

2023-04-25

JPG Store

NFT Lending Contracts

π Lanningham



Sundae Labs

Contents

1 - Audit Manifest	3
2 - Specification	5
2.a - High Level Objectives	5
2.b - Detailed Specification	6
2.c - Core Invariants	17
3 - Findings Summary	20
JPG-100 - ID tokens can escape allowed script addresses	23
JPG-101 - Incorrect implementation of config token	24
JPG-102 - Bid pools unusable	26
JPG-103 - UTXO packing vulnerability	27
JPG-104 - Loans can be repaid even after they have expired	28
JPG-200 - Lender not paying marketplace fee	29
JPG-201 - Inefficient / Inflexible use of Address owners	30
JPG-202 - Cannot update a loan offer	31
JPG-203 - Cannot cancel Loan Offer s and Loan Request s in the same transaction	32
JPG-204 - Borrower can extend duration of loan	33
JPG-205 - Small DDOS attack vector	34
JPG-206 - Double satisfaction bug when repaying loan	35
JPG-207 - Loan Requests with ID tokens can be used to access borrowers bidpools	36
JPG-300 - Implementation of ID token limits flexibility	37
JPG-301 - Inefficiency in double satisfaction code	38
JPG-302 - Exposure to future risk of ID Token usage	39
JPG-303 - Marketplace Address is hard coded	41
JPG-304 - Small social engineering attack vector	42
JPG-305 - Composability suffers when accepting loan requests	43
JPG-306 - Lender can hijack ADA stake when accepting loan request	44
JPG-400 - Argument for: Bid Pool s can be reclaimed	45
JPG-401 - Argument for: Loan Offer s and Loan Request s can be reclaimed	47
JPG-402 - Argument for: ID Token s cannot escape	49
JPG-403 - Argument for: ID Token s cannot be forged	52
JPG-404 - Argument for: The Config can only be updated by JPG Store once	55
JPG-405 - Argument for: A Bid Pool cannot be embezzled	57
JPG-406 - Argument for: A Loan Offer can be accepted	60
JPG-407 - Argument for: A Loan Request can be accepted	64
JPG-408 - Argument for: Loan Offer s, Loan Request s, and Bid Pool s cannot be double-satisfied.	67
JPG-409 - Argument for: A Loan can be repaid	71
JPG-410 - Argument for: A Loan can default	74
JPG-411 - Argument for: The marketplace fee is paid	76
4 - Appendix	78
4.a - Disclaimer	78
4.b - Issue Guide	79
4.c - Revisions	81
4.d - About Us	81

1 - Audit Manifest

Please find below the list of pinned software dependencies and files that were covered by the audit.

Software	Version	Commit
Lending Contracts	0.0.0	ad0850717bc4de2d57c6be218cc98d66e0fd27b4

Filename	Hash (SHA256)
validators/id_token.ak	e42859dfa50d9c1fc4feee3c7c127b9fff9551cef8da13e67d8b20916f4aeedb
validators/loan.ak	c88d05da19cf858cfa0109e85fcab8fb04f43e5703bf22446e42bf416a8ff680
validators/pool.ak	25d788f2bedc2f8e8d853cc76118390d28758e083d69c61558a1eae78dd1be01
validators/config.ak	f511f9af937fa1845ecd24d478378f3c58817afbbae8ea7ba7f636267a9912b
lib/lending/types/loan_types.ak	cd919253aea189ee0041f31049cfd90d310f9ac545b3fbb90bbef91bec9a8d95
lib/lending/types/pool_types.ak	cbb24790f507b709ec881e397ae76dbbf3919d0e547b4b74391de07081914bc7
lib/lending/types/types.ak	997b876ae7d67295c4f541085fa8191fad029bfb9adf8db4f322278ec401b1f8
lib/lending/utills/loan.ak	b03b3b74e0d6c677e562225c1af5d6336511b24483afd1d673eca3a7de0f22be
lib/lending/utills/pool.ak	5a16343ff64e04262cdf0bc9ac5c5f90ef23700308f819849476c1b743e92793
lib/lending/utills/utills.ak	d9b5916e9c5cb36d5af2b4a60463f664a57eb82e12ca4e8c70a6abbefa243333
lib/lending/constants.ak	3910e642efc07f8a2eb8d23785511ccabd0de253acbf947d3f9582804d3d699

Parameter	Value
utxo_ref	8be7d08a82219a5bb9bd2a68a690c341e8d4e767799fd9c2ae15383faecd21a6#0
admin_kh	5a26c03b596f0e97273a881ad43db493d53cd27bfdcafaa23ce60848

Validator	Method	Hash (Blake2b-224)
config	mint	ec291a303b834997b9e4d9d5d661f9dcc1677967cd7893dc4c0d0cc5
id_token	mint	77019d836d9a5f1e1225ad0ea8262f99c14c89ae74a750bce29cee93
pool	spend	d064690601080228f9cbd54ca70a1590e9498c913b9fc7cbd845e888

loan	spend	fba91f46fc289ad8abe08d07488d8bb5a2c15c41d6903ce70b0ded94
------	-------	--

2 - Specification

The first part of any audit that Sundae Labs performs involves developing a deep and intimate understanding for what the client is trying to accomplish.

We first work with the client to develop a clear high level mission statement capturing the business goals of the project. Then, we translate that into an informal specification, which loosely outlines how to achieve that goal. Finally, we translate that, often with the help of the code they've written, into a detailed specification of how the protocol works.

As we make recommendations and findings, we update this specification.

We present the final version of each of these as part of our audit report.

2.a - High Level Objectives

- Allow Lenders to offer ADA loans to Borrowers, using the borrowers valuable NFTs as collateral
 - Lenders can set relevant parameters for their loans, such as principal, interest, acceptance collections, and duration
- Allow Borrowers to request a loan, offering their valuable NFTs as collateral
 - Lenders can set relevant parameters for their loans, such as principal, interest, NFT collateral, and duration
- Allow a lender to use the same underlying ADA for multiple different loan offers
 - For example, the lender may have 1000 ADA, and would be happy to loan it for **either** a Claymates NFT at 3% interest, or a Spacebudz NFT at 5% interest
- Support extension for a future feature:
 - In particular, the JPG Store team plans to add exactly one other feature that should be able to leverage the same underlying ADA
 - To minimize risk, extensibility beyond this is not needed
- Provide bulk execution, targeting roughly 5-10 of the same action in a single transaction

2.b - Detailed Specification

2.b.i - Definitions

- **Lender**

- A user offering ADA for a short term loan, and willing to receive a valuable NFT as collateral in case of a **Default**.

- **Borrower**

- A user placing an NFT as collateral, in return for a short term loan of ADA.

- **Default**

- When a borrower does not pay back the ADA plus interest before the loan expires, and the NFT offered as collateral is forfeited to the **Lender**.

- **Interest**

- An additional ADA balance added to the borrowed amount when repaying a loan

- **Loan Offer**

- A loan offered by a **Lender**, that can be accepted by a **Borrower**. Configured to some set of **Terms**. When accepted, the ADA is drawn from a **Bid Pool**.

- **Loan Request**

- A request, placed by a **Borrower**, which requests some amount of ADA for a set interest and period of time, offering a specific NFT as collateral.

- **Loan**

- A **Loan Offer** or **Loan Request** that has been accepted, and is waiting for repayment or default.

- **Destination**

- The address and datum that the ADA or NFT are sent to when repaying or defaulting on a loan. Includes the datum for composability with other protocols.

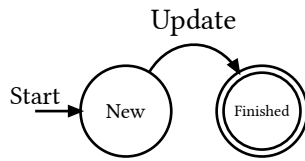
- **Terms**

- The parameters of a loan offer or request, such as the principal, interest rate, duration, and acceptable collateral.
- **Bid Pool**
 - A pool of ADA that can be issued out to satisfy a **Loan Offer** from the same owner.
- **Batcher**
 - An automated entity that defaults loans on behalf of the **Lender** .
- **ID Token**
 - A token that represents the owner of a loan contract; Used to authenticate usage of that owners bid pools.
- **Config**
 - A UTXO holding the protocol configuration, namely the list of allowed scripts that can hold ID tokens.

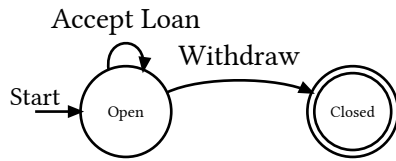
2.b.ii - State Machine

Overall, there are four state machines implemented by the JPG Store lending protocol.

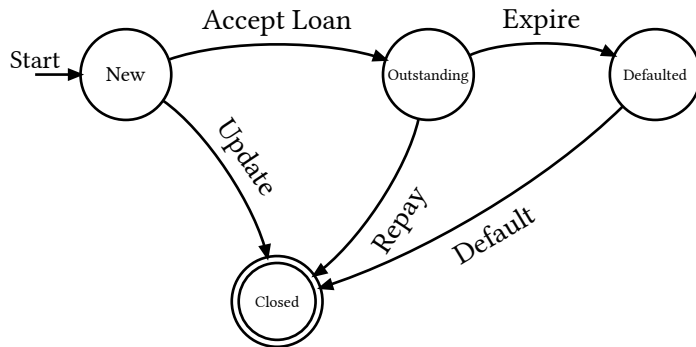
First, the Config UTXO, which can be initialized, and then updated once.



Second, a Lender can create a Bid Pool which holds ADA that can be used to satisfy one of the Loan Offers the Lender has created. These bid pools follow the following simple state machine:



A Lender can create a Loan Offer, which follows the following state machine:



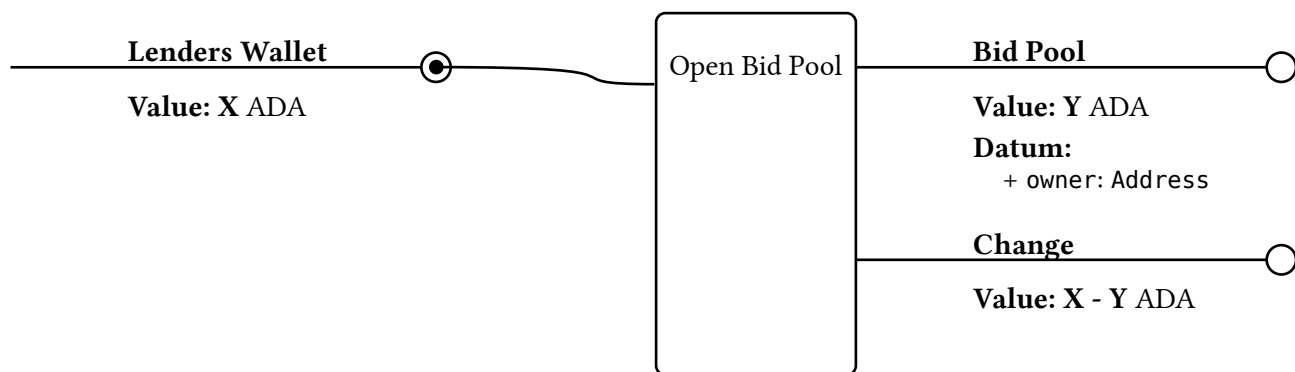
Finally, a Borrower can create a Loan Request, which follows the same state machine as above.

2.b.iii - Transactions

There are 9 core transactions that are possible in the JPG Store lending protocol. We provide a sketch of each one below:

1. Open Bid Pool

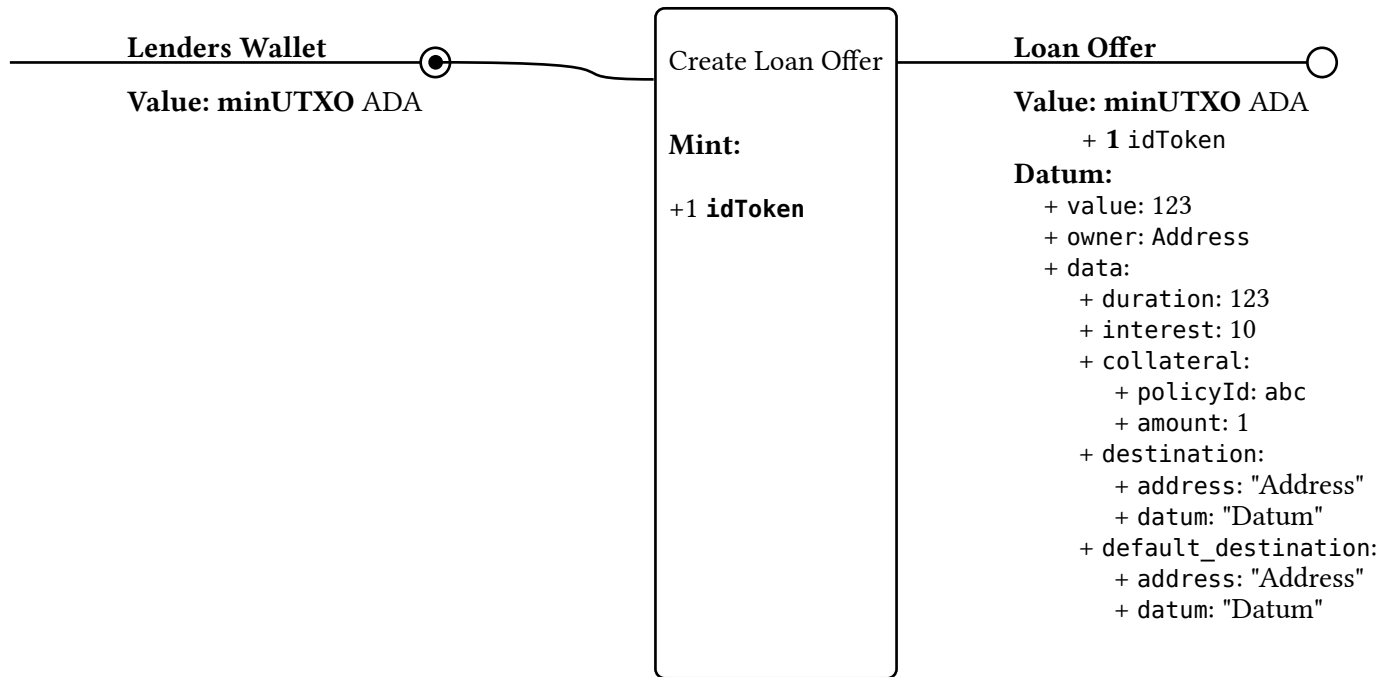
1. A Lender opens a bid-pool
2. This transaction is very simple; there is no validation, and the user is expected to construct the datum correctly.



1. Create Loan Offer

1. A Lender creates a Loan Offer

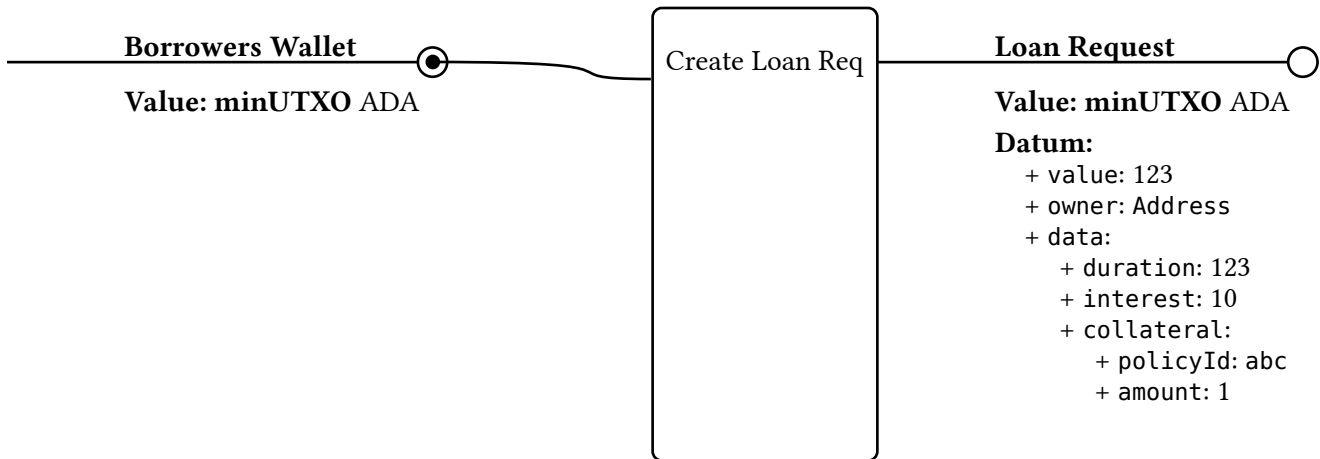
2. Lender is expected to mint an ID Token into the output, along with a valid datum representing their offered Terms .



1. Create Loan Request

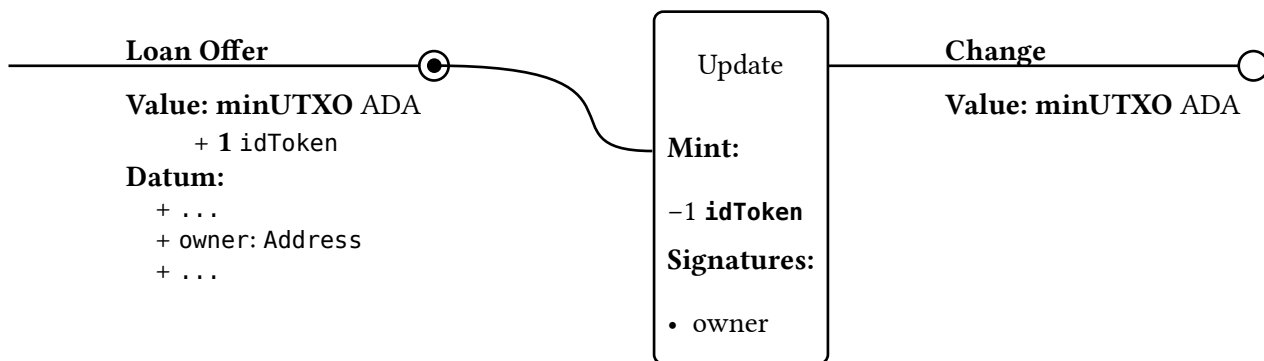
1. A Borrower creates a Loan Request

2. Unlike a Loan Offer, a Loan Request has no ID token.



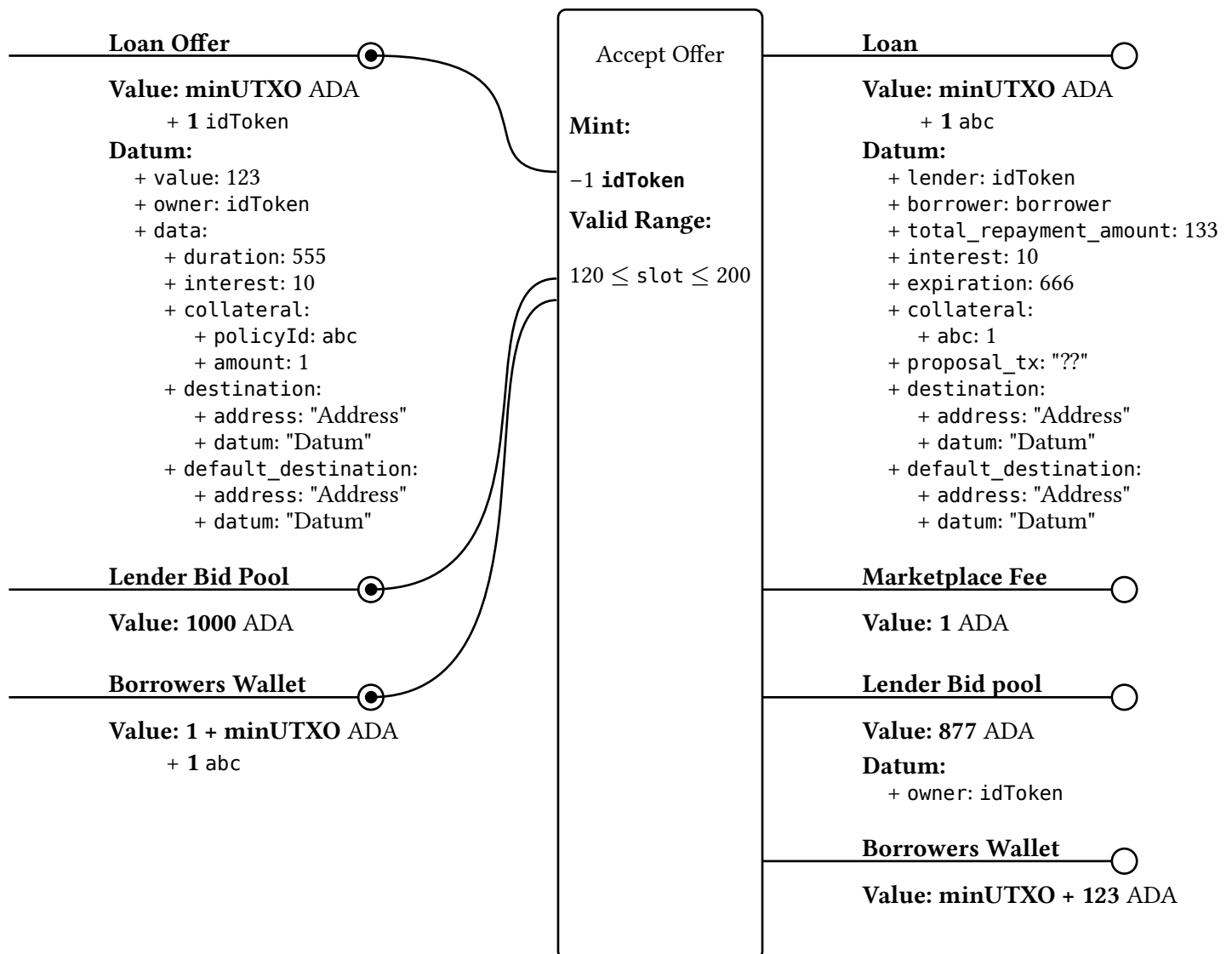
1. Cancel Loan Offer / Loan Request

1. The Owner of a Loan Offer or Loan Request can sign the transaction to cancel it.
2. There must be no ID Token s present on the ouputs, to prevent escapement. This means the contracts do not support updates.
3. A Loan Request is similar.



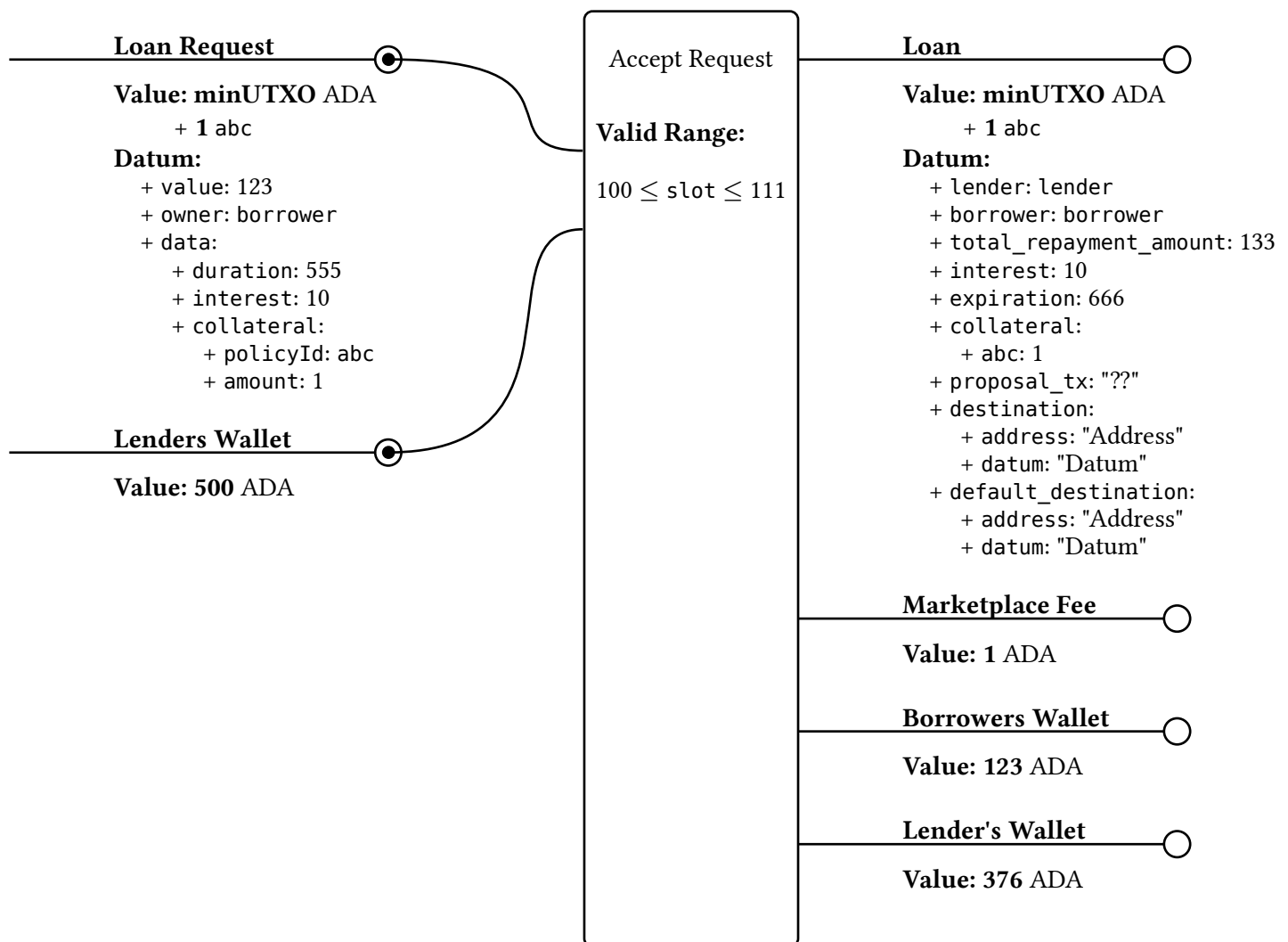
1. Accept Loan Offer

1. A Borrower can accept a Loan Offer by providing the necessary collateral
2. There must be no ID Token s present on the outputs, to prevent escapement (see Core Invariant 3).
3. The Borrower is expected to pay the marketplace fee.
4. Valid range must be no longer than 3 hours.
5. Must set the datum correctly (see Core Invariant 7)



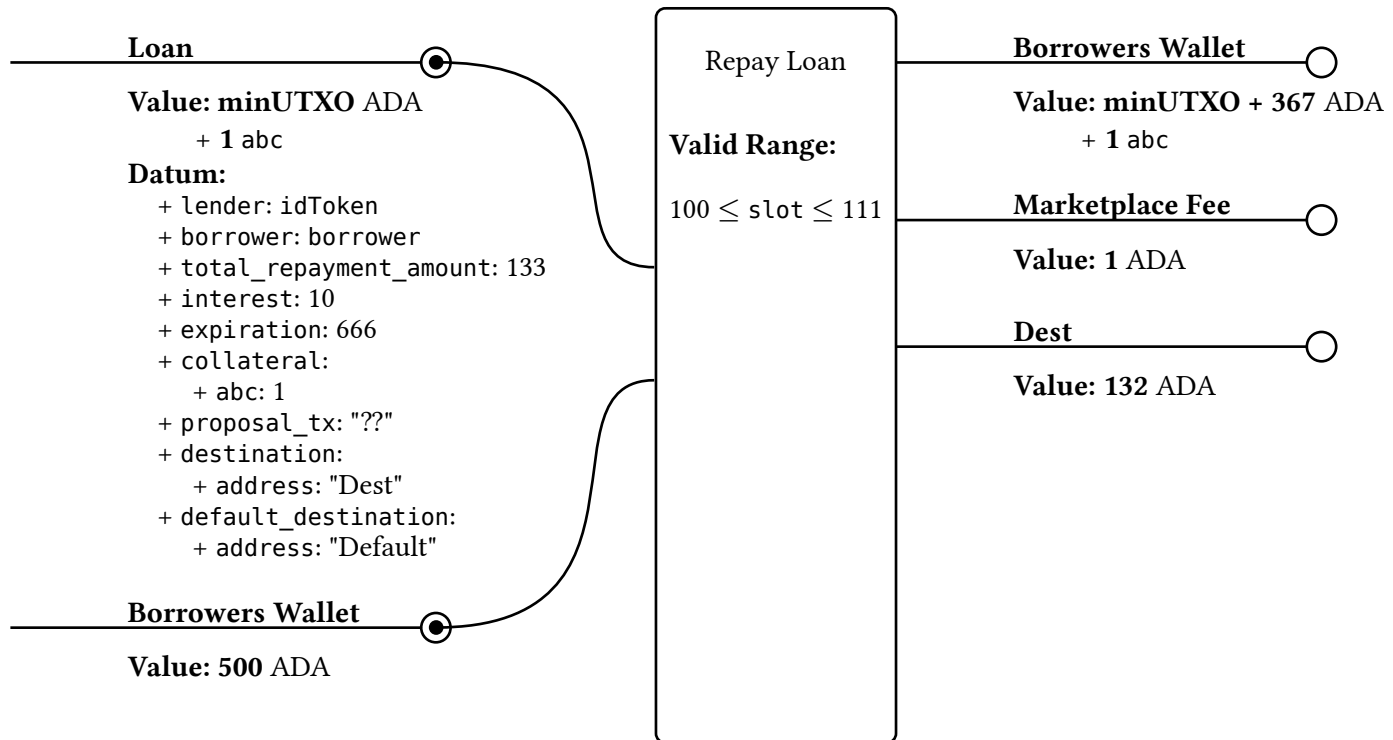
1. Accept Loan Request

1. A Lender can accept a Loan Request, providing the requested ADA and securing the collateral against default.
2. There must be no ID Tokens present in the outputs, to prevent escapement (see Core Invariant 3).
3. The Lender is expected to cover the marketplace fee.
4. Valid range must be no longer than 3 hours.
5. Must set the datum fields correctly (see Core Invariant 8).



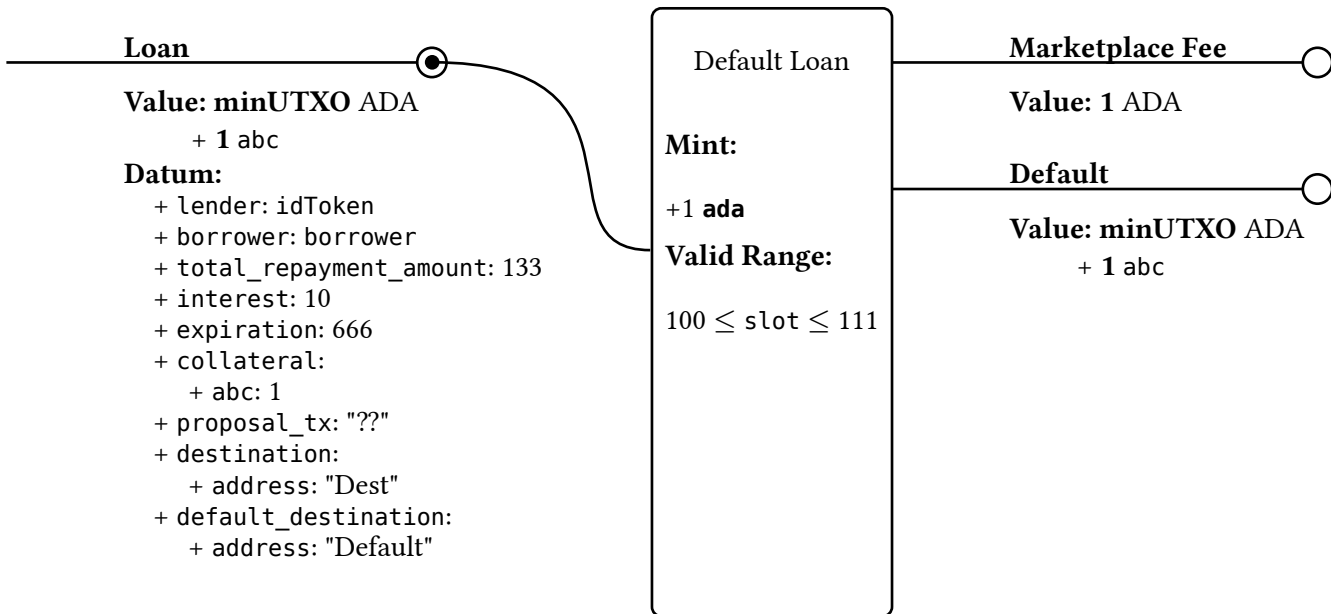
1. Repay Loan

1. Before the expiration has elapsed, the Borrower can repay the Loan plus interest to reclaim their NFT collateral.
2. The lender pays the marketplace fee out of their Loan proceeds.



1. Default Loan

1. After the expiration has elapsed, The Lender (or a batcher on their behalf) can claim the NFT held as collateral.
2. The Lender or Batcher is expected to pay the Marketplace fee.



2.c - Core Invariants

We've identified the following core invariants that must hold for the JPG Store Lending protocol.

1. A **Bid Pool** s can be reclaimed.
 1. That is, a UTXO at the **Bid Pool** address with a well-formed datum specifying the owner as Address must be spendable with the Withdraw redeemer if the transaction is signed by the payment credential of Address and is otherwise valid.
 2. See **JPG-400** for an argument of correctness.
2. A **Loan Offer** and **Loan Request** can be reclaimed.
 1. That is, a UTXO at the **Loan** contract address with a well-formed **Loan Offer** or **Loan Request** datum must be spendable with the CancelLoanOffer or CancelLoanDatum redeemer provided that it is signed by the payment credential of the owner field in the datum, and in the case of a **Loan Offer** , the ID token is burned.
 2. See **JPG-401** for an argument of correctness.
3. An **ID Token** cannot escape.
 1. That is, if a **ID Token** exists, it must be held at a JPG Store approved contract address.
 2. See **JPG-402** for an argument of correctness.
4. An **ID Token** cannot be forged.
 1. That is, if an **ID Token** with an asset name of a is minted, then the transaction must be signed by a.
 2. See **JPG-403** for an argument of correctness.
5. The **Config** can only be updated by JPG Store once.
 1. The **Config** is left open so that JPG Store can implement another feature utilizing ID tokens.
 2. However, it can only be updated once, and only by JPG Store, to place an upper bound on the risk this poses.
 3. See **JPG-404** for an argument of correctness.
6. A **Bid Pool** cannot be embezzled.
 1. That is, if a **Bid Pool** is spent, it must be because:
 1. The owner has signed the transaction;
 2. or to fund the acceptance one of a **Loan Offer** containing the owners **ID Token** .
 2. See **JPG-405** for an argument of correctness.
7. A **Loan Offer** can be accepted.

1. That is, a UTXO at the Loan contract address with a well-formed Loan Offer datum must be spendable with the `AcceptOffer` redeemer provided that:
 1. The output datum at the Loan contract is well-formed
 1. The lender, interest, collateral, destination, and `default_destination` are set to the corresponding values from the Loan Offer
 2. The total repayment amount is set to the Loan Offer value plus the Loan Offer interest
 3. The expiration is set to a POSIX time within a constant bound α of the transaction time plus the Loan Offer duration
 2. No ID Token s appear on any of the outputs
 3. Collateral matching the criteria from the Loan Offer is provided in the output locked with the Loan contract
 4. The specified ADA (and no more) is withdrawn from the Lender s Bid Pool
2. See JPG-406 for an argument of correctness.
8. A Loan Request can be accepted.
 1. That is, a UTXO at the Loan contract address with a well-formed Loan Request datum must be spendable with the `AcceptRequest` redeemer provided that:
 1. The output datum at the Loan contract is well-formed
 1. The borrower, interest, and collateral are set to their respective fields from the input Loan Request
 2. The total repayment amount is set to the Loan Request value plus the Interest
 3. The expiration is set to a POSIX time within a constant bound α of the transaction time plus the Loan Request duration
 2. The specified collateral is present in the output UTXO locked at the Loan contract
 3. The specified ADA is paid to the owner field from the Loan Request
 2. See JPG-407 for an argument of correctness.
9. Loan Offer s, Loan Request s, and Bid Pool s cannot be double satisfied.
 1. That is, if multiple operations are performed in one transaction, they must each be satisfied in aggregate, without relying on overlapping state.
 2. That is, the one output cannot be used to satisfy two different inputs.
 3. See JPG-408 for an argument of correctness.

10. A Loan can be repaid.
 1. That is, a UTXO at the Loan contract address with a well-formed Loan datum must be spendable before the expiration time, so long as:
 1. `total_repayment_amount` is paid to the lender at the `destination` address with the `destination` datum attached.
 2. The collateral NFTs are returned to the borrower address.
 2. See JPG-409 for an argument of correctness.
11. A Loan can default.
 1. That is, a UTXO at the Loan contract address with a well-formed Loan datum must be spendable after the expiration time, so long as:
 1. The collateral NFTs are transferred to the `default_destination` address with the `default_destination` datum attached.
 2. See JPG-410 for an argument of correctness.
12. The marketplace fee is paid.
 1. That is, when accepting a Loan Offer , Loan Request , and when repaying or defaulting a Loan , 1% of the interest in ADA must be paid to the JPG store marketplace address.
 2. See JPG-411 for an argument of correctness.

3 - Findings Summary

ID	Title	Severity	Status
<u>JPG-100</u>	ID tokens can escape allowed script addresses	Major	Resolved
<u>JPG-101</u>	Incorrect implementation of config token	Major	Resolved
<u>JPG-102</u>	Bid pools unusable	Major	Resolved
<u>JPG-103</u>	UTXO packing vulnerability	Major	Resolved
<u>JPG-104</u>	Loans can be repaid even after they have expired	Major	Resolved
<u>JPG-200</u>	Lender not paying marketplace fee	Minor	Acknowledged
<u>JPG-201</u>	Inefficient / Inflexible use of Address owners	Minor	Acknowledged
<u>JPG-202</u>	Cannot update a loan offer	Minor	Resolved
<u>JPG-203</u>	Cannot cancel <u>Loan Offer</u> s and <u>Loan Request</u> s in the same transaction	Minor	Resolved
<u>JPG-204</u>	Borrower can extend duration of loan	Minor	Resolved
<u>JPG-205</u>	Small DDOS attack vector	Minor	Resolved
<u>JPG-206</u>	Double satisfaction bug when repaying loan	Minor	Resolved

<u>JPG-207</u>	Loan Requests with ID tokens can be used to access borrowers bidpools	Minor	Resolved
<u>JPG-300</u>	Implementation of ID token limits flexibility	Info	Acknowledged
<u>JPG-301</u>	Inefficiency in double satisfaction code	Info	Acknowledged
<u>JPG-302</u>	Exposure to future risk of <u>ID Token</u> usage	Info	Acknowledged
<u>JPG-303</u>	Marketplace Address is hard coded	Info	Acknowledged
<u>JPG-304</u>	Small social engineering attack vector	Info	Acknowledged
<u>JPG-305</u>	Composability suffers when accepting loan requests	Info	Acknowledged
<u>JPG-306</u>	Lender can hijack ADA stake when accepting loan request	Info	Acknowledged
<u>JPG-400</u>	Argument for: <u>Bid Pool</u> s can be reclaimed	Witness	Resolved
<u>JPG-401</u>	Argument for: <u>Loan Offer</u> s and <u>Loan Request</u> s can be reclaimed	Witness	Resolved
<u>JPG-402</u>	Argument for: <u>ID Token</u> s cannot escape	Witness	Resolved
<u>JPG-403</u>	Argument for: <u>ID Token</u> s cannot be forged	Witness	Resolved
<u>JPG-404</u>	Argument for: The <u>Config</u> can only be updated by JPG Store once	Witness	Resolved

<u>JPG-405</u>	Argument for: A <u>Bid Pool</u> cannot be embezzled	Witness	Resolved
<u>JPG-406</u>	Argument for: A <u>Loan Offer</u> can be accepted	Witness	Resolved
<u>JPG-407</u>	Argument for: A <u>Loan Request</u> can be accepted	Witness	Resolved
<u>JPG-408</u>	Argument for: <u>Loan Offer</u> s, <u>Loan Request</u> s, and <u>Bid Pool</u> s cannot be double-satisfied	Witness	Resolved
<u>JPG-409</u>	Argument for: A <u>Loan</u> can be repaid	Witness	Resolved
<u>JPG-410</u>	Argument for: A <u>Loan</u> can default	Witness	Resolved
<u>JPG-411</u>	Argument for: The marketplace fee is paid	Witness	Resolved

JPG-100 - ID tokens can escape allowed script addresses

Severity	Status	Commit
Major	Resolved	fcca423c195afa39e6c1e57a8790910048c72610

Description

ID Token are access keys to a Bid Pool . As such, they must always be held at approved contract addresses which ensure that they are burned as they are spent. This is to prevent a free-floating ID token, which could be used to drain a users Bid Pool .

However, both Loan Offer s and Loan Request s are held at the same whitelisted address; the minting policy for the token only checks for the address, not the datum. Thus, a user could mint an ID token when opening a Loan Request .

A Loan Request does not check that the ID token is burned when it is canceled or accepted.

We classify this is a Major vulnerability, as opposed to critical, because it does require the owner of the Bid Pool to mistakenly mint their own ID token when they're not "supposed to"; However, we still consider this Major rather than Minor because it would be fairly easy to socially engineer someone into doing so.

Recommendation

We recommend one of two alternatives:

- remove the ID token altogether, and replace the BidPool with an extensible "authorization" script specified in the datum
- remove the check for quantity burned, and replace it with a check in the loan contract (which happens regardless of redeemer) that any ID tokens present in the outputs are at allowed addresses. This prevents the ID token from escaping.

Resolution

This issue was resolved as of commit fcca423c195afa39e6c1e57a8790910048c72610.

JPG-101 - Incorrect implementation of config token

Severity	Status	Commit
Major	Resolved	0e07ff3a2288a99f5038d6cb5ad1efd46379469e

Description

The **Config** token is intended to allow the allow list to be updated once.

To do this, the following code is employed:

```
fn spend(data: AllowList, ...) -> Bool {
  expect Some(script_input) = list.find(inputs, fn(in) { in.output_reference ==
out_ref })
  let output = script_input.output
  let Output { address, datum, .. } = output
  expect InlineDatum(dat) = datum
  expect datum: AllowList = dat
  ...
  let AllowList { allow_list } = data
  and {
    ...
    allow_list == list.push(datum.allow_list, redeemer),
  }
}
```

In particular, both `allow_list` and `datum.allow_list` come from the datum of the input being spent.

This means it would never be possible to update the **Config**.

Recommendation

Instead, you should scan for the output and compare the datum of the output, like so:

```
expect Some(script_output) = list.find(outputs, fn(out) { value.quantity_of(out.value,
policy, asset_name_constant) == 1 })
expect InlineDatum(out_datum) = script_output.datum
expect out_datum: AllowList = out_datum
let AllowList(output_allow_list) = out_datum
...
let AllowList(input_allow_list) = data
```

```
and {  
    ...  
    output_allow_list == list.push(input_allow_list, redeemer)  
    ...  
}
```

Resolution

This issue was resolved as of commit [0e07ff3a2288a99f5038d6cb5ad1efd46379469e](#).

JPG-102 - Bid pools unusable

Severity	Status	Commit
Major	Resolved	cdf55e23cecd04f38a1139f75cd3dc97151d6fe7

Description

Currently, the bid pool feature is effectively useless, as they cannot be spent in a transaction with an input from a script that is not in the allow list... and it is not safe to put the pool contract in the allow list!

```
expect address_in_allowlist(input_pkh, allowlist)
```

Recommendation

Either remove this check, or move it after the `if input_pkh == contract_pkh` check on the next line. It is safe to remove this check because:

- Any input from another script will not be the bid pool, and will not have an ID token
- Therefore it will fall through to the `else` clause, and be treated like any normal wallet input
- None of the other invariants are impacted by the presence of an extra script

Resolution

This issue was resolved as of commit `cdf55e23cecd04f38a1139f75cd3dc97151d6fe7`.

JPG-103 - UTXO packing vulnerability

Severity	Status	Commit
Major	Resolved	37c7febd7588c0984b1413b83207b179a5662463

Description

When accepting a loan offer, the validator folds over the collateral value from the loan output datum (which must be equal to the loan output value sans lovelace).

For each `[(policyId, assetName, quantity)]` tuple, we increment the accumulator if `policyId` is the collateral requested by the lender, and pass the accumulator through unchanged if it is not.

This means that the borrower can add arbitrary other tokens and NFTs to the loan output as collateral. They could do this to force garbage, scammy, or otherwise offensive tokens on the lender; More importantly, in theory, they could pack the UTXO so full of assets that trying to default or repay the loan exceeds cardano execution units. We have not confirmed if this is actually the case for the loan contract through testing, but the risk is there so we consider it a major vulnerability.

Recommendation

Simply fail if any token other than the requested policy ID is provided. This forces the borrower to provide **only** the requested collateral, and no others.

Resolution

This issue was resolved as of commit 37c7febd7588c0984b1413b83207b179a5662463.

JPG-104 - Loans can be repaid even after they have expired

Severity	Status	Commit
Major	Resolved	96e1421e4a37960fc1fc92cc2e80035f5894a0b2

Description

When repaying a loan, we have the following check:

```
expect !check_loan_expired(...)`
```

However, this returns false if the transaction lower bound is negative infinity; indeed, a user could just set this to 10 years ago, and voila, the transaction lower bound would not be after the expiration date.

Recommendation

You need to either:

- enforce a maximum length on the transaction valid range, such that the borrower cannot set the transaction valid from too far in the past
- have a separate helper, `check_loan_not_expired` that checks that the transaction upper bound is before the expiration

Resolution

This issue was resolved as of commit 96e1421e4a37960fc1fc92cc2e80035f5894a0b2.

JPG-200 - Lender not paying marketplace fee

Severity	Status	Commit
Minor	Acknowledged	

Description

In the requirements, when a loan Defaults, the Lender is responsible for paying the JPG Store marketplace fee.

Also, the Default can be executed by either the Lender or a Batcher.

However, since the loan UTXO only holds the minUTXO ada, then if the Batcher executes the default, there will not be enough ADA on the UTXO to cover the fee and pay the marketplace fee.

Recommendation

We recommend requiring the Lender or the Borrower to leave enough ADA for the minUTXO, the transaction fee, and the marketplace fee in the case of a default.

Resolution

This issue was acknowledged by the project team with the comment: Noted. This is a longer term solution we thought of (batchers that return your ADA) however probably something we could invest out of pocket.

JPG-201 - Inefficient / Inflexible use of Address owners

Severity	Status	Commit
Minor	Acknowledged	

Description

The **Bid Pool**, **Loan Offer** and **Loan Request** datums all use an Address as the owner. However, this has three main drawbacks:

1. It is less efficient; you only check for a signature from the payment credential, and the **Destination** is stored separately. The attached staking key isn't used, resulting in potentially significant performance costs. If the intention is to use the staking script to index things offchain, just store the staking credential, and request a signature from the staking credential to reclaim the order.
2. It is inflexible: it only allows normal wallets to own these UTXOs, and you cannot have a "protocol owned bidpool" for example.
3. Perhaps most concerningly, if someone pays into the contract with a script payment credential (which is suggested as valid by the type), the funds will be locked forever.

Recommendation

We recommend specify the owner using something like the `aicone` library provided by Sundae Labs; this would allow you to store a single public key hash in the normal case, or allow arbitrary scripts to own the UTXO.

Resolution

This issue was acknowledged by the project team with the comment: Acknowledged.

JPG-202 - Cannot update a loan offer

Severity	Status	Commit
Minor	Resolved	fcca423c195afa39e6c1e57a8790910048c72610

Description

One stated requirement was that the owner of a loan offer should be able to update the details by signing the transaction. However, such a transaction is only valid if the ID token is burned, meaning you would first have to pay things back to your wallet, and then reopen a new loan offer in a separate transaction.

Additionally, if a user accidentally sends assets to the load offer address with the correct datum, but doesn't mint the ID token, that would permanently lock their assets at this UTXO.

Recommendation

Instead of checking that the ID token is burned, check that the only ID tokens in the outputs are held at the allowed addresses; This allows you to carry through an ID token to an updated loan offer, while still forcing that the token gets burned if there are no outputs to the allowed addresses.

This also resolves the secondary burned assets issue.

Resolution

This issue was resolved as of commit fcca423c195afa39e6c1e57a8790910048c72610, with the comment: Given the solution to [JPG-100](#), this is no longer a prioritized requirement.

JPG-203 - Cannot cancel Loan Offer s and Loan Request s in the same transaction

Severity	Status	Commit
Minor	Resolved	fcca423c195afa39e6c1e57a8790910048c72610

Description

The implementation of `check_one_token_burned_per_offer_input` currently makes the assumption that every input with the Loan contract payment credential is a Loan Offer. If this function is called, and there is a Loan Request in the inputs, the expect at `lib/lending/utils/loan.ak:29` will fail. This means that you must only be cancelling Loan Offer s, or only cancelling Loan Request s in a single transaction, and you cannot mix them.

Recommendation

We recommend replacing the expect `datum: LoanOfferDatum = dat` with a `when dat is`, to gracefully handle other datums.

Resolution

This issue was resolved as of commit `fcca423c195afa39e6c1e57a8790910048c72610`, with the comment: Given the solution to JPG-100, this is no longer a prioritized requirement.

JPG-204 - Borrower can extend duration of loan

Severity	Status	Commit
Minor	Resolved	189690d8392a18a792abf504542a4753c3b5cf08

Description

The contracts enforce that when a Loan Request or Loan Offer is accepted, the loan expiration is set to `valid_to + request_duration`.

However, there is no check on the `valid_to` field: in theory, a borrower could set the transaction `valid_range` to (10 minutes ago, 5 years from now). Then, the `valid_to + request_duration` for a 5 day loan would be 5 years and 5 days in the future.

Luckily, the cardano ledger does not allow a transaction with a `valid_to` further than one stability window in the future. This limits the length of time that a borrower can add to a loan to roughly 36 hours. Because of this, we categorize this as a minor vulnerability, rather than a major or critical vulnerability.

Recommendation

We recommend enforcing a maximum length of the validity range before checking the loan duration. For example, a window of 3 hours means that the borrower can delay the loan by at most 3 hours.

Resolution

This issue was resolved as of commit 189690d8392a18a792abf504542a4753c3b5cf08.

JPG-205 - Small DDOS attack vector

Severity	Status	Commit
Minor	Resolved	<code>cdf55e23cecd04f38a1139f75cd3dc97151d6fe7</code>

Description

The bidpool script ensures that the sum of ada on the bidpool inputs from a given owner, minus the amount withdrawn for each loan request, equals the sum of the ADA on bidpool outputs for a given owner.

This allows the borrower to re-balance a users bidpools, potentially maliciously. They could, for example, split the users bidpool into dozens or potentially hundreds of tiny bidpools to annoy the lender and force them to pay transaction fees on re-aggregating or withdrawing the funds; They could also aggregate multiple bidpools used for concurrency into a single bidpool, which would hurt the user experience of other borrowers interacting with a popular lender.

Note that this can be done even without accepting a loan offer, meaning a malicious user can essentially just constantly spend the same UTXO, so long as they don't withdraw any funds from the bid pool.

Recommendation

To prevent this, we recommend one of the following:

- Ensure that at least one loan offer is being accepted, which would mean that at the very least an attacker would have to satisfy the loan offer
- Ensure that the amount of ADA in the bidpool on the outputs is strictly less than the amount on the inputs; this is equivalent to bullet point 1, as in order for remaining to decrease, there must be a loan offer (or equivalent contract) being accepted.
- If there are no loan offers, ensure that the lovelace at the bidpool **increases** by 1 ADA, so that at the very least it benefits the lender.

Resolution

This issue was resolved as of commit `cdf55e23cecd04f38a1139f75cd3dc97151d6fe7`.

JPG-206 - Double satisfaction bug when repaying loan

Severity	Status	Commit
Minor	Resolved	eff799cc9e5b01733c7b80be513f1630a53fd8b9

Description

If two loans use the same asset as collateral (i.e. if this is ever used for semi-fungible or fungible assets), then someone else can repay your loan and keep a portion of the collateral.

For example, suppose there are two loans, each using 1 A as collateral.

Each one will check that the sum of the assets paid to borrower is greater than or equal to 1. But that means that someone else can repay the loan, and keep the extra A.

Now imagine there are 5 such loans. The attacker could keep 4 of the 5 A tokens.

Recommendation

Since it is expected that the borrower is repaying their own loans, remove the check that the collateral is paid to the borrower address, and instead check that the borrower signs the transaction. They can authorize whatever they want to happen to their collateral, such as paying it into another loan request.

The important thing is to prevent someone else paying off your loans without your permission.

Alternatively, if you do want to allow paying off someone elses loan, ensure that only one Loan is on the inputs to prevent double satisfaction.

Resolution

This issue was resolved as of commit `eff799cc9e5b01733c7b80be513f1630a53fd8b9`.

JPG-207 - Loan Requests with ID tokens can be used to access borrowers bid-pools

Severity	Status	Commit
Minor	Resolved	270f5adc70388c38d955cd3ae4aa072ebcc0f451

Description

ID tokens have an allow list of contracts that they can be minted into. This includes the loan contract, which includes Loan requests, even though loan requests have no purpose for an ID token.

The ID token is meant to allow loan **offers** to authorize access to a bid pool of ADA.

However, if a borrower accidentally mints an ID token into the loan request contract (for example, through social engineering), this would allow a malicious actor to access any bidpools they have outstanding.

This is because the bidpools don't discriminate: they simply look at the first integer field of the datum holding the ID token, and allow that amount to be withdrawn.

So someone could lend the borrower their own money to lock up the collateral, which would then have to be repaid to the lender, or the loan defaulted.

Recommendation

While this is low risk, because the steps required to get there involve a social engineering attack, if you want to address this, you need to disallow the loan request from being considered for the bid pool. One simple way to do this is to make the `LoanRequestDatum` incompatible with the `LoanOfferDatum`, such as by swapping the `value` and `owner` fields.

Another way would be to disallow ID tokens on the inputs when accepting a loan request.

Another way would be to use separate contracts for the loan requests and loan offers.

Resolution

This issue was resolved as of commit `270f5adc70388c38d955cd3ae4aa072ebcc0f451`.

JPG-300 - Implementation of ID token limits flexibility

Severity	Status	Commit
Info	Acknowledged	

Description

The stated goal of the **ID Token** is to give flexibility, so that other features and contracts could make use of the **Bid Pool**. However, the **Bid Pool** contract enforces at least some structure to the datum and layout of transactions that utilize it.

Recommendation

A more flexible approach would be for the **Bid Pool** to specify an authorization field in the datum, which refers to a script hash that must be present in the transaction withdrawals.

Then, by withdrawing “0” staking rewards, a custom script could implement **arbitrary** logic regarding the transaction; new features could be opted-into by users via a new authorization script.

Resolution

This issue was acknowledged by the project team with the comment: Noted; The large refactor is out of scope.

JPG-301 - Inefficiency in double satisfaction code

Severity	Status	Commit
Info	Acknowledged	

Description

The guard against double satisfaction happens for every single spent input; Each one of these UTXOs is going to repeat the same logic to scan over the outputs for the correct UTXO. This results in roughly quadratic increase in execution unit costs.

Recommendation

If you moved most of the logic for `AcceptLoanRequest`, `AcceptLoanOffer`, `Default`, and `Repay` into a staking script, and then had each UTXO spend validator check for the presence of that staking script, then they could check that all orders are satisfied in a single pass, without output tagging. This would significantly boost the number of operations you could perform in a single transaction.

Resolution

This issue was acknowledged by the project team with the comment: Acknowledged.

JPG-302 - Exposure to future risk of ID Token usage

Severity	Status	Commit
Info	Acknowledged	

Description

In order for a Loan Request to be cancelled or accepted, the attached ID token must be burned. This helps maintain one of the core invariants, and ensures that a users Bid Pool cannot be touched without their permission.

However, while the logic for this is currently sound, it is exposed to future protocol risk. Because the list of allowed scripts into which the ID Token can be paid is dynamic, **and not opt-in at the time the user creates their bid pool**, the interactions between these scripts must be considered with exponentially increasing care.

For example, if another allowed script is added, it must check that no loan offers are present in the inputs before doing the same check.

Imagine two allowed contracts that are allowed to hold ID tokens: the Loan contract and XYZ, some future feature the protocol is extended with.

Now imagine that a user mints two ID tokens, one paid into the loan contract, and the other paid into XYZ.

When the user cancels their loan offer, they must burn a number of ID tokens equal to the number of LoanOffer inputs. But, if they also spend the UTXO at contract XYZ, there are two ID tokens to burn. The second contract must be careful to disallow LoanOffer inputs, otherwise they might be subject to a double satisfaction problem, and the ID Token might be allowed to escape.

Recommendation

We recommend any of the following solutions:

- Refactor away from the “id token” model, and use an “authorization” parameter on the bid pool, so the user opts-in to the features they support.
- Make one ID token per contract, and have the bid pool list the policy IDs it wants to allow as an access token.

- Hard-code the allowed destinations for the ID token, and ensure those contracts ensure the ID token is not present on the outputs.

Resolution

This issue was acknowledged by the project team with the comment: We recognize the risk of future updates, but we believe this is acceptable because it would require all of the following:

- JPG Store keys to be compromised
- The allow list to be updated using those keys
- A user to be tricked into minting an ID token into an unsupported contract

As a compromise, we are going to lock the configuration after one update. This will allow us to leave the feature set open for a future planned feature, but prevent unbounded updates many years into the future.

JPG-303 - Marketplace Address is hard coded

Severity	Status	Commit
Info	Acknowledged	

Description

The marketplace fee is hard coded. If this address is compromised, because of the use of bidpools and an allow list, this might make deployment of a replacement contract difficult.

Recommendation

Store the marketplace address in the config UTXO, and allow this to be updated if the allow list is not updated.

Resolution

This issue was acknowledged by the project team with the comment: Acknowledged

JPG-304 - Small social engineering attack vector

Severity	Status	Commit
Info	Acknowledged	

Description

When accepting a loan offer, the loan contract scans for the **first** loan output satisfying the condition. However, it is possible to pay out into the loan contract multiple times for a loan, or pay out for a loan that isn't involved in the transaction, or pay out as an already-expired loan UTXO, etc.

This means that there is a social-engineering attack where the user is tricked into paying extra collateral into the loan contract without borrowing the ADA they would be entitled to from it. Normally a user is reviewing a transaction carefully, but given that it's a script output with a datum that is likely not human readable, and the second output looks similar to the first, the user might just assume it's a visual glitch.

Recommendation

If this is seen as worth fixing, the way we recommend fixing it is to check that the number of loans on the output is equal to the number of inputs at the loan contract. This, combined with the existing check that each one has an output tagged with `proposal_tx`, will ensure they are 1:1.

Resolution

This issue was acknowledged by the project team with the comment: Acknowledged

JPG-305 - Composability suffers when accepting loan requests

Severity	Status	Commit
Info	Acknowledged	

Description

The check against double satisfaction when accepting a loan request involves identifying the output with a specific datum; However, this prevents the borrower from directing the borrowed ADA to any other DeFi protocol. For example, if a borrower wanted to immediately use the borrowed ADA to swap on a DEX, they would have to do so in two separate transactions: one to accept the loan request, and another to initiate the swap.

Recommendation

It would likely be difficult to solve this without dramatically rewriting the scripts, or compromising on performance.

If you do want to solve this with minimal changes, we recommend applying the same trick as is used in bidpools: sum up all loan request inputs with a specific destination, and then sum up all ADA being sent to that destination, and ensure one is greater than or equal to the other.

Resolution

This issue was acknowledged by the project team with the comment: Acknowledged

JPG-306 - Lender can hijack ADA stake when accepting loan request

Severity	Status	Commit
Info	Acknowledged	

Description

The lender accepting a loan request is forced to pay out the requested principle to the borrowers payment key hash. However, no restriction is placed on the staking key, meaning the lender can still leave their stake key attached. If the borrower doesn't notice and re-attach their own stake key, they won't earn ADA staking rewards on this ADA, despite paying interest on the loan. Additionally, the franken address may disrupt some tools in the ecosystem from correctly identifying the ADA in the users wallet.

Recommendation

Use a `Destination`, which specifies the whole address, rather than just checking the payment credential.

Resolution

This issue was acknowledged by the project team with the comment: Acknowledged

JPG-400 - Argument for: Bid Pools can be reclaimed

Severity	Status	Commit
Witness	Resolved	

Description

Here, we make an argument for the first core invariant, that a Bid Pool can always be reclaimed.

- If a Bid Pool UTXO is being spent, the pool validator will be executed with a script purpose of Spend, along with the UTXO datum, the redeemer, and the script context.
 - See `validators/pool.ak:18`
- A well-formed datum will correctly pass the injected typecheck code (`validators/pool.ak:18`), meaning it will have an owner field of type Address
- The script context, constructed and presented by the node, must be well-formed and in a known format, so the destructuring at `validators/pool.ak:19` cannot fail.
- The redeemer has two simple variants, Accept and Withdraw (`lib/types/pool_types.ak:9`); we assume for this invariant that it is Withdraw, meaning the branch condition for Withdraw will execute. (`validators/pool.ak:95`)
- We additionally assume that “well-formed datum” must have a `VerificationKeyCredential` for the payment credential; A script credential will result in funds locked forever (see JPG-201, `validators/pool.ak:96`)
- `list.has` is a standard library method which checks every element of a list to see if it contains a given element.
- In this case (`validators/pool.ak:99`), it returns True (and thus returns True for the whole validator) if the `datum_owner` payment credential is present in `transaction.extra_signatories`
- If the `datum_owner` payment credential is present in `transaction.extra_signatories`, then the Cardano ledger enforces that the transaction has a valid signature from a public key that hashes to this payment credential.
- Conversely, if the transaction is not signed by a public key which hashes to the `datum_owner` payment credential (and not otherwise rejected by the ledger), then said public key hash must not be in the `transaction.extra_signatories` field, and the `list.has` check will fail.

- At no point did we check any other condition, or require any other signature.
- Thus, the **Bid Pool** UTXO is spendable if and only if the transaction is signed by the payment credential of the `datum.owner` field.
 - Note: Technically, if the transaction is incidentally signed by nature of one of the inputs being from the `datum.owner`, but the `datum.owner` is not in the `extra_signatories` field, the transaction will fail. However, this is not a concern, as the user could just as easily include their credential in the `extra_signatories` field.

JPG-401 - Argument for: Loan Offer s and Loan Request s can be reclaimed

Severity	Status	Commit
Witness	Resolved	

Description

Here, we make an argument for the second core invariant, that a Loan Offer or Loan Request can always be reclaimed.

- We define a Loan Offer or Loan Request as a UTXO at the Loan contract address with the corresponding well-formed datum.
- Spending either of these UTXOs invokes the validator at `validators/loan.ak:21`, with the script purpose of `Spend`, along with the UTXO datum, the redeemer, and the script context.
- The datum is declared as type `Data`, meaning no additional typechecking is injected. (`validators/loan.ak:22`)
- The script context, constructed and presented by the node, must be well-formed and in a known format, so the destructuring at `validators/loan.ak:23+25` cannot fail.
- The expectation at `validators/loan.ak:28` must pass; this calls a utility function defined at `lib/lending/utls/utls.ak:35`.
 - This function returns true if any of the outputs (`lib/lending/utls/utls.ak:36`) have any assets (`lib/lending/utls/utls.ak:40+41`) with the `id_token_policy` (`lib/lending/utls/utls.ak:41`).
 - This constant must be set to `36b864176007c1a22ab3ebe87bce3a78436367eed07c4414d2f3e23c` to be valid.
 - The loan contract ensures that this method returns false; Therefore, any ID tokens present on the inputs must be burned for the transaction to be accepted by the Cardano ledger.
- The redeemer can be one of 6 different variants (`lib/types/loan_types.ak:56`), and the two that are relevant to this invariant are `CancelLoanOffer` and `CancelLoanRequest`
- The implementation of these two redeemers is nearly identical, so we will first address Loan Offer s (using the `CancelLoanOffer` redeemer), and then look at the differences.

- If the `CancelLoanOffer` redeemer is present, the branch at `validators/loan.ak:30` will be executed.
- At `validators/loan.ak:31`, the code `expect datum: LoanOfferDatum = raw_datum` will check that the datum is Plutus Data Constructor 1, with 3 fields: an integer, an address, a field of type `LoanRequestData`
 - Notably, `LoanOffers`, `LoanRequests`, and `Loans` are held at the same address, but use different types, meaning they all use constructor 0.
 - This means that there's a risk that if they are structurally equivalent, this typecheck will succeed.
 - However, each of the three will fail to typecheck as the others:
 - `LoanDatum` will fail to typecheck as the other two because the first field is an address, while `LoanOfferDatum` and `LoanRequestDatum` have an integer for the first field.
 - `LoanRequestDatum` and `LoanOfferDatum` will fail to typecheck as the other two because the third field, `data`, is of type `LoanRequestData` and `LoanOfferData` respectively.
 - `LoanRequestData` and `LoanOfferData` fail to typecheck as each other because the third field is `Value` and `(PolicyId, Int)` respectively.
- At `validators/loan.ak:32`, `datum.owner.payment_credential` is deconstructed as a `VerificationKeyCredential`
 - If the owner of a loan is set to a `ScriptCredential`, which is suggested as valid by the type, the funds will be locked forever. (See [JPG-201](#)).
- At `validators/loan.ak:35`, the contract ensures that `extra_signatories` contains the `payment_credential` from the `datum.owner` field. See the last two bullets from [JPG-400](#) for an argument as to why this is secure.
- Cancelling a **Loan Request** is identical, except for the datum type assertion. The exact same argument applies.
- Thus we can conclude that both **Loan Offer**s and **Loan Request**s with valid datums can be reclaimed if and only if someone is able to produce a signature from the owners wallet.

JPG-402 - Argument for: ID Token s cannot escape

Severity	Status	Commit
Witness	Resolved	

Description

Here we make an argument for the third core invariant , that an ID Token cannot escape. Specifically, we show that if an ID token exists, it must be held at a JPG Store approved contract address. Importantly, this property is critical to the argument in JPG-405 that ensures a users Bid Pool cannot be drained.

- First, we analyze transactions that mint an ID Token
 - The minting policy of an ID token is located at `validators/id_token.ak:8`
 - When the mint field of a transaction contains the ID token policy ID with either a positive or negative count, this validator will be executed with the provided redeemer and the script context.
 - The redeemer must be void, and the ledger will ensure that the script context is valid, meaning the destructure at `validators/id_token.ak:9` cannot fail.
 - At `validators/id_token.ak:12`, the validator ensures that it is being invoked with the script purpose of `Mint`. This prevents it from being used as a spending validator, to publish any certificates, or to withdraw any staking rewards, so we don't need to consider those cases.
 - The destructure of the transaction at `validators/id_token.ak:13` cannot fail.
 - At `validators/id_token.ak:20`, the validator invokes a helper function defined at `lib/lending/utls/utls.ak:8` to find the allowlist UTXO in the reference inputs.
 - This function uses a hard coded constant `config token` defined at `lib/lending/constants.ak:5`
 - There must be at least one reference input such that the quantity of the hard coded `config token` is greater than or equal to 1
 - This reference input must have an inline datum
 - This datum must be of type `AllowList`
 - This allow list is a list of script hashes
 - At `validators/id_token.ak:22`, the validator uses `list.foldl` over the transaction outputs, starting with 0

- For each output, we check if the output contains the ID token (using the policy ID from the script purpose, which is guaranteed by the ledger to be our own policy ID)
- If this output does not contain the ID token, we return the accumulator unchanged
- If it does, we ensure the payment credential of this output is in the allow list; this is done via a helper defined at `lib/lending/utls/utls.ak:25`
 - The helper checks that any element of the allow list is equal to `ScriptCredential(output.payment_credential)`
 - This prevents paying the ID token into a `VerificationKeyCredential`
 - It also prevents paying the ID token into any script not in the `allowList`
 - We assume this allow list is set correctly on startup by the JPG store team, and updated only once.
- We then increment the accumulator by 1
- ▶ Thus, `outputs_count` contains the number of outputs that contain at least one ID token.
- ▶ At `validators/id_token.ak:42` we match on `dict.get(to_dict(from_minted_value(transaction.mint)), policy)`
 - Note, this cannot be `None`, or the ledger would not be running this minting policy
 - We then `foldl` over the `(asset_name, quantity)` pairs returned by the `dict.get`, starting from 0
 - If the quantity is less than 1 (i.e. the ID token is being burned, since we know its not zero), we return the accumulator unchanged, effectively allowing anyone to burn an ID token
 - If the quantity is greater than or equal to 1 (i.e. a token is being minted), we ensure that `asset_name` is in the `extra_signatories`, which means the ledger will enforce that the transaction is signed by a public key with that hash.
 - We then increment the accumulator by the quantity minted.
- ▶ Thus, `mint_count` contains the sum of all ID tokens being minted across all inputs.
- ▶ The minting validator only passes if `mint_count` (the number of distinct ID tokens minted) is equal to `outputs_count` (the number of outputs containing ID tokens); effectively, this ensures that each output has exactly one ID token.

- Note that if there were ID tokens on the inputs, this could violate the above argument: the number of minted ID tokens may be three, and there may be two ID tokens on the inputs, so if three outputs contain ID tokens, at least one of them has more than one ID token.
- This doesn't allow the ID token to be present an unauthorized address, however, and as we will see in the rest of the argument, it would be impossible for ID tokens to be present on the inputs.
- We now know that any ID tokens that come into existence must exist at an address in the `allow_list`. Thus, if we demonstrate that each of the contracts in `allow_list` prevent the token from escaping, we have proven that the ID token cannot exist at another contract address, and thus cannot be used to steal funds from a users bidpool.
- The current contract intended to be in the `allow_list` is the **Loan** contract.
- However, at `validators/loan.ak:28`, the contract ensures that, if a UTXO holding an ID token is spent, there are no ID tokens on the outputs. This means that either:
 - There are ID tokens present on the inputs, and they are all burned, so that the transaction balances.
 - There are no ID tokens present on the inputs, and none are minted.
- Thus it is impossible for any ID tokens on the inputs to be spent to a different address.
- We cannot know what the behavior of the second contract that will be put in the `allow_list` is. Therefore, there is a risk that:
 - A malicious actor obtains access to the admin keys
 - A malicious contract is added to the `allow_list` which does not enforce this constraint
 - A user is asked to sign a transaction minting an ID token to this malicious contract
 - This malicious contract then allows the users bidpool to be drained
- This risk is deemed managable by the JPG Store team, but any future contract should be carefully audited to ensure it does not allow ID tokens to escape.

JPG-403 - Argument for: ID Tokens cannot be forged

Severity	Status	Commit
Witness	Resolved	

Description

Here we make an argument for the **fourth core invariant**, that an ID Token cannot be forged.

Specifically, this means that if an ID Token with a specific asset name is minted, it must be signed by (and thus authorized by) the user it refers to.

- When a token is minted, the cardano node ensures that the specific minting policy script is run with a script purpose of Mint, and invokes it with a redeemer and a script context.
- In this case, that is the ID token minting policy, located at `validators/id_token.ak:7`
- The script context, constructed and presented by the node, must be well-formed and in a known format, so the destructuring at `validators/id_token.ak:8` cannot fail.
- The script purpose must be mint, and so the expectation at `validators/id_token.ak:11` must pass.
- The script then uses a `find_allowlist` utility function, defined at `lib/lending/utls/utls.ak:8` to find the appropriate reference input
 - This method fetches a hard-coded constant policy ID and token name (`lib/lending/utls/utls.ak:9`, defined `lib/lending/constants.ak:5`)
 - It then scans the transaction reference inputs for the first reference input that has at least one of these tokens
 - We assume that the JPG Store team is the only one that can mint or spend these tokens; the truth of this, or the security of the keys holding these reference tokens is outside the scope of this audit.
 - This reference input must have an inline datum of type `AllowList` (defined at `lib/lending/types/types.ak:7`)
 - This UTXO is expected to contain a carefully selected list of script hashes; Refer to the concerns in JPG-302.
- The utility function above must return a valid allow list, or it must fail.
- The minting policy then folds over the transaction outputs, starting with 0 (`validators/id_token.ak:22`):

- If the output contains any token with a policy ID equal to the ID token being minted (`validators/id_token.ak:27-32`)
 - The address attached to that output must be in the `allow_list` (`validators/id_token.ak:33`);
 - This utility simply checks that the output payment credential is a script credential with the same script hash as one of the scripts in the `allow_list`.
 - The accumulator is incremented
- Note that the `policy_id` check above **might** be susceptible to the “zero value” ledger quirk.
 - It is possible to construct the bytes for a transaction with a mint field that has the ID token policy ID, but with a value of 0
 - In theory, the ledger is supposed to filter this out and so `value |> policies |> list.has(policy) (validators/id_token.ak:27)` should return false.
 - However, Sundae Labs is not 100% sure this is the case; We have anecdotally seen transactions fail because of a 0 value in the mint field, meaning there may be situations where the ledger presents an unnormalized Value.
 - For the avoidance of doubt, you may want to check that `quantity_of` is `> 0` instead.
- Thus, `outputs_count` is the number of outputs that have **any** `id_token`, while ensuring that those ID tokens are present at allowed addresses.
- Next, the minting policy performs a similar fold over the minted assets starting from 0
 - At `validators/id_token.ak:42`, the code `minted_assets |> from_minted_value |> flatten` produces a list of `(policy_id, asset_name, amount)` triples
 - For each asset, if the `policy_id` is equal to the policy being minted, and the quantity is greater than or equal to one
 - we expect the `extra_signatories` to include the `asset_name`; i.e. in order to mint an ID token, the “owner” of that token must sign the transaction;
 - Then we increment the accumulator by the quantity being minted
- We thus have a sum of the number of ID tokens being minted (ignoring those that are burned), and we know that each one is authorized for minting.
- The minting is allowed if the number being minted is equal to the amount on the outputs; In a practical sense, since all contracts that are in the allow list force no ID tokens to be present on the outputs, this means that every ID token must be in a separate output.

- Thus, ID tokens cannot be forged or duplicated, as the transaction must be signed by the user it refers to.

JPG-404 - Argument for: The Config can only be updated by JPG Store once

Severity	Status	Commit
Witness	Resolved	

Description

Here we make an argument for the **fifth core invariant**, that the Config can only be updated by the JPG Store once.

Specifically, after the the protocol is booted with a single script in the `allow_list`, it is important to place a finite upper bound on the risk from updates to the `allow_list`.

If the `allow_list` could be updated arbitrarily, this becomes an open risk to the protocol indefinitely.

To address this, the JPG Store team has written a contract to hold the Config UTXO which only allows the allow list to be updated once.

- The Config token can only ever have a single instance.
 - At `validators/config.ak:19`, in the config token minting policy, the script checks the mint field of the transaction for tokens with its own policy ID.
 - At `validators/config.ak:29`, one of the inputs must be the parameterized `utxo_ref`, which can only be spent once
 - At `validators/config.ak:28`, this amount must be 1
 - At `validators/config.ak:29`, the transaction must be signed by the admin public key hash
 - At `validators/config.ak:30`, the single config token must be paid to the config address
 - This uses a utility function at `validators/config.ak:67`
 - The utility checks that every output either does not contain the config token, or if it does, the payment credential must be a script credential with the payment script hash equal to the policy ID.
 - Thus, only one config token can ever exist, and it must be initially paid to itself.
- The config token can only be spent by the admin, and only to add one extra contract to the allow list
 - The spend validator is implemented at `validators/config.ak:36`
 - The script finds it's own output, using the `out_ref` from the script purpose in the standard way, at `validators/config.ak:37-52`

- The script ensures that the config token is paid back into the config script using the same utility as the mint, at `validators/config.ak:59`
- The script ensures the transaction is signed by the admin, at `validators/config.ak:60`
- The new allow list is identified by looking for the config token, of which we know there must be only one, at `validators/config.ak:44`
- The new output must have an inline datum that exactly matches the previous allow list, plus one new script hash passed as the redeemer; this is checked at `validators/config.ak:61`
- The allow list must be 2 items or less, checked at `validators/config.ak:62`
- Thus, the following paths are the only valid sequences for this UTXO
 - It is initialized with 0 elements in the allow list, and then up to two scripts are added in two separate transactions
 - It is initialized with 1 element in the allow list, and then one more is added later on
 - It is initialized with 2 or more elements in the allow list, and can never be updated.

JPG-405 - Argument for: A Bid Pool cannot be embezzled

Severity	Status	Commit
Witness	Resolved	

Description

Here we make an argument for the sixth core invariant, that a Bid Pool cannot be embezzled.

Specifically, this means that, other than a withdraw, a bid pools funds can only be spent if an ID token belonging to the owner is present in the inputs, and only in an amount in accordance with the loan offer. Combined with the fact that ID tokens can only be at approved addresses, and cannot escape, this ensures that it is safe to rely on the logic in the loan offer contract to ensure the correct collateral is secured.

- The Bid Pool UTXO is locked by the contract defined at `validators/pool.ak:18`
- When this UTXO is spent, the validator is invoked with the script purpose of `Spend`, along with the UTXO datum, the redeemer, and the script context.
- We assume the datum is valid, thus it passes the injected type check for the `Datum` type defined at `lib/lending/types/pool_types.ak:3`
- We assume the redeemer is `Accept`, as we've already explored the security of `Withdraw` for another invariant.
- The script context passed from the node must be well formed, so the destructures at `validators/pool.ak:19+25` must be valid.
- The UTXO is being spent, so the script purpose must be `Spend`, at `validators/pool.ak:26`
- The script then finds the allow list in the reference inputs, at `validators/pool.ak:27`; We make an argument for the soundness of this in JPG-402
- The script then identifies the input being spent using the standard library function for this purpose at `validators/pool.ak:29`
- We then `foldl` over the transaction inputs starting with 0
 - The objective of this fold is to count how much ADA should be **left** in the bidpools owned by `datum.owner` at the end
 - That is, we will increase the accumulator by all the bidpools owned by `datum.owner`, and decrease it by each loan offer being accepted with an ID token from `datum.owner`

- First, the script destructures each input at `validators/pool.ak:39-45`
- The script then checks if the input is coming from a script at `validators/pool.ak:47`
 - If it is not, then it's just a normal wallet address, and it can't be a bidpool or a loan offer, so we can pass the accumulator through unchanged (`validators/pool.ak:67`)
 - If it is a script input, then it is either the bidpool script, a loan offer, or some other irrelevant script
 - If the input is locked by the Bid Pool script (`validators/pool.ak:48`)
 - We check if the owner of the bidpool is the same as the owner from the datum being spent. Note that the utility method used to check this is slightly confusingly named. (`validators/pool.ak:49` and `lib/lending/utls/pool.ak:17`)
 - The `is_this_contract_input` method does a type assertion on the input datum; however, this method is only called if the input is already determined to be at the bid pool contract, so this will not fail on other non-bidpool datums.
 - This means if we have multiple bidpools from the same owner, they will all sum to the total being considered (`validators/pool.ak:50`)
 - And if we have bidpools from other owners, the accumulator is passed through unchanged.
 - Otherwise, we check if it's a relevant offer input (`validators/pool.ak:54`)
 - This uses a helper method defined at `lib/lending/utls/pool.ak:23`
 - The datum must be an inline datum, of type `Offer` (`lib/lending/utls/pool.ak:29+30`)
 - Note that `Offer` datum is anything with a `Value` and an `Owner` field, to leave room for future extensions. (`lib/lending/types/types.ak:11`)
 - If the owner in the datum is not equal to the owner of the bidpool, this is not a relevant offer for the purposes of allowing withdrawals from the bidpool, and the accumulator is unchanged (`lib/lending/utls/pool.ak:32+37`)
 - This **is** a relevant offer if the datum owner is the same as the owner of the bidpool, and the UTXO has an ID token from the owner (`lib/lending/utls/pool.ak:35`)
 - If it is not a relevant offer input, we return the accumulator unchanged (this offer is not allowed to withdraw from this bidpool; it may withdraw from other bidpools, or be satisfied directly) (`validators/pool.ak:64`)

- If it is relevant, then we decrease the accumulator by the offer value. In the case of a Loan Offer, this allows the loan amount to be withdrawn from the bidpool.
- The presence of the ID token ensures that the validator holding the ID token will enforce the appropriate logic (in the case of a Loan Offer, that the appropriate collateral is provided).
- We now know the amount of ADA that must be left in the bidpools owned by `datum.owner` on the outputs.
- This remainder must not be negative (`validators/pool.ak:74+89`)
- In the case that the remainder is positive, the script then sums up the total ADA from bid pools on the outputs owned by `datum.owner` (`validators/pool.ak:76`). This uses a helper function defined at (`lib/lending/pool.ak:41`).
 - The helper function folds over the outputs, starting with 0
 - For each output, if the address is the bid pool address, the datum must be an inline datum of type `PoolDatum`.
 - If so, and the owner is equal to the **input** datum owner, the accumulator is increased by the amount of lovelace on this output. "Bid Pools" pools to have **more** ADA on the outputs; This allows a borrower to meet the minUTXO requirement by providing a small amount of extra ADA.
- Thus, we've demonstrated that:
 - Bidpools can only be spent if the amount of ADA withdrawn is less than or equal to the amount of ADA authorized by satisfied Loan Offers containing the owners ID token.

JPG-406 - Argument for: A Loan Offer can be accepted

Severity	Status	Commit
Witness	Resolved	

Description

Here, we make an argument for the **seventh core invariant**, that a Loan Offer can be accepted.

That is, a UTXO serving as a Loan Offer can be spent alongside a bid pool to borrow the offered ADA, so long as sufficient collateral matching the lenders criteria is locked by the loan script in the outputs.

- To accept a Loan, the borrower spends a Loan Offer UTXO, locked by the validator at `validators/loan.ak:22`.
- Thus, the ledger will invoke the script with the datum, redeemer, and script context.
 - The datum on this validator is declared as `Data`, so no typecheck is injected for the datum.
- The script context and transaction must correctly destructure. `validators/loan.ak:23+25`
- The script then checks to ensure that there are no ID tokens on the outputs. In the case of a Loan Offer, this ensures that the incoming ID tokens are burned. This is important for the argument in JPG-402.
- The script then branches on redeemer; the one relevant to this invariant is `AcceptLoanOffer`, which contains an index of the fee output.
- The UTXO is being spent, so the script purpose must be `Spend`, at `validators/loan.ak:134`
- The script computes a “datum tag”, which is the `blake2b_256` hash of the serialized `out_ref` being spent, which uniquely identifies this input. This is then attached to the output paying the jpg store fee, to prevent double satisfaction of the marketplace fee. This is important for the argument in JPG-411.
- The script then destructures the datum and the loan offer data at `validators/loan.ak:143`.
 - If the datum is invalid, pointing to a Loan Request, or a Loan, the script will fail here because the types are structurally different, ensuring that we’re working with a valid Loan Offer.
 - Note that a completely invalid datum will also fail, which we consider acceptable.
- The script identifies the exact input being spent by scanning the transaction inputs for the `out_ref` from the script purpose. (`validators/loan.ak:153`)

- The script then finds the output that corresponds to this accepted **Loan Offer**, using a utility function. (validators/loan.ak:159)
 - This utility function is defined at lib/lending/utls/loan.ak:13
 - It is passed the list of transaction outputs, the payment credential of the Loan contract, and the out_ref of the **Loan Request** that is being accepted.
 - It uses the list.find standard library function to scan the outputs for the first output matching this criteria:
 - The payment credential must equal the payment credential of the Loan contract
 - The datum must be an inline datum
 - the datum must be of type LoanDatum
 - The proposal_tx field on the output datum must equal proposal_tx
 - The above ensures that if multiple loan offers are accepted in the same transaction, they cannot be satisfied by the same output loan.
 - Note that there may be multiple outputs which have proposal_tx set to out_ref; in this case, only the first one will be found. The second loan would be equivalent to just paying directly to the script, and is not validated in any way. This opens the door to a minor social engineering attack, as described in **JPG-304**
 - The helper then expects the output datum to be an inline datum of type LoanDatum, and returns the output and the datum lib/lending/utls/loan.ak:31-33
- The destructure of the output loan must succeed, as we already asserted on the type in find_loan (validators/loan.ak:162)
- Both ends of the transaction must be Finite bounds, and the difference between the upper and lower bound must be at most 3 hours (validators/loan.ak:173-177). Because the ledger will enforce the valid range of the transaction, this means that the transaction will settle on chain at most 3 hours before the valid_to, and at most 3 hours after the valid_from
- At validators/loan.ak:193 and validators/loan.ak:195, we check that $\text{math.abs}(\text{loan_expiration} - (\text{valid_to} + \text{offer_duration})) \leq 10$
 - valid_to is at most 3 hours (10800000 milliseconds) after the actual slot the transaction is included in;

- that means that `loan_expiration` can be at most 10800010 milliseconds after `slot + offer_duration`
- `valid_to` is at a minimum equal to the slot the transaction is included in.
- that means that `loan_expiration` can be at most 10 milliseconds before `slot + offer_duration`
- At `validators/loan.ak:172` we check that the loan output value without lovelace is exactly equal to the value recorded in the loan datum; that is, the datum must contain the cardano native tokens at its own output.
- At `validators/loan.ak:178`, we destructure the collateral requested by the **Loan Offer** this is the policy ID and quantity of collateral that the lender has indicated they desire for the loan.
- Thus, the script needs to check that `loan_output.value` must contain at least quantity tokens of policy `policy_id`
 - At `validators/loan.ak:180` we perform a fold over `loan_collateral`, the value from the datum; but, we just checked that `loan_collateral` is equal to the `loan_output.value` without lovelace, therefore we will iterate over every cardano native token on the output.
 - We fold, starting from 0
 - Each element is a `(policy, assetName, quantity)` tuple. (`validators/loan.ak:185`)
 - If the policy ID is the one requested by the lender (`validators/loan.ak:186`), we increment the accumulator by the quantity of that token present on the output (`validators/loan.ak:187`)
 - Otherwise, there is a token that wasn't requested as collateral on the loan output, so we fail the transaction.
 - With the quantity of requested collateral tokens computed (`validators/loan.ak:179`), we check that this quantity is equal to the requested quantity in the large expectation at `validators/loan.ak:195`.
- Additionally, in that large block, we check that:
 - The owner of the Loan Request is recorded as the lender in the output datum
 - (Note, the borrower can be set to anything, because the borrower is the one executing the transaction)
 - The total repayment amount on the output datum is equal to the borrowed principle plus the interest
 - The loan interest is copied from the offer interest

- the destination and default destination are copied from the offer
- Finally, we check that the marketplace fee is paid. See **JPG-411** for more details on this.
- Thus, we the loan offer can be accepted if:
 - The requested collateral is provided into a loan output datum
 - The loan datum is set correctly
 - The owner, collateral amount, interest, and destination are copied over
 - The total repayment amount is set to the principal plus the interest
 - The expiration is set to within 3 hours and 10 milliseconds of the actual transaction time plus the duration
 - The JPG Store marketplace fee is paid
- And this invariant holds.

JPG-407 - Argument for: A Loan Request can be accepted

Severity	Status	Commit
Witness	Resolved	

Description

Here, we make an argument for the **eighth core invariant**, that a **Loan Request** can be accepted. That is, a UTXO serving as a **Loan Request** can be spent to lock the offered collateral, so long as the lender pays the borrower the requested amount of ADA.

- In order to accept a Loan request, a lender spends a UTXO locked by the validator at `validators/loan.ak:22`.
- Thus, the ledger runs the contract with the input datum, the redeemer, and the script context.
- We validate that there are no ID tokens on the outputs (`validators/loan.ak:28`);
 - In this case, an ID token isn't used to authorize the transaction, so there should be none on the inputs, but if there are, they are simply burned with no effect.
- In order to accept a loan request, the redeemer will be the `AcceptLoanRequest` variant, and the script purpose must be `Spend`. (`validators/loan.ak:48+49`).
- The script then computes a “datum tag”, which is the `blake2b_256` hash of the serialized `out_ref` being spent, which uniquely identifies this input (`validators/loan.ak:52`). In this case, it will be used in two places, addressed below.
- The Script identifies the exact input being spent by scanning the transaction inputs for the `out_ref` from the script purpose. (`validators/loan.ak:57`)
- We assert that the datum being spent is a `LoanRequestDatum` (`validators/loan.ak:62-71`)
 - A loan offer or bid pool will have structurally different datums, and so we can assume this is a `Loan Request`, and that the datum is valid.
- We use the `find_loan` utility function described in the previous invariant to find the first output that corresponds to the loan request being spent. (`validators/loan.ak:73`)
- Both the upper and lower bounds on the loan must be finite (`validators/loan.ak:79+80`)
 - We also check that `valid_to - valid_from` is at most 3 hours (10800000 milliseconds) (`validators/loan.ak:121`)

- This ensures that `valid_to` is at most 3 hours in the future.
- Additionally, we check that `math.abs(loan_expiration - (valid_to + request_duration))` is no more than 10. (`validators/loan.ak:122+125`)
- This ensures that the loan expiration is at most 3 hours and 10 milliseconds after the actual transaction time, and at least `request_duration` duration - 10 milliseconds after the actual transaction time.
- The script checks that the collateral recorded in the datum is exactly equal to the output value sans lovelace (`validators/loan.ak:95`)
 - Unlike when accepting a loan offer, we also check that the loan collateral exactly equals the requested collateral (`validators/loan.ak:125`). This is because the loan request specifies the exact value its offering, rather than just being a policy ID and quantity as in the loan offer case.
 - This means that the lender cannot pack the UTXO with unwanted NFTs / assets.
- The script now checks that the amount of ADA paid to the borrower by the lender accepting the request is sufficient
 - It folds over the transaction outputs, starting from 0 (`validators/loan.ak:97`)
 - For each output, if the output is paying funds to the borrower payment credential, and is tagged with the `datum_tag`, we increment the accumulator by the lovelace on the output (`validators/loan.ak:108-110`)
 - Otherwise, it is paying ADA to some other address, or for some other loan, so shouldn't count towards satisfying this loan.
 - This accumulated total must be greater than or equal to the amount requested by the borrow minus 1% (to cover the marketplace fee) (`validators/loan.ak:115`)
 - Next, several fields on the datum are checked at `validators/loan.ak:125`:
 - The `loan_borrower` must be set to the `request_owner`
 - (Note: the lender can be set to anything, because the lender is the one executing the transaction)
 - The loan repayment amount must be set to the requested amount plus the interest
 - The loan interest must be recorded correctly
 - Finally, the marketplace fee must be paid, which is checked at `validators/loan.ak:129-131`; See **JPG-411** for more details on this.

Thus, the loan request can be accepted if:

- The collateral is held in the loan output
- The datum is constructed correctly
- The borrower is paid the principal they requested

And this invariant holds.

JPG-408 - Argument for: Loan Offer s, Loan Request s, and Bid Pool s cannot be double-satisfied.

Severity	Status	Commit
Witness	Resolved	

Description

Here, we make an argument for the ninth core invariant, that a Loan Offer, Loan Request, or Bid Pool cannot be double-satisfied.

That is, if multiple Loan Offer s or Loan Request s are being claimed, a Loan is being repaid or defaulted, or multiple Bid Pool s are being used to claim one or more Loan Offer s, then each one is satisfied in aggregate: the sum of obligations from the inputs is satisfied, rather than just each obligation being satisfied independently.

In order to demonstrate this, we show two independent but complementary properties:

- None of the cases above can accidentally satisfy the other (ex: a loan offer being satisfied by a bid pool, etc.)
- None of the cases above can be double-satisfied by other instances of themselves (ex: two loan offers being satisfied by the same output)
- Loan Offer
 - Accepting a loan offer requires locking collateral into the loan script, and paying the JPG store fee.
 - For locking the collateral:
 - The collateral specified in the offer must be locked with a Loan datum;
 - The only other input that cares about a loan output is a Loan Request.
 - Since the loan output datum contains a `proposal_tx` field, and this must be set to the `out_ref` of the loan request or offer being accepted, it is impossible for one loan output to satisfy either another loan request or another loan offer.
 - Thus, this criteria is not vulnerable to double satisfaction.
 - For paying the jpg store fee:
 - The fee output is identified by the datum tag.
 - Thus, no two inputs can use the same fee output.

- The fee output cannot be confused for the loan output, which isn't tagged with the datum tag regardless.
- Thus, this criteria is not vulnerable to double satisfaction.
- Cancelling a loan offer requires the transaction to be signed by the owner, and imposes no restrictions on the outputs that are susceptible to double satisfaction.
- **Loan Request**
 - Accepting a loan request requires locking the collateral into the loan script, paying the principal amount to the borrower, and paying the JPG store fee.
 - For locking the collateral,
 - The collateral specified in the request must be locked with a Loan datum;
 - The only other input that cares about a loan output is a Loan Offer.
 - By the same `proposal_tx` argument, the loan output datum for a loan request must be distinct for all inputs.
 - Thus, this criteria is not vulnerable to double satisfaction.
 - For paying the principal amount to the borrower
 - All outputs paid to the borrower with the `datum_tag` are summed, and this sum must be greater than the principal amount minus the JPG store fee
 - Since we're only summing outputs with the `datum_tag`, this cannot consider outputs intended for any other input, regardless of the purpose.
 - In theory, if the jpg store marketplace address was the borrower, the fee UTXO would get included in this, but this is deemed unimportant, as at worst it would allow JPG store to avoid paying themselves the marketplace fee.
 - Thus, this criteria is not vulnerable to double satisfaction.
 - For paying the JPG store fee:
 - The fee output is identified by the datum tag.
 - Thus, a given fee output can only be used for a specific loan request input, and cannot be confused with another loan request, loan offer, loan repayment, or loan default fee payment.
 - Similarly to above, since the fee output is tagged with the same datum tag as the borrower payment, if the borrower was the marketplace address, the borrower payment could be used

to satisfy the JPG store fee requirement; But this would just be JPG store avoiding paying themselves, and so is not considered a risk.

- Thus, this criteria is not vulnerable to double satisfaction.
- Cancelling a loan request requires the transaction to be signed by the owner, and imposes no restrictions on the outputs that are susceptible to double satisfaction.
- **Loans**
 - Defaulting a loan requires paying the collateral to the specified destination, and the JPG store fee must be paid.
 - For paying the jpg store fee
 - The relevant payment to the marketplace address is identified by a datum tag; therefore it must be unique to this input, and another JPG store fee payment would not suffice.
 - If the destination was set to the marketplace address, with the datum tag as the datum, then the default payment could get confused with the jpg store payout;
 - However, this is impossible: it would require predicting what the transaction hash is, which is influenced by the destination datum you set, and would require calculating a hash fixed point.
 - Thus, this criteria is not vulnerable to double satisfaction.
 - For paying the collateral to the specified destination
 - The scripts look at the output directly after the JPG store fee output.
 - Since the jpg store fee output is unique, so too is the output immediately after it.
 - If it is another JPG store fee output, it will not be to the correct destination address.
 - If it were reused for another loan default, then the JPG store fee output would have to simultaneously have two separate datums.
 - Thus, this criteria is not vulnerable to double satisfaction.
 - Repaying a loan requires paying the principal and interest to the lender, and paying the JPG store fee.
 - For paying the jpg store fee
 - The same argument as above, using the datum tag, applies here.
 - For paying the principal and interest to the lender
 - The same argument as above, relating to the output immediately after the fee output, applies here.

- Note that the borrower is in charge of where the collateral goes; the script just checks that the borrower has signed the transaction. Thus, repayment of the collateral is not subject to any criteria, and cannot be vulnerable to double satisfaction.

- **Bid Pools**

- Bidpools perform a global check: the sum the lovelace from all bid pool inputs owned by their owner, and subtract off the amounts being withdrawn for each loan-like input authorized by an ID token.
 - Note that it is important that no ID token authorizes an input that have a datum structurally similar to Loan offers, but where the first integer field should be interpreted as anything other than “amount allowed to be withdrawn from the pool”.
- Because this scans over all inputs and all outputs, partitioning by owner, they will correctly consider the amount available in the bidpool, the amount withdrawn by loans, and the amount remaining in the bidpool, even if there are multiple bidpools or multiple loans on the transaction.

JPG-409 - Argument for: A Loan can be repaid

Severity	Status	Commit
Witness	Resolved	

Description

Here we make an argument for the **tenth core invariant**, that a Loan can be repaid correctly.

That is, after a loan is created, but before it has expired, the borrower can spend the UTXO to reclaim their collateral, so long as the borrower pays the loan amount plus interest to the lenders preferred repayment address.

- To repay a loan, we spend a Loan UTXO, locked by the validator at `validators/loan.ak:22`
- The ledger will run the script with the input datum, the redeemer, and the script context.
- The datum is declared as `Data`, so no typecheck for the datum is injected.
- The script context and transaction must correctly destructure. `validators/loan.ak:23+25`
- The script then checks to ensure that there are no ID tokens on the outputs. In the case of a Loan, this ensures that the incoming ID tokens are burned. This is important for the argument in JPG-402.
- In order to repay a loan, the redeemer will be the `Repay` variant, and the script purpose must be `Spend.(validators/loan.ak:240+241)`.
- The script computes a `datum_tag`, used to prevent double satisfaction, as the `blake2b_256` hash of the serialized `out_ref` being spent.
- We then check that the datum is a `LoanDatum` (`validators/loan.ak:251`); Because the Loan Request and Loan Offer are structurally different types, this will fail, and we can assume that the datum is valid.
- We then check that the transaction is signed by the borrower at `validators/loan.ak:263`. This ensures that only the borrower can pay back their own loan, and also that they authorize where the collateral is directed.
- We then look for the output at `fee_output_index` from the redeemer. (`validators/loan.ak:261`)
 - This must refer to an existing output, or the `list.at` would fail.
 - This must refer to an output the pays the marketplace fee, as it is checked at `validators/loan.ak:263`

- This must refer to a **distinct** output, if multiple loans are being repaid, because the fee output must be tagged with the `datum_tag`. That is, if two loans are being repaid, they must each refer to a distinct output for the marketplace fee, because that output cannot simultaneously have two different datums attached to it.
 - Therefore, the output **immediately after** this one must also be distinct. Keep this in mind for a point below.
- We check that the loan hasn't yet expired at `validators/loan.ak:267` using a helper function defined at `lib/lending/utls/loan.ak:115`
 - Note that this helper function checks if it **is** expired, and we check that it returns false.
- Finally, we check that the loan is repaid with interest at `validators/loan.ak:268`
 - The `check_loan_is_repaid` helper is defined at `lib/lending/utls/loan.ak:127`
 - We find the output that is 1 after the `fee_output_index` (`lib/lending/utls/loan.ak:134`)
 - This output must exist (or the transaction will fail) and be unique, because the fee output must be unique (remember the point from before)
 - This enforces an interleaved structure on the outputs: `feeOutput`, `loanRepayment`, `feeOutput`, `loanRepayment`
 - We then check three things:
 - The output must be paid to the lenders destination address (`lib/lending/utls/loan.ak:144+149`)
 - The output datum must be equal to the lenders destination datum (`lib/lending/utls/loan.ak:145+149`)
 - The lovelace on the output plus the marketplace fee must be greater than or equal to the total repayment amount (`lib/lending/utls/loan.ak:146+149`); This effectively checks that the principal, plus the interest are paid, and the a portion of the repayment is used to pay the JPG store marketplace fees.
- Thus, we have demonstrated that a loan can be repaid if:
 - The loan has not expired
 - The collateral is returned to the borrower
 - The principal and interest are paid to the lender at the correct destination
 - The JPG store marketplace fee is paid

And the invariant is satisfied.

JPG-410 - Argument for: A Loan can default

Severity	Status	Commit
Witness	Resolved	

Description

Here we make an argument for the **eleventh core invariant**, that a Loan can be defaulted correctly. That is, an outstanding loan can be spent after it has expired, so long as the correct collateral is sent to the lenders preferred default address.

- To default a loan, you must spend a Loan UTXO locked by the validator at `validators/loan.ak:22`
- The ledger will run the script with the input datum, the redeemer, and the script context.
- The datum is declared as `Data`, so no typecheck for the datum is injected.
- The script context and transaction must correctly destructure. `validators/loan.ak:23+25`
- The script then checks to ensure that there are no ID tokens on the outputs. In the case of a Loan, no ID tokens are expected to exist in the inputs, but this ensures that if they do, they are harmlessly burned.
- In order to default a loan, the redeemer will be the `Default` variant, and the script purpose must be `Spend`. (`validators/loan.ak:29+202+203`)
- Next, we calculate the `datum_tag`, which is the `blake2b_256` hash of the serialized `out_ref` being spent, which uniquely identifies this input. This is then attached to the output paying the `jpg` store fee, to prevent double satisfaction of the marketplace fee. (`validators/loan.ak:206`)
- Next, the datum is type-checked as a `LoanDatum`; A Loan Request or a Loan Offer have structurally different datums, so with this check, we can assume that the UTXO is actually a loan, and the datum is valid. (`validators/loan.ak:211`)
- We then check if the loan is actually expired (`validators/loan.ak:222`), using a utility function defined at `lib/lending/utls/loan.ak:115`.
 - If the transaction lower bound is infinite, we return false, and the transaction fails.
 - If the transaction lower bound (i.e. the earliest time this transaction can be) is finite, it must be after the loan expiration.

- We check that the JPG store fee is satisfied at `validators/loan.ak:224+225`. See **JPG-411** for an argument that this is correct.
- Finally, we check that the collateral was paid to the default address at `validators/loan.ak:231`, using a helper function defined at `lib/lending/utls/loan.ak:58`
 - Similar to repaying a loan, described in **JPG-409**, the collateral payment is expected to be exactly after the fee UTXO. Because the fee UTXO is datum tagged, it is unique, and so one output cannot satisfy multiple input loans.
 - We check that the collateral is paid to the default destination address (`lib/lending/utls/loan.ak:73+89`)
 - We check that the output has the default datum attached (`lib/lending/utls/loan.ak:74+89`)
 - We check that every asset listed in the collateral of the loan must be present on this output, with a quantity greater than or equal to the amount specified by the loan (`lib/lending/utls/loan.ak:75+89`)
- Thus, a loan can be defaulted if:
 - The loan has expired
 - The collateral is sent to the default address
 - The JPG store marketplace fee is paid

And the invariant holds.

JPG-411 - Argument for: The marketplace fee is paid

Severity	Status	Commit
Witness	Resolved	

Description

Here we make an argument for the **final core invariant**, that the JPG Store marketplace fees are always paid.

Specifically, a minimum of 1% of the loan principal must be paid to a known JPG store contract when

- A Loan Request is accepted
- A Loan Offer is accepted
- A Loan defaults
- A Loan is repaid

We assume that everything else about the transaction is valid, and all previous invariants hold.

- The marketplace fee is enforced via a `check_fee` utility function implemented at `lib/lending/`
`utils/loan.ak:41`
- This method is called in the following locations:
 - The `AcceptLoanRequest` redeemer branch at `validators/loan.ak:131`
 - The `AcceptLoanOffer` redeemer branch at `validators/loan.ak:199`
 - The `Default` redeemer branch at `validators/loan.ak:225`
 - The `Repay` redeemer branch at `validators/loan.ak:263` with
- In each case, the parameters are:
 - `fee_output`, found by indexing into the outputs with `fee_output_index` from the redeemer.
 - `datum_tag`, the `blake2b_256` hash of the serialized `out_ref` of the `Loan` input being spent, pulled from the script purpose
 - The loan value, calculated as:
 - Directly from the input datum when using the `AcceptLoanRequest` and `AcceptLoanOffer` redeemers, at `validators/loan.ak:64+131` and `validators/loan.ak:143+199` respectively

- `loan_total_repayment_amount` - `loan_interest` using the `Loan` datum when using the `Default` and `Repay` redeemers, at `validators/loan.ak:211+229` and `validators/loan.ak:250+263` respectively
- So, if this method correctly enforces that the JPG store fee is paid, then the invariant holds.
 - The fee output is deconstructed at `lib/lending/utills/loan.ak:42`
 - The address this output is paid to must be equal to the `fee_addr()` constant defined at `lib/lending/constants.ak:22`
 - The `datum_tag` must be equal to the datum attached to fee output.
 - This is meant to prevent double satisfaction: namely if multiple loans are repayed, defaulted, etc. in the same transaction, each one will have a distinct `out_ref`, and thus a distinct `datum_tag`, and thus must point to a distinct output.
 - If two redeemers point to the same output, the hash will be different, and one of them will fail.
 - If a redeemer points to an output that is not paying the fee, it will fail because it has the wrong address or datum.
 - Finally, the lovelace attached to the output must be greater than or equal to the fee calculated by `get_fee`, which simply divides by 100 to calculate 1 percent of the loan principle.
- Thus, the JPG store fee must always be paid in order to close a loan.
 - Note that it doesn't matter who pays this; A batcher processing a defaulted 100k ADA loan must pay 1k ADA to the JPG store, rather than the Lender. This puts both parties in a stand-off waiting for the other to pay the fee.

4 - Appendix

4.a - Disclaimer

This Smart Contract Security Audit Report (“Report”) is provided on an “as is” basis, for informational purposes only, and should not be construed as investment advice or any other kind of advice on legal, financial, or other matters. The entities and individuals involved in preparing this Report (“Auditors”) do not guarantee the accuracy, completeness, or usefulness of the information provided herein and shall not be held liable for any contents, errors, omissions, or inaccuracies in this Report or for any actions taken in reliance thereon.

The Auditors make no claims, promises, or guarantees about the absolute security of the smart contracts audited and the underlying code. The findings, interpretations, and conclusions presented in this Report are based on the best efforts of the Auditors and reflect their professional judgment at the time of the audit. The blockchain and cryptocurrency landscape is rapidly evolving, and new vulnerabilities may emerge that were not identified or considered at the time of the audit. As such, this Report should not be considered as a comprehensive guarantee of the audited smart contracts’ security.

The Auditors disclaim, to the fullest extent permitted by law, any and all warranties, whether express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement. The Auditors shall not be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this Report, even if advised of the possibility of such damage.

This Report is not exhaustive and is subject to change without notice. The Auditors reserve the right to update, modify, or revise this Report based on new information, subsequent developments, or further analysis. The Auditors encourage all interested parties to conduct their own independent research and due diligence when evaluating the security of smart contracts.

By using or relying on this Report, you agree to indemnify and hold harmless the Auditors from any claim, demand, action, damage, loss, cost, or expense, including attorney fees, arising out of or relating to your use of or reliance on this Report.

If you have any questions or require further clarification regarding this Report, please contact the contact@sundaeswap.finance.

4.b - Issue Guide

4.b.i - Severity

Severity	Description
Critical	Critical issues highlight exploits, bugs, loss of funds, or other vulnerabilities that prevent the dApp from working as intended. These issues have no workaround.
Major	Major issues highlight exploits, bugs, or other vulnerabilities that cause unexpected transaction failures or may be used to trick general users of the dApp. dApps with Major issues may still be functional.
Minor	Minor issues highlight edge cases where a user can purposefully use the dApp in a non-incentivized way and often lead to a disadvantage for the user.
Info	Info are not issues. These are just pieces of information that are beneficial to the dApp creator, or should be kept in mind for the off-chain code or end user. These are not necessarily acted on or have a resolution, they are logged for the completeness of the audit.
Witness	Witness findings are affirmative findings, which covers bizarre corner cases we considered and found to be safe. Not all such cases are covered, but when something is considered interesting, or might be a common question, we try to include it.

4.b.ii - Status

Status	Description
Resolved	Issues that have been fixed by the project team.
Mitigated	Issues that have a partial mitigation , and are now vulnerable in only extreme corner cases.

Acknowledged

Issues that have been **acknowledged** or **partially fixed** by the **project** team.
Projects can decide to not **fix** issues for whatever reason.

Identified

Issues that have been **identified** by the **audit** team. These are waiting for a response from the **project** team.

4.c - Revisions

This report was created using a git based workflow. All changes are tracked in a github repo and the report is produced using [typst](#). The report source is available [here](#). All versions with downloadable PDFs can be found on the [releases page](#).

4.d - About Us

Sundae Labs stands at the forefront of innovation within the Cardano ecosystem, distinguished by its pioneering development of the first Automated Market Maker (AMM) Decentralized Exchange (DEX) on Cardano. As a trusted leader in blockchain technology, we offer a comprehensive suite of products and services designed to enhance the Cardano network's functionality and security. Our offerings include Sundae Rewards, Sundae Governance, Sundae Exchange, and Sundae Taste Test—an automated price discovery platform—all available on a Software as a Service (SaaS) basis. These solutions empower other high-profile projects within the ecosystem by providing them with turnkey rewards and governance capabilities, thereby fostering a more robust and scalable blockchain infrastructure.

Beyond our product offerings, Sundae Labs is deeply committed to the advancement of the Cardano community and its underlying technology. We contribute significantly to research and development efforts aimed at improving Cardano's security and scalability. Our engagement with Input Output Global (IOG) initiatives, such as Voltaire, and participation in core technological discussions underscore our dedication to the Cardano ecosystem's growth. Additionally, our expertise extends to software development consulting services, including product design and development, and conducting security audits. Sundae Labs is not just a contributor but a vital partner in Cardano's journey towards achieving its full potential.

4.d.i - Links

Sundae Labs - <https://sundae.fi>

Sundae Public Audits - <https://github.com/SundaeSwap-finance/sundae-audits-public>