Arcade V2 Audit

Author: Roku

Audit Start: May 24, 2022

Commit Hash for Audit: f1c9c43

Scope: every contract in the `contacts` folder, except `test` folder

Report Submitted: June 02, 2022

Fixes Submitted: June 14, 2022

Final Commit Hash: f7b2c6c

Overview

I was asked by kvk from the Arcade.xyz team to do a smart contract audit on the v2 of the Pawn Protocol. This is a report that concludes what I find and provides suggestions that aim to improve security or reduce the complexity of the codebase.

Pawn is a protocol built for decentralized NFTs lending. The core protocol offers legacy lending features and a more advanced kind of loan with “installment”, which gives lenders a more stable income stream throughout the duration of the loan.

The codebase is **very well documented and tested**. The test suits achieved 100% statement coverage and 100% branch coverage on core functions, (98% in total). The test suite is also very **well organized** and covers various kinds of scenarios, which make it very easy to understand what the system is supposed to behave, also gave me an ease while trying to write up some exploit scenarios. Overall, the codebase gave me a high confidence in the engineering team in terms of security process and expertise.

I concluded the 6-day audit with 1 high severity, 1 medium severity and 2 low severity findings.

Findings

High Severity

(H-1) Borrower with loans with installment can keep paying the minimum interest, and never repay the capital nor get “liquidated”.

In the function `_verifyDefaultedLoan`, if it’s a loan with installment, it only checks if the number of missed installments is < 40% of total number of installments.

This leads to scenarios when the borrower still keeps paying the ‘minimum’ interest, it will make the loan “non defaultable”, and the lender will have no way to get his principle back even after the duration.

Suggestion:

consider adding a deadline check (similar to legacy loans), and let the lender use claim function after deadline + grace period.

Response:

Status: **Fixed**

Fix Commit: 18d25ce57bc7fdd70ab722e8ec9f56abad05b4fb

PR: https://github.com/Non-fungible-Technologies/v2-contracts/pull/43

Medium Severity

(M-1) No protection on PunkRouter.depositPunk

In the PunkRouter, the contract wraps the Punk NFT into wrappedPunk, and sends it into the `AssetVault` address user specified. The code is relatively straight forward:

```
_wrappedPunks.safeTransferFrom(address(this), address(uint160(bundleId)), punkIndex
```

However, given the relatively complex design of `bundleId` and asset vault, it is possible that a user might specify the wrong `bundleId` and result in a wrong address receiving the wrapPunk. (Using vault address as NFT id is smart, but it is somehow counterintuitive).

Suggestion

I suggest doing the one following to lower the risk of making a mistake:

- instead of using `boundleId` (`uint256`), specifying recipient as an address is more intuitive and general.
- keep the `boundleId`, but add an additional check with `VaultFactory` to see if this `bundleId` is valid. (expose an `exist()` function)

Response:

Status: Fixed

Fix Commit: f40e357879e9f7687c381bd99976645f401b11b3

PR: https://github.com/Non-fungible-Technologies/v2-contracts/pull/33

Comments from the team:

Arcade.xyz team decided to remove PunkRouter entirely. Any lack of native support for CryptoPunks in an AssetVault would be error-prone and could temporarily freeze assets, in case a user mistakenly sent a punk to their vault. By adding native support for CryptoPunks in AssetVault via the withdrawPunk function, the PunkRouter became obsolete. We believe this is a less error-prone solution that removes both the risk of sending unwrapped punks to the vault, and providing the incorrect vault ID in the old PunkRouter depositPunk function.

Low Severity

(L-1) Borrower can force to rollover loans with installments and let the lender get less than expected.

In the rollover function, there's no restriction around the loan's duration. So there's no way for a lender to stop a rollover, as long as the borrower pays back what is owed for the period borrowed. For a legacy loan, there is no issue the borrower has to pay back full interest. But for loans with installments, it's possible that the lender agrees on a longer period and expects a passive income out of it, but then the loan is forced to rollover into a new one and he only gets paid the interest for the first few "periods".

Suggestion

It seems like it can be either a "feature" or "bug", but it has to be communicated well to the lender and documented in the code. If this is not an expected behavior, consider adding a constraint on minimum % of installments being "passed" before rollover can be triggered.

Response

Status: Acknowledged

Comments from the team:

As discussed in the initial report, this is a desired feature. The goal of the protocol is to encourage a competitive lending market - thus, if the borrower receives a "better offer", they should be able to use rollovers to switch to the new lender with the better terms. It's true that early repayment results in receiving less than the full term's interest - this is true in the case of both rollovers and standard repayment.

Comments from Roku:

I initially categorized this as a Med severity finding, because it seems like an unexpected behavior for lenders. But after discussion with the team, they confirmed that this is a desired feature and I agree that this will make the overall system more healthy. Just needed to be communicated properly on their interface.

(L-2) Fee is not capped

there's no upper bound on `originationFee` or `rollOverFee`, introducing unnecessary centralization risk (or key risk). The owner of FeeCollector has the ability to set fee to 100% and instantly withdraw principal, making the borrower getting 0 out of a loan.

Since the owner of LoanCore can update the feeCollector, the risk of losing key for feeCollector and freezing funds is low. But still, it would make the system safer if the upper bound is enforced by the contract.

Response

Status: Fixed

Fix Commit: 94c2302c98e113fc544167ffcbe87bd8e154a6f2

PR: https://github.com/Non-fungible-Technologies/v2-contracts/pull/28

Minor Issues / Suggestions

(1) Wrong comments

in `errors/Lending.sol`:

- in natspec of error `OC_NumberInstallments`
  Loan terms must have an even number of installments. but this isn't checked, nor applied the actual logic calculation
- wrong comment on `LC_BalanceGTZero` and `LC_NonceUsed`

in `OriginationController`:

```
// interest rate must be greater than or equal to 0.01%

// and less than 10,000% (1e8 basis points)

if (terms.interestRate < 1e18 || terms.interestRate > 1e24) revert
OC_InterestRate(terms.interestRate);
```

- should be 1e18 basis points.

Response:

Status: Fixed

Fix Commit: f078e4a9a669bd307cd1a3d807880ea19ce8a3c8

PR: https://github.com/Non-fungible-Technologies/v2-contracts/pull/42

(2) Remove hardcoded function selector & fix invalid tests in whitelist module

The following syntax increases readability, and doesn't increase gas cost because everything will be set at compile time.

```
bytes4 private constant ERC20_TRANSFER = ERC20.transfer.selector;
bytes4 private constant ERC20_ERC721_APPROVE = ERC20.approve.selector;
bytes4 private constant ERC20_ERC721_TRANSFER_FROM = ERC20.transferFrom.selector;

bytes4 private constant ERC721_SAFE_TRANSFER_FROM =
    bytes4(keccak256("safeTransferFrom(address, address, uint256)"));
bytes4 private constant ERC721_SAFE_TRANSFER_FROM_DATA =
    bytes4(keccak256("safeTransferFrom(address, address, uint256,bytes)"));
bytes4 private constant ERC721_ERC1155_SET_APPROVAL =
    ERC721.setApprovalForAll.selector;

bytes4 private constant ERC1155_SAFE_TRANSFER_FROM =
    ERC1155.safeTransferFrom.selector;
bytes4 private constant ERC1155_SAFE_BATCH_TRANSFER_FROM =
    ERC1155.safeBatchTransferFrom.selector;
```

Also, some issues in the test files: The tests in `CallWhitelist.ts` around blacklisted selectors are **invalid**, because it uses `isWhitelisted` interface to check if it returns false, but the function won't return true unless it's whitelisted && not blacklisted. The effective tests should be as follow:

```
describe("Global blacklist", function () {
        it("erc20 transfer", async () => {
            const { whitelist, mockERC20 } = await loadFixture(fixture);
            const selector = mockERC20.interface.getSighash("transfer");
            expect(await whitelist.isBlacklisted(selector)).to.be.true;
        });

        it("erc20 approve", async () => {
            const { whitelist, mockERC20 } = await loadFixture(fixture);
            const selector = mockERC20.interface.getSighash("approve");
            expect(await whitelist.isBlacklisted(selector)).to.be.true;
        });
        ...
})
```

Or even add tests that even if the selector is being added as whitelisted by the owner, isWhitelisted will still return `false`.

Response

Status: Fixed

Fix Commit: 3643c4c357ba08a721baf469483c2aa5a5dfcc9b

PR: https://github.com/Non-fungible-Technologies/v2-contracts/pull/38

(3) Optimize repay and repayPart functions

in `repay` function of `LoanCore`, it can reduce gas spent on transfer if core function just do one safeTransferFrom to get money from msg.sender to lender at the end of the function, instead of

```
IERC20Upgradeable(data.terms.payableCurrency)
    .safeTransferFrom(_msgSender(), address(this), returnAmount);
IERC20Upgradeable(data.terms.payableCurrency)
    .transfer(lender, returnAmount);
```

This also applies better to the check-effect-interaction pattern.

It can further be optimized by using transferFrom(borrower, lender) directly, instead of having the Repayment Controller as a proxy.

Similarly, in function `repayPart`, amount of token `paymentTotal` is charged from the user first, but then `boundedPaymentTotal` is sent to the lender:

```
IERC20Upgradeable(data.terms.payableCurrency)
    .safeTransferFrom(_msgSender(), address(this), paymentTotal);

// calculate boundedPaymentTotal which <= paymentTotal

IERC20Upgradeable(data.terms.payableCurrency)
    .transfer(lender, boundedPaymentTotal);
```

This introduces extra logic to deal with refunds.

A better pattern is to only pull the boundedPaymentTotal, and transfer to the lender directly.

This would only require changing borrower to approve the core contract instead of RepayController, instead of dealing with refund.

Response

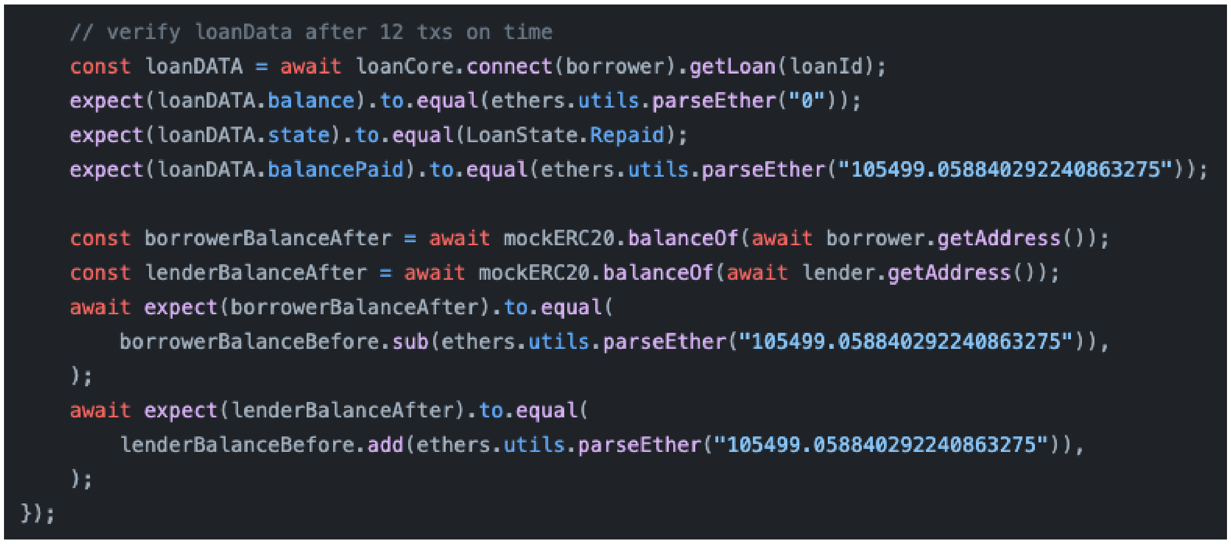
Status: Acknowledged

Comments from the team

The changes for both minor fixes 3 and 4 are very similar and investigated at the same time. The suggestions related to the checks-effects-interactions pattern was also suggested by the Quantstamp audit. These comments are related to moving `safeTransferFrom` at the top of both the repay and repayPart functions to the bottom where the other transfer methods are located. This would allow `repay and `repayPart` to conform to the checks-effects-interactions pattern. After this change was made, every test was run (`yarn test`) and there were no new failing tests.

Subsequently, in the comment in minor 3 and 4 for Roku there is also the suggestion to remove the first transfer method in each `repay` and `repayPart` and only have one transfer method that goes directly from the borrower to the lender. As opposed to in the current setup LoanCore contract is the middle man for these transfers. Theoretically, this makes sense but in the implementation we ran into a concerning issue. After this change was made, all tests were run (`yarn test`) and there were 2 new failing tests which were not failing previously. After investigation we found that at the end of the 2 tests,the balance checks for the borrower and lender were failing. After some more investigation it was found that the tests would only pass when their parameters were not equal whereas before they were. See below. Image 1 is the old test before change, image 2 is the new test after change.

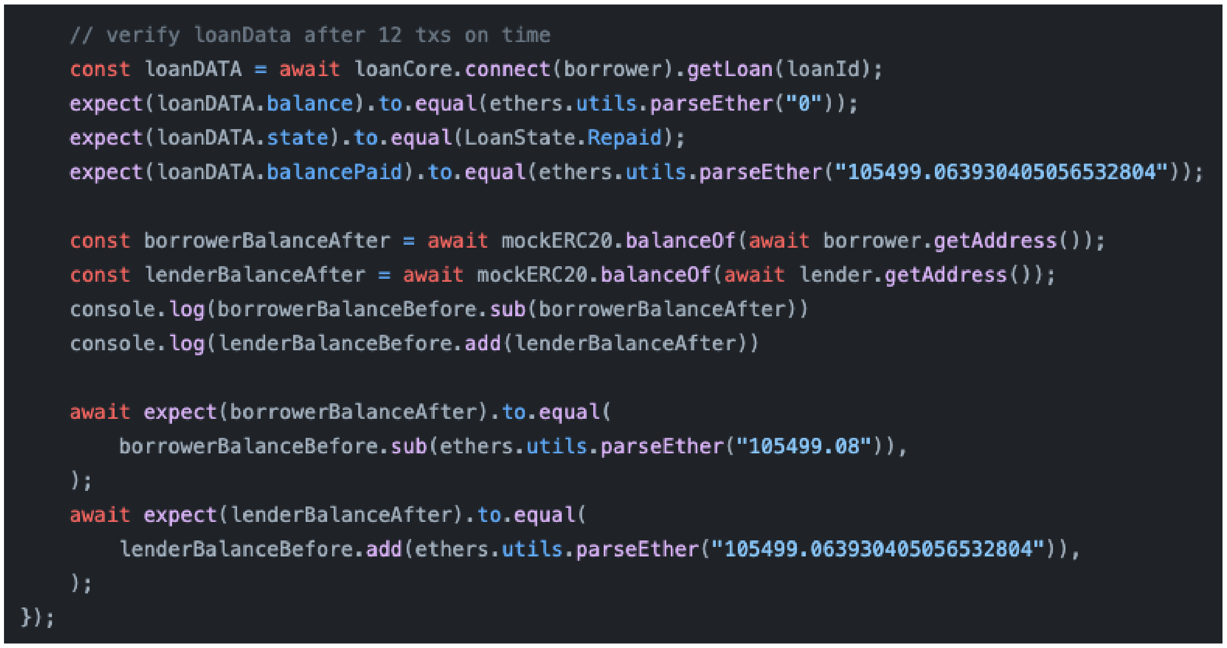
Before audit changes



```
    // verify loanData after 12 txs on time
    const loanDATA = await loanCore.connect(borrower).getLoan(loanId);
    expect(loanDATA.balance).to.equal(ethers.utils.parseEther("0"));
    expect(loanDATA.state).to.equal(LoanState.Repaid);
    expect(loanDATA.balancePaid).to.equal(ethers.utils.parseEther("105499.058840292240863275"));

    const borrowerBalanceAfter = await mockERC20.balanceOf(await borrower.getAddress());
    const lenderBalanceAfter = await mockERC20.balanceOf(await lender.getAddress());
    await expect(borrowerBalanceAfter).to.equal(
        borrowerBalanceBefore.sub(ethers.utils.parseEther("105499.058840292240863275")),
    );
    await expect(lenderBalanceAfter).to.equal(
        lenderBalanceBefore.add(ethers.utils.parseEther("105499.058840292240863275")),
    );
});
```

After audit changes



```
    // verify loanData after 12 txs on time
    const loanDATA = await loanCore.connect(borrower).getLoan(loanId);
    expect(loanDATA.balance).to.equal(ethers.utils.parseEther("0"));
    expect(loanDATA.state).to.equal(LoanState.Repaid);
    expect(loanDATA.balancePaid).to.equal(ethers.utils.parseEther("105499.063930405056532804"));

    const borrowerBalanceAfter = await mockERC20.balanceOf(await borrower.getAddress());
    const lenderBalanceAfter = await mockERC20.balanceOf(await lender.getAddress());
    console.log(borrowerBalanceBefore.sub(borrowerBalanceAfter))
    console.log(lenderBalanceBefore.add(lenderBalanceAfter))

    await expect(borrowerBalanceAfter).to.equal(
        borrowerBalanceBefore.sub(ethers.utils.parseEther("105499.08")),
    );
    await expect(lenderBalanceAfter).to.equal(
        lenderBalanceBefore.add(ethers.utils.parseEther("105499.063930405056532804")),
    );
});
```

Both of these tests passed with their respective protocols, which makes the team concerned that the order and routing of payments has been done with a purpose since we have inherited this payment pattern from the legacy (V1) contracts. With this being said we opted to not change the contracts to make a direct transfer from borrower to lender upon repayment and have opted to go with our current pattern of using LoanCore as a middleman between the two parties.

(4) Minor suggestion on canCallOn optimization

Function `canCallOn` should let the calling contract (or user) specify an id that proves it has ownership over the call, instead of using an exhausted on-chain loop.

Response

Status: Acknowledged

Comments from the team

After reviewing this change, the team opted to stick with our current implementation of `canCallOn`. The reasoning behind this decision was for a few reasons, but mainly we saw this change as being invasive enough as to create breaking changes, and needing to possibly specify a loan ID as well as the vault address. This could become confusing for users and we do not expect borrowers to have hundreds of borrower notes at one time since they are burned after the loans lifecycle is over. Along the same lines, we did not see a big benefit to adding additional storage for this logic and opted to reuse the data available.

Closing & Disclaimer

This report should only be used as a material for users to better assess the risk of using the protocol, not as any sort of proof or guarantee of safety of investments.

It’s been an honor working with the Arcade.xyz team, they have been helpful throughout my audit process, also carefully reviewing all the issues I raised. I would suggest auditors to work with them.