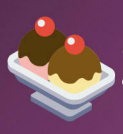
2024-10-01

Crashr

Orderbook Contracts

π Lanningham



Sundae Labs

Contents

1 - Audit Manifest

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

Software	Version	Commit
Orderbook Contracts	`1.0.0`	`e84e8bbc9bc3ea46a7e599a46e2870ec6c26120d`
Aiken Compiler	`v1.0.24-alpha`	`982eff4`

Filename	Hash (SHA256)
validators/ask.ak	`fc87591c0899a973dd2c59a41313a0ff0bb6475d9eca427ceda62d36e1057cc2`
lib/crashr/constants.ak	`baacd6bf1e90fe99734010e932bdbd58739821b7b2a85f18d84f4d6e8cc32265`
lib/crashr/types.ak	`9ff281712883a99a238ce2ea1b7257d58145c4848da04818e1a94d3e90b12056`
lib/crashr/utils.ak	`0b96eba36ba120ec7a327449a76163ebf8efd00264f09b75051610b94a7845ae`

Validator	Method	Hash (Blake2b-224)
ask	spend	`1e9fb7b37ebd6b54488031575fc78771d7f921c6b088c891565395eb`

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

The chief objective of the Crashr protocol is to establish a token orderbook marketplace on the Cardano blockchain.

2.b - Informal Specification

- Allow a seller to list a bundle of assets
  - These assets are locked in a UTXO, and can be purchased by anyone
  - The bundle of tokens can be claimed so long as a set of payout criteria are met
  - Each payout is an address, and an arbitrary amount of tokens that must be paid to that address
  - Payouts should also support wildcards, such as "at lest 4 NFTs from this policy ID"
- A seller should be able to cancel and reclaim their order, or update the details, with the appropriate signature
- Allow a buyer to claim the bundle, sending them to their own wallet
  - In order to do so, they must satisfy each of the payout criteria
- Collect a fee for the Crashr marketplace
  - Colloquially, we want to charge 2% of the "fungible value", if selling tokens
  - And 1 ADA for each non-fungible NFT sold

The following goals were discussed, and are explicitly out of scope:

- Alternatives; The seller cannot list different options for payments

2.c - Detailed

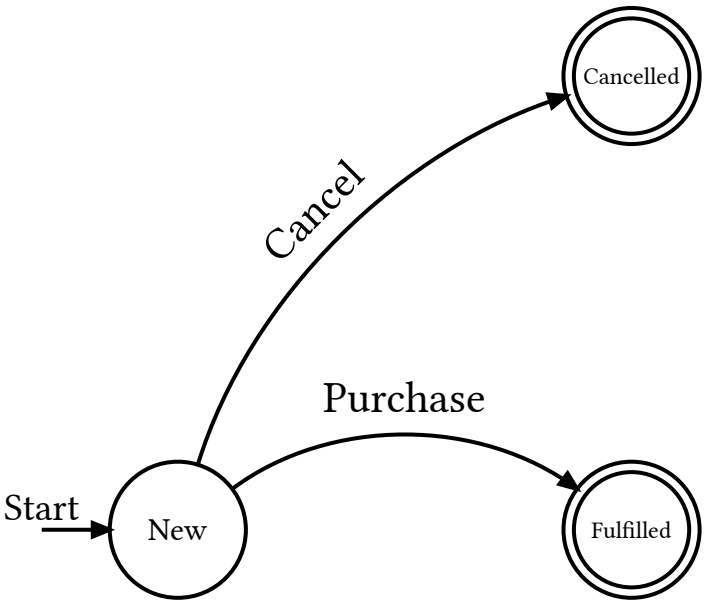
2.c.i - Definitions

- **Seller**
  - A user offering a bundle of assets for sale in return for satisfying a set of **Payout Criteria** .
- **Buyer**
  - A user who purchases a bundle of assets from a **Seller** by satisfying the **Payout Criteria** .
- **Listing**
  - A specific bundle of assets offered by a **Seller** for sale.
- **Payout Criteria**
  - A list of addresses, and values that must be paid to those addresses in order to claim a bundle of assets. Also supports wildcards by Policy ID
- **Fee**
  - The fee collected by the Crashr marketplace in return for executing a sale

2.d - State Machine

Before digging into each specific transaction, we outline the high-level state machines in the protocol.

The **Listing** UTXO implements the following state machine:

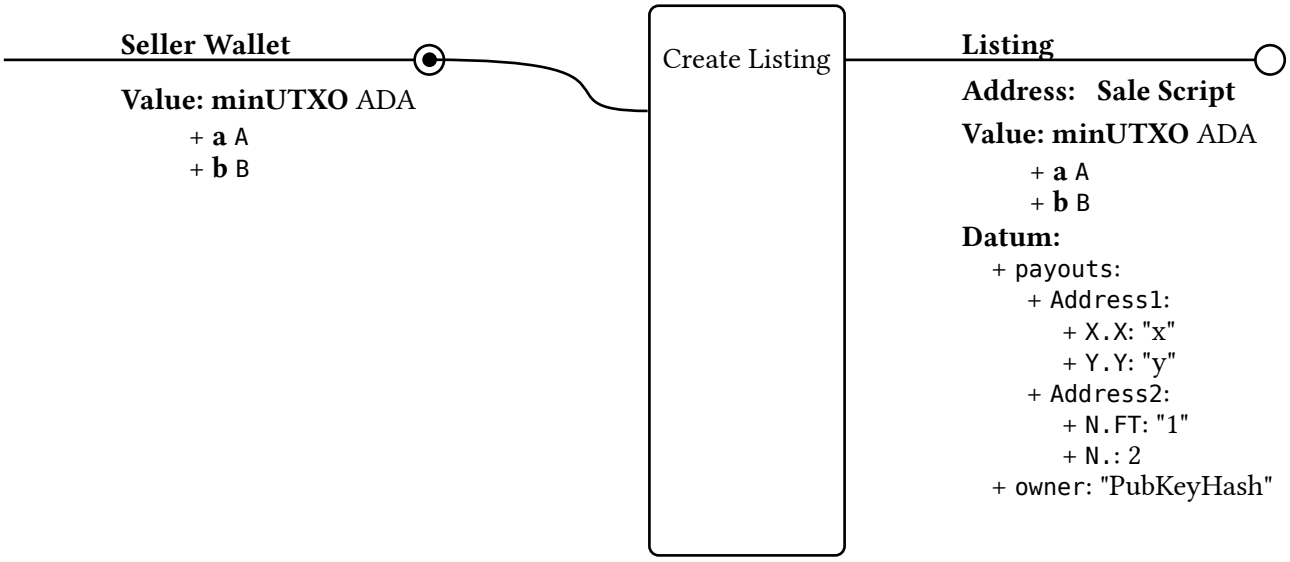


2.d.i - Transactions

There are 3 relevant transaction archetypes. Here is a brief overview of each.

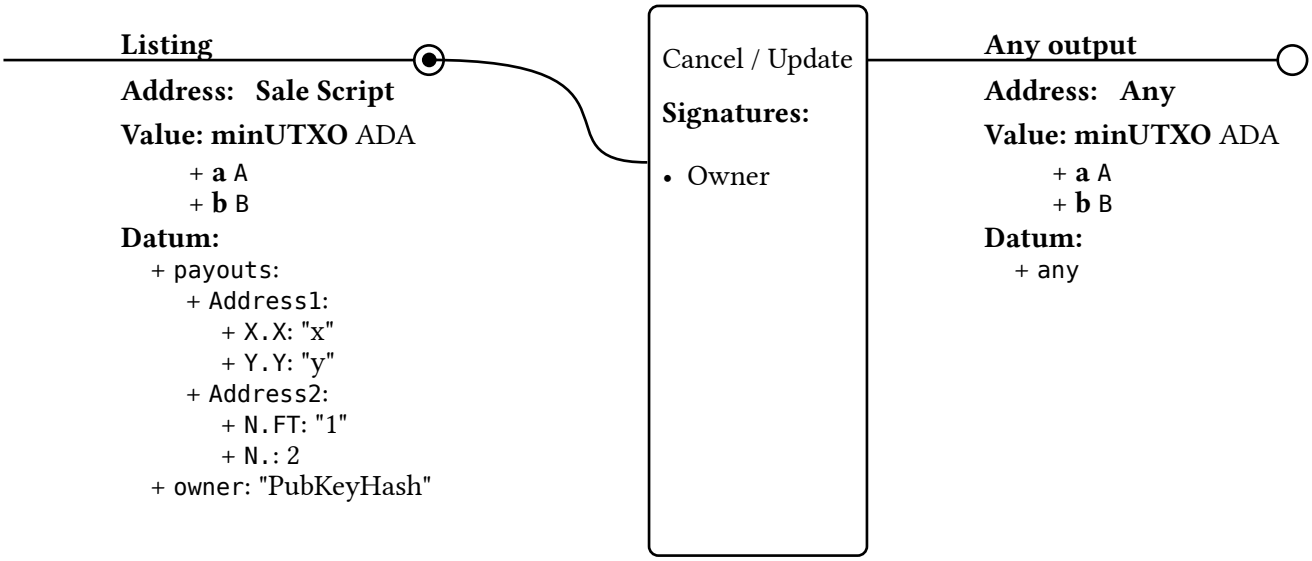
List

1. A **Seller** lists a bundle of tokens for sale with a set of payment criteria



Cancel / Update

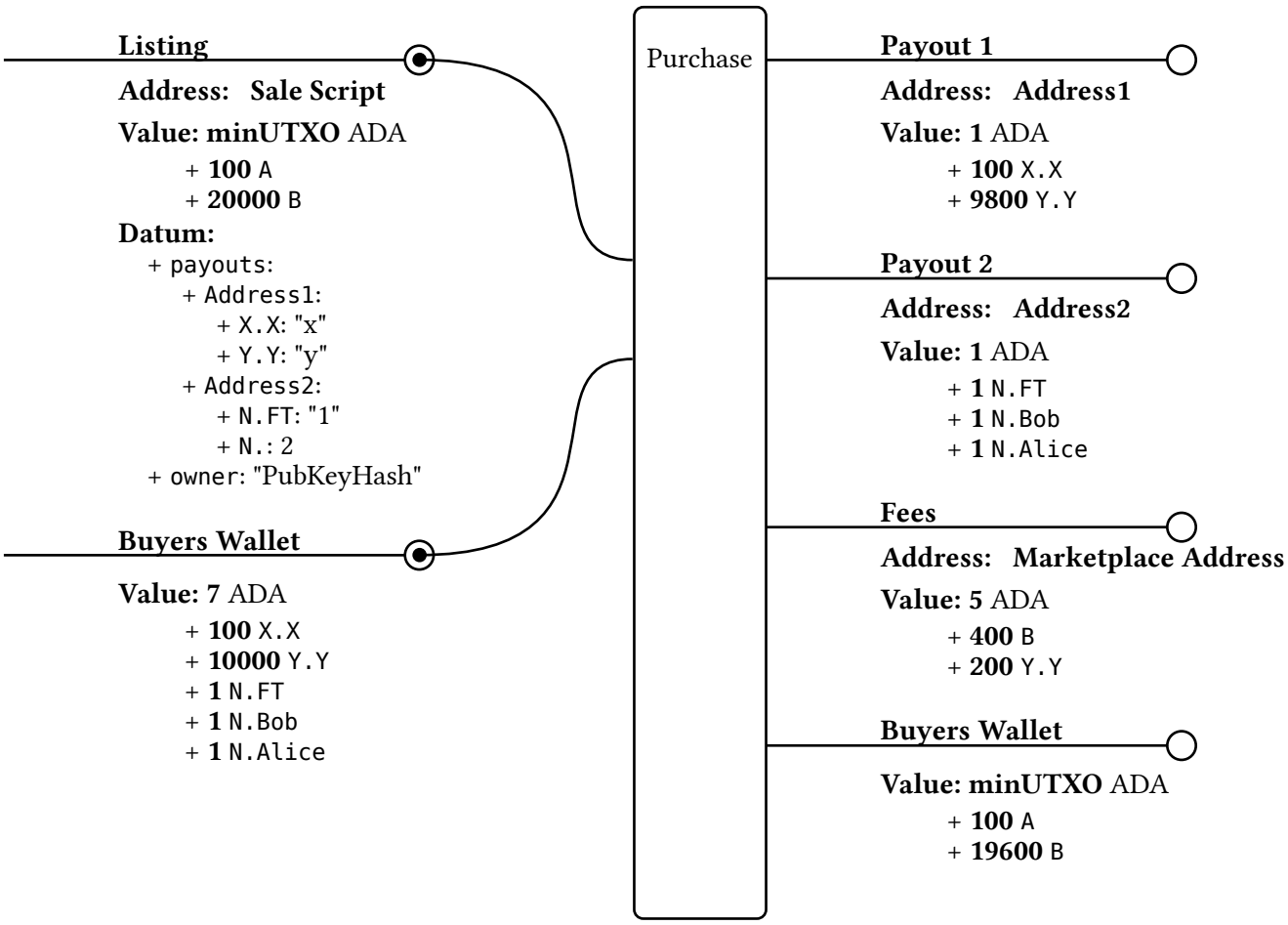
1. A **Seller** cancels an outstanding **Listing** , reclaiming the tokens back to their wallet
2. Alternatively, the **Seller** updates the transaction, paying the funds back into the contract with new parameters



Note: Any output can be the same listing contract, with extra / fewer assets, and a different datum, to update the listing.

Purchase

1. A **Buyer** satisfies each of the **Payout Criteria** in order to claim the bundle of assets.



2.e - Core Invariants

We identify the following core invariants that must hold in order for the protocol to achieve it's goals.

1. The **Seller** , and only the **Seller** , for a well-formed and un-purchased **Listing** must be able to cancel or update their listing at any time.
    1. That is to say, the **Seller** should be able to spend the UTXO **Listing** by attaching a signature from the appropriate key pair.
    2. Someone will be considered "the **Seller** " if they can produce a valid signature for a public key which hashes to the `owner` field of the listing datum.
    3. They should be able to pay those assets back into their wallet, another wallet, or directly back into the listing script with new parameters.
    4. A "well-formed" listing will be a UTXO locked at the `ask.ak` validator script with a datum that parses correctly as the `Datum` type from `ask.ak:22`, and has the **Seller** s verification key hash in the `owner` field.
2. If a **Listing** is purchased, the **Seller** must receive assets that collectively satisfy the obligations of the **Payout Criteria** .
    1. That is to say, for each payout, there must be a corresponding output, paid to the correct payout address, with greater than or equal to the specified amount of each asset.
    2. If a payout has a policy ID, but the asset name is empty string, it is treated as a wildcard; each asset with that policy ID will contribute towards satisfaction of the wildcard `amount`.
    3. Each unit of assets on the output must be credited towards satisfying only one unit of one payout; that is, you must not be able to satisfy two different payouts, or both a wildcard and specific required asset with the same unit of token.
3. When a **Listing** is purchased, the marketplace fee must be paid.
    1. That is, in any Purchase transaction, there must be a payout to the marketplace address with at least:
        1. 2% of the total ADA, plus 1 ADA per unique token with a quantity less than 100, plus 2% of each token with a quantity greater than or equal to 100
        2. If multiple **Listing** s are purchased in the same transaction, the fee must be paid for each **Listing** separately.

3 - Findings Summary

ID	Title	Severity	Status
CRASHR-000	Assets can be souble satisfied if using wild-cards	Critical	Resolved
CRASHR-001	Critical bug in check_output_cover_pay-outs	Critical	Resolved
CRASHR-100	Assets with empty asset names would be vulnerable to substitution	Major	Acknowledged
CRASHR-200	Protocol susceptible to fee-dodging	Minor	Acknowledged
CRASHR-201	Incorrect rounding in marketplace fees	Minor	Resolved
CRASHR-202	Risk in hard-coding the marketplace ad-dress	Minor	Acknowledged
CRASHR-300	minUTXO not accounted for	Info	Acknowledged
CRASHR-301	Low composibility with other protocols	Info	Acknowledged
CRASHR-400	Argument for Core Invariant 1	Witness	Resolved
CRASHR-401	Argument for Core Invariant 2	Witness	Resolved
CRASHR-402	Argument for Core Invariant 3	Witness	Resolved

CRASHR-000 - Assets can be souble satisfied if using wildcards

Severity	Status	Commit
Critical	Resolved	fc0a0c95a2b02c85456ff27ecd009e920a572355

Description

Consider a payout that requires “One X.Y, and three other X’s of any asset name”; A buyer can fulfil this order with just 2 other X’s, because the X.Y would count for one of the wildcards.
The root of this probelm is in the use of `is_additional_assets_covered`, which checks for other assets, but doesn’t discount those already covered by another payout.

Recommendation

As you evaluate each asset in a payout, subtract it from the output `output_assets` value / dict. This will ensure that no one asset can satisfy multiple payout conditions in the same payout.
Note that different payouts aren’t vulnerable to double satisfaction because each payout corresponds to exactly one output.

Resolution

This issue was resolved as of commit `fc0a0c95a2b02c85456ff27ecd009e920a572355`.

CRASHR-001 - Critical bug in check_output_cover_payouts

Severity	Status	Commit
Critical	Resolved	fc0a0c95a2b02c85456ff27ecd009e920a572355

Description

On line 38, 43, and 46 of `utils.ak`, the inner `dict.fold` references `res` from the **outer** fold. This could cause the function to incorrectly report that the payout is satisfied, when it is not.

Consider, for example, the following scenario:

`payout_dict`:

- `A`:
  - `a`: 123
  - `b`: 456

`output_dict`:

- `A`:
  - a: 1
  - b: 666

The code would fold over the elements of `payout_dict`, starting with an accumulator value of `True`.

The first (and only) iteration for the outer fold would be for the key `A`, and the `payout_assets` of `{ a: 123, b: 456 }`, with an accumulator value of `True`.

The code would extract the `output_assets` with policy ID `A` from `output_dict`, which would yield `{ a: 1, b: 666 }`. The code would then fold over `payout_assets`, starting with an accumulator value of `True`.

The first iteration of the inner fold would be for the key `a`, the value `123`, and an accumulator of `True`.

The code would check if `output_assets` contained `a`, which it does with as quantity 1, and return `True && 1 >= 123`, which evaluates to `False`.

The second iteration of the inner fold would be for the key `b`, the value `456`, and an accumulator of `False`. However, this inner fold uses `_` for this accumulator, so it is discarded. The outer accumulator is still `True`.

The code would check if `output_assets` contained `b`, which it does with a quantity of 666, and return `True && 666 >= 456`, which evaluates to `True`.

This `True` would then propagate out of the inner and then outer fold, incorrectly reporting that the payout is satisfied.

Recommendation

Ensure that the inner fold accumulates on its own value, rather than the outer fold.
We provide a patch and a test that solves this issue here: https://gist.github.com/Quantumplation/64556d040a6de33cb5c904968a2df98d

Resolution

This issue was resolved as of commit `fc0a0c95a2b02c85456ff27ecd009e920a572355`.

CRASHR-100 - Assets with empty asset names would be vulnerable to substitution

Severity	Status	Commit
Major	Acknowledged	

Description

If the listing contract is used with an asset that has no asset name, it will be treated as a wildcard. For example, if the payout tries to accept OADA, or qADA, then any other asset under that policy ID could be used, such as qDJED, etc.

Recommendation

Use an explicit representation of the payout condition, rather than trying to piggyback on another type definition.
Something like:

```
pub type Criteria {
  Exact(policyId, assetName, amount),
  Policy(policyId, amount),
}

pub type Payout {
  address: Address,
  amount: List<Criteria>,
}
```

would allow you to avoid ambiguity and exactly capture the semantics of what you are trying to express.

Resolution

This issue was acknowledged by the project team with the comment: We have addressed this issue on the UI/frontend by limiting the assets that can be selected for 'wildcard' to verified policies (NFT Collections) only.

CRASHR-200 - Protocol susceptible to fee-dodging

Severity	Status	Commit
Minor	Acknowledged	

Description

The protocol fee is only charged on the **payouts**, and not on the assets being listed. So an NFT sold for 1000 ADA might generate a 20 ADA fee, but users can just list 100 ADA in return for the NFT instead, and pay only 1 ada instead.

Recommendation

Apply the same fee logic on the assets in the listing, as those on the payouts, splitting the fee half to the buyer and half to the seller.
This ensures that both parties are charged, and that the correct value of the trade is always accounted for.

Resolution

This issue was acknowledged by the project team with the comment: We have already built the offchain component of the protocol and this change might include a lot of changes to the offchain component as well, so we are gunna stick with the current implementation. But we will address this in the next version.

CRASHR-201 - Incorrect rounding in marketplace fees

Severity	Status	Commit
Minor	Resolved	b2367e4acd87c6fbccff3b5fa5942a14626d28fe

Description

The marketplace fees should charge a 2% fee on any ADA paid in the payouts.

However, the code at `lib/crashr/utils.ak:260` calculates this as:

```
let ada_fee = (value.lovelace_of(total_payout) * 20 + 50) / 1000;
```

If this is an attempt to round to the nearest lovelace, the correct formula would be:

```
let ada_fee = (value.lovelace_of(total_payout) * 20 + 500) / 1000;
```

If the intention is to round up to the next highest lovelace, the correct formula would be:

```
let ada_fee = (value.lovelace_of(total_payout) * 20 + 999) / 1000;
```

A similar problem impacts the token fee at `lib/crashr/utils.ak:295`.

Recommendation

Decide on a rounding policy and correct as appropriate.

Resolution

This issue was resolved as of commit `b2367e4acd87c6fbccff3b5fa5942a14626d28fe`.

CRASHR-202 - Risk in hard-coding the marketplace address

Severity	Status	Commit
Minor	Acknowledged	

Description

The marketplace address is hard coded into the contract. This means that if these keys are compromised, the earnings from all outstanding orders will be at risk. Typically contracts will have the ability to rotate the relevant keys to new ones in the case of a compromise, or have multisig capabilities.

Recommendation

We don't feel this risk is particularly high; If you do want to address it, we suggest putting the marketplace fee in the datum of a reference input.

Resolution

This issue was acknowledged by the project team with the comment: We fully understand the risk of hardcoding the address and we plan to fix this on the next version.

CRASHR-300 - minUTXO not accounted for

Severity	Status	Commit
Info	Acknowledged	

Description

The contracts don't ensure that the various `minUTXO` requirements are met by the seller. This means it becomes the responsibility of the buyer to satisfy the minUTXO of the various outputs (of which there could be many), and becomes a hidden fee to the buyer.

Recommendation

Ensure that the Listing UTxO has enough ADA to cover each of the payouts, and have the contract enforce that those are paid from the Listing UTxO.

Resolution

This issue was acknowledged by the project team with the comment: The team is aware of this and decided to stick with the current implementation due to tight schedule. But we plan to address this in the next version.

CRASHR-301 - Low composibility with other protocols

Severity	Status	Commit
Info	Acknowledged	

Description

The contracts are missing several features that will make it very difficult to integrate and compose with other protocols. For example, the destination of a payout is just an address, not an address and datum, meaning you can't send these payouts to a smart contract treasury, etc.

Recommendation

We recommend:

- Make the payout `Destination` an address plus a datum, so you can specify paying to a smart contract
- Use something like `sundae/multisig` (https://github.com/SundaeSwap-finance/aicone) for the order owner, so that listings can be owned by other smart contracts

Resolution

This issue was acknowledged by the project team with the comment: This is not yet a priority for the current version, but we'll revisit this in the future.

CRASHR-400 - Argument for Core Invariant 1

Severity	Status	Commit
Witness	Resolved	

Description

We make an argument that the audited code satisfies core invariant 1.

Assume that there exists a well formed and unspent listing UTxO.

- The datum is well formed, and of the type described in `validators/ask.ak:22-29`
- The UTxO is locked with a script address equal to the audited ask validator.
- The `owner` is set to the blake2b-224 hash of a public key for which the seller can produce a valid ed25519 signature.

In order to spend the UTxO, the ledger will invoke the validator function at `validators/ask.ak:44` with the current (well formed) datum, the user provided redeemer, and a relevant script context.

- The aiken compiler will inject a check for the validity of the datum, which it will pass via our assumption above.
- The aiken compiler will inject a check for the validity of the redeemer, which is user provided, and must be set to the `WithdrawOrUpdate` variant.

The validator will destructure the script context at `validators/ask.ak:45`, extracting the transaction and purpose.

- The ledger constructs the script context for the current plutus version, so we assume this is correct and cannot fail.

The validator then destructures the transaction at `validators/ask.ak:47`, extracting the outputs and extra signatories.

- Again, the ledger constructed this type, so we assume this cannot fail.

The validator then matches on the redeemer at `validators/ask.ak:50`.

In this case, the redeemer must be `WithdrawOrUpdate`, and so the code at `validators/ask.ak:82` will run.

This uses the aiken `list.has` function to check that `extra_signatories` contains `datum.owner`. This checks that at least one element of the list is equal to the owner on the datum, and is returned as the result of the function. A True will allow the UTxO to be spent, and False will reject the transaction.

We then have two cases to check:

- The **Seller can** spend this UTxO
  - `extra_signatories` is a field on the transaction
  - For each value in this array, there must be a matching signature in the witness set from a public key which has a blake2b-224 hash equal to the value in the array
  - Therefore, the owner can sign the transaction, place their public key hash in the extra_signatories field, set the redeemer to `WithdrawOrUpdate`, and spend the UTxO.
  - There are no other restrictions we encountered, such as valid range restrictions, and so if the transaction is otherwise valid, it will be accepted by the ledger.
- **Only** the **Seller** can spend this UTxO
  - If someone other than the seller adds the sellers public key hash to the extra signatories, the ledger will validate that the transaction witness set contains a signature from a corresponding public key.
  - We assume that only the seller can produce such a signature.
  - Thus, nobody else can spend the UTxO with the `WithdrawOrUpdate` redeemer.
  - Other invariants are enforced on the `Buy` redeemer, which can be someone else, but so long as those conditions are met, we don't consider this condition violated.

Finally, as an addendum, we note that there are no restrictions placed on the outputs of the transaction; Therefore, the **Seller** can choose to pay the assets back into their wallet, another wallet, a smart contract, or even back into the same order with an updated datum.

The **Seller** can also spend multiple UTxOs in the same transaction, and all of them must be satisfied in this way. Nothing in the above argument is invalidated by multiple inputs on the same transactions.

Thus, we consider the first invariant to hold.

CRASHR-401 - Argument for Core Invariant 2

Severity	Status	Commit
Witness	Resolved	

Description

We make an argument that the audited code satisfies core invariant 2.

We assume that:

- A listing UTxO has been created, with a valid datum according to the type defined at `validators/ask.ak:22-29`
- The UTxO is being spent, not neccesarily by the **Seller** , and the redeemer is set to `Buy` with some integer offset.

Since the UTxO is being spent, the ledger will invoke the validator at `validators/ask.ak:44` and apply 3 parameters: the current datum, the user provided redeemer, and the script context.

Since we assume that the datum on the UTxO is well formed, and its provided by the ledger, the aiken injected type checks will pass.

The validator destructures the script context and the transaction at `validators/ask.ak:45+47`.

- Since this is provided by the ledger, we assume these are well formed

The validator then matches on the redeemer at `validators/ask.ak:50`, and in all cases relevant to this invariant, the code at `validators/ask.ak:52-76` will be run.

First, the validator expects that the `purpose` is `Spend`; Since we are spending this UTxO, this will always succeed.

The validator then computes a `datum_tag` at `validators/ask.ak:55`, which is the out_ref being spent, serialized to CBOR, and passed to the blake2b_256 hash function.

- This produces a unique value for this particular UTxO being spent

The validator then destructures the datum at `validators/ask.ak:61`, extracting just the payouts field.

- Since we assume that the datum is well formed according to this type, this will always succeed.

The validator then uses a utility function to find the payout at `redeemer.payout_outputs_index`

- If this index is negative, it will continuously recurse at `lib/crashr/utils.ak:221` until the builtin `tail_list` fails, or the transaction exceeds its budget; in either case, the transaction is invalid.

- If the index is between 0 and the number of outputs on the transaction, this utility will return a list of the outputs starting at that index (0 invokes tail_list 0 times, 1 invokes it 1 time, etc.)
- If the index is equal to the number of outputs, it will return an empty list, as it will call tail_list until there is an empty list, and current_index will be 0, so it will return that empty list
- If the index is greater than the number of outputs, it will eventually call tail_list on an empty list and error, or exceed the transaction budget; Given the cost of this function, and the practical bounds on number of transaction outputs in a transaction, this is unlikely to prevent a listing from being bought. In the worst case, the transaction builder can ensure that the outputs are ordered so that the payouts appear early in the list of outputs. We do not judge this a major risk.
- This utility function performs an optimization to avoid extra bounds checking when the index is very large, but none of this invalidates the above description.

So, if the transaction succeeds, this utility will return a valid (though potentially empty) list of outputs from the transaction.

The validator then destructures the output into a marketplace output, and the remaining outputs, at `validators/ask.ak:66`

- If the list returned from `find_payout_outputs` is empty, this will fail; this means there must always at least be an output dedicated to marketplace fees.

`rest_outputs` is now either an empty list, or suffix list of outputs from the transaction.

The validator uses a utility function called `check_outputs` at `validators/ask.ak:69` to check that each payout is satisfied, passing it this list of outputs, the list of payouts from the datum, and NoDatum.

This function is defined at `lib/crashr/utils.ak:421`.

First, it destructures the outputs to a first output, and the remaining outputs. This means there must always be at least one payout and one output corresponding to that payout.

Second, this first output is destructured into an address, a value, and a datum.

The datum must equal the datum tag passed in to the function, which in this case is always NoDatum.

Note that this prevents payouts to smart contracts.

Then, the validator destructures the payouts to a first payout and the remaining payouts.

Thus, if this function is invoked, there must have both an output and a payout to compare.

The validator then checks, at `lib/crashr/utils.ak:439`, that the output address is equal to the payout address.

It checks that the output covers the payout, that the lovelace on the output is greater than 0, and that the payout lovelace amount is greater than or equal to 0. We will explore the "output covers the payout" check in a moment.

The validator then recurses:

- If `rest_payouts` is empty, then every payout has been satisfied, and we return the payout amount
- If the `rest_payouts` is not empty, there are still payouts to validate, so we recurse with `rest_outputs` and `rest_payouts`, and `NoDatum`, and return the payout amount merged with the result of the recursive call (effectively summing all the payouts)

Thus:

- Every payout is checked, or we exceed our execution units. The team should check for the maximum number of payouts that can be processed before the execution units are exceeded, but we expect it to be practically bounded by the size of transactions, rather than the execution units.
- Every payout had a corresponding output (otherwise the destructure would fail)
- we return the sum of all payouts

So, what does it mean for the `output to cover the payout`? This is handled by a utility function, invoked with the `payout.amount` and the `output.value` at `lib/crashr/utils.ak:443`.

This utility is defined at `lib/crashr/utils.ak:19`, and expects a payout amount as the first argument, and an output value as the second argument, which matches how it is being invoked.

The validator flattens each of these values to dictionaries; this converts the aiken Value type to a Dictionary from Policy ID to dictionaries from Asset Name to quantity.

The validator then folds over the payout policy IDs, starting from an accumulator of True; each policy ID will need to be satisfied, so the accumulator gets &&'d in each iteration of the fold at `lib/crashr/utils.ak:31`.

For each policy in the payout value, the same policy must be present in the output value. `lib/crashr/utils.ak:29` checks this using `dict.get` and expecting to get a `Some(..)`.

The validator folds over each asset in the **payout** with that policy ID, starting with separate accumulator value of True; each asset name must be satisfied, so the accumulator gets &&'d in each iteration of the fold at `lib/crashr/utils.ak:43+46`. Collectively, this means that each asset in the payout gets checked for satisfaction.

The validator checks whether the asset name is in the output assets for that policy ID at `lib/crashr/utils.ak:35`.

If the asset is present, then we check that the `output_amount` is greater than or equal to the required `payment_amount`; If so, the payout is satisfied for that output and we move on. Because the policy IDs and asset names in a ledger `Value` cannot have duplicates, therefore we don't have to worry about double counting at this level.

However, if the payout asset is not present in the output, we check two cases.

- If the asset name is not empty, or this is ADA, then this payout is just plain not satisfied, so we return false, meaning the whole assets fold will return false, meaning the whole policy fold will return false, meaning does_output_cover_payout will return false, and the transaction will fail.
- If the asset name **is** empty, but the policy ID **is not**, we treat this special, as a wildcard. We have raised concerns about this approach elsewhere in the audit, but ignoring those concerns, we then check if the wildcards are satisfied at `lib/crashr/utils.ak:46`.

This utility is passed the payout assets (for this specific policy), the output assets (for this specific policy), and the payout amount. It filters out any assets with an empty asset name (i.e. the wildcard); This will never be invoked on the ADA policy ID because of a previous check.

It then counts the number of asset names in the output that **aren't** in the list of filtered requested assets. That is, if an asset is explicitly requested by the payout, it should not be able to satisfy a wildcard requirement.

This count of asset names (irrespective of the quantities) must be greater than or equal to the `amount_to_cover`, which was the quantity specified in the payout wildcard case. Thus, the payout must include the correct number of NFTs specified by the wildcard. We have highlighted to the team that this wildcard behavior breaks down for fungible tokens, or assets with an empty asset name like qADA or oADA, and the team has acknowledged the risk.

Thus, we believe we can conlude that the `does_output_cover_payout` utility method correctly checks whether the `value` of an output satisfies a particular payout, and the `check_payouts` will correctly check whether a range of outputs satisfy a range of payouts.

Finally, the validator at `validators/ask.ak:76`, by way of a utility function defined at `lib/crashr/utils.ak:314`. We will ignore most of this method, as it's not relevant for this invariant, but at `lib/crashr/utils.ak:320`, it checks that the datum on the marketplace payout is equal to the datum tag that was computed earlier.

Because each UTxO validator expects the payout pointed at by the redeemer to be a marketplace payout with a unique datum tag, it is impossible to double-satisfy payouts:

- No two UTxOs can be pointed to the same marketplace payout because of the datum tag
- Each output used to satisfy an order payout must have NoDatum, so we can't overlap into another orders marketplace payouts without the transaction failing.

Thus, we have argued:

- If a listing UTxO is spent, and not as part of a cancellation or update transaction, every `Payout` specified in the datum must be satisfied:
  - The address in each payout must receive, at a minimum, the assets specified in the payout
  - Because we use >=, more of an asset can be provided, meaning the minUTXO requirement can be met
  - If multiple listing UTxOs are filled, they must be satisfied by distinct outputs

Thus, the second core invariant holds.

CRASHR-402 - Argument for Core Invariant 3

Severity	Status	Commit
Witness	Resolved	

Description

We make an argument that the audited code satisfies core invariant 3.

We assume that:

- A listing UTxO has been created, with a valid datum according to the type defined at `validators/ask.ak:22-29`
- The UTxO is being spent, not neccesarily by the **Seller** , and the redeemer is set to `Buy` with some integer offset.
- All other payouts are satisfied as per core invariant 2.

We now must check that the marketplace fee must be paid.

At `validators/ask.ak:66`, the payouts are destructured into a marektplace payout and a list of other outputs.

This marketplace output must have a datum equal to the datum tag computed at `validators/ask.ak:55` (checked in `check_marketplace_payout` at `lib/crashr/utils.ak:320`), which is unique to this UTxO. Additionally, each payout output must have no datum, meaning:

- No marketplace UTxO can be used to satisfy multiple orders
- Each order must have a corresponding marketplace UTxO
- An output cannot be used to satisfy both a marektplace payout and one of the listing payouts.

Checking each payout computed the sum of the values on those payouts, stored in `payouts_sum` at `validators/ask.ak:69`. See the argument for core invariant 2 for the soundness of this sum.

The validator uses that to calculate the marketplace fee, using a utility function defined at `lib/crashr/utils.ak:257`.

- This first calculates a 2% fee at `lib/crashr/utils.ak:260`; see **CRASHR-201** for a small rounding bug.
- It then calculates the number of unique tokens, by flattening the value, excluding lovelace, and counting all tokens below a heuristic 100 tokens; Tokens with an asset name are counted as 1, while tokens with an empty asset name (treated like a wildcard) add their quantity to the sum

- The ADA fee is then the 2% fee plus 1 ADA (1 million lovelace) per unique token (defined at `lib/crashr/utils.ak:282`), or 1 ADA, whichever is greater.
- Additionally, the token fee is defined as 2% of any tokens which pay out more than 100 units. See **CRASHR-201** for a similar small rounding bug.

This utility then returns a Value that comes from merging the total ADA fee and the total token fee.

The above calculations satisfy the business requirements we were given, so now we just need to check that it is paid to the marketplace address.

Recall that the marketplace payout is checked via a utility defined at `lib/crashr/utils.ak:314`.

This first destructures the output into an address, a value, and a datum. The datum must have the datum tag, as described before. The address of the output must be the marketplace address, which is hard coded as a constant. This ensures that the marketplace fee is actually paid to the Crashr treasury, and not some other address.

Finally, we use the same `does_output_cover_payout`, with the marketplace fee as the payout value, and the marketplace output value as the value. The same argument from **CRASHR-401** for the correctness of this method applies here.

Thus, if the datum is the correct datum tag, the funds are paid to the marketplace address, and the output value has greater than or equal to the quantity of each asset defined by the marketplace fee computed above, the transaction will validate, and the core invariant holds.

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