Maple Finance Security Review

Auditors

Christoph Michel, Lead Security Researcher

0xleastwood, Lead Security Researcher

Riley Holterhus, Security Researcher

Devtooligan, Junior Security Researcher

Jonatas Martins, Junior Security Researcher

Report prepared by: Pablo Misirov & Jonatas Martins

December 2, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Maple Finance is an institutional crypto-capital network built on Ethereum and Solana. Maple provides the infrastructure for credit experts to efficiently manage and scale crypto lending businesses and connect capital from institutional and individual lenders to innovative, blue-chip companies. Built with both traditional financial institutions and decentralized finance leaders, Maple is transforming capital markets by combining industry-standard compliance and due diligence with the frictionless lending enabled by smart contracts and blockchain technology. Maple is the gateway to growth for financial institutions, pool delegates and companies seeking capital on-chain.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Maple V2 according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 15 days in total, Maple Finance engaged with Spearbit to review Maple V2. In this period of time a total of 52 issues were found.

Note: Scope was subject to change one week into the engagement as per client's request. It must be assumed that all issues have been found against the latest provided scope and commit hashes. Additionally, all liquidity migration related contracts and issues have been moved into the consequent security review done on Maple's "Liquidity Migration process", and therefore will be documented on that particular audit report.

Summary

Project Name	Maple Finance
Repository	Maple V2
Initial Commit	912fb38d924efdee4676d...
Type of Project	DeFi, Lending
Audit Timeline	Oct 17th - Nov 4th
Two weeks fix period	Nov 4th - Nov 18th
Methods	Manual Review

Second week scope change

Repository	Commit
maple-labs/globals-v2 (v1.0.0-rc.0)	3645455
maple-labs/liquidations (v2.0.0-rc.1) UPDATED	e6e01e2
maple-labs/loan (v4.0.0-rc.1) UPDATED	626a8d1
maple-labs/maple-proxy-factory (v1.1.0-rc.0)	ba01041
maple-labs/pool-v2 (v1.0.0-rc.1) UPDATED	a194647
maple-labs/withdrawal-manager (v1.0.0-rc.1) UPDATED	b3fdf27e

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	4	3	1
Low Risk	15	10	5
Gas Optimizations	9	9	0
Informational	23	21	2
Total	**52**	**44**	**8**

5 Findings

5.1 High Risk

5.1.1 First pool depositor can be front-run and have part of their deposit stolen

Severity: *High Risk*

Context: pool-v2::Pool.sol#L73

Description: The first deposit with a `totalSupply` of zero shares will mint shares equal to the deposited amount. This makes it possible to deposit the smallest unit of a token and profit off a rounding issue in the computation for the minted shares of the next depositor: `(shares_ * totalAssets()) / totalSupply_`

Example:

- The first depositor (victim) wants to deposit 2M USDC (2e12) and submits the transaction.
- The attacker front runs the victim's transaction by calling `deposit(1)` to get 1 share. They then transfer 1M USDC (`1e12`) to the contract, such that `totalAssets = 1e12 + 1`, `totalSupply = 1`.
- When the victim's transaction is mined, they receive `2e12 / (1e12 + 1) * totalSupply = 1` shares (rounded down from `1.9999...`).
- The attacker withdraws their 1 share and gets `3M USDC * 1 / 2 = 1.5M USDC`, making a 0.5M profit.

  During the migration, an `_initialSupply` of shares to be airdropped are already minted at initialization and are not affected by this attack.

Recommendation: Require a minimum initial shares amount for the first deposit by adjusting the initial mint (when `totalSupply == 0`) such that:

- mint an `INITIAL_BURN_AMOUNT` of shares to a dead address like address zero.
- only mint `depositAmount - INITIAL_BURN_AMOUNT` to the recipient.

`INITIAL_BURN_AMOUNT` needs to be chosen large enough such that the cost of the attack is not profitable for the attacker when minting low share amounts but low enough that not too many shares are stolen from the user. We recommend using an `INITIAL_BURN_AMOUNT` of `1000`.

Spearbit: Here's the script that I used to check the values for different scenarios.

Maple: What do you guys think of this approach? #210.

Basically, instead of stealing a small amount of assets from the first depositor to mint shares to a nonexistent address, thereby throwing those assets away forever, and asymmetrically harming just the first honest depositor, this PR just "assumes" an amount of shares was minted backed by an equal amount of assets. This way, the starting rate is still 1 share per asset, but no real assets are locked away behind any irredeemable shares.

Spearbit: It makes the code more complicated, you need to add this amount to assets and `totalSupply` everywhere and may not forget it. Now there's suddenly a discrepancy between `pool.totalAssets()` and `PM.totalAssets()`. It is also problematic when assets & `totalSupply` contain "virtual" balances that are not in the contract as it breaks invariants like `asset.balanceOf(pool) + LM.assetsUnderManagement() == pool.totalAssets()` and you need to make sure that it's impossible to transfer out these virtual balances, as it would revert. Which is an issue in your case.

Let's say you start with a virtual balance of 1000 supply / 1000 assets, then I deposit 1000 assets and get another 1000 shares. A loan is funded with my 1000 assets but they default and only 500 assets are recovered. Now I withdraw my 1000 shares and should get a transfer out of `shares * (totalAssets + BOOTSTRAP_MINT) / (totalSupply + BOOTSTRAP_MINT) = 1000 * (500 + 1000 virtual) / 2000 = 750` yet there's only 500 in the contract. This only happens because you have a virtual balance that is not backed by actual assets. If the first depositor deposited these funds you could get out your entire 750.

In our opinion, it's okay to "steal" less than 1 cent from the first depositor, they're already paying $5 in gas fees. What you could think about is to have the `PoolDeployer` always be the first depositor, keep a small funds balance in it, and do a `funds.transferFrom` + mint for them in the pool constructor.

Maple: This is our current working PR #219 note there are still some comments to be addressed but you can see the approach we are taking.

Merged PR: #219 ready for review.

Merged this as well to make the mint param globally configurable by the governor: #40.

Spearbit: Fixed.

5.2 Medium Risk

5.2.1 Users depositing to a pool with unrealized losses will take on the losses

Severity: *Medium Risk*

Context: pool-v2::Pool.sol#L278, pool-v2::Pool.sol#L275

Description: The pool share price used for deposits is always the `totalAssets() / totalSupply`, however the pool share price when redeeming is `totalAssets() - unrealizedLosses() / totalSupply`. The `unrealizedLosses` value is increased by loan impairments (`LM.impairLoan`) or when starting triggering a default with a liquidation (`LM.triggerDefault`). The `totalAssets` are only reduced by this value when the loss is realized in `LM.removeLoanImpairment` or `LM.finishCollateralLiquidation`.

This leads to a time window where deposits use a much higher share price than current redemptions and future deposits. Users depositing to the pool during this time window are almost guaranteed to make losses when they are realized. In the worst case, a `Pool.deposit` might even be (accidentally) front-run by a loan impairment or liquidation.

Recommendation: Make it very clear to the users when there are unrealized losses and communicate that it is a bad time to deposit. Furthermore, consider adding an `expectedMinimumShares` parameter that is checked against the actual minted shares. This ensures that users don't accidentally lose shares when front-run. Note that this would need to be a new `deposit(uint256 assets_, address receiver_, uint256 expectedMinimumShares_)` function to not break the ERC4626 compatibility.

> *The `Pool.mint` function has a similar issue, whereas the `Pool.mintWithPermit` function already accepts a `maxAssets_` parameter.*

Maple: Yes our team is aware of this issue and plan on making it very clear to users through our front end and documentation that it is not recommended depositing while there are unrealized losses. The alternative of not using two exchange rates introduces another vulnerability which is that users could front run a payment or reversion of an impairment and make a large amount off of the exchange rate change.

Spearbit: Acknowledged.

5.2.2 `TransitionLoanManager.add` does not account for accrued interest since last call

Severity: *Medium Risk*

Context: pool-v2::TransitionLoanManager.sol#L74

Description: The `TransitionLoanManager.add` advances the domain start but the accrued interest since the last domain start is not accounted for. It therefore wrongly tracks the `_accountedInterest` variable. If `add` is called several times, the accounting will be wrong.

Recommendation: Consider tracking the accrued interest or ensure that the `MigrationHelper.addLoansToLM` is called only once in the final migration script, adding all loans at the same time.

```
function add(address loan_) external override nonReentrant {
    ...
    uint256 domainStart_ = domainStart;
+   uint256 accruedInterest;
    if (domainStart_ == 0 || domainStart_ != block.timestamp) {
+       accruedInterest = getAccruedInterest();
        domainStart = _uint48(block.timestamp);
    }
    ...
+   _updateIssuanceParams(issuanceRate += newRate_, accountedInterest + accruedInterest);
}
```

This mimics `LoanManager._advanceGlobal` as long as there are no late payments but that's also the case for `TransitionLoanManager` as one of the preconditions for the migration is that loans have at least 5 days for any payment to be due.

Maple: In theory yes, but realistically we'll add all loans atomically. Even in the largest pool, we have around 30 active loans, which is feasible to do in one transaction. This is not an issue since all loans are added atomically, but we can add this functionality to be defensive on the TLM side.

Spearbit: Not fixed, see comment.

Maple: Here is the PR for the fix to your comment #218.

Spearbit: Fixed.

5.2.3 Unaccounted collateral is mishandled in `triggerDefault`

Severity: *Medium Risk*

Context: pool-v2::LoanManager.sol#L563

Description: The control flow of `triggerDefault` is partially determined by the value of `MapleLoanLike(loan_).collateral() == 0`. The code later assumes there are 0 collateral tokens in the loan if this value is true, which is incorrect in the case of *unaccounted* collateral tokens. In non-liquidating repossessions, this causes an overestimation of the number of `fundsAsset` tokens repossessed, leading to a revert in the `_disburseLiquidationFunds` function. Anyone can trigger this revert by manually transferring 1 Wei of `collateralAsset` to the loan itself. In liquidating repossessions, a similar issue causes the code to call the liquidator's `setCollateralRemaining` function with only accounted collateral, meaning unaccounted collateral will be unused/stuck in the liquidator.

Recommendation: In both cases, use the collateral token's `balanceOf` function to measure the amount of collateral tokens in the loan, for example:

```
- if (IMapleLoanLike(loan_).collateral() == 0 || IMapleLoanLike(loan_).collateralAsset() == fundsAsset)
↪  {
+ address collateralAsset_ = IMapleLoanLike(loan_).collateralAsset();
+ if (IERC20Like(collateralAsset_ ).balanceOf(loan_) == 0 || collateralAsset_== fundsAsset) {
```

Maple: Fixed in #211.

Spearbit: Fixed.

5.2.4 Initial cycle time is wrong when queuing several config updates

Severity: *Medium Risk*

Context: withdrawal-manager::WithdrawalManager.sol#L123

Description: The initial cycle time will be wrong if there's already an upcoming config change that changes the cycle duration.

Example:

```
currentCycleId: 100
config[0] = currentConfig = {initialCycleId: 1, cycleDuration = 1 days}
config[1] = {initialCycleId: 101, cycleDuration = 7 days}
```

Now, scheduling will create a config with `initialCycleId: 103` and `initialCycleTime = now + 3 * 1 days`, but the cycle durations for cycles (100, 101, 102) are 1 days + 7 days + 7 days.

Recommendation: Optimistically "apply" (just for the computation, not actually activate it) any pending configs for a cycle ID and then sum up the cycle durations for the cycles `[currentCycleId, currentCycleId + 1, currentCycleId + 2]`. Add the result to `getWindowStart(currentCycleId_)`.

Maple: Fixed in #50.

Spearbit: Fixed.

5.3 Low Risk

5.3.1 Users cannot resubmit a withdrawal request as per the wiki

Severity: *Low Risk*

Context: pool-v2::Pool.sol#L210-L224, pool-v2::PoolManager.sol#371-L382, withdrawal-manager::WithdrawalManager.sol#L151-L176,

Description: As per Maple's wiki:

- Refresh: The withdrawal request can be resubmitted with the same amount of shares by calling `pool.requestRedeem(0)`.

However, the current implementation prevents `Pool.requestRedeem()` from being called where the `shares_` parameter is zero.

Recommendation: Consider removing the below require statement found in `Pool._requestRedeem()`.

```
function _requestRedeem(uint256 shares_, address owner_) internal returns (uint256 escrowShares_) {
-     require(shares_ != 0, "P:RR:ZERO_SHARES");
...
```

Maple: Issue addressed in #PR 183.

Spearbit: The PR introduced an issue where anyone can DoS the withdrawal process by front-running calls to process a share redemption with a call to `requestRedeem()`, thereby refreshing the request before the user is able to process a withdrawal.

The fix here does not sufficiently cover the edge case where a non-owner account can arbitrarily push a user's request to redeem shares back by two cycles. Any call to process a redemption can be front-run because a new request can be made as long as `block.timestamp >= getWindowStart(exitCycleId_)`.

Making a new request to redeem shares with zero as a parameter can be done by any user as it does not require them to be the owner, nor does it expect the caller to have an approved amount.

If a user can perpetually front-run a user's attempt to redeem shares, it would be a valid DoS attack on the withdrawal manager.

Maple: But if you check out `Pool.sol` in the `_requestRedeem()` helper function we do check if the `msg.sender != owner` and use an allowance in that case Pool.sol#L230. Does this address your concern?

Spearbit: We would be explicitly requesting to redeem zero shares. The allowance checks will not revert because we don't need a pre-approved amount for this to execute.

Maple: We'll have stand up in a few hours so I can discuss with the team how best to fix it, do you folks have a suggestion?

Spearbit: The only suggestion that makes the most sense is to only allow the owner to refresh a request.

Maple: Fixed in #PR 232 by only allowing the owner or an approved EOA to refresh their request.

Spearbit: Fixed.

5.3.2 Accrued interest may be calculated on an overstated payment

Severity: *Low Risk*

Context: `migration-helpers::AccountingChecker.sol#L50-L51`

Description: The `checkTotalAssets()` function is a useful helper that may be used to make business decisions in the protocol. However, if there is a late loan payment, the total interest is calculated on an incorrect payment interval, causing the accrued interest to be overstated. It is also important to note that late interest will also be excluded from the total interest calculation.

Recommendation: Consider capping the time delta maximum to the payment interval. If it is also intended to include late interest, it may also be useful to add this calculation to the total interest amount.

Maple: Acknowledged, won't address as this contract will be only used during migration during which we will have no late loans.

Spearbit: Acknowledged.

5.3.3 No deadline when liquidating a borrower's collateral

Severity: *Low Risk*

Context: `liquidations::Liquidator.sol#L86-L103`

Description: A loan's collateral is liquidated in the event of a late payment or if the pool delegate impairs a loan due to insolvency by the borrower. If the loan contains any amount of collateral (assuming it is different to the funds' asset), the liquidation process will attempt to sell the collateral at a discounted amount.

Because a liquidation is considered active as long as there is remaining collateral in the liquidator contract, a user can knowingly liquidate all but `1 wei` of collateral. As there is no incentive for others to liquidate this dust amount, it is up to the loan manager to incur the cost and responsibility of liquidating this amount before they can successfully call `LoanManager.finishCollateralLiquidation()`.

Recommendation: Consider adding a deadline to the liquidation process. After this time period has passed, any leftover amount can be claimed by the protocol's treasury, allowing for liquidation finalization. This function hook can be added to the existing `LoanManager.finishCollateralLiquidation()` as an edge case.

Maple: Reevaluated, and will not address as it is solvable by performing a transaction to liquidate remaining amount. Inconvenient but not an issue really.

Spearbit: Acknowledged.

5.3.4 Loan impairments can be unavoidably unfair for borrowers

Severity: *Low Risk*

Context: loan::MapleLoan.sol#L859, pool-v2::LoanManager.sol#L233

Description: When a pool delegate impairs a loan, the loan's `_nextPaymentDueDate` will be set to the min of `block.timestamp` and the current `_nextPaymentDueDate`. If the pool delegate later decides to remove the impairment, the original `_nextPaymentDueDate` is restored to its correct value. The borrower can also remove an impairment themselves by making a payment. In this case, the `_nextPaymentDueDate` is *not* restored, which is always worse for the borrower. This can be unfair since the borrower would have to pay late interest on a loan that was never actually late (according to the original payment due date). Another related consequence is that a borrower can be liquidated before the original payment due date even passes (this is possible as long as the loan is impaired more than `gracePeriod` seconds away from the original due date).

Recommendation: Ensure that these subtleties are documented and made clear to borrowers. Also, consider mitigating this unfair behavior. This can be done by restoring `_nextPaymentDueDate` when a borrower makes a payment on an impaired loan, and by explicitly enforcing that a loan can never be liquidated before the next payment's original due date.

Maple: Discussed with business team and we want to keep as is, this will be in the loan agreements with the borrowers.

Spearbit: Acknowledged.

5.3.5 `withdrawCover()` vulnerable to reentrancy

Severity: *Low Risk*

Context: pool-v2 (v1.0.0-rc.1)PoolManager.sol#L407

Description: `withdrawCover()` allows for reentrancy and could be abused to withdraw below the minimum cover amount and avoid having to cover protocol insolvency through a bad liquidation or loan default.

The `moveFunds()` function could transfer the asset amount to the recipient specified by the pool delegate. Some tokens allow for callbacks before the actual transfer is made. In this case, the pool delegate could reenter the `withdrawCover()` function and bypass the balance check as it is made before tokens are actually transferred. This can be repeated to empty out required cover balance from the contract.

It is noted that the `PoolDelegateCover` contract is a protocol controlled contract, hence the low severity.

Recommendation: This can be mitigated by moving the balance check below the `moveFunds()` line:

```
@@ -397,15 +398,15 @@ contract PoolManager is IPoolManager, MapleProxiedInternals, PoolManagerStorage

 require(msg.sender == poolDelegate, "PM:WC:NOT_PD");
-    require(
-    amount_ <= (IERC20Like(asset).balanceOf(poolDelegateCover) -
↪  IMapleGlobalsLike(globals()).minCoverAmount(address(this))),
- "PM:WC:BELOW_MIN"
-    );
     recipient_ = recipient_ == address(0) ? msg.sender : recipient_;
     IPoolDelegateCoverLike(poolDelegateCover).moveFunds(amount_, recipient_);
+    require(
+    IERC20Like(asset).balanceOf(poolDelegateCover) >=
↪  IMapleGlobalsLike(globals()).minCoverAmount(address(this)),
+    "PM:WC:BELOW_MIN"
+    );
```

Maple: Fixed in #188.

Spearbit: Fixed.

5.3.6 Bad parameter encoding and deployment when using wrong initializers

Severity: *Low Risk*

Context: pool-v2 (v1.0.0-rc.1)::PoolDeployer.sol#L32-L77

Description: The initializers used to encode the arguments, when deploying a new pool in `PoolDeployer`, might not be the initializers that the proxy factory will use for the default version and might lead to bad parameter encoding & deployments if a wrong initializer is passed.

Recommendation: Should check the initializers while deploying a new pool:

```
function deployPool(
  address[3] memory factories_,
  address[3] memory initializers_,
  address asset_,
  string memory name_,
  string memory symbol_,
  uint256[6] memory configParams_
) external override returns (...)
{
  address poolDelegate_ = msg.sender;

  IMapleGlobalsLike globals_ = IMapleGlobalsLike(globals);

  require(globals_.isPoolDelegate(poolDelegate_), "PD:DP:INVALID_PD");

  require(globals_.isFactory("POOL_MANAGER",       factories_[0]), "PD:DP:INVALID_PM_FACTORY");
  require(globals_.isFactory("LOAN_MANAGER",       factories_[1]), "PD:DP:INVALID_LM_FACTORY");
  require(globals_.isFactory("WITHDRAWAL_MANAGER", factories_[2]), "PD:DP:INVALID_WM_FACTORY");

+ require(initializers[0] == factories[0].migratorForPath(factories[0].defaultVersion(),
↪  factories[0].defaultVersion()), "PD:DP:INVALID_PM_INITIALIZER");
+ require(initializers[1] == factories[1].migratorForPath(factories[1].defaultVersion(),
↪  factories[1].defaultVersion()), "PD:DP:INVALID_LM_INITIALIZER");
+ require(initializers[2] == factories[2].migratorForPath(factories[2].defaultVersion(),
↪  factories[2].defaultVersion()), "PD:DP:INVALID_WM_INITIALIZER");

  bytes32 salt_ = keccak256(abi.encode(poolDelegate_));
  ...
}
```

Maple: Fixed in #200.

Spearbit: Fixed.

5.3.7 Event `LoanClosed` might be emitted with the wrong value

Severity: *Low Risk*

Context: loan (v4.0.0-rc.1::MapleLoan.sol#L92-L124)

Description: In function `closeLoan` function, the fees are got by the `getClosingPaymentBreakdown` function and it is not adding refinances fees after in code are paid all fee by `payServiceFees` which may include refinances fees. The event `LoanClose` might be emitted with the wrong fee value.

Recommendation: It is recommended to change `getClosingPaymentBreakdown` by adding refinances fees:

```
function getClosingPaymentBreakdown() public view override returns (uint256 principal_, uint256
↪  interest_, uint256 fees_) {
  uint256 paymentsRemaining_  = _paymentsRemaining;

- uint256 delegateServiceFee_ = IMapleLoanFeeManager(_feeManager).delegateServiceFee(address(this)) *
↪  paymentsRemaining_;
- uint256 platformServiceFee_ = IMapleLoanFeeManager(_feeManager).platformServiceFee(address(this)) *
↪  paymentsRemaining_;
+ (
+   uint256 delegateServiceFee_,
+   uint256 delegateRefinanceServiceFee_,
+   uint256 platformServiceFee_,
+   uint256 platformRefinanceServiceFee_
+ ) = IMapleLoanFeeManager(_feeManager).getServiceFeeBreakdown(address(this), paymentsRemaining_);

- fees_ = delegateServiceFee_ + platformServiceFee_;
+ fees_ = delegateServiceFee_ + platformServiceFee_ + delegateRefinanceServiceFee_ +
↪  platformRefinanceServiceFee_;
  ...
}
```

Maple: Fixed in #238.

Spearbit: Fixed, should add a test for the event value.

5.3.8 Bug in makePayment() reverts when called with small amounts

Severity: *Low Risk*

Context: loan (v4.0.0-rc.1)::MapleLoan.sol#L167

Description: When `makePayment()` is called with an amount which is less than the fees payable, then the transaction will always revert, even if there is an adequate amount of drawable funds. The revert happens due to an underflow in `getUnaccountedAmount()` because the token balance is decremented on the previous line without updating drawable funds.

Recommendation: Consider updating the logic so that drawable funds are decremented by the fees before paying them:

```
@@ -160,11 +160,13 @@ contract MapleLoan is IMapleLoan, MapleProxiedInternals, MapleLoanStorage {

     uint256 principalAndInterest = principal_ + interest_;
-    IMapleLoanFeeManager(_feeManager).payServiceFees(_fundsAsset, 1);
// The drawable funds are increased by the extra funds in the contract, minus the total needed for
↪  payment.
// NOTE: This line will revert if not enough funds were added for the full payment amount.
-    _drawableFunds = (_drawableFunds + getUnaccountedAmount(_fundsAsset)) - principalAndInterest;
+    _drawableFunds = (_drawableFunds + getUnaccountedAmount(_fundsAsset)) - principalAndInterest -
↪  fees_;
+    require(IMapleLoanFeeManager(_feeManager).payServiceFees(_fundsAsset, 1) == fees_,
↪  "ML:MP:INCORRECT_FEES");
```

Alternatively, if this behavior is desired, consider updating the NatSpec and documentation and as well as a `require(amount >= fees))` check.

Maple: Fixed in #250.

Spearbit: Looks good. In the `_handleServiceFeePayment()` helper, what's the reason for the final `else` clause at the end?

#250-diff

```
if (balanceBeforeServiceFees_ > balanceAfterServiceFees_) {
    _drawableFunds -= (fees_ = balanceBeforeServiceFees_ - balanceAfterServiceFees_);
} else {
    _drawableFunds += balanceAfterServiceFees_ - balanceBeforeServiceFees_;
}
```

Not seeing how the `balanceAfterServiceFees` would ever be greater than `balanceBeforeServiceFees` when the only thing done in the middle is paying fees.

The balance will either decrease by the fees paid or remain the same if fees are zero. So technically this works since it will hit the `else` and then add `0` to the `_drawableFunds`.

Maple: This was just to be defensive in our design, this would never happen with the current loan. If for whatever reason the balance did increase, we simply return zero for `fees_` and increment drawableFunds.

Spearbit: Fixed.

5.3.9 `Pool.previewWithdraw` always reverts but `Pool.withdraw` can succeed

Severity: *Low Risk*

Context: withdrawal-manager::WithdrawalManager.sol#L400, pools-v2::Pool.sol#L117

Description: The `Pool.previewWithdraw => PM. previewWithdraw => WM.previewWithdraw` function call sequence always reverts in the `WithdrawalManager`. However, the `Pool.withdraw` function can succeed. This behavior might be unexpected, especially, as integrators call `previewWithdraw` before doing the actual `withdraw` call.

Recommendation: Either disable `Pool.withdraw` calls or implement `Pool.previewWithdraw` calls.

Maple: Yes this is an issue, we should be using `processWithdraw` in the Pool => PM and reverting it. We don't want to support `withdraw` because of the exact shares issue. We can add this change.

Spearbit: Refactor pool to use `requestWithdraw` and `processWithdraw`.

Solution to #68 leads to previewWithdraw always returning zero. However, can call withdraw with a specific amount so it does not revert and returns more than `0`. We do not consider the core issue of unifying the behavior of these two functions fully fixed.

Maple: We merged a fix in for the issue of `previewWithdraw` always returning issue in #53.

Spearbit: The fix in that PR is for a different issue. In #53, the fix is for the `previewRedeem` bug that was recently introduced where `&&` should have been `||`. The issue above is separate and is saying that `previewWithdraw` (on the pool) should not be reverting/returning `0` if the actual `withdraw` function *does* allow you to withdraw shares.

Maple: Fixed and merged in #225.

Spearbit: The PM now reverts for `Pool.requestWithdraw` & `Pool.withdraw`. The `previewWithdraw` function returns 0. Therefore, the behavior is still a little different but it's acceptable and indeed unclear if ERC4626 view functions should revert or return 0 in this case, see *IERC426 Implementation of `preview` and `max` functions* for more discussion.

> I find it a bit strange that `getEscrowParams` works for both `assets_` and `shares_` (see `_requestRedeem`). In an actual implementation, how are you going to differentiate if the second argument are shares or assets?

#225.

Please have a look, might be relevant for future PM upgrades that indeed return something different for each `getEscrowedParams` call.

Maple: Good catch we are just discussing this on our call we just want to convert and just pass along shares. We're debating if we should use `convertToShares` or `convertToExitShares` on the `assets_` then pass it along to `getEscrowParams`. We were wondering if you had a recommendation in regard to this for future upgrades?

Spearbit: What are your intentions with this escrow feature and how is it supposed to work? It should be `convertToExitShares` because to withdraw 1000 assets from the WM you need to burn 1000 assets converted to exit shares. So this amount should be escrowed.

Maple: That should cover it #229.

Spearbit: Fixed.

5.3.10 Setting a new WithdrawalManager locks funds in old one

Severity: *Low Risk*

Context: `pool-v2::PoolManager.sol#L237`

Description: The `WithdrawalManager` only accepts calls from a `PoolManager`. When setting a new withdrawal manager with `PoolManager.setWithdrawalManager`, the old one cannot be accessed anymore. Any user shares locked for withdrawal in the old one are stuck.

Recommendation: Before calling this function, ensure that there are no locked shares in the current withdrawal manager.

Maple: Acknowledged, will manage operationally and will capture in documentation. Addressed in Withdrawal-Mechanism#configuration-management.

Spearbit: Acknowledged.

5.3.11 Use `whenProtocolNotPaused` on `migrate()` instead of `upgrade()` for more complete protection

Severity: *Low Risk*

Context:

- Liquidator.sol
- MapleLoan.sol
- WithdrawalManager.sol

Description: `whenProtocolNotPaused` is added to `migrate()` for the `Liquidator`, `MapleLoan`, and `WithdrawalManager` contracts in order to protect the protocol by preventing it from upgrading while the protocol is paused. However, this protection happens only during upgrade, and not during instantiation.

Recommendation: Consider moving the `whenProtocolNotPaused` modifier from `upgrade()` to `migrate()` since `migrate()` is called during both the upgrade process and the instantiation process.

```
--- a/contracts/Liquidator.sol
+++ b/contracts/Liquidator.sol
@@ -52,7 +52,7 @@ contract Liquidator is ILiquidator, LiquidatorStorage, MapleProxiedInternals {
-    function migrate(address migrator_, bytes calldata arguments_) external override {
+    function migrate(address migrator_, bytes calldata arguments_) external override
↪   whenProtocolNotPaused {
         require(msg.sender == _factory(),        "LIQ:M:NOT_FACTORY");
         require(_migrate(migrator_, arguments_), "LIQ:M:FAILED");
     }
@@ -63,7 +63,7 @@ contract Liquidator is ILiquidator, LiquidatorStorage, MapleProxiedInternals {
         _setImplementation(implementation_);
     }
-    function upgrade(uint256 version_, bytes calldata arguments_) external override
↪   whenProtocolNotPaused {
+    function upgrade(uint256 version_, bytes calldata arguments_) external override {
```

Maple: Fixed in these PRs:

- #245

- #213
- #214
- #215
- #52
- #54

This actually has introduced a problem, if there is no migrator contract, this will not revert on pause for either upgrade or deploy. ProxyFactory.sol#L26. Thinking of the best approach to this, thinking it might be worth adding a pause in the MPF itself.

Spearbit: Adding a pause to the MPF would mean you can never deploy/upgrade any contract while the protocol is paused? But maybe you want to be able to do that for some contracts/migrations. If you want to do the same as now and keep the responsibility of whether you allow migrations during pause in the implementation contract, there's also another approach: Always call `proxy.migrate` in MLF even with a zero-address for initializer but then do nothing in `ProxiedInternals._migrate` (more specifically, return true if `migrator_ == 0` and `arguments_` is empty).

Maple: We have learnt towards adding the pause to MPF, in the case of a protocol pause we can always bundle transactions if we need to upgrade. Here is the PR for the update #38

To summarize, we will be reverting the changes in the rest of the repos to remove pausing functionality from both upgrade and migrate since they will be in the MPF. These reversions will be made everywhere except the Loan, since the `LoanFactory` is already deployed on mainnet.

#38 is merged.

#214 closed this as it is no longer relevant.

Spearbit: - pause added to MPF in #38.

- pause removed from PM/LM here: #222.
- pause removed from WM here: #54.
- pause removed from Liquidator here: #55.
- Loan now has pause on migrate or on upgrade like it was before? In current main, it's still on `upgrade` but ok. Just making sure you're aware in case you intended to change that.

Maple: Yes that change was intended and to have it on `upgrade` as we won't be updating the Loan factory in production. We're fine with new loans being created during a protocol pause.

Spearbit: Fixed.

5.3.12 Missing post-migration check in PoolManager.sol could result in lost funds

Severity: *Low Risk*

Context: `pool-v2 (v1.0.0-rc.1)::PoolManager.sol#L59-L62`

Description: The protocol employs an upgradeable/migrateable system that includes upgradeable initializers for factory created contracts. For the most part, a storage value that was left uninitialized due to an erroneous initializer would not be affect protocol funds. For example forgetting to initialize `_locked` would cause all `nonReentrant` functions to revert, but no funds lost.

However, if the `poolDelegateCover` address were unset and `depositCover()` were called, the funds would be lost as there is no `to != address(0)` check in transferFrom.

Recommendation: Consider adding an explicit check to the `migrate()` function:

```
function migrate(address migrator_, bytes calldata arguments_) external override {
     require(msg.sender == _factory(),        "PM:M:NOT_FACTORY");
     require(_migrate(migrator_, arguments_), "PM:M:FAILED");
+    require(poolDelegateCover != address(0),      "PM:M:DELEGATE_NOT_SET");
 }
```

Alternatively consider adding `to != address(0)` check to the `transferFrom()` for more robust protection.

Maple: Fixed in #204.

Spearbit: Fixed.

5.3.13 `Globals.poolDelegates[delegate_].ownedPoolManager` mapping can be overwritten

Severity: *Low Risk*

Context: globals-v2/MapleGlobals.sol#L112

Description: The `Globals.poolDelegates[delegate_].ownedPoolManager` keeps track of a *single* pool manager for a pool delegate. It can happen that the same pool delegate is registered for a second pool manager and the mapping is overwritten, by calling `PM.acceptPendingPoolDelegate -> Globals.transferOwnedPoolManager` or `Globals.activatePoolManager`.

Recommendation: Consider checking that the pool delegate does not own a pool manager yet.

```
function activatePoolManager(address poolManager_) external override isGovernor {
    address delegate_ = IPoolManagerLike(poolManager_).poolDelegate();
+   require(poolDelegates[delegate_].ownedPoolManager == 0, "MG:APM:ALREADY_OWNS");
    emit PoolManagerActivated(poolManager_, delegate_);
    poolDelegates[delegate_].ownedPoolManager = poolManager_;
}

function transferOwnedPoolManager(address fromPoolDelegate_, address toPoolDelegate_) external override
↪   {
    PoolDelegate storage fromDelegate_ = poolDelegates[fromPoolDelegate_];
    PoolDelegate storage toDelegate_   = poolDelegates[toPoolDelegate_];
    require(fromDelegate_.ownedPoolManager == msg.sender, "MG:TOPM:NOT_AUTHORIZED");
    require(toDelegate_.isPoolDelegate,                   "MG:TOPM:NOT_POOL_DELEGATE");
+   require(toDelegate_.ownedPoolManager == 0,            "MG:TOPM:ALREADY_OWNS");
    fromDelegate_.ownedPoolManager = address(0);
    toDelegate_.ownedPoolManager   = msg.sender;
    emit PoolManagerOwnershipTransferred(fromPoolDelegate_, toPoolDelegate_, msg.sender);
}
```

Maple: Fixed in #38.

Spearbit: Fixed.

5.3.14 Pool withdrawals can be kept low by non-redeeming users

Severity: *Low Risk*

Context: `withdrawal-manager::WithdrawalManager.sol#L331`

Description: In the current pool design, users request to exit the pool and are scheduled for a withdrawal window in the withdrawal manager. If the pool does not have enough liquidity, their share on the available pool liquidity is proportionate to the total shares of all users who requested to withdraw in that withdrawal window.

It's possible for griefers to keep the withdrawals artificially low by requesting a withdrawal but not actually withdrawing during the withdrawal window. These griefers are not penalized but their behavior leads to worse withdrawal amounts for every other honest user.

Recommendation: There's no straightforward solution in the current design of the withdrawal process. Short-term, monitor the withdrawal situation for this issue. Long-term, consider enforcing withdrawals during the withdrawal window or penalizing withdrawers that requested a withdrawal but did not withdraw.

Maple: Yes we are aware of this and discussed it during our design phase. We will continuously evaluate this over time and determine if a WM upgrade is necessary with an alternative design.

Spearbit: Acknowledged.

5.3.15 `_getCollateralRequiredFor` should round up

Severity: *Low Risk*

Context: `loan::MapleLoan.sol#L700`

Description: The `_getCollateralRequiredFor` rounds down the collateral that is required from the borrower. This benefits the borrower.

Recommendation: Consider rounding up.

```
- (collateralRequired_ * (principal_ - drawableFunds_)) / principalRequested_
+ (collateralRequired_ * (principal_ - drawableFunds_) + principalRequested_ - 1) / principalRequested_
```

Maple: Fixed in #243.

Spearbit: Fixed.

5.4 Gas Optimization

5.4.1 Use the cached variable in `makePayment`

Severity: *Gas Optimization*

Context: `loan (v4.0.0-rc.1):: MapleLoan.sol#L185`

Description: The `claim` function is called using `_nextPaymentDueDate` instead of `nextPaymentDueDate_`

Recommendation: Should use cached `nextPaymentDueDate_` variable to save gas.

```
- ILenderLike(_lender).claim(principal_, interest_, previousPaymentDueDate_, _nextPaymentDueDate);
+ ILenderLike(_lender).claim(principal_, interest_, previousPaymentDueDate_, nextPaymentDueDate_);
```

Maple: Fixed in #237.

Spearbit: Fixed.

5.4.2 No need to explicitly initialize variables with default values

Severity: *Gas Optimization*

Context: pool-v2 (v1.0.0-rc.1)::LoanManager.sol#L367 pool-v2 (v1.0.0-rc.1)::PoolManager.sol#L196

Description: By default a value of a variable is set to `0` for `uint`, `false` for `bool`, `address(0)` for `address`... Explicitly initializing/setting it with its default value wastes gas.

Recommendation: In `LoanManager.sol` is recommended to remove line 367:

```
-  liquidationComplete_ = false;
```

In `PoolManager.sol`

```
- uint256 i_ = 0;
+ uint256 i_;
```

Maple: Fixed in #193.

Spearbit: Fixed.

5.4.3 Cache calculation in `getExpectedAmount`

Severity: *Gas Optimization*

Context: pool-v2 (v1.0.0-rc.1)::LoanManager.sol#L850 pool-v2 (v1.0.0-rc.1)::LoanManager.sol#L853

Description: The decimal precision calculation is used twice in the `getExpectedAmount` function, if you cache into a new variable would save some gas.

Recommendation: Follow the code to fix this issue:

```
- uint8 collateralAssetDecimals_ = IERC20Like(collateralAsset_).decimals();
+ uint256 collateralAssetDecimals_ = uint256(10) ** uint256(IERC20Like(collateralAsset_).decimals());

uint256 oracleAmount =
  swapAmount_
    * IMapleGlobalsLike(globals_).getLatestPrice(collateralAsset_)  // Convert from `fromAsset` value.
    * uint256(10) ** uint256(IERC20Like(fundsAsset).decimals())     // Convert to `toAsset` decimal
    ↪  precision.
    * (HUNDRED_PERCENT - allowedSlippageFor[collateralAsset_])      // Multiply by allowed slippage
    ↪  basis points
    / IMapleGlobalsLike(globals_).getLatestPrice(fundsAsset)        // Convert to `toAsset` value.
-   / uint256(10) ** uint256(collateralAssetDecimals_)              // Convert from `fromAsset` decimal
↪  precision.
+   / collateralAssetDecimals_                                      // Convert from `fromAsset` decimal
↪  precision.
    / HUNDRED_PERCENT;                                              // Divide basis points for slippage.

- uint256 minRatioAmount = (swapAmount_ * minRatioFor[collateralAsset_]) / (uint256(10) **
↪  collateralAssetDecimals_);
+ uint256 minRatioAmount = (swapAmount_ * minRatioFor[collateralAsset_]) /  collateralAssetDecimals_;
```

Maple: Fixed in #194.

Spearbit: Fixed.

5.4.4 For-Loop Optmization

Severity: *Gas Optimization*

Context: pool-v2::TransitionLoanManager.sol#L103 pool-v2::TransitionLoanManager.sol#L111 pool-v2 (v1.0.0-rc.1)::PoolManager.sol#L542 pool-v2 (v1.0.0-rc.1)::PoolManager.sol#L595

Description: The for-loop can be optimized in 4 ways:

1. Removing initialization of loop counter if the value is 0 by default.
2. Caching array length outside the loop.
3. Prefix increment (++i) instead of postfix increment (i++).
4. Unchecked increment.

```
- for (uint256 i_ = 0; i_ < loans_.length; i_++) {
+ uint256 length = loans_.length;
+ for (uint256 i_; i_ < length; ) {
  ...
+ unchecked { ++i; }
}
```

Recommendation: Optimize the for-loops.

Maple: Fixed in #195.

Spearbit: There is one more optimization in PR:

```
- for (uint256 i_ = 0; i_ < length_;) {
+ for (uint256 i_; i_ < length_;) {
```

Maple: Updated in #221.

Spearbit: Fixed

5.4.5 `Pool._divRoundUp` can be more efficient

Severity: *Gas Optimization*

Context: pool-v2::Pool.sol#L195

Description: The gas cost of `Pool._divRoundUp` can be reduced in the context that it's used in.

Recommendation: Consider changing to:

```
function _divRoundUp(uint256 numerator_, uint256 divisor_) internal pure returns (uint256 result_) {
-   result_ = (numerator_ / divisor_) + (numerator_ % divisor_ > 0 ? 1 : 0);
+   result_ = (numerator_ + divisor_ - 1) / divisor_;
}
```

Maple: Fixed in #201.

Spearbit: Fixed.

5.4.6 Liquidator uses different reentrancy guards than rest of codebase

Severity: *Gas Optimization*

Context: liquidations::Liquidator.sol#L28

Description: All other reentrancy guards of the codebase use values `1/2` instead of `0/1` to indicate `NOT_LOCKED/LOCKED`.

Recommendation: Consider using `1/2` values as well for gas efficiency reasons and to unify the codebase. It's important to update the `LiquidatorInitializer._initialize` function and set it to the non-zero NOT_LOCKED value then.

Maple: Fixed in #52.

Spearbit: Fixed.

5.4.7 Use `block.timestamp` instead of `domainStart` in `removeLoanImpairment`

Severity: *Gas Optimization*

Context: pool-v2::LoanManager.sol#L291

Description: The `removeLoanImpairment` function adds back all interest from the payment's start date to `domainStart`. The `_advanceGlobalPaymentAccounting` sets `domainStart` to `block.timestamp`.

Recommendation: Consider accruing the interest from `paymentInfo_.startDate` to `block.timestamp` directly to make the code easier to understand and a gas improvement.

```
// Discretely update missing interest as if payment was always part of the list.
_updateIssuanceParams(
  issuanceRate + paymentInfo_.issuanceRate,
- accountedInterest + _uint112(_getPaymentAccruedInterest(paymentInfo_.startDate, domainStart,
↪  paymentInfo_.issuanceRate, paymentInfo_.refinanceInterest))
+ accountedInterest + _uint112(_getPaymentAccruedInterest(paymentInfo_.startDate, block.timestamp,
↪  paymentInfo_.issuanceRate, paymentInfo_.refinanceInterest))
);
```

Maple: Fixed in #206.

Spearbit: Fixed.

5.4.8 `setTimelockWindows` checks `isGovernor` multiple times

Severity: *Gas Optimization*

Context: globals-v2/MapleGlobals.sol#L241-L246

Description: The `Globals.setTimelockWindows` function calls `setTimelockWindow` in a loop and each time `setTimelockWindow`'s `isGovernor` is checked.

Recommendation: Only check `isGovernor` once to optimize the function. Consider creating and calling an internal `_setTimeLockWindow` function that does not call this modifier.

Maple: Fixed in #39.

Spearbit: Fixed, just a potential cleanup: `setTimelockWindow` could also call the internal `_setTimelockWindow`

5.4.9 `fullDaysLate` computation can be optimized

Severity: *Gas Optimization*

Context: `loan::MapleLoan.sol#L843`

Description: The `fullDaysLate` computation can be optimized.

Recommendation: Consider changing to:

```
- (((currentTime_ - nextPaymentDueDate_ - 1) / 1 days) + 1) * 1 days
+ ((currentTime_ - nextPaymentDueDate_ + (1 days - 1)) / 1 days) * 1 days
```

Maple: Fix PR added #248.

Spearbit: Fixed.

5.5 Informational

5.5.1 Users can prevent repossessed funds from being claimed

Severity: *Informational*

Context: `debt-locker::DebtLocker.sol#L325-L329`

Description: The `DebtLocker.sol` contract dictates an active liquidation by the following *two* conditions:

- The `_liquidator` state variable is a non-zero address.
- The current balance of the `_liquidator` contract is non-zero.

If an arbitrary user sends `1 wei` of funds to the liquidator's address, the borrower will be unable to claim repossessed funds as seen in the `_handleClaimOfRepossessed()` function.

> While the scope of the audit only covered the diff between `v3.0.0` and `v4.0.0-rc.0`, the audit team decided it was important to include this as an `informational` issue. The Maple team will be addressing this in their V2 release.

Recommendation: Consider using `collateralRemaining` as an indicator for when a liquidation has finished.

Maple: Acknowledged, won't address because `DebtLockers` are getting deprecated upon launch and migration.

Spearbit: Acknowledged.

5.5.2 MEV whenever `totalAssets` jumps

Severity: *Informational*

Context: `pool-v2::Pool.sol#L278`, `pool-v2::Pool.sol#L275`

Description: An attack users can try to capture large interest payments is sandwiching a payment with a deposit and a withdrawal. The current codebase tries to mostly eliminate this attack by:

- Optimistically assuming the next interest payment will be paid back and accruing the interest payment linearly over the payment interval.
- Adding a withdrawal period.

However, there are still circumstances where the `totalAssets` increase by a large amount at once:

- Users paying back their payment early. The jump in `totalAssets` will be the `paymentAmount - timeElapsedSincePaymentStart / paymentInterval * paymentAmount`.
- Users paying back their entire loan early (`closeLoan`).
- Late payments increase it by the late interest fees and the accrued interest for the next payment from its start date to now.

Recommendation: These opportunities are rather rare and hard to completely mitigate because they are under the borrower's control. In a future version of the protocol, consider streaming single large interest payments to the pool over a certain time.

Maple: We are aware that it is possible, but with our accounting mechanism for the `LoanManager`, the `WithdrawalManager`, and sufficient diversification in the loan portfolio, this issue will be very minor in terms of percent value change. We acknowledge and will not address.

Spearbit: Acknowledged.

5.5.3 Use ERCHelper `approve()` as best practice

Severity: *Informational*

Context: `loan (v4.0.0-rc.1)::LoanManager.sol#L317`

Description: The ERC20 `approve` function is being used by `fundsAsset` in `fundLoan()` to approve the max amount which does not check the return value.

Recommendation: Use ERCHelper `approve()` as a best practice.

Maple: Fixed in #236.

Spearbit: Fixed.

5.5.4 Additional verification in `removeLoanImpairment`

Severity: *Informational*

Context: `pool-v2::LoanManager.sol#L266`

Description: Currently, if `removeLoanImpairment` is called *after* the loan's original due date, there will be no issues because the loan's `removeLoanImpairment` function will revert. It would be good to add a comment about this logic or duplicate the check explicitly in the loan manager. If the loan implementation is upgraded in the future to have a non-reverting `removeLoanImpairment` function, then the loan manager as-is would account for the interest incorrectly.

Recommendation: Consider adding a comment to the loan manager such as:

```
+ // NOTE: This call will revert if we are past the original due date
  IMapleLoanLike(loan_).removeLoanImpairment();
```

Also, consider duplicating the check explicitly in the loan manager.

Maple: Fixed in #192.

Spearbit: Fixed.

5.5.5 Can check `msg.sender != collateralAsset/fundsAsset` for extra safety

Severity: *Informational*

Context: `liquidations(v2.0.0-rc.1)::Liquidator.sol#L93`

Description: Some old ERC tokens (e.g. the Sandbox's `SAND` token) allow arbitrary calls from the token address itself. This odd behavior is usually a result of implementing the ERC677 `approveAndCall` and `transferAndCall` functions incorrectly. With these tokens, it is technically possible for the low-level `msg.sender.call(...)` in the `liquidator` to be executing arbitrary code on one of the tokens, which could let an attacker drain the funds.

Recommendation: Although it is very unlikely for Maple to whitelist one of these tokens as the collateral/asset, consider adding an explicit check to the `liquidatePortion` function:

```
+ require(msg.sender != collateralAsset && msg.sender != fundsAsset);
```

It is also recommended keeping this in mind when whitelisting future tokens.

Maple: Fixed in #51.

Spearbit: Fixed.

5.5.6 IERC426 Implementation of `preview` and `max` functions

Severity: *Informational*

Context: `IERC4626.previewDeposit()` `IERC4626.previewMint()` `IERC4626.previewRedeem()` `IERC4626.previewWithdrawal()`

Description: For the `preview` functions, EIP 4626 states:

> MAY revert due to other conditions that would also cause the deposit [mint/redeem, etc.] to revert.

But the comments in the interface currently state:

> MUST NOT revert.

In addition to the comments, there is the actual behavior of the preview functions. A commonly accepted interpretation of the standard is that these `preview` functions should revert in the case of conditions such as `protocolPaused`, `!active`, `!openToPublic` `totalAssets > liquidityCap` etc. The argument basically states that the `max` functions should return `0` under such conditions and the `preview` functions should revert whenever the `amount` exceeds the `max`.

Recommendation: At a minimum, consider clarifying the NatSpec. Also, carefully consider the behavior of the `preview` functions. As an early adopter of 4626, decisions by Maple now could have ripple effects on the industry.

Maple: Fixed in #51.

Spearbit: The team decided to return 0 for `preview*` functions instead of reverting. There's a bug in the PR #51#discussion_r1020201462 'previewWithdraw' also always returns 0 but it should be possible to call 'withdraw' in a way to return non-zero assets. See *`Pool.previewWithdraw` always reverts but `Pool.withdraw` can succeed* for a related issue to unify the behavior of these two functions.

Did you decide not to account for `protocolPaused` and `_canDeposit()` in the preview / max functions?

Maple: We merged a fix that you mentioned with this PR #53.

We decided not to have any reverts for the preview functions as we believe that would be better for integrators. Also, the spec states "May revert" in those conditions such as a protocol pause so we believe we are still in spec if we are choosing not to revert.

Spearbit: That should be ok. There is so much ambiguity in the standard I think it's fine to make this judgment call which reduces complexity in the code of integrators.

One small nit, you may want to consider updating the NatSpec on the preview functions in the interface as mentioned in #68#issue-1435405355. It says "MUST NOT REVERT" but the standard says "MAY REVERT" so I could see some confusion around that.

You could either change that to "MAY REVERT" or write something like "Maple has decided that these will not revert" or even just delete the line that says "MUST NOT REVERT" on the preview() functions would eliminate any future confusion.

Maple: Acknowledged we'll make the update and come back to you folks. Fixed in #226.

Spearbit: Fixed.

5.5.7 Set `domainEnd` correctly in intermediate `_advanceGlobalPaymentAccounting` steps

Severity: *Informational*

Context: pool-v2::LoanManager.sol#L675

Description: In the `_advanceGlobalPaymentAccounting` function, `domainEnd` is set to `payments[paymentWithEarliestDueDate].paymentDueDate`, which is possibly zero if the last payment has just been accrued past. This is currently not an issue, because in this scenario `domainEnd` would never be used before it is set back to its correct value in `_updateIssuanceParams`. However, for increased readability, it is recommended to prevent this odd intermediate state from ever occurring.

Recommendation: In `_advanceGlobalPaymentAccounting`, copy the same logic that `_updateIssuanceParams` has for setting `domainEnd`:

```
- domainEnd_   = payments[paymentWithEarliestDueDate].paymentDueDate;
+ domainEnd_ = paymentWithEarliestDueDate == 0
+     ? _uint48(block.timestamp)
+     : payments[paymentWithEarliestDueDate].paymentDueDate;
```

Maple: Fixed in #190.

Spearbit: Fixed.

5.5.8 Replace hard-coded value with `PRECISION` constant

Severity: *Informational*

Context: pool-v2::LoanManager.sol#L468

Description: The constant `PRECISION` is equal to `1e30`. The hard-coded value `1e30` is used in the `_queueNextPayment` function, which can be replaced by `PRECISION`.

Recommendation: Change `1e30` to `PRECISION`:

```
- uint256 incomingNetInterest_ = newRate_ * (nextPaymentDueDate_ - startDate_) / 1e30;  // NOTE: Use
↪  issuanceRate to capture rounding errors.
+ uint256 incomingNetInterest_ = newRate_ * (nextPaymentDueDate_ - startDate_) / PRECISION;  // NOTE:
↪  Use issuanceRate to capture rounding errors.
```

Maple: Fixed in #191.

Spearbit: Fixed.

5.5.9 Use of floating pragma version

Severity: *Informational*

Context: globals-v2 (v1.0.0-rc.0)::IMapleGlobals.sol#L2 globals-v2 (v1.0.0-rc.0)::Interfaces.sol#L2

Description: Contracts should be deployed using a fixed `pragma` version. Locking the `pragma` helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Recommendation: Lock the `pragma` version and also consider upgrading the project `pragma` version to a newer stable version, currently 0.8.7.

Maple: PR for the fix #37.

Spearbit: Fixed.

5.5.10 `PoolManager` has low-level shares computation logic

Severity: *Informational*

Context: pool-v2::PoolManager.sol#L570, pool-v2::PoolManager.sol#L578

Description: The `PoolManager` has low-level shares computation logic that should ideally only be in the ERC4626 Pool to separate the concerns.

Recommendation: Consider refactoring the `PoolManager` to call functions on the `Pool` instead:

maxMint

```
function maxMint(address receiver_) external view virtual override returns (uint256 maxShares_) {
    uint256 totalAssets_ = totalAssets();
    uint256 totalSupply_ = IPoolLike(pool).totalSupply();
    uint256 maxAssets_   = _getMaxAssets(receiver_, totalAssets_);

-   maxShares_ = totalSupply_ == 0 ? maxAssets_ : maxAssets_ * totalSupply_ / totalAssets_;
+   maxShares_ = pool.previewDeposit(maxAssets_)
}
```

maxWithdraw

```
function maxWithdraw(address owner_) external view virtual override returns (uint256 maxAssets_) {
  uint256 lockedShares_ = IWithdrawalManagerLike(withdrawalManager).lockedShares(owner_);
  uint256 maxShares_    = IWithdrawalManagerLike(withdrawalManager).isInExitWindow(owner_) ?
  ↪  lockedShares_ : 0;
- maxAssets_            = maxShares_ * (totalAssets() - unrealizedLosses()) /
↪  IPoolLike(pool).totalSupply();
+ maxAssets_ = pool.convertToExitAssets(maxShares);
}

// in Pool, add this function
+ function convertToExitAssets(uint256 shares_) public view override returns (uint256 assets_) {
+   assets_ = shares_ * (totalAssets() - unrealizedLosses()) / totalSupply;
+ }
```

Maple: Fixed in #208.

Spearbit: Fixed.

5.5.11 Add additional checks to prevent refinancing/funding a closed loan

Severity: *Informational*

Context: pool-v2::PoolManager.sol#L432

Description: It's important that an already liquidated loan is not reused by refinancing or funding again as it would break a second liquidation when the second liquidator contract is deployed with the same arguments and salt.

Recommendation: Consider explicitly checking for this scenario in the `PoolManager._validateAndFundLoan` function. After `closeLoan`, making the last payment with `makePayment`, or a liquidation's `repossess` call, `_clearLoanAccounting` is called. For example, check that the loan's `_paymentsRemaining` is not zero.

Maple: Fixed in #203.

Spearbit: Fixed.

5.5.12 `PoolManager.removeLoanManager` errors with out-of-bounds if loan manager not found

Severity: *Informational*

Context: pool-v2::PoolManager.sol#L188

Description: The `PoolManager.removeLoanManager` errors with an out-of-bounds error if the loan manager is not found.

Recommendation: Consider reverting with a more expressive error.

```
function removeLoanManager(address loanManager_) external override {
  _whenProtocolNotPaused();

  require(msg.sender == poolDelegate, "PM:RLM:NOT_PD");
+ require(isLoanManager[loanManager_], "PM:RLM:INVALID_LM");

  ...
}
```

Maple: Fixed in #196.

Spearbit: Fixed.

5.5.13 `PoolManager.removeLoanManager` does not clear `loanManagers` mapping

Severity: *Informational*

Context: pool-v2::PoolManager.sol#L188

Description: The `PoolManager.removeLoanManager` does not clear the reverse `loanManagers[mapleLoan] = loanManager` mapping.

Recommendation: In the current version there's no efficient way to enumerate all `mapleLoans` that have the removed `loanManager` set. As this function is just a contingency function, consider simply providing the loans to be cleared as an additional argument.

Maple: Acknowledged, will need team discussion on whether we implement. Fixed in #212

Spearbit: The `loanManagers` field has been removed from the PM. The `loan -> loanManager` mapping is now done by calling `loan.lender()` on verified loans. Note that when the loan manager is removed by `removeLoanManager` , loans will still return the removed loan manager through `loan.lender()`. I'll assume that this is indeed the desired behavior for now and set this to fixed.

5.5.14 `Pool._requestRedeem` reduces the wrong approval amount

Severity: *Informational*

Context: pool-v2::Pool.sol#L213

Description: The `requestRedeem` function transfers `escrowShares_` from `owner` but reduces the approval by `shares_`. Note that in the current code these values are the same but for future `PoolManager` upgrades this could change.

Recommendation: Reduce the approval by `escrowShares_`.

Maple: Fixed in #202.

Spearbit: Fixed.

5.5.15 Issuance rate for double-late claims does not need to be updated

Severity: *Informational*

Context: pool-v2::LoanManager.sol#L223

Description: The `previousRate_` for the 8c) case in `claim` is always zero because the payment is late (`!onTimePayment_`). The subtraction can be removed

I'd suggest removing the subtraction here as it's confusing. The first payment's IR was reduced in `_advanceGlobalPaymentAccounting`, the newly scheduled one that is also past due date never increased the IR.

Recommendation: The first payment's IR was reduced in `_advanceGlobalPaymentAccounting`, the newly scheduled payment that is also past the due date never increased the IR. For readability, consider changing the code to:

```
// 8c. If the current timestamp is greater than the RESULTING `nextPaymentDueDate`, then the next
↪  payment must be
//     FULLY accounted for, and the new payment must be removed from the sorted list.
//     Payment `issuanceRate` is used for this calculation as the issuance has occurred in isolation
↪  and entirely in the past.
//     All interest from the aggregate issuance rate has already been accounted for in
↪  `_advanceGlobalPaymentAccounting`.
else {
    ( uint256 accountedInterestIncrease_, ) = _accountToEndOfPayment(paymentIdOf[msg.sender], newRate_,
    ↪  previousPaymentDueDate_, nextPaymentDueDate_);

    return _updateIssuanceParams(
-       issuanceRate - previousRate_,
+       issuanceRate,
        accountedInterest + _uint112(accountedInterestIncrease_)
    );
}
```

Maple: Fixed in #199.

Spearbit: Fixed.

5.5.16 Additional verification that `paymentIdOf[loan_]` is not 0

Severity: *Informational*

Context: pool-v2::LoanManager.sol

Description: Most functions in the loan manager use the value `paymentIdOf[loan_]` without first checking if it's the default value of 0. Anyone can pay off a loan at any time to cause the `claim` function to set `paymentIdOf[loan_]` to 0, so even the privileged functions could be front-run to call on a loan with `paymentIdOf` 0. This is not an issue in the current codebase because each function would revert for some other reasons, but it is recommended to add an explicit check so future upgrades on other modules don't make this into a more serious issue.

Recommendation: Each time `paymentIdOf[loan_]` is used in a function, ensure that the value is non-zero before proceeding.

Maple: Fixed in #207.

Spearbit: Looks good to me, a question about this error string, not sure why it's IL (`impairLoan`?) when it's called in `_handleXYZ`. Fixed.

5.5.17 `LoanManager` redundant check on late payment

Severity: *Informational*

Context: pool-v2::LoanManager.sol#L208

Description: The `claim` function has a check for `block.timestamp > previousPaymentDueDate_ && block.timestamp <= nextPaymentDueDate_` in one of the `if` statements. The payment is already known to be late at this point in the code, so `block.timestamp > previousPaymentDueDate_` is always true.

Recommendation: Consider removing the check in the code for increased readability and a small gas saving.

```
- if (block.timestamp > previousPaymentDueDate_ && block.timestamp <= nextPaymentDueDate_) {
+ if (block.timestamp <= nextPaymentDueDate_) {
```

Maple: Fixed in #198.

Spearbit: Fixed.

5.5.18 Add `encodeArguments/decodeArguments` to `WithdrawalManagerInitializer`

Severity: *Informational*

Context: withdrawal-manager::WithdrawalManagerInitializer.sol pool-v2 (v1.0.0-rc.1)::PoolDeployer.sol#L68

Description: Unlike the other `Initializers`, the `WithdrawalManagerInitializer.sol` does not have public `encodeArguments/decodeArguments` functions, and `PoolDeployer` need to be changed to use these functions correctly

Recommendation: Consider adding these functions and using them in `PoolDeployer`.

```
//PoolDeployer.sol
function deployPool(...)
    external override returns (...)
{
  ...
  // Deploy Withdrawal Manager
- arguments          = abi.encode(pool_, configParams_[3], configParams_[4]);
+ arguments          = IWithdrawalManagerInitializerLike(initializers_[2]).encodeArguments(pool_,
↪  configParams_[3], configParams_[4]);
  withdrawalManager_ = IMapleProxyFactory(factories_[2]).createInstance(arguments, salt_);
  ...
}
```

Maple: Fixed in #205 and #49.

Spearbit: Fixed.

5.5.19 Reorder `WM.processExit` parameters

Severity: *Informational*

Context: withdrawal-manager::WithdrawalManager.sol#L210

Description: All other WM and Pool function signatures start with `(uint256 shares/assets, address owner)` parameters but the `WM.processExit` has its parameters reversed (`address, uint256`).

Recommendation: Consider reordering the parameters for consistency and rename the `account_` parameter to `owner_`.

Maple: Fixed in #48.

Spearbit: Only have visibility into the PR for the WM, but it should require changes in the pool module. Do you have the PR for the pool module?

Maple: The accompanying PR on pools-v2 #217.

Spearbit: Fixed.

5.5.20 Additional verification in `MapleLoanInitializer`

Severity: *Informational*

Context: `loan::MapleLoanInitializer.sol#L87`

Description: The `MapleLoanInitializer` could verify additional arguments to avoid bad pool deployments.

Recommendation: Consider verifying:

```
require(IMapleGlobalsLike(globals_).isPoolAsset(assets_[1]), "");
require(_paymentInterval > 0, "");
require(_paymentsRemaining > 0, "");
```

Maple: Fixed in #244.

Spearbit: Fixed.

5.5.21 Clean up `updatePlatformServiceFee`

Severity: *Informational*

Context: `loan::MapleLoanFeeManager.sol#L109-L110`

Description: The `updatePlatformServiceFee` can be cleaned up to use an existing helper function

Recommendation: Consider changing the code to:

```
function updatePlatformServiceFee(uint256 principalRequested_, uint256 paymentInterval_) external
↪  override {
-   uint256 platformServiceFeeRate_ =
↪  IMapleGlobalsLike(globals).platformServiceFeeRate(_getPoolManager(msg.sender));
-   uint256 platformServiceFee_     = principalRequested_ * platformServiceFeeRate_ * paymentInterval_
↪  / 365 days / HUNDRED_PERCENT;
+   uint256 platformServiceFee_ = getPlatformServiceFeeForPeriod(msg.sender, principalRequested_,
↪  paymentInterval_);

    platformServiceFee[msg.sender] = platformServiceFee_;

    emit PlatformServiceFeeUpdated(msg.sender, platformServiceFee_);
}
```

Maple: Fixed in #242.

Spearbit: Fixed.

5.5.22 Document restrictions on `Refinancer`

Severity: *Informational*

Context: `loan::MapleLoan.sol#L290`, `loan::Refinancer.sol`

Description: The refinancer may not set unexpected storage slots, like changing the `_fundsAsset` because `_drawableFunds`, `_refinanceInterest` are still measured in the old fund's asset.

Recommendation: The provided `Refinancer` does indeed not allow this but in theory, any refinancer can be used. It's good to document restrictions on `Refinancer`s or even enforce them in code by doing pre/post checks.

Maple: Addressed in Refinancing.

Spearbit: Docs mention it so consider this fixed:

> Note that the `Refinancer` contract should never set the `collateralAsset` or `fundsAsset` and should never directly set `drawableFunds`, principal or collateral.

It still talks about `DebtLocker` which seems outdated.

> This call is permissioned in the `DebtLocker` to only be called by the `PoolDelegate`.

5.5.23 Typos / Incorrect documentation

Severity: *Informational*

Context: See below.

Description: The code and comments contain typos or are sometimes incorrect.

Recommendation: Consider fixing the typos:

- `loan::MapleLoan.sol#L93`: "to be transfer" => "to be transferred"
- `withdrawal-manager::WithdrawalManager.sol#L222`: The comment is incorrect, it only transfer the redeemable shares. "Transfer both returned shares and redeemable shares, burn only the redeemable shares in the pool." => "Transfer redeemable shares, burn the redeemable shares in the pool, relock remaining shares."

Maple: Fixed in #241 and #47.

Spearbit: Fixed.