Maple Findings Workshop

Presentation by Riley Holterhus

Maple V2

MAPLE

- **Spearbit audit:** October 17 - November 4th
  - Christoph Michel, 0xleastwood, Riley Holterhus, Devtooligan, Jonatas Martins
- **Mainnet launch:** December 14th
- **Some changes from V1**:
  - Improved withdrawal mechanics
  - Simplified “First Loss Capital”
  - Adopting ERC4626 standard

First pool depositor front-running | *High Risk*

- Maple pools adopt the ERC4626 “tokenized vault standard”
- Essentially, can deposit tokens to get vault shares, can burn vault shares to get fraction of the underlying tokens

```
function deposit(uint256 assets_, address receiver_) external returns (uint256 shares_) {
    _mint(shares_ = previewDeposit(assets_), assets_, receiver_, msg.sender);
}


function _mint(uint256 shares_, uint256 assets_, address receiver_, address caller_) internal {
    require(receiver_ != address(0), "P:M:ZERO_RECEIVER");
    require(shares_   != uint256(0), "P:M:ZERO_SHARES");
    require(assets_   != uint256(0), "P:M:ZERO_ASSETS");

    _mint(receiver_, shares_);

    emit Deposit(caller_, receiver_, assets_, shares_);

    require(ERC20Helper.transferFrom(asset, caller_, address(this), assets_), "P:M:TRANSFER_FROM");
}

function previewDeposit(uint256 assets_) public returns (uint256 shares_) {
    // As per https://eips.ethereum.org/EIPS/eip-4626#security-considerations,
    // it should round DOWN if it’s calculating the amount of shares to issue to a user, given an amount of assets provided.
    shares_ = convertToShares(assets_);
}

function convertToShares(uint256 assets_) public returns (uint256 shares_) {
    uint256 totalSupply_ = totalSupply;

    shares_ = totalSupply_ == 0 ? assets_ : (assets_ * totalSupply_) / totalAssets();
}


function totalAssets() public returns (uint256 totalAssets_) {
    totalAssets_ = IPoolManagerLike(manager).totalAssets();
}
```

First pool depositor front-running | *High Risk*

Basically:

```
function deposit(uint256 assets, address receiver) external {
    uint256 sharesToMint;
    if (totalSupply == 0) {
        sharesToMint = assets;
    } else {
        sharesToMint = (assets * totalSupply) / depositToken.balanceOf(address(this));
    }
    require(sharesToMint != uint256(0));

    depositToken.transferFrom(msg.sender, address(this), assets);

    _mint(receiver, sharesToMint);
}
```

First pool depositor front-running | *High Risk*

Problem:

- Integer division negatively affects user
- Can be manipulated to cause a large loss, specifically for victim first depositor
- Requiring non-zero sharesToMint doesn't solve this completely

```
function deposit(uint256 assets, address receiver) external {
    uint256 sharesToMint;
    if (totalSupply == 0) {
        sharesToMint = assets;
    } else {
        sharesToMint = (assets * totalSupply) / depositToken.balanceOf(address(this));
    }
    require(sharesToMint != uint256(0));

    depositToken.transferFrom(msg.sender, address(this), assets);

    _mint(receiver, sharesToMint);
}
```

First pool depositor front-running | *High Risk*

	Before			After	
Tx	**totalSupply**	**balanceOf**	**sharesToMint**	**totalSupply**	**balanceOf**
Attacker deposit 1 wei of WETH	0	0	1	1	1
Attacker transfers 100 WETH to contract	1	1	N/A	1	$1 + 100 \times 10^{18}$
Victim deposits 200 WETH	1	$1 + 100 \times 10^{18}$	$1 \times 200 \times 10^{18} / (1 + 100 \times 10^{18})$ = floor(1.99…) = 1	2	$1 + 300 \times 10^{18}$
Attacker withdraws 1 share	2	$1 + 300 \times 10^{18}$	N/A	1	$1 + 150 \times 10^{18}$

```
function deposit(uint256 assets, address receiver) external {
    uint256 sharesToMint;
    if (totalSupply == 0) {
        sharesToMint = assets;
    } else {
        sharesToMint = (assets * totalSupply) / depositToken.balanceOf(address(this));
    }
    require(sharesToMint != uint256(0));

    depositToken.transferFrom(msg.sender, address(this), assets);

    _mint(receiver, sharesToMint);
}
```

First pool depositor front-running | *High Risk*

	Before			After	
Tx	**totalSupply**	**balanceOf**	**sharesToMint**	**totalSupply**	**balanceOf**
Attacker deposit 1 wei of WETH	0	0	1	1	1
Attacker transfers 100 WETH to contract	1	1	N/A	1	$1 + 100 \times 10^{18}$
Victim deposits 200 WETH	1	$1 + 100 \times 10^{18}$	$1 \times 200 \times 10^{18} / (1 + 100 \times 10^{18})$ = floor(1.99…) = 1	2	$1 + 300 \times 10^{18}$
Attacker withdraws 1 share	2	$1 + 300 \times 10^{18}$	N/A	1	$1 + 150 \times 10^{18}$

```
function deposit(uint256 assets, address receiver) external {
    uint256 sharesToMint;
    if (totalSupply == 0) {
        sharesToMint = assets;
    } else {
        sharesToMint = (assets * totalSupply) / depositToken.balanceOf(address(this));
    }
    require(sharesToMint != uint256(0));

    depositToken.transferFrom(msg.sender, address(this), assets);

    _mint(receiver, sharesToMint);
}
```

First pool depositor front-running | *High Risk*

	Before			After	
Tx	**totalSupply**	**balanceOf**	**sharesToMint**	**totalSupply**	**balanceOf**
Attacker deposit 1 wei of WETH	0	0	1	1	1
Attacker transfers 100 WETH to contract	1	1	N/A	1	$1 + 100 \times 10^{18}$
Victim deposits 200 WETH	1	$1 + 100 \times 10^{18}$	$1 \times 200 \times 10^{18} / (1 + 100 \times 10^{18})$ = floor(1.99…) = 1	2	$1 + 300 \times 10^{18}$
Attacker withdraws 1 share	2	$1 + 300 \times 10^{18}$	N/A	1	$1 + 150 \times 10^{18}$

```
function deposit(uint256 assets, address receiver) external {
    uint256 sharesToMint;
    if (totalSupply == 0) {
        sharesToMint = assets;
    } else {
        sharesToMint = (assets * totalSupply) / depositToken.balanceOf(address(this));
    }
    require(sharesToMint != uint256(0));

    depositToken.transferFrom(msg.sender, address(this), assets);

    _mint(receiver, sharesToMint);
}
```

First pool depositor front-running | *High Risk*

	Before			After	
Tx	**totalSupply**	**balanceOf**	**sharesToMint**	**totalSupply**	**balanceOf**
Attacker deposit 1 wei of WETH	0	0	1	1	1
Attacker transfers 100 WETH to contract	1	1	N/A	1	$1 + 100 \times 10^{18}$
Victim deposits 200 WETH	1	$1 + 100 \times 10^{18}$	$1 \times 200 \times 10^{18} / (1 + 100 \times 10^{18})$ = floor(1.99…) = 1	2	$1 + 300 \times 10^{18}$
Attacker withdraws 1 share	2	$1 + 300 \times 10^{18}$	N/A	1	$1 + 150 \times 10^{18}$

```
function deposit(uint256 assets, address receiver) external {
    uint256 sharesToMint;
    if (totalSupply == 0) {
        sharesToMint = assets;
    } else {
        sharesToMint = (assets * totalSupply) / depositToken.balanceOf(address(this));
    }
    require(sharesToMint != uint256(0));

    depositToken.transferFrom(msg.sender, address(this), assets);

    _mint(receiver, sharesToMint);
}
```

First pool depositor front-running | *High Risk*

	Before			After	
Tx	**totalSupply**	**balanceOf**	**sharesToMint**	**totalSupply**	**balanceOf**
Attacker deposit 1 wei of WETH	0	0	1	1	1
Attacker transfers 100 WETH to contract	1	1	N/A	1	$1 + 100 \times 10^{18}$
Victim deposits 200 WETH	1	$1 + 100 \times 10^{18}$	$1 \times 200 \times 10^{18} / (1 + 100 \times 10^{18})$ = floor(1.99…) = 1	2	$1 + 300 \times 10^{18}$
Attacker withdraws 1 share	2	$1 + 300 \times 10^{18}$	N/A	1	$1 + 150 \times 10^{18}$

```
function deposit(uint256 assets, address receiver) external {
    uint256 sharesToMint;
    if (totalSupply == 0) {
        sharesToMint = assets;
    } else {
        sharesToMint = (assets * totalSupply) / depositToken.balanceOf(address(this));
    }
    require(sharesToMint != uint256(0));

    depositToken.transferFrom(msg.sender, address(this), assets);

    _mint(receiver, sharesToMint);
}
```

First pool depositor front-running | *High Risk*

	Before			After	
Tx	**totalSupply**	**balanceOf**	**sharesToMint**	**totalSupply**	**balanceOf**
Attacker deposit 1 wei of WETH	0	0	1	1	1
Attacker transfers 100 WETH to contract	1	1	N/A	1	$1 + 100 \times 10^{18}$
Victim deposits 200 WETH	1	$1 + 100 \times 10^{18}$	$1 \times 200 \times 10^{18} / (1 + 100 \times 10^{18})$ = floor(1.99…) = 1	2	$1 + 300 \times 10^{18}$
Attacker withdraws 1 share	2	$1 + 300 \times 10^{18}$	N/A	1	$1 + 150 \times 10^{18}$

Profit: 50 WETH

```
function deposit(uint256 assets, address receiver) external {
    uint256 sharesToMint;
    if (totalSupply == 0) {
        sharesToMint = assets;
    } else {
        sharesToMint = (assets * totalSupply) / depositToken.balanceOf(address(this));
    }
    require(sharesToMint != uint256(0));

    depositToken.transferFrom(msg.sender, address(this), assets);

    _mint(receiver, sharesToMint);
}
```

First pool depositor front-running | *High Risk*

Well-known issue that has been mentioned in audits before:

- **UniswapV2:** https://dapp.org.uk/reports/uniswapv2.html
  - LP tokens instead of vault shares
- **Bunni by Timeless Finance:** https://www.rileyholterhus.com/writing/bunni
  - Manually supply liquidity instead of manual transfer of tokens
- **xSushi-like contracts:** https://media.dedaub.com/latent-bugs-in-billion-plus-dollar-code-c2e67a25b689
  - Pretty similar, and you can see a first deposit of $60M USD 🤯

First pool depositor front-running | *High Risk*

Solution:

- Need to enforce a minimum deposit… that can’t be withdrawn
- So, mint some of the initial amount to the zero address
- Most legit first depositors will mint thousands of shares (token decimals typically ≥ 6), so not a big cost

```
function deposit(uint256 assets, address receiver) external {
    uint256 sharesToMint;
    if (totalSupply == 0) {
        sharesToMint = assets;
    } else {
        sharesToMint = (assets * totalSupply) / depositToken.balanceOf(address(this));
    }
    require(sharesToMint != uint256(0));

    if (totalSupply == 0) {
        _mint(address(0), BOOTSTRAP_MINT);

        sharesToMint -= BOOTSTRAP_MINT;
    }

    depositToken.transferFrom(msg.sender, address(this), assets);

    _mint(receiver, sharesToMint);
}
```

Manually sent tokens breaks things | *Medium Risk*

```
contract Loan {

    /* ...omitted... */

    uint256 _collateral;
    address _borrower;

    function collateral() external view returns (uint256) {
        return _collateral;
    }

    function postCollateral(uint256 amount) public {
        IERC20(collateralToken).transferFrom(msg.sender, address(this), amount);
        _collateral += getUnaccountedAmount(collateralToken);
    }

    function removeCollateral(uint256 amount, address destination) external {
        require(msg.sender == _borrower);

        _collateral -= amount;
        IERC20(collateralToken).transfer(destination, amount);
        require(_isCollateralMaintained());
    }

    function getUnaccountedAmount(address asset) public view returns (uint256) {
        return IERC20(asset).balanceOf(address(this))
            - (asset == collateralToken ? _collateral     : uint256(0))
            - (asset == borrowToken     ? _borrowedAmount : uint256(0));
    }


}
```

```
contract LoanManager {

    /* ...omitted... */

    function triggerDefault(address loan) external {
        if (IMapleLoanLike(loan).collateral() == 0 || IMapleLoanLike(loan).collateralToken() == borrowToken) {
            _handleNonLiquidatingRepossession(...);
        } else {
            _handleLiquidatingRepossession(...);
        }
    }

}
```

- Two types of repossessions in a default - liquidating and nonliquidating
- Depends on if loan has any collateral

Manually sent tokens breaks things | *Medium Risk*

```
contract Loan {

    /* ...omitted... */

    uint256 _collateral;
    address _borrower;

    function collateral() external view returns (uint256) {
        return _collateral;
    }

    function postCollateral(uint256 amount) public {
        IERC20(collateralToken).transferFrom(msg.sender, address(this), amount);
        _collateral += getUnaccountedAmount(collateralToken);
    }

    function removeCollateral(uint256 amount, address destination) external {
        require(msg.sender == _borrower);

        _collateral -= amount;
        IERC20(collateralToken).transfer(destination, amount);
        require(_isCollateralMaintained());
    }

    function getUnaccountedAmount(address asset) public view returns (uint256) {
        return IERC20(asset).balanceOf(address(this))
            - (asset == collateralToken ? _collateral     : uint256(0))
            - (asset == borrowToken     ? _borrowedAmount : uint256(0));
    }


}
```

```
contract LoanManager {

    /* ...omitted... */

    function triggerDefault(address loan) external {
        if (IMapleLoanLike(loan).collateral() == 0 || IMapleLoanLike(loan).collateralToken() == borrowToken) {
            _handleNonLiquidatingRepossession(...);
        } else {
            _handleLiquidatingRepossession(...);
        }
    }

}
```

- Two types of repossessions in a default - liquidating and nonliquidating
- Depends on if loan has any collateral

Manually sent tokens breaks things | *Medium Risk*

```
contract LoanManager {

    /* ...omitted... */

    function triggerDefault(address loan) external {
        if (IMapleLoanLike(loan).collateral() == 0 || IMapleLoanLike(loan).collateralToken() == borrowToken) {
            _handleNonLiquidatingRepossession(...);
        } else {
            _handleLiquidatingRepossession(...);
        }
    }

}
```

```
contract Loan {

    /* ...omitted... */

    uint256 _collateral;
    address _borrower;

    function collateral() external view returns (uint256) {
        return _collateral;
    }

    function postCollateral(uint256 amount) public {
        IERC20(collateralToken).transferFrom(msg.sender, address(this), amount);
        _collateral += getUnaccountedAmount(collateralToken);
    }

    function removeCollateral(uint256 amount, address destination) external {
        require(msg.sender == _borrower);

        _collateral -= amount;
        IERC20(collateralToken).transfer(destination, amount);
        require(_isCollateralMaintained());
    }

    function getUnaccountedAmount(address asset) public view returns (uint256) {
        return IERC20(asset).balanceOf(address(this))
            - (asset == collateralToken ? _collateral     : uint256(0))
            - (asset == borrowToken     ? _borrowedAmount : uint256(0));
    }


}
```

- The collateral() function represents the collateral that is *accounted for,* but anyone can transfer tokens manually to the loan
- This doesn't match what is expected in _handleNonLiquidatingRepossesion □ reverts
- The loan default is temporarily delayed (can be fixed by syncing the collateral variable)

Manually sent tokens breaks things | *Medium Risk*

Solution:

- Use balanceOf instead of internal accounting variable

```
contract LoanManager {

    /* ...omitted... */

    function triggerDefault(address loan) external {
        address collateralToken = IMapleLoanLike(loan).collateralToken();
        if (IERC20(collateralToken).balanceOf(loan) == 0 || collateralToken == borrowToken) {
            _handleNonLiquidatingRepossession(...);
        } else {
            _handleLiquidatingRepossession(...);
        }
    }

}
```

Error queuing many config updates | *Medium Risk*

- New withdrawal mechanism has some configuration parameters that can be changed
  - Importantly the “cycleDuration” can be configured
- Changes take effect after 2 additional “cycles”

```
function setExitConfig(uint256 cycleDuration_, uint256 windowDuration_) external onlyOwner {
    CycleConfig memory config_ = getCurrentConfig();


    // The new config will take effect only after the current cycle and two additional ones elapse.
    // This is done in order to to prevent overlaps between the current and new withdrawal cycles.
    uint256 currentCycleId_   = getCurrentCycleId();
    uint256 initialCycleId_   = currentCycleId_ + 3;
    uint256 initialCycleTime_ = getWindowStart(currentCycleId_) + 3 * config_.cycleDuration;
    uint256 latestConfigId_   = latestConfigId;

    // If the new config takes effect on the same cycle as the latest config, overwrite it. Otherwise create a new config.
    if (initialCycleId_ != cycleConfigs[latestConfigId_].initialCycleId) {
        latestConfigId_ = ++latestConfigId;
    }

    cycleConfigs[latestConfigId_] = CycleConfig({
        initialCycleId:   _uint64(initialCycleId_),
        initialCycleTime: _uint64(initialCycleTime_),
        cycleDuration:    _uint64(cycleDuration_),
        windowDuration:   _uint64(windowDuration_)
    });
}
```

Error queuing many config updates | *Medium Risk*

Problem:

- What if you change the config again within these 2 intermediate cycles?
- config_.cycleDuration is only accurate for those 2 cycles, otherwise you should be using the updated cycleDuration

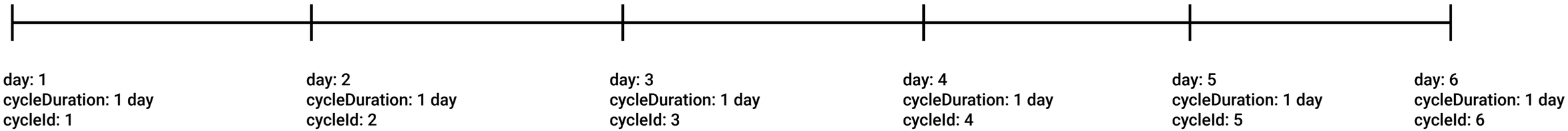
```
function setExitConfig(uint256 cycleDuration_, uint256 windowDuration_) external onlyOwner {
    CycleConfig memory config_ = getCurrentConfig();


    // The new config will take effect only after the current cycle and two additional ones elapse.
    // This is done in order to to prevent overlaps between the current and new withdrawal cycles.
    uint256 currentCycleId_    = getCurrentCycleId();
    uint256 initialCycleId_    = currentCycleId_ + 3;
    uint256 initialCycleTime_ = getWindowStart(currentCycleId_) + 3 * config_.cycleDuration;
    uint256 latestConfigId_    = latestConfigId;

    // If the new config takes effect on the same cycle as the latest config, overwrite it. Otherwise create a new config.
    if (initialCycleId_ != cycleConfigs[latestConfigId_].initialCycleId) {
        latestConfigId_ = ++latestConfigId;
    }

    cycleConfigs[latestConfigId_] = CycleConfig({
        initialCycleId:   _uint64(initialCycleId_),
        initialCycleTime: _uint64(initialCycleTime_),
        cycleDuration:    _uint64(cycleDuration_),
        windowDuration:   _uint64(windowDuration_)
    });
}
```

Error queuing many config updates | *Medium Risk*



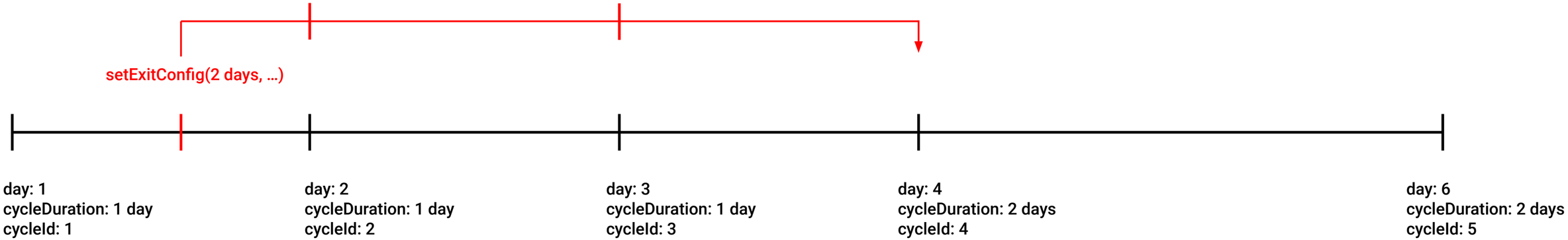

```
function setExitConfig(uint256 cycleDuration_, uint256 windowDuration_) external onlyOwner {
    CycleConfig memory config_ = getCurrentConfig();


    // The new config will take effect only after the current cycle and two additional ones elapse.
    // This is done in order to to prevent overlaps between the current and new withdrawal cycles.
    uint256 currentCycleId_    = getCurrentCycleId();
    uint256 initialCycleId_    = currentCycleId_ + 3;
    uint256 initialCycleTime_  = getWindowStart(currentCycleId_) + 3 * config_.cycleDuration;
    uint256 latestConfigId_    = latestConfigId;

    // If the new config takes effect on the same cycle as the latest config, overwrite it. Otherwise create a new config.
    if (initialCycleId_ != cycleConfigs[latestConfigId_].initialCycleId) {
        latestConfigId_ = ++latestConfigId;
    }

    cycleConfigs[latestConfigId_] = CycleConfig({
        initialCycleId:   _uint64(initialCycleId_),
        initialCycleTime: _uint64(initialCycleTime_),
        cycleDuration:    _uint64(cycleDuration_),
        windowDuration:   _uint64(windowDuration_)
    });
}
```

Error queuing many config updates | *Medium Risk*



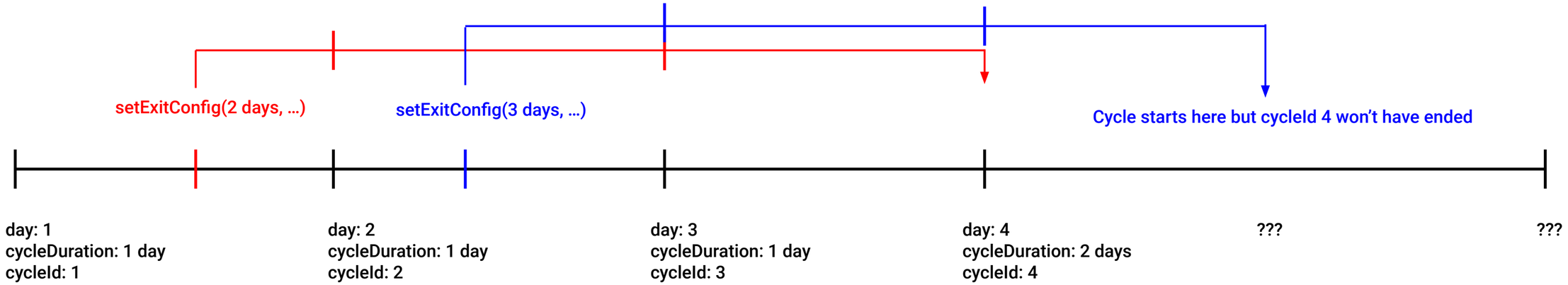

```
function setExitConfig(uint256 cycleDuration_, uint256 windowDuration_) external onlyOwner {
    CycleConfig memory config_ = getCurrentConfig();


    // The new config will take effect only after the current cycle and two additional ones elapse.
    // This is done in order to to prevent overlaps between the current and new withdrawal cycles.
    uint256 currentCycleId_    = getCurrentCycleId();
    uint256 initialCycleId_    = currentCycleId_ + 3;
    uint256 initialCycleTime_  = getWindowStart(currentCycleId_) + 3 * config_.cycleDuration;
    uint256 latestConfigId_    = latestConfigId;

    // If the new config takes effect on the same cycle as the latest config, overwrite it. Otherwise create a new config.
    if (initialCycleId_ != cycleConfigs[latestConfigId_].initialCycleId) {
        latestConfigId_ = ++latestConfigId;
    }

    cycleConfigs[latestConfigId_] = CycleConfig({
        initialCycleId:   _uint64(initialCycleId_),
        initialCycleTime: _uint64(initialCycleTime_),
        cycleDuration:    _uint64(cycleDuration_),
        windowDuration:   _uint64(windowDuration_)
    });
}
```

Error queuing many config updates | *Medium Risk*




```
function setExitConfig(uint256 cycleDuration_, uint256 windowDuration_) external onlyOwner {
    CycleConfig memory config_ = getCurrentConfig();


    // The new config will take effect only after the current cycle and two additional ones elapse.
    // This is done in order to to prevent overlaps between the current and new withdrawal cycles.
    uint256 currentCycleId_    = getCurrentCycleId();
    uint256 initialCycleId_    = currentCycleId_ + 3;
    uint256 initialCycleTime_  = getWindowStart(currentCycleId_) + 3 * config_.cycleDuration;
    uint256 latestConfigId_    = latestConfigId;

    // If the new config takes effect on the same cycle as the latest config, overwrite it. Otherwise create a new config.
    if (initialCycleId_ != cycleConfigs[latestConfigId_].initialCycleId) {
        latestConfigId_ = ++latestConfigId;
    }

    cycleConfigs[latestConfigId_] = CycleConfig({
        initialCycleId:   _uint64(initialCycleId_),
        initialCycleTime: _uint64(initialCycleTime_),
        cycleDuration:    _uint64(cycleDuration_),
        windowDuration:   _uint64(windowDuration_)
    });
}
```

Error queuing many config updates | *Medium Risk*

Solution:

- Fetch the pending cycleDuration values and sum those to get the initialCycleTime

```
function setExitConfig(uint256 cycleDuration_, uint256 windowDuration_) external onlyOwner {
    CycleConfig memory config_ = getCurrentConfig();


    // The new config will take effect only after the current cycle and two additional ones elapse.
    // This is done in order to to prevent overlaps between the current and new withdrawal cycles.
    uint256 currentCycleId_    = getCurrentCycleId();
    uint256 initialCycleId_    = currentCycleId_ + 3;

    uint256 initialCycleTime_ = getWindowStart(currentCycleId_);
    for (uint256 i = currentCycleId_; i < initialCycleId_; i++) {
        CycleConfig memory config = getConfigAtId(i);
        initialCycleTime_ += config.cycleDuration;
    }

    uint256 latestConfigId_    = latestConfigId;

    // If the new config takes effect on the same cycle as the latest config, overwrite it. Otherwise create a new config.
    if (initialCycleId_ != cycleConfigs[latestConfigId_].initialCycleId) {
        latestConfigId_ = ++latestConfigId;
    }

    cycleConfigs[latestConfigId_] = CycleConfig({
        initialCycleId:   _uint64(initialCycleId_),
        initialCycleTime: _uint64(initialCycleTime_),
        cycleDuration:    _uint64(cycleDuration_),
        windowDuration:   _uint64(windowDuration_)
    });
}
```

MEV/unfair behaviour considerations | *Informational*

The Maple V2 contracts successfully prevents against many different types of MEV/unfair behaviour, e.g:

- If a borrower is known to be insolvent & the loan due date is far enough in the future:
  - Lenders can withdraw before loss is realized in the internal accounting, can lead to a race to withdraw with losers taking on bulk of the losses
    - So, loans can be “impaired” by an admin and loss is realized immediately

- If a loan payment updated the internal accounting with a discrete jump:
  - Lenders could sandwich these payment to unfairly capture value
    - So, within the internal accounting the payment is streamed linearly

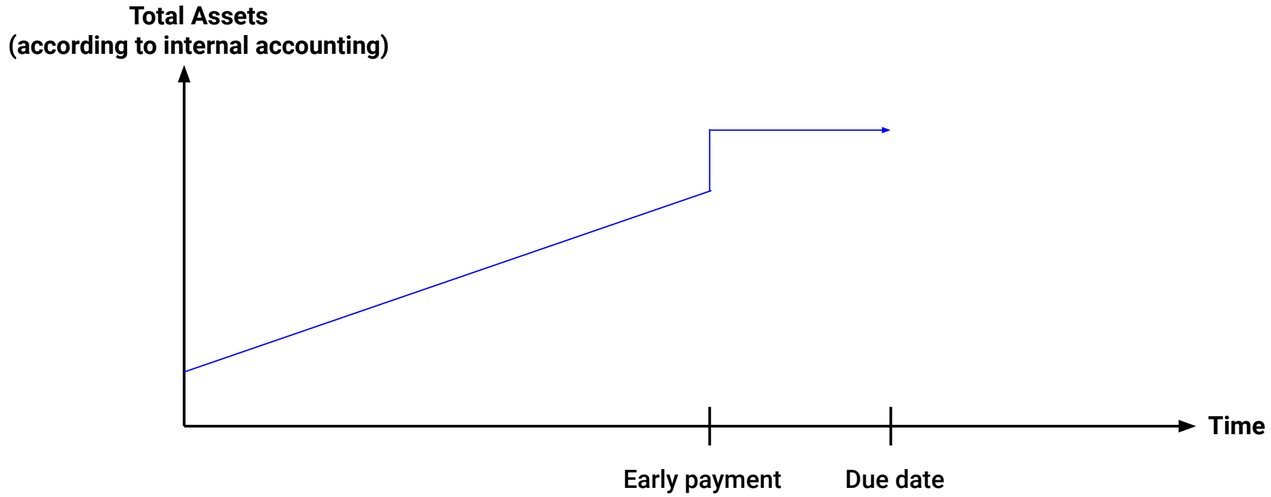
MEV/unfair behaviour considerations | *Informational*

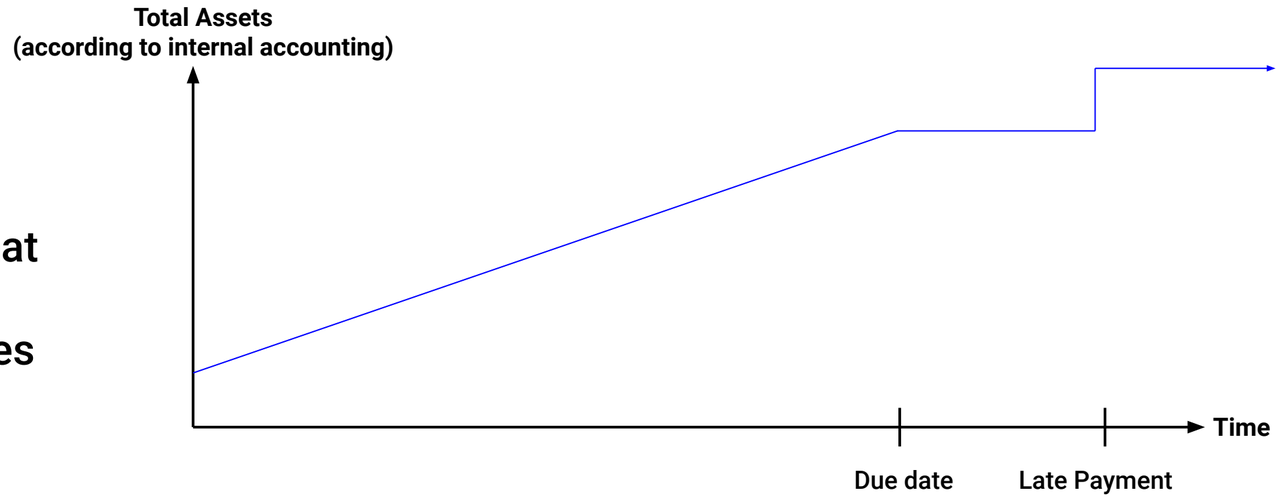
Although, there do exist some remaining scenarios that are more difficult to completely mitigate:

- If a borrower makes a loan payment **early**:
  - Linear accrual is cut short, results in small discrete jump

- If a borrower makes a loan payment **late**:
  - Late interest fee (and accrued interest on multi-payment loans) results in small discrete jump

 These jumps are rare enough and small enough that exploitation is unlikely. Also, the sandwichers would need to go through the withdrawal cycles, which makes exploitation even more unlikely





Questions?