Zellic

Voyage

Smart Contract Security Assessment

November 1st, 2022

Prepared for:

Ian Tan

Voyage Finance

Prepared by:

Filippo Cremonese and Oliver Murray

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 email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Voyage Finance from August 22nd to September 2nd, 2022.

Zellic thoroughly reviewed the Voyage codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

Specifically, taking into account Voyage's threat model, we performed a deep analysis of vault interactions including considering multiple variants of interaction sequences between senior and junior depositors and NFT purchasers as well as the security of upgrades, critical function calls, and the oracle implementation.

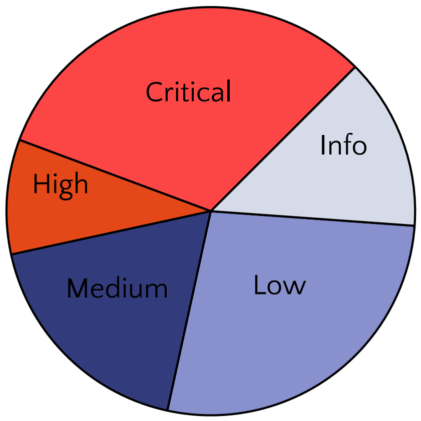
During our assessment on the scoped Voyage contracts, we discovered 22 findings. There were seven critical issues found. Of the remaining issues, two were high impact, four were medium impact, six were low impact, and the remaining were informational.

Additionally, Zellic recorded its notes and observations from the audit for Voyage Finance's benefit in the Discussion section (4) at the end of the document.

We sometimes observe a high amount of findings in projects undergoing rapid development. Our recommendation to the Voyage team is to adopt a security-focused development workflow. The codebase should be augmented with a comprehensive test suite ensuring the code is behaving as intended under real-world conditions. We also encourage Voyage to freeze the codebase and perform another independent audit before launch.

Breakdown of Finding Impacts

Impact Level	Count
Critical	7
High	2
Medium	4
Low	6
Informational	3



2 Introduction

2.1 About Voyage

Voyage allows users to obtain short-term loans for purchasing from whitelisted NFT collections. Multiple payment installments are used to reduce the cost of market entry by spreading payments over time. Senior and junior risk tranches are used to incentivize LPs with different risk tolerances. A traditional free market mechanism is used to drive the ratios of assets deposited into senior and junior tranches.

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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile 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. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible 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 forth.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Voyage Contracts

Repository	https://github.com/voyage-finance/voyage-contracts
Versions	`6068d640a2a39607bf32c73e5026080398056140`
Programs	• Voyage Smart Constracts
Type	Solidity
Platform	EVM

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 two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese, Engineer
fcremo@zellic.io

Oliver Murray, Engineer
oliver@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

August 22, 2022	Kick-off call
August 22, 2022	Start of primary review period
September 2, 2022	End of primary review period

3 Detailed Findings

3.1 Public `pullToken` function allows to steal ERC20 tokens for which Voyage has approval

- **Target**: PeripheryPayments.sol
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `PeripheryPayments::pullToken` function does not perform any access control and can be used to invoke `transferFrom` on any token.

```
function pullToken(
    IERC20 token,
    uint256 amount,
    address from,
    address recipient
) public payable {
    token.safeTransferFrom(from, recipient, amount);
}
```

Furthermore, we have two additional observations about this function:

- It is unnecessarily marked as `payable`.
- It allows to call `transferFrom` on any contract, not just ERC20; since ERC721 tokens also have a compatible `transferFrom` function, `pullToken` could be used to invoke `transferFrom` on ERC721 contracts as well. At the time of this review, the Voyage contract does not hold nor is supposed to have approval for any ERC721 assets, so this issue has no impact yet.

Impact

An attacker can use this function to invoke `tranferFrom` on any contract on behalf of Voyage, with arbitrary arguments. This can be exploited to steal any ERC20 token for which Voyage has received approval.

Recommendations

Apply strict access control to this function, allowing only calls from `address(this)`.

Remediation

Voyage has applied the appropriate level of access control to this function by making it internal. Furthermore, the contract has been removed and its functionality factored into a library as reflected in commit 9a2e8f42.

3.2 Signature clash allows calls to `transferReserve` to steal NFT collateral

- **Target**: VaultFacet.sol
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `VaultFacet` contract has a `transferReserve(_vault, _currency, _to, _amount)` function meant to be used by the vault owner for recovering any ERC20 assets held by their vault.

The function calls `execute` on the given vault, instructing it to call `transferFrom(from, to, amount)` on the address specified by the `_currency` argument, with the `to` and `amount` arguments specified by the `transferReserve` caller.

An attacker can take advantage of this capability by making the vault call `transferFrom` on the ERC721 contract controlling a collateral held by the vault. This is possible since ERC20 `transferFrom` and ERC721 `transferFrom` signatures are identical; therefore, the calldata format required by both functions is the same.

Impact

An attacker can transfer any NFT held by a vault without having fully repaid the debt for which the NFT was held as collateral.

Recommendations

Ensure the contract being called is not the contract of an NFT being held as collateral.

An additional recommended hardening measure would be to entirely deny calls to ERC721 contracts. A possible approach to accomplish this is to try to call a harmless ERC721 method on the contract and reverting the transaction if the call does *not* fail.

Remediation

Commit 7460dc9a was indicated as containing the remediation. The commit appears to correctly fix the issue. The `transferReserve` function has been renamed to `transferCurrency` and now takes as input the address of an NFT collection. The currency to be transferred is obtained from the metadata associated to the collection in Voyage storage.

Voyage updated the code in a subsequent commit, and (as of commit f558e630) the `t` `ransferCurrency` function again receives a `_currency` argument representing an ERC20 contract address. The change was made to ensure users can always withdraw ERC20 tokens that would otherwise be at risk of being stuck in their vault. That address is checked against data contained in Voyage storage to ensure it is not the address of an ERC721 contract used by Voyage, and the code seems still safe.

We note that 25 commits exist between 7460dc9a and the one subject to our audit, applying changes that are both irrelevant as well as others potentially relevant to the remediation, increasing the difficulty of the review. In total, the diff between the reviewed and remediation commits amounts to 18 solidity files changed, with 137 insertions and 385 deletions. The changes include adding a `checkCurrencyAddr` function also used in `transferCurrency`, then renamed to `checkCollectionAddr`, which only ensures that the address given as an argument is a deployed contract, not that it exists in the metadata stored by Voyage as the name could imply.

3.3 Missing calldata validation in `buyNow` results in stolen NFT

- **Target**: LoanFacet.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `_data` calldata parameter passed to `buyNow( ... )` requires a consistency check against the `_tokenId` parameter.

The `_tokenId` passed is recorded in

```
LibAppStorage.ds().nftIndex[param.collection][
    param.tokenId
] = composeNFTInfo(param);
```

where `nftInfo.isCollateral` is set to `true` by `composeNFTInfo( ... )`.

However, the actual NFT ordered from the market is specified in `_data`, which is not validated to match the given `_tokenId`. This allows an attacker to purchase a mismatching NFT, which is sent to the vault. The NFT corresponding to the `_tokenId` argument is marked as collateral for the loan instead of the one that was actually received. Therefore, the token can be withdrawn from the vault, as the following check in `VaultFacet::withdrawNFT( ... )` will not revert the transaction:

```
if (LibAppStorage.ds().nftIndex[_collection][_tokenId].isCollateral) {
    revert InvalidWithdrawal();
}
```

Impact

This vector makes the current implementation vulnerable to several attacks. For example, `buyNow` can be called with `tokenId = 10` and `calldata _data` containing `tokenId = 15`. The order will process and an NFT with `tokenId = 15` will be purchased. The NFT can then be withdrawn while having only paid the down payment.

Recommendations

Validation checks should be added to ensure that the `tokenId` and `collection` passed in are consistent with the `tokenId` and `collection` passed in `calldata _data`. Additionally, validation modules should be added to the `LooksRareAdapter` and `SeaportAdapter` to validate all other order parameters. Currently, only the order selectors are validated. This additional lack of checks opens up the possibility for more missing validation exploits on other variables. However, the core of the vulnerability is the same, and so we have grouped them all into one finding.

Remediation

Voyage has incoporated the necessary validation checks for `tokenId` and `collection` in commit 7937b13a. They have also included additional validation checks for `isOrderAsk` and `taker` in `LooksRareAdapter` and `fulfillerConduitKey` in `SeaportAdapter` in commit 7937b13a. These are critical validations checks to have included and we applaud Voyage for their efforts. However, there may still be other parameters that require validation checks in the orders and we suggest Voyage perform a comprehensive review of all of the parameters in determine if there are any outstanding validation checks that may be necessary.

3.4 Missing timelocks can result in stolen NFTs

- **Target**: VToken.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

To start, there are no timelocks on the senior and junior depositor vaults. Furthermore, the share of vault assets lenders are entitled to when they withdraw is based on the share of assets at the time of deposit:

```
function pushWithdraw(address _user, uint256 _shares) internal {
    unbondings[_user].shares += _shares;
    unbondings[_user].maxUnderlying += convertToAssets(_shares);
    totalUnbonding += _shares;
}
```

And these funds are held in `bonding` until they are claimed:

```
function claim() external {
    uint256 maxClaimable = unbondings[msg.sender].maxUnderlying;
    [ ... ]
```

This means a malicious user can potentially steal an outsized share of the principal and interest payments by manipulating their vault shares through deposits, withdraws, and claims.

Impact

Given that a lender can also purchase an NFT, the above opens up a novel approach for NFT theft as follows:

1. Take out a flash loan.
2. Call `deposit( ... )` - make a large vault deposit and get awarded an outsized number of shares (shares are proportional to total asset share).
3. Call `buyNFT( ... )` - purchase an NFT by making the first principal and interest payments.

4. Call `withDraw( ... )` - with a large enough flash loan, you should be able to lock for withdraw the majority of the principal and interest payments you just made.
5. Call `claim( ... )` - remove your funds from the vault and pay back your flash loan. You will need a separate source of funds to pay interest on the flash loan (longer-term loan, whale).
6. Repeat steps 2, 4, and 5 until the maturity of the loan has passed.
7. Call `withdrawNFT( ... )` to take posession of your NFT. Sell it and repay any outstanding debts.

Recommendations

The vector above can be blocked by preventing lenders from also purchasing NFTs; however, this would be a naive fix. The ability to deposit and withdraw funds without timelocks in order to create a `maxClaimable` slip that can be used to claim interest and principal payments at any time is a fundamental design flaw. It means depositors can game the system, claiming principal and interest payments for which they hold no credit risk.

We suggest implementing a timelock mechanism on depositors' shares to ensure they are "paying their dues." This will help ensure depositors take on levels of credit risk commensurate with their returns. It is true that depositors who come in at later dates may end up covering losses on assets lent out earlier; our interpretation is that this is part of the pooling design. However, we feel the ability to game this exposure is a design flaw and should be removed.

Remediation

Commit a5bfd675 was indicated as containing the remediation. Rewiewing the remediation for this issue has proven to be challenging due to the pace of the development. A total of 181 commits exist between the commit under review and a5bfd675; the diff between the two commits amounts to 43 solidity files changed, with 2416 insertions and 1684 deletions. The issue appears to be correctly fixed at the given commit. We largely based this evaluation on the description provided by the Voyage team due to the considerable amount of changes, which aligns with what can be observed in the commits. In particular, the code tracking the balances of the amounts deposited by the users has been updated to keep track of the unbonding amounts; further, we observed no anomalies in the evolution of the balances during the execution of a proof of concept developed to demonstrate the issue when executed against the commit containing the remediation.

We note that it is still technically possible to reclaim a disproportionate amount of the interests portion of the installments by depositing a very large amount of assets before buying an NFT and withdrawing after repayments are made. The Voyage team has argued that this strategy does not seem to be exploitable to gather a profit. Their assessment is likely to be correct for the economic conditions in which Voyage is expected to operate, although profitability might be possible if some parameters such as flash loan interest rates, Voyage pool asset sizes and NFT values were to assume unexpected values.

3.5 Junior depositor funds mistakenly sent to senior depositors

- **Target**: LoanFacet.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

Calls to `liquidate( ... )` made when the discounted price of the underlying NFT is greater than the price paid when buying it will move funds from junior depositors and send them to senior depositors. In `liquidate( ... )`,

```
param.remaningDebt = param.totalDebt;
// [ ... ]
param.receivedAmount = discountedFloorPriceInTotal;
// [ ... ]
if (param.totalDebt > discountedFloorPriceInTotal) {
    param.remaningDebt = param.totalDebt - discountedFloorPriceInTotal;
} else {
    uint256 refundAmount = discountedFloorPriceInTotal -
        param.totalDebt;
    IERC20(param.currency).transfer(param.vault, refundAmount);
    param.receivedAmount -= refundAmount;
}
```

If `param.totalDebt > discountedFloorPriceInTotal`, then `param.receivedAmount = param.totalDebt` and `param.remaningDebt = param.totalDebt`. The following code will therefore execute the following:

```
if (param.remaningDebt > 0) {
    param.totalAssetFromJuniorTranche = ERC4626(
        reserveData.juniorDepositTokenAddress
    ).totalAssets();

    if (param.totalAssetFromJuniorTranche >= param.remaningDebt) {
        IVToken(reserveData.juniorDepositTokenAddress)
            .transferUnderlyingTo(address(this), param.remaningDebt);
        param.juniorTrancheAmount = param.remaningDebt;
        param.receivedAmount += param.remaningDebt;
    } else {
        IVToken(reserveData.juniorDepositTokenAddress)
            .transferUnderlyingTo(
                address(this),
                param.totalAssetFromJuniorTranche
            );
        param.juniorTrancheAmount = param.totalAssetFromJuniorTranche;
        param.receivedAmount += param.totalAssetFromJuniorTranche;
        param.writeDownAmount =
            param.remaningDebt -
            param.totalAssetFromJuniorTranche;
    }
}
```

It can be verified that `param.receivedAmount = 2 * param.totalDebt` or `param.receivedAmount = param.totalDebt + param.totalAssetFromJuniorTranche` depending on whether `param.totalAssetFromJuniorTranche ≥ param.remaningDebt`.

Voyage will be in possession of assets equal to `param.receivedAmount`; furthermore, `param.receivedAmount` will be sent to the senior depositors:

```
IERC20(param.currency).safeTransfer(
    reserveData.seniorDepositTokenAddress,
    param.receivedAmount
);
```

Impact

The finding has been rated as critical because it could have catastrophic consequences for the performance of the protocol.

1. Junior depositors would be missing funds with potentially no explanation. This is likely to be realized by users over time and may result in near complete abandonment of the junior tranche and hence loss of core protocol functionality and purpose.
2. It would raise the prospect of additional yet unfound issues that could end up affecting senior depositors. This could result in a complete loss of confidence in the project and team.

Recommendations

Make the following code change in `liquidate( … )`:

```
} else {
    uint256 refundAmount = discountedFloorPriceInTotal -
        param.totalDebt;
    IERC20(param.currency).transfer(param.vault, refundAmount);
    param.receivedAmount -= refundAmount;
    param.remaningDebt = 0;
}
```

Remediation

Voyage has since made considerable changes to the code base in order to fundamentally alter the way funds are distributed in the event of liquidations. We view the changes to the code base as extending beyond remediation efforts targeting the basic coding mistake we have identified and as constituting extensions to the code base that would require extending the scope of the audit engagement. For context, there have been 81 commits made since the scoping of the audit and this remediation commit provided by Voyage 654a9242. Across these commits a total of 30 solidity files have been changed with a total of 1,406 insertions and 1008 deletions. Out of respect for the scope of the initial engagement we have not been able to fully audit these changes and confirm whether the underlying issue identified here has indeed been remediated. However, we can confirm that Voyage has acknowledged the issue and has claimed to have fixed it in these architectural changes.

3.6 Inconsistent usage of `totalUnbonding` leads to lost or under-utilized lender assets

- **Target**: Vtoken.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

Functions in VToken assume the variable `totalUnbonding` keeps track of the total amount of underlying shares in the unbonding state. However, the rest of the Voyage protocol assumes these variables keep track of the amount of underlying asset in the unbonding state. For example, VToken::pushWithdraw(...) uses shares:

```
function pushWithdraw(address _user, uint256 _shares) internal {
    unbondings[_user].shares += _shares;
    unbondings[_user].maxUnderlying += convertToAssets(_shares);
    totalUnbonding += _shares;
}
```

Whereas `JuniorDepositToken::totalAssets()` assumes the variable expresses an amount of the underlying asset:

```
contract JuniorDepositToken is VToken {
    function totalAssets() public view override returns (uint256) {
        return asset.balanceOf(address(this)) - totalUnbonding;
    }
}
```

Impact

This issue has far-reaching consequences, as it influences the amount of assets deposited to both the senior and junior pools. Depending on the exchange rate used to convert between assets and shares, `totalUnbonding` could become greater or smaller than the correct value.

For example, if `convertToAssets(_shares) > _shares` then `totalUnbonding` will be set to a lower-than-intended amount by `pushWithdraw( ... )`. This means that Voyage will assume the pool has more assets available than it really does. So, for example, liquidity checks in `buyNow( ... )` will pass when they should not, and purchase orders can mysteriously fail. Additionally, assets locked by lenders for withdraw will still be lent out. This can lead to calls to `claim( ... )` failing and lost assets for lenders.

If `convertToAssets(_shares) < _shares` then `totalUnbonding` is instead set to a greater-than-intended value by `pushWithdraw( ... )`. Voyage will therefore assume the pool has fewer assets than it really does. Depositor assets will become underutilized by borrowers, and depending on the magnitude of the difference, funds could become effectively locked.

Moreover, since `totalUnbonding` factors into `SeniorDepositToken::totalAssets( ... )` this can also have an impact on the general accuracy of deposit and withdraw calculations, as the conversion ratio between shares and assets depends on the value returned by `totalAssets( ... )`.

Recommendations

Consistently use `totalUnbonding` to express an amount of assets or an amount of shares.

Assuming the variable is intended to keep track of an amount of assets, at least two modifications to the code would have to be made.

One to `VToken::pushWithdraw( ... )`:

```
function pushWithdraw(address _user, uint256 _shares) internal {
     unbondings[_user].shares += _shares;
     unbondings[_user].maxUnderlying += convertToAssets(_shares);
    totalUnbonding += _shares;
    totalUnbonding += convertToAssets(_shares);
 }
```

And another to `VToken::claim()`:

```
function claim() external {
     // [ ... ]
     if (availableLiquidity > maxClaimable) {
         // [ ... ]
     } else {
         // [ ... ]
     }
    totalUnbonding -= transferredShares;
    totalUnbonding -= convertToAssets(transferredShares);
    asset.safeTransfer(msg.sender, transferredAsset);
}
```

Remediation

Commits 3320ba3c and acbe5001 were indicated as containing remediations for this issue.

Reviewing the remediation for this issue has proven to be challenging due to the pace of the development. A total of 29 commits exist between the commit under review and 3320ba3c; the diff between the two commits amounts to 24 solidity files changed, with 324 insertions and 525 deletions. Another 86 commits exist between 3320ba3c and acbe5001, with a diff amounting to 29 solidity files changed, 1279 insertions, and 969 deletions.

The two commits appear to correctly fix the issue; we largely based this evaluation on the description of the applied changes provided by the Voyage team due to the considerable amount of changes, which seems compatible with what can be observed in the commits. The `totalUnbonding` function is not used anymore in the computation of `totalAssets`; two other functions, `totalUnbondingAsset` and `unbonding`, were introduced, respectively computing the amount of assets and shares that are in the unbonding state.

3.7 Share burn timing in Vtoken can lead to complete loss of funds

- **Target**: Vtoken.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

In general, the ERC4626 vault uses the current ratio of total shares to total assets for pricing conversion from assets to shares for deposits and conversions from shares to assets for withdrawals.

The VToken vault in Voyage implements a novel two-step withdrawal process.

Users first call `withdraw(...)`, which calls `pushWithdraw(...)`, to record the number of shares being withdrawn and the corresponding value in asset terms and to reserve the total amount of assets being withdrawn by updating `totalUnbonding`. In the current implementation, `burn(...)` occurs before this call is made:

```
shares = previewWithdraw(_amount); // No need to check for rounding error,
    previewWithdraw rounds up.
if (msg.sender != _owner) {
    _spendAllowance(_owner, msg.sender, shares);
}
beforeWithdraw(_amount, shares);
_burn(_owner, shares);
pushWithdraw(_owner, shares);
```

This inadvertently alters the total shares and hence the conversion from shares to assets that occurs in `pushWithdraw(...)`:

```
function pushWithdraw(address _user, uint256 _shares) internal {
    unbondings[_user].shares += _shares;
    unbondings[_user].maxUnderlying += convertToAssets(_shares);
    totalUnbonding += _shares;
}
```

Users then call `claim(...)` in order to receive their funds. For the case where `availableLiquidity > maxClaimable`, the incorrect conversion from the previous step will carry over. Furthermore, if `availableLiquidity ≤ maxClaimable`, another conversion will also be based on an incorrect total shares:

```
if (availableLiquidity > maxClaimable) {
    transferredAsset = maxClaimable;
    transferredShares = unbondings[msg.sender].shares;
    resetUnbondingPosition(msg.sender);
} else {
    transferredAsset = availableLiquidity;
    uint256 shares = convertToShares(availableLiquidity);
    reduceUnbondingPosition(shares, transferredAsset);
    transferredShares = shares;
}
```

Impact

Calling `deposit( ... )` and `withdraw( ... )` in the same transaction repeatedly can lead to draining of the tranches.

For example, in general, let deposit of assets of amount equal to `assetDeposited` result in an amount of shares equal to `sharesReceived` being sent to the depositor.

It is expected behavior (and has been verified) that an immediate call (same transaction) to `withdraw( ... )` made with `assetDeposited` will set the amount of shares to be burned as `sharesReceived`:

```
function withdraw(
    uint256 _amount,
    address _receiver,
    address _owner
) public override(ERC4626, IERC4626) returns (uint256 shares) {
    shares = previewWithdraw(_amount); // No need to check for rounding
    error, previewWithdraw rounds up.

    beforeWithdraw(_amount, shares);

    _burn(_owner, shares);
    pushWithdraw(_owner, shares);

    emit Withdraw(msg.sender, _receiver, _owner, _amount, shares);
```

The call to `_burn( ... )` in `withdraw( ... )` reduces the `totalSupply` of shares by `sharesReceived` so that the call to `pushWithdraw( ... )` overprices the asset when calculating the amount of asset owed to the depositor in `unbondings[_user].maxUnderlying += convertToAssets(_shares)` and also reserves the assets for withdraw in `totalUnbonding += convertToAssets(_shares)`:

```
function pushWithdraw(address _user, uint256 _shares) internal {
    unbondings[_user].shares += _shares;
    unbondings[_user].maxUnderlying += convertToAssets(_shares);
    totalUnbonding += convertToAssets(_shares);
}
```

The call to `convertToAssets(_shares)` necessarily overprices the asset. We have fully proven this mathematically, but there is a sufficiently strong intuitive argument.

The price of shares in units of assets is based on the ratio of the balance of assets to shares. From the base implementation of ERC4626 we have

```
function convertToAssets(uint256 shares)
    public
    view
    virtual
    returns (uint256)
{
    uint256 supply = totalSupply(); // Saves an extra SLOAD if totalSupply
    is non-zero.

    return supply == 0 ? shares : shares.mulDivDown(totalAssets(), supply
    );
}
```

Therefore, if the supply is reduced by a premature `_burn( ... )`, we necessarily overinflate the amount of assets the depositor can withdraw. This allows an attacker to drain the funds from the vaults through repeated atomic `deposit( ... ) + withdraw( ... )` transactions.

Recommendations

Move share burning until the end of `claim( ... )` as suggested below:

```
if (availableLiquidity > maxClaimable) {
        transferredAsset = maxClaimable;
        transferredShares = unbondings[msg.sender].shares;
```

```
            resetUnbondingPosition(msg.sender);
        } else {
            transferredAsset = availableLiquidity;
            uint256 shares = convertToShares(availableLiquidity);
            reduceUnbondingPosition(shares, transferredAsset);
            transferredShares = shares;
        }
        totalUnbonding -= transferredAsset;
        asset.safeTransfer(msg.sender, transferredAsset);
        _burn(_owner, transferredShares);
    }
```

This positioning should also avoid conflicts with other processes in Voyage.

Remediation

Voyage has moved the call to `_burn` so that it occurs after the reduction in the `unbonding` position in `claim` in commit 63099db1. This aligns the implementation with the intended design and avoids overvaluing the assets in the preview and conversion functions.

3.8 Buyers make first interest payment twice

- **Target**: LoanFacet.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

Callers of `buyNow( ... )` will always pay the first interest payment twice.

The first time happens when they pay the down payment—they can pay it in either ETH or WETH.

The down payment is equal to `params.downpayment = params.pmt.pmt` where `pmt` is given by

```
function calculatePMT(Loan storage loan)
    internal
    view
    returns (PMT memory)
{
    PMT memory pmt;
    pmt.principal = loan.principal / loan.nper;
    pmt.interest = loan.interest / loan.nper;
    pmt.pmt = pmt.principal + pmt.interest;
    return pmt;
}
```

The second time happens when `distributeInterest( ... )` is called:

```
LibLoan.distributeInterest(
    reserveData,
    params.pmt.interest,
    _msgSender()
);
```

This pulls the same amount, but only WETH, directly from the buyer.

Impact

Users will be discouraged from using the protocol due to the extra large payment arising from high interest rates.

Recommendations

Remove the interest component from the down payment.

Remediation

Commit 3320ba3c was indicated as containing the remediation for this issue. The `params.downpayment` variable is now set to `params.pmt.principal` instead of `params.pmt.pmt`, meaning it will contain the value corresponding to the principal (without interest) of a single installment.

We note that a total of 29 commits exist between the commit under review and 3320ba3c; the diff between the two commits amounts to 24 solidity files changed, with 324 insertions and 525 deletions, containing other potentially relevant changes.

3.9 Missing stale price oracle check results in outsized NFT price risk

- **Target**: LoanFacet.sol
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The price oracle stores the average price of NFTs in a given collection. Calls to `updateTWAP( ... )` set the average price and the `block.timestamp`:

```
function updateTwap(address _currency, uint256 _priceAverage)
    external
    auth
{
    prices[_currency].priceAverage = _priceAverage;
    prices[_currency].blockTimestamp = block.timestamp;
}
```

The timestamp is returned from calls to `getTWAP( ... )`:

```
function getTwap(address _currency)
    external
    view
    returns (uint256, uint256)
{
    return (
        prices[_currency].priceAverage,
        prices[_currency].blockTimestamp
    );
}
```

Unfortunately, the time stamp is never used in `buyNow( ... )`.

Since the protocol expects only NFTs satisfying the following two conditions to be purchased,

```
if (params.fv == 0) {
    revert InvalidFloorPrice();
}

if (params.fv < params.totalPrincipal) {
    revert InvalidPrincipal();
}
```

an out-of-date price oracle means these conditions could be violated.

Furthermore, there are no stale price checks in `liquidate(...)`:

```
IPriceOracle priceOracle = IPriceOracle(
    reserveData.priceOracle.implementation()
);
(param.floorPrice, param.floorPriceTime) = priceOracle.getTwap(
    param.collection
);
if (param.floorPrice == 0) {
    revert InvalidFloorPrice();
}
[ ... ]
param.totalDebt = param.principal;
param.remaningDebt = param.totalDebt;
param.discount = getDiscount(param.floorPrice, param.liquidationBonus);
param.discountedFloorPrice = param.floorPrice - param.discount;


uint256 discountedFloorPriceInTotal = param.discountedFloorPrice *
    collaterals.length;
IERC20(param.currency).safeTransferFrom(
    param.liquidator,
    address(this),
    discountedFloorPriceInTotal
);
```

Impact

If an NFT was purchased with a price greater than the average price (i.e., `params.fv < params.totalPrincipal`), then lenders may end up backing much riskier assets than intended. Additionally, if stale prices are below current prices a liquidator would be able to purchase the NFTs at a discount and sell them for a profit. On the other hand, if stale prices were above market prices, NFTs could stay locked in the system.

The utility of credit products for users depends immensely on good alignment between the underlying credit dynamics and user expectations. This logic error can result in a rate of loan defaults that is largely outsized to investor expectations.

Additionally, upon liquidation it can also result in vault loss of funds through selling NFTs at submarket prices.

Recommendations

Introduce stale price checks in `buyNow( ... )` and `liquidate(..)`. The protocol operators need to determine the appropriate length of the time window to be accepted for the last average price. Because these are NFT markets, it is important to ensure the window is long enough so that it reflects a sufficient number of trades while at the same time not including out-of-date trades.

Remediation

Voyage has introduced stale price checks in both `buyNow( ... )` and `liquidate( ... )` in the following commits 80a681a2 and 654a9242.

3.10 Calls to `Redeem( ... )` can result in lost depositor funds

- **Target**: VToken.sol
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: Medium

Description

We would like to credit Voyage for finding the following critical exploit while the audit was ongoing and in its early stages.

Calls to the base ERC4626 `redeem( ... )` can be made by anyone. Unfortunately, `redeem( ... )` does not implement any of the `pushWithdraw( ... )`:

```
function pushWithdraw(address _user, uint256 _shares) internal {
    unbondings[_user].shares += _shares;
    unbondings[_user].maxUnderlying += convertToAssets(_shares);
    totalUnbonding += convertToAssets(_shares);
}
```

Impact

Any calls to claim after calling `redeem( ... )` would result in no funds be transferred to the user.

Recommendations

We suggest modifying `redeem( ... )` to accordingly incorporate the `pushWithdraw( ... )` functionality.

Remediation

Commit 2ebf6278 was indicated as containing the remediation. The issue appears to be correctly fixed in the given commit, having `redeem` implement the correct logic including a call to `pushWithdraw`.

We note that the actual remediation was performed in 3320ba3c and that 2ebf6278 actually performs a minor refactoring on the lines responsible for the fix.

3.11 Incorrect calculation in `refundGas`

- **Target**: Vault.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

The `Vault::refundGas` function performs an incorrect calculation of the `amountRefundable` variable if the WETH amount to unwrap is greater than the available balance. The code is reported below for convenience:

```
function refundGas(uint256 _amount, address _dst) external onlyPaymaster
    {
    uint256 amountRefundable = _amount;
    uint256 ethBal = address(this).balance;
    // we need to unwrap some WETH in this case.
    if (ethBal < _amount) {
        IWETH9 weth9 = IWETH9(LibVaultStorage.ds().weth);
        uint256 balanceWETH9 = weth9.balanceOf(address(this));
        uint256 toUnwrap = _amount - ethBal;
        // this should not happen, but if it does, we should take what we
    can instead of reverting
        if (toUnwrap > balanceWETH9) {
            weth9.withdraw(balanceWETH9);
            amountRefundable = amountRefundable - toUnwrap - balanceWETH9
    ;
        } else {
            weth9.withdraw(toUnwrap);
        }
    }
    // [code continues ... ]
```

Consider the following numerical example:

- `_amount` is 100
- `ethBal` is 60
- `balanceWETH9` is 30
- `toUnwrap` will be calculated as 100 - 60 = 40
- `amountRefundable` will be calculated as 100 - 40 - 30 = 30, instead of the expected 90

Impact

The function will refund to the treasury less than the expected amount.

Recommendations

Fix the calculation by applying parentheses around `toUnwrap - balanceWETH9` on the line calculating `amountRefundable`.

Remediation

Voyage has followed the recommendation and corrected the calculation in commit 6e44df5f.

3.12 Missing access control on `postRelayedCall` leading to ETH transfer from Vault

- **Target**: VoyagePaymaster.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: Medium

Description

The `VoyagePaymaster::postRelayedCall` function is lacking any access control check. The function invokes `refundGas` on a vault supplied by the caller to refund a caller controlled amount of ETH to the treasury address.

```
function postRelayedCall(
    bytes calldata context,
    bool success,
    uint256 gasUseWithoutPost,
    GsnTypes.RelayData calldata relayData
) external virtual override {
    address vault = abi.decode(context, (address));
    // calldata overhead = 21k + non_zero_bytes * 16 + zero_bytes * 4
    // ~= 21k + calldata.length * [1/3 * 16 + 2/3 * 4]
    uint256 minimumFees = (gasUseWithoutPost +
        21000 +
        msg.data.length *
        8 +
        REFUND_GAS_OVERHEAD) * relayData.gasPrice;
    uint256 refund = vault.balance >= minimumFees
        ? minimumFees
        : minimumFees + 21000 * relayData.gasPrice; // cover cost of
    unwrapping WETH
    IVault(vault).refundGas(refund, treasury);
}
```

Impact

A malicious user can invoke `postRelayedCall` to transfer ETH from any vault to the treasury address.

Recommendations

Apply strict access control to the function.

Remediation

Commit 791b7e63 was indicated as containing the remediation. The commit correctly fixes the issue by enforcing access control to `postRelayedCall`.

3.13 Functions cannot be removed during upgrades

- **Target**: DiamondVersionFacet.sol
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

In `getUpgrade( ... )`,

```
bytes4[] storage existingSelectors = LibAppStorage
    .ds()
    .upgradeParam
    .existingSelectors[msg.sender];
```

it is populated with null values.

Therefore, the following loop to set the remove functions will never initiate:

```
for (uint256 i = 0; i < existingSelectors.length; ) {
    if (!newSelectorSet[existingSelectors[i]]) {
        LibAppStorage.ds().upgradeParam.selectorsRemoved[i].push(
            existingSelectors[i]
        );
    }
```

And the final `IDiamondCut.FacetCut[]` returned will not contain any of the remove instructions.

Impact

It will not be possible to remove functions from Voyage's interface using the intended functionality. It would be possible, however, to replace them with functions that do not perform any operations. This approach will, however, result in a very cluttered and confusing interface and should be avoided.

Recommendations

Populate `existingSelectors` by populating adding `existingSelectors.push(selector)` to the following:

```
for (uint256 i = 0; i < currentFacets.length; ) {
    IDiamondLoupe.Facet memory facet = currentFacets[i];
    for (uint256 j = 0; j < facet.functionSelectors.length; ) {
        bytes4 selector = facet.functionSelectors[j];
        newSelectors.push(selector);
        existingSelectorFacetMap[selector] = facet.facetAddress;
        existingSelectors.push(selector);
        unchecked {
            ++j;
        }
    }
    unchecked {
        ++i;
    }
}
```

Remediation

The upgrade functionality has been dropped entirely from the project, the issue has therefore been remediated.

3.14 Missing access control on multiple `PaymentsFacet` functions

- **Target**: PaymentsFacet.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: Low

Description

Multiple functions in `PaymentsFacet` are lacking any access control checks:

- `unwrapWETH9` unwraps and sends WETH owned by Voyage to an arbitrary address
- `wrapWETH9` wraps all the ETH balance owned by Voyage into WETH
- `sweepToken` transfers any ERC20 token owned by Voyage to an arbitrary address
- `refundETH` transfers all the ETH balance owned by Voyage to `msg.sender`

Impact

Those functions can be used to steal or transfer ETH and ERC20 assets held by the main Voyage contract. The contract only holds assets temporarily while processing transactions (e.g., `buyNow`), so an attacker cannot generally gain anything by using them. However, since there is no reentrancy guard, there is a risk of an attacker finding a way to reenter the contract while the contract is holding some assets.

Recommendations

Since these functions are not meant to be publicly exposed, they represent an unnecessary risk. We recommend to enforce access control to restrict usage only to the intended user.

Remediation

Commit 9a2e8f42 was indicated as containing the remediation. The issue is correctly fixed in the given commit. The four functions have been marked as internal.

3.15 Multicall can be used to call `buyNow` with untrusted `msg.value`

- **Target**: Multicall.sol, LoanFacet.sol
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: Low

Description

The main Voyage contract also exposes the methods of the Multicall contract via this chain:

- `Voyage` is an instance of `Diamond` (by inheritance)
- `Diamond` allows to `delegatecall` any registered facet
  - One of the facets is `PaymentsFacet`
- `PaymentsFacet` is `multicall` by inheritance
  - `Multicall` has a `multicall` method that performs an arbitrary amount of `delegatecalls` to `address(this)`, with arbitrary calldata

Any function called by `multicall` must not trust `msg.value`, since Multicall allows to perform multiple calls in the same transaction, preserving the same `msg.value`.

A function trusting `msg.value` might assume that the contract has received `msg.value` ETH from the caller and can spend it exclusively, which is not true in case the function is called multiple times in the same transaction by leveraging `multicall`.

Multicall allows to call any method exposed by any Voyage facet, including `LoanFacet::buyNow`, which assumes that `msg.value` ETH were sent by the caller as down payment for the requested NFT.

Impact

The `buyNow` function assumes the caller has sent `msg.value` ETH as down payment for the NFT. Luckily, an attacker cannot exploit this flawed assumption and use funds from the protocol pools to buy NFTs at a reduced price, as the contract will not have enough ETH to buy the NFT, causing a revert.

Recommendations

Adopt an explicit allowlist to limit which functions can be invoked by Multicall and ensure `msg.value` is not used by any of these functions. The `buyNow` function is the only one using `msg.value` in the commit under review.

Remediation

Multicalls and self permit have been removed from the code base entirely, the issue has therefore been remediated.

3.16 The maxWithdraw functionality is broken

- **Target**: LiquidityFacet.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Low
- **Impact**: Low

Description

Depositors will be unable to use the intended maxWithdraw functionality in `withdraw( ... )`:

```
uint256 userBalance = vToken.maxWithdraw(msg.sender);
uint256 amountToWithdraw = _amount;
if (_amount == type(uint256).max) {
    amountToWithdraw = userBalance;
}
BorrowState storage borrowState = LibAppStorage.ds()._borrowState[
    _collection
][reserve.currency];
uint256 totalDebt = borrowState.totalDebt + borrowState.totalInterest;
uint256 avgBorrowRate = borrowState.avgBorrowRate;
IVToken(vToken).withdraw(_amount, msg.sender, msg.sender);
```

Impact

Users will need to make withdraw requests for exact amounts in order to retrieve all of their deposited funds. If the `_amount` provided in the function call exceeds the available balance, the function will fail with no clear error message. This can create a frustrating and unexpected user experience.

Recommendations

Change

```
IVToken(vToken).withdraw(_amount, msg.sender, msg.sender);
```

to

```
IVToken(vToken).withdraw(amountToWithdraw, msg.sender, msg.sender);
```

Also, modify the `_amount` check to the following:

```
if (_amount == type(uint256).max || _amount > userBalance) {
    amountToWithdraw = userBalance;
}
```

Remediation

Commits aac23ae9 and 0e00c990 were indicated as containing the remediation. The commits correctly fix the issue by applying the suggested remediations.

3.17 Calls to `previewBuyNow(...)` do not return correct order previews

- **Target**: LoanFacet.sol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Low
- **Impact**: Low

Description

The implemented functionality to preview NFT orders is incomplete. For example, the call below does not pass the `_data` and `_tokenIds`, which are required to determine the `totalPrincipal`. As it currently stands, even the most critical fields like `totalPrincipal` are not populated:

```
function previewBuyNowParams(address _collection)
    public
    view
    returns (ExecuteBuyNowParams memory)
{
    ExecuteBuyNowParams memory params;
    ReserveData memory reserveData = LibLiquidity.getReserveData(
        _collection
    );
    ReserveConfigurationMap memory reserveConf = LibReserveConfiguration
        .getConfiguration(_collection);

    (params.epoch, params.term) = reserveConf.getBorrowParams();
    params.nper = params.term / params.epoch;

    params.outstandingPrincipal =
        params.totalPrincipal -
        params.totalPrincipal /
        params.nper;
```

Impact

There is a high probability that users would rely on the intended functionality of `previewBuyNow(...)` to improve their user experience.

Currently, the operation is non-functional and users would not be able to preview orders.

This could discourage user engagement.

Recommendations

We suggest fully specifying the desired functionality in `previewBuyNow(...)` and then updating the function accordingly. For example, parameters like `_data` and `_tokenId` should be passed to return the purchase price of the NFT and the average trading price of the NFTs in the collection. This would further allow fields like `params.totalPrincipal` to be populated and hence result in correct interest rate calculations.

Remediation

Voyage has refactored the function to populate a new struct `PreviewBuyNowParams` in commit f3db2541. It has been verified that the struct has been populated in the following commits 2f4da9c9 and e1892115.

3.18 Public `approve` function allows to give approval for any ERC20 tokens held by Voyage

- **Target**: PeripheryPayments.sol
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Low
- **Impact**: Low

Description

The `PeripheryPayments::approve` function does not perform any access control and can be used to invoke `approve` on any token on behalf of the Voyage contract.

```
function approve(
    IERC20 token,
    address to,
    uint256 amount
) public payable {
    token.safeApprove(to, amount);
}
```

Furthermore, we have two additional observations about this function:

- It is unnecessarily marked as `payable`.
- It allows to call `approve` on any contract, not just ERC20; since ERC721 tokens also have a compatible `approve` function, `PeripheryPayments::approve` could be used to invoke `approve` on ERC721 contracts as well. At the time of this review, the Voyage contract does not hold any ERC721 assets, so this specific issue has no impact yet.

Impact

An attacker can use this function to invoke `approve` on any contract on behalf of Voyage, with arbitrary arguments. This can be exploited to gain approval for any ERC20 or ERC721 token owned by Voyage. At the time of this review, the main Voyage contract only temporarily holds assets (e.g., while processing `buyNow`), so this could only be exploited if an external call to a malicious contract was to be performed while Voyage is in possession of an asset.

While this issue might not be exploitable in the code as reviewed, we strongly recommend against exposing this function, as approval for a token has a persistent effect that might become relevant with a future code update.

Recommendations

Apply strict access control to this function, allowing only calls from `address(this)`.

Remediation

Commit 9a2e8f42 was indicated as containing the remediation. The commit correctly fixes the issue by moving the `approve` function a new `LibPayments` library as an internal function. The subsequent commit 9a2e8f42 removes the `PeripheryPayments.sol` file entirely, leaving only the library version of the function.

3.19 Junior tranche receives interest with zero risk exposure

- **Target**: LibLoan.sol
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

There are no checks to confirm non-zero risk exposure of the junior tranche during the distribution of interest.

Impact

Interest is sent to the junior tranche even if there are no assets deposited. The interest is not entirely lost and can be recovered through calls to `transferUnderlyingTo( ... )` made by the admin.

We would like to further note that the distribution of IR payments between junior and senior tranches is fixed and not a risk-weighted exposure of the tranches. This creates a dynamic where the JR tranche may only contain $1 backing a $1MM NFT and still receive a fixed share of the interest. Such an opportunity would attract other investors. In theory, they would support the junior tranche until an equilibrium level is found that reflects the market appetite for IR returns and the credit profile of the protocol.

Recommendations

We would like Voyage to please confirm this is the dynamic they seek. In order for this dynamic to be realized, the following recommendation should be observed.

Add a check to ensure that the junior tranche has non-zero exposure to assets paying interest in `distributeInterest( ... )`.

Remediation

Voyage has included checks to ensure that the balance of the jr tranche exceeds an optimal liquidity ratio in calls to `buyNow` in commit 76f21d00.

3.20 Missing validation check on ERC20 `transfer`

- **Target**: loanFacet.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Low
- **Impact**: Informational

Description

Currently `liquidate( ... )` does not revert the transaction if the following ERC20 `transfer` fails:

```
if (param.totalDebt > discountedFloorPriceInTotal) {
    param.remaningDebt = param.totalDebt - discountedFloorPriceInTotal;
} else {
    uint256 refundAmount = discountedFloorPriceInTotal -
        param.totalDebt;
    IERC20(param.currency).transfer(param.vault, refundAmount);
    param.receivedAmount -= refundAmount;
}
```

Impact

The call should never fail as the funds will always be in the account.

Recommendations

Add a check and revert on a `false` return value from the ERC20 `transfer` call.

Remediation

Commit 654a9242 was indicated as containing the remediation. The commit correctly fixes the issue by using `safeTransfer` instead of `transfer`, which does revert if the transfer fails.

3.21 Lack of reentrancy guards

- **Target**: Voyage
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Medium
- **Impact**: Informational

Description

Most of the public and external functions lack reentrancy guards. Applying a guard to all functions that are not intended to be reentrant greatly simplifies reasoning about the actions that a malicious contract could perform on Voyage and reduces the attack surface.

Impact

The lack of reentrancy guards increases the attack surface reachable by any malicious contract that could be invoked by Voyage.

Recommendations

We recommend applying guards to all functions that are not designed to be reentrant. We note that the diamond pattern adopted by Voyage might require a custom implementation of reentrancy guards, in order to use the shared diamond storage contract to store the flag tracking the contract state. We further note that the diamond pattern requires allowing direct self-reentrancy, slightly limiting how restrictive a reentrancy guard could be.

Remediation

Voyage has indicated they have applied reentrancy gaurds to the majority of external functions. They have further clarified that they beleive that all external functions which do not have reentrancy gaurds are not vulnerable.

3.22 The contract SubVault is incomplete and not integrated into Voyage

- **Target**: SubVault.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Neither Voyage or the Vault contract can deploy a subvault, making any functionality currently implemented by the SubVault informational to the current design.

Impact

Currently there is no impact.

However, if the SubVault were to be integrated into Voyage, it would likely cause severe issues.

For example, anyone is able to call `pause(...)` and `unpause(...)` on the vault and hence render it inoperable as they please.

Recommendations

Please clarify the intended purpose of the SubVault. If it is not to be integrated into Voyage, we suggest removing it from the code base.

If the intention is to include the SubVault in Voyage, we can then evaluate the current implementation against this intention.

Remediation

Commit d15ae96d was indicated as containing the remediation. The commit fixes the issue by entirely removing all code related to subvaults.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Centralization risk

By design, Voyage entrusts its developers and operators. The diamond design pattern allows to upgrade any part of the contract, and the code as reviewed contains multiple functions that allow authorized administrators to, for example, transfer assets and set critical parameters. This design allows to add new features to the protocol and fix bugs, but it also exposes Voyage to the risk of a developer or administrator being compromised.

We encourage Voyage to disclose this aspect of the design to the users of the protocol and to adopt typical best practices to reduce the risk of a compromise:

- Use a multisig wallet to authorize smart contract upgrades or when making calls to sensitive protocol parameter setting functions such as those in ConfigurationFacet or LiquidityFacet
    - individual keys should ideally be stored on hardware wallets
- Require code review by multiple developers before deploying upgrades
- Use only peer-reviewed scripts to perform upgrades

4.2 Code maturity

Various areas of the codebase can be improved to increase maintainability, robustness, and readability.

Testsuite

The code as reviewed lacks a comprehensive testsuite. The Voyage team was already aware and planning to expand the testsuite with both positive and negative tests, ensuring the contracts appropriately handle both correct and incorrect inputs and state.

Missing events

The protocols lack sufficient event triggers. For example, there are no events triggered in critical VToken functions added to the base ERC4626 contract.

The overall codebase should be reviewed for missing events and updated accordingly.

Input validation

The codebase exhibits a general lack of input validation. One pervasive example is the retrieval of elements from maps without explicit checks to ensure the key (provided by the `msg.sender`) exists in the map. We strongly encourage the Voyage team to perform explicit checks against every external input and to extend the testsuite to ensure those checks are and remain effective as the codebase evolves.

The lack of input validation has been the root cause of critical issues in the codebase, such as finding 3.3 on the missing consistency check between the `_tokenId` argument and the token ID embedded in the `_data` argument provided to the `buyNow` function, allowing to buy an NFT at the cost of only the down payment.

Unused code

The codebase contains multiple pieces of unused code, ranging from individual lines, functions, and even whole files (such as the subvault-related features). This makes it harder to understand the codebase, increases code size (and hence deployment cost), and increases the available attack surface. Some unused functions even proved dangerous, such as `PeripheryPayments::approve`. Our recommendation is to remove any unused code from the main branch of the codebase.

Mix of generic ERC20 and hardcoded WETH code

At the time of review, the codebase contained both code designed to handle any generic ERC20 asset and code that assumes usage of WETH. The Voyage team communicated they are evaluating plans to support other ERC20 assets, such as various stablecoins, to enable processing NFT orders using these assets. Since the support for assets other than WETH is incomplete, we reviewed the code under the assumption that only WETH is used. If multiple assets are to be allowed, we recommend to review carefully the code interacting with the generic assets to ensure the contract being interacted with is consistent throughout the whole process of buying, repaying, and liquidating an NFT as well as depositing and withdrawing said asset.

4.3 Fail-open design of some core functions

Some functions critical to the operation of the protocol are designed with a fail-open behavior. For example, `withdrawNFT` relies on the `isCollateral` flag to be set in the `nftIndex` map of the shared diamond storage to prevent NFTs being held as collateral from being withdrawn.

It is arguably easier to forget to set or mistakenly set to the default value a variable than it is to set it to a nondefault value. We suggest to consider adopting a fail-closed design, in which dangerous actions are prevented by default. In the specific case of `withdrawNFT`, an `isWithdrawable` variable could be introduced, which would have to be set to `true` (a nondefault value) to allow an NFT to be transferred.

The fail-open design of `withdrawNFT` made possible to exploit the issue presented in finding 3.3, where a lack of input validation in `buyNow` led to setting the incorrect entry in `nftIndex`, allowing to withdraw the NFT bought with funds from the reserve without repaying it in full.

4.4 Functions marked as `payable`

The codebase contains multiple functions marked as `payable` without requiring them to be marked as such. Examples include

- `PaymentsFacet::refundETH`
- `PeripheryPayments::pullToken`
- `PeripheryPayments::approve`
- all `SelfPermit` functions

4.5 Slither

Slither returns multiple warnings. Most warnings are already reported separately or false positives. However, we recommend fixing the following:

- strictly following the checks-effects-interactions pattern
- LoanFacet.liquidate(address,address,uint256) (contracts/voyage/facets/LoanFacet.sol#360-524) ignores return value by IERC20(param.currency).transfer(param.vault,refundAmount) (contracts/voyage/facets/LoanFacet.sol#460)

4.6 Oracle hardening

Oracles should be hardened so they are robust for their intended purposes and potential attack vectors. This raises the following important considerations:

- How is the TWAP calculated? Consideration needs to be made to ensure there are a sufficient number of trades in the window for the average, while at the same time ensuring the window does not contain stale prices.
- Is the NFT marketplace itself safe and secure? Does it always give accurate prices under all scenarios?

- Will the price oracle ever fail? Under what conditions? In other words, will it fail if it cannot return a valid TWAP?
- What is the nature of the NFT collection? Are there quality tiers that can throw off the TWAP via different trading volumes or price ranges?

We recommend that Voyage provide insight to Zellic regarding the points above so that we can provide more insight for the report.

4.7 Findings discovered by Voyage

The following issues were discovered and shared with Zellic by the Voyage team while the audit was ongoing.

- For one, `_burn` should be called in `claim`, not `withdraw`. This causes the exchange rate to become inconsistent.
- Also, `SeniorDepositToken::redeem` is allowed to bypass the unbonding mechanism as it is not overridden.

5 Audit Results

During our investigation we found a total of 22 findings. By impact, of the 22 findings 7 were critical, 2 were high, 4 were medium, 6 were low and 3 were informational in nature.

We sometimes observe a high amount of findings in projects undergoing rapid development. Our recommendation to the Voyage team is to adopt a security-focused development workflow. The codebase should be augmented with a comprehensive test suite ensuring the code is behaving as intended under real-world conditions. We also encourage Voyage to freeze the codebase and perform another independent audit before launch.

5.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 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.

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.