage
- Check that the deadline is in the future.

☑ Test coverage

- Call `_approve` with the owner, spender, and value, so that tokens are approved.
  ☑ Test coverage
- Assure that all other aspects of the signature are valid. That is ensured in the `isValidSignatureNow` call.
  ☑ Test coverage

Negative behavior

- Do NOT allow reusing the signature. That is ensured through the nonce.
  ☑ Negative test

5.9 Module: EnFi4626.sol

Function: `_deposit(uint256 assets, address receiver)`

This facilitates depositing assets into the vault.

Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that enough was transferred in `safeTransferFromWithAmount`.
  - **Impact**: The amount of assets to deposit into the vault.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to mint the shares to.

Branches and code coverage (including function calls)

Intended branches

- Deplete the balance of `msg.sender` by `assets`.
  ☑ Test coverage
- Increase the balance of `address(this)` by `assets`.
  ☑ Test coverage
- Mint `shares` to `receiver`.
  ☑ Test coverage

Negative behavior

- Should not allow depositing more assets than what was transferred. That is ensured in `safeTransferFromWithAmount`.
  ☑ Negative test
- Should not allow the deposit inflation attack. Currently that is not enforced in code.
  ☐ Negative test

Function: `_mint(uint256 shares, address receiver)`

This facilitates minting shares into the vault.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of shares to mint.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to mint the shares to.

Branches and code coverage (including function calls)

Intended branches

- Deplete the balance of `msg.sender` by `assets`.
  ☑ Test coverage
- Increase the balance of `address(this)` by `assets`.
  ☑ Test coverage
- Mint `shares` to `receiver`.
  ☑ Test coverage

Negative behavior

- Should not allow depositing more assets than what was transferred. That is ensured in `safeTransferFromWithAmount`.
  ☑ Negative test
- Should not allow the deposit inflation attack. Currently that is not enforced in code.
  ☐ Negative test
- Should not allow for huge discrepancies between `previewMint` and actual mint. Currently that is not enforced in code.

☐ Negative test

Function: `_redeem(uint256 shares, address receiver, address owner)`

This allows the redemption of shares from the vault.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Limited to the balance of `owner`.
  - **Impact**: The amount of assets to withdraw from the vault.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to mint the shares to.
- `owner`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address owning the assets to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Deplete the balance of assets for `address(this)` by `assets`.
  ☑ Test coverage
- Increase the balance of assets for `receiver` by `assets`.
  ☑ Test coverage
- Burn `shares` from `owner`.
  ☑ Test coverage

Negative behavior

- Should not allow spending more than the allowance if the caller is not the owner.
  ☑ Negative test
- Should not allow withdrawing more assets than what is burned.
  ☑ Negative test

Function: `_withdraw(uint256 assets, address receiver, address owner)`

This allows withdrawing assets from the vault.

Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Limited to the balance of `owner`.
  - **Impact**: The amount of assets to withdraw from the vault.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to mint the shares to.
- `owner`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address owning the assets to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Deplete the balance of assets for `address(this)` by `assets`.
  ☑ Test coverage
- Increase the balance of assets for `receiver` by `assets`.
  ☑ Test coverage
- Burn `shares` from `owner`.
  ☑ Test coverage

Negative behavior

- Should not allow spending more than the allowance if the caller is not the owner.
  ☑ Negative test
- Should not allow withdrawing more assets than what is burned.
  ☑ Negative test

5.10 Module: LendingStrategy.sol

Function: `getCurrentlyUtilized()`

This gets the amount of funds utilized as a percentage (precision).

Branches and code coverage (including function calls)

Intended branches

- Scales linearly with the utilization rate.
  ☑ Test coverage

Negative behavior

- Boundary tests.
  ☐ Negative test

Function: `peek()`

This peeks the to-be rate of interest.

Branches and code coverage (including function calls)

Intended branches

- Interest scales with utilization rate.
  ☑ Test coverage
- Min/max thresholds are enforced.
  ☑ Test coverage

5.11 Module: Market.sol

Function: `borrow(address authorizer, address market, uint256 price, uint256 deadline, uint256 amount, address receiver, address owner, uint8 v, bytes32 r, bytes32 s)`

This allows borrowing with signature.

Inputs

- `authorizer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be the owner of the contract and part of a valid signature.
  - **Impact**: Authorizer of the borrow.
- `market`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be the address of the contract.
  - **Impact**: Market address.
- `price`
  - **Control**: Fully controlled by the caller.

    - **Constraints**: Checked to be a valid price.
    - **Impact**: Price of the borrow.
- `deadline`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked to be in the future and part of a valid signature.
    - **Impact**: Deadline of the borrow.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked to be a valid amount — not included in the signature.
    - **Impact**: Amount of the borrow.
- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None here — not included in the signature.
    - **Impact**: Receiver of the borrow.
- `owner`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None here — not included in the signature.
    - **Impact**: Owner of the collateral used for the borrow.
- `v`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked to be a valid signature.
    - **Impact**: Signature v.
- `r`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked to be a valid signature.
    - **Impact**: Signature r.
- `s`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked to be a valid signature.
    - **Impact**: Signature s.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the signature constructed from `authorizer`, `market`, `price`, `deadline`, `v`, `r`, and `s` is valid.
  ☑ Test coverage
- Ensure the authorizer is the owner of the contract.

  - ☑ Test coverage
- Ensure that the market is the address of the contract.
  - ☑ Test coverage
- Ensure that the deadline is in the future.
  - ☑ Test coverage
- Validate the amount to be borrowed against the price.
  - ☑ Test coverage
- Debt balance of the owner should be increased by the calculated amount.
  - ☑ Test coverage
- Credit balance of the receiver should be increased by the amount borrowed.
  - ☑ Test coverage

Negative behavior

- Signature should not be replayable. Currently it is.
  - ☐ Negative test
- Assumes `_owner` cannot be `address(0)`.
  - ☐ Negative test

Function: `checkSignature(address authorizer, address market, uint256 price, uint256 deadline, uint8 v, bytes32 r, bytes32 s)`

This checks a signature.

Inputs

- `authorizer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be the owner of the contract and part of a valid signature.
  - **Impact**: Authorizer of the signature.
- `market`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be the address of the contract.
  - **Impact**: Market address.
- `price`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid price.
  - **Impact**: Price of the signature.
- `deadline`
  - **Control**: Fully controlled by the caller.

  - **Constraints**: Checked to be in the future and part of a valid signature.
  - **Impact**: Deadline of the signature.
- `v`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid signature.
  - **Impact**: Signature v.
- `r`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid signature.
  - **Impact**: Signature r.
- `s`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid signature.
  - **Impact**: Signature s.

Branches and code coverage (including function calls)

Intended branches

- Verify that the signature constructed from `authorizer`, `market`, `price`, `deadline`, `v`, `r`, and `s` is valid.
  ☑ Test coverage

Negative behavior

- Should not allow the replay of a signature. This is not enforced, as there is no nonce put in place.
  ☐ Negative test

Function: `leverage(address authorizer, address market, uint256 price, uint256 deadline, uint8 v, bytes32 r, bytes32 s)`

This allows leveraging with signature.

Inputs

- `authorizer`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be the owner of the contract and part of a valid signature.
  - **Impact**: Authorizer of the leverage.
- `market`

  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be the address of the contract.
  - **Impact**: Market address.
- `price`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid price.
  - **Impact**: Price of the leverage.
- `deadline`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be in the future and part of a valid signature.
  - **Impact**: Deadline of the leverage.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid amount — not included in the signature.
  - **Impact**: Amount of the leverage.
- `v`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid signature.
  - **Impact**: Signature v.
- `r`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid signature.
  - **Impact**: Signature r.
- `s`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid signature.
  - **Impact**: Signature s.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the signature constructed from `authorizer`, `market`, `price`, `deadline`, `v`, `r`, and `s` is valid.
  ☑ Test coverage
- Ensure the authorizer is the owner of the contract.
  ☑ Test coverage
- Ensure that the market is the address of the contract.
  ☑ Test coverage

- Ensure that the deadline is in the future.
  ☑ Test coverage
- Validate the amount to be leveraged against the price.
  ☑ Test coverage
- Increase the collateral balance of the sender by the amount leveraged.
  ☑ Test coverage
- Increase the debt balance of the sender by the amount leveraged.
  ☑ Test coverage
- Mint to the Market the credit fee amount.
  ☑ Test coverage

Negative behavior

- Signature should not be replayable. Currently it is.
  ☐ Negative test
- Assumes `_owner` cannot be `address(0)`.
  ☐ Negative test

Function: `recall(uint256 _value)`

This allows a user to close a position (e.g., deleverage the value of what was deposited).

Inputs

- `_value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid amount of collateral to be recalled.
  - **Impact**: Amount of collateral to be recalled.

Branches and code coverage (including function calls)

Intended branches

- Verify that the collateral amount is valid.
  ☑ Test coverage
- If the collateral amount is valid, then deleverage the value of what was deposited.
  ☑ Test coverage
- If what was deposited is more than what can be repaid, then repay all the debt and send the surplus back to the sender.
  ☑ Test coverage
- Otherwise, repay what is possible and make sure user is still within correct col-

lateralization limits.
  ☑ Test coverage
- Receive the swapped collateral as a result of the deleverage (essentially `max_deposit`).
  ☑ Test coverage
- Ensure that debt is zero after a full recall.
  ☐ Test coverage

Negative behavior

- If the collateral amount is invalid, then revert.
  ☑ Negative test
- Should not allow user to recall more than what they have deposited.
  ☑ Negative test
- Should not allow recalling on behalf of another user.
  ☑ Negative test
- Should not allow calling if debt is zero.
  ☐ Negative test

Function: `registerCollateral(address collateral_)`

This sets the Collateral contract address.

Inputs

- `collateral_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid address.
  - **Impact**: Collateral address.

Branches and code coverage (including function calls)

Intended branches

- Should set the Collateral address.
  ☑ Test coverage
- Ensure that the Collateral's market is the same as this address. Similar additional cross-checks could be added.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test

- Presumably should not be callable more than once.
  ☐ Negative test

Function: `registerCredit(address credit_)`

This sets the Credit contract address.

Inputs

- `credit_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid address.
  - **Impact**: Credit address.

Branches and code coverage (including function calls)

Intended branches

- Should set the Credit address.
  ☑ Test coverage
- Ensure that the Credit's market is the same as this address. Similar additional cross-checks could be added.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test
- Presumably should not be callable more than once.
  ☐ Negative test

Function: `registerDebt(address debt_)`

This sets the Debt contract address.

Inputs

- `debt_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid address.
  - **Impact**: Debt address.

Branches and code coverage (including function calls)

Intended branches

- Should set the Debt address.
  ☑ Test coverage
- Ensure that the Debt's market is the same as this address. Similar additional cross-checks could be added.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test
- Presumably should not be callable more than once.
  ☐ Negative test

Function: `registerVsCredit(address vsCredit_)`

This sets the vsCredit contract address.

Inputs

- `vsCredit_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid address.
  - **Impact**: vsCredit address.

Branches and code coverage (including function calls)

Intended branches

- Should set the vsCredit address.
  ☑ Test coverage
- Ensure that the vsCredit's market is the same as this address. Similar additional cross-checks could be added.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test
- Presumably should not be callable more than once.
  ☐ Negative test

Function: `setAuctions(address auctions_)`

This sets the Auctions contract address.

Inputs

- `auctions_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid address.
  - **Impact**: Auctions address.

Branches and code coverage (including function calls)

Intended branches

- Should set the Auctions address.
  ☑ Test coverage
- Assure that Auctions's market is the same as this address. Similar additional cross-checks could be added.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test
- Presumably should not be callable more than once.
  ☐ Negative test

Function: `setStrategy(address strategy_)`

Sets the LendingStrategy contract address.

Inputs

- `strategy_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid address.
  - **Impact**: LendingStrategy address.

Branches and code coverage (including function calls)

Intended branches

- Should set the LendingStrategy address.

☑ Test coverage

- Ensure that the LendingStrategy's market is the same as this address. Similar additional cross-checks could be added.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test
- Presumably should not be callable more than once.
  ☐ Negative test

Function: `setSwapper(address swapper_)`

Sets the swapper contract address.

Inputs

- `swapper_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid address.
  - **Impact**: Swapper address.

Branches and code coverage (including function calls)

Intended branches

- Should set the swapper address.
  ☑ Test coverage
- Ensure that the swapper's market is the same as this address. Similar additional cross-checks could be added.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test
- Presumably should not be callable more than once.
  ☐ Negative test

Function: `setTreasury(address treasury_)`

Sets the treasury contract address.

Inputs

- `treasury_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid address.
  - **Impact**: Treasury address.

Branches and code coverage (including function calls)

Intended branches

- Should set the treasury address.
  ☑ Test coverage
- Assure that the treasury's market is the same as this address. Similar additional cross-checks could be added.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test
- Presumably should not be callable more than once.
  ☐ Negative test

5.12 Module: ProxyManager.sol

Function: `create(byte[32] implementation_id, bytes[32] init_data, bytes[32] salt)`

This allows the creation of clones of a master contract.

Inputs

- `implementation_id`
  - **Control**: Fully controlled by owner
  - **Constraints**: Assumed to be a valid implementation.
  - **Impact**: Allows the owner to create clones of the implementation.
- `init_data`
  - **Control**: Fully controlled by owner.
  - **Constraints**: None.
  - **Impact**: The owner can pass arbitrary data to the initializer of the clone.
- `salt`

- **Control**: Fully controlled by owner.
- **Constraints**: None.
- **Impact**: The salt used for the `CREATE2` opcode for deploying the clone.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the master contract being cloned is a valid contract.
  ☑ Test coverage
- Add the clone to the list of instances.
  ☑ Test coverage
- Set the reverse mapping of the clone to the implementation ID.
  ☑ Test coverage
- Bootstrap the clone.
  ☑ Test coverage
- Initialize the clone.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the owner to create clones.
  ☑ Negative test
- Should not allow the creation of clones of nonexistent implementations.
  ☑ Negative test

Function: `setImplementation(byte[32] implementation_id, address implementation, bytes[32] upgrade_data, bool callUpdate)`

This allows the owner to register a new master logic contract.

Inputs

- `implementation_id`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The ID where the implementation will be registered.
- `implementation`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Checked that the implementation is a valid contract.
  - **Impact**: The implementation to be registered.
- `upgrade_data`

  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The upgrade data to be passed to the clone's upgrade function.
- `callUpdate`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: Whether to call the upgrade function on the clones or not.

Branches and code coverage (including function calls)

Intended branches

- If `implementation_id` already exists, update the implementation's clone instances.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test
- Should NOT allow upgrading to an implementation that has a different storage layout. This is not checked.
  ☐ Negative test

Function: `upgradeContract(address clone_address, bytes[32] upgrade_data)`

This allows the owner to manually upgrade a clone to a new implementation.

Inputs

- `clone_address`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The clone to be upgraded.
- `upgrade_data`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: None.
  - **Impact**: The upgrade data to be passed to the clone's upgrade function.

Branches and code coverage (including function calls)

Intended branches

- Call the clone's upgrade function.
  ☑ Test coverage
- Ensure that the clone is a valid clone.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  ☑ Negative test

5.13 Module: Roles.sol

Function: `hasRole(address account, bytes32 role)`

This checks whether the account has a specific role.

Inputs

- `account`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The account whose role is being checked.
- `role`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The role being checked for the account.

Branches and code coverage (including function calls)

Intended branches

- Check whether account is the owner or not.
  ☑ Test coverage
- If not, check whether the account has the role (as a form of ERC-721 token ID ownership) or not.
  ☑ Test coverage

Function: `_setRoleNFT(uint256 id, bytes32 _role)`

This sets a role to a specific ERC-721 token ID.

Inputs

- `id`
  - **Control**: Controlled by calling function.
  - **Constraints**: Checked that it is not zero.
  - **Impact**: The ID of the ERC-721 token whose role is being set.
- `_role`
  - **Control**: Controlled by calling function.
  - **Constraints**: Checked that it is not zero.
  - **Impact**: The role being set to the ERC-721 token ID.

Branches and code coverage (including function calls)

Intended branches

- Assumes that if the `accessControl` is changed, this function will be called to set the new `accessControl`'s role.
  ☐ Test coverage
- Should allow changing/removing the role of an ERC-721 token ID.
  ☑ Test coverage

5.14 Module: SaferERC20.sol

Function: `safeTransferToSelf(IERC20 token, uint256 value)`

This performs a safe transfer to self. If the token is ETH, then the value is transferred to the contract. If the token is not ETH, then the value is transferred from the sender to the contract.

Inputs

- `token`
  - **Control**: Controlled by upper-level function.
  - **Constraints**: Must be a valid IERC20 or native ETH.
  - **Impact**: The token to be transferred.
- `value`
  - **Control**: Controlled by upper-level function.
  - **Constraints**: In case of ETH, must be greater than or equal to the `msg.value`.
  - **Impact**: The amount of tokens to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Perform a safe transfer of ETH to the contract. This means that the `msg.value` must be greater than or equal to the value.
  ☑ Test coverage
- Returns anything left over to the sender.
  ☑ Test coverage
- Perform a safe transfer of tokens from the sender to the contract. This means that the sender must have approved the contract to spend the value amount of.
  ☑ Test coverage

Negative behavior

- Should not allow receiving ETH if the token is not ETH. Currently not checked.
  ☐ Negative test

Function: `safeTransfer(IERC20 token, address to, uint256 value)`

This allows performing of a safe transfer of tokens. If the token is ETH, then the value is transferred to the address. If the token is not ETH, then the value is transferred from the contract to the address.

Inputs

- `token`
  - **Control**: Controlled by calling function.
  - **Constraints**: Has to be either a valid IERC20 or native ETH.
  - **Impact**: The token to be transferred.
- `to`
  - **Control**: Controlled by calling function.
  - **Constraints**: Must be a valid address.
  - **Impact**: The address to receive the tokens.
- `value`
  - **Control**: Controlled by calling function.
  - **Constraints**: None; presumably the calling function will do the checks.
  - **Impact**: The amount of tokens to be transferred.

Branches and code coverage (including function calls)

Intended branches

- Assume that `to` is entitled the `value` and that the contract has enough balance to transfer.
  - ☐ Test coverage
- Assume that higher-level calling function will check that the `to` does not receive more than they deserve.
  - ☐ Test coverage

Negative behavior

- Should not allow receiving ETH if the token is not ETH. Currently not checked.
  - ☐ Negative test

5.15 Module: UniswapV2Swapper.sol

Function: `swap(address token, address owner, address receiver, uint256 amount, uint256 slippage)`

This swaps.

Inputs

- `token`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: None.
- `owner`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Owner.
- `receiver`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Receiver.
- `amount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount to swap.
- `slippage`
  - **Control**: Full.
  - **Constraints**: None.

- **Impact**: Slippage in percentage (not correct).

Branches and code coverage (including function calls)

Intended branches

- Swap functions as intended.
  ☑ Test coverage

Negative behavior

- Slippage testing.
  ☐ Negative test

Function call analysis

- `swap()` → `pair.token0()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** `token0`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `swap()` → `pair.token1()`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** `token1`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `swap()` → `token.safeTransferFrom(owner, pair, amount)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `swap()` → `pair.swap(amount0Out, amount1Out, receiver, new bytes(0))`
  - **What is controllable?** Receiver.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `swap()` → `token.safeDecimals()`
  - **What is controllable?** Nothing.

  - **If return value controllable, how is it used and how can it go wrong?** Token decimals.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `swap()` → `valueOf(token, in_precision + slippage)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Value for calculating slippage (this is erroneous).
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Nukem Loans contracts, we discovered nine findings. Three critical issues were found. Three were of high impact, two were of medium impact, and one was of low impact. Nukem Loans acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.