ANASTASIA LABS

Security Audit Report

Date: May 1, 2024
Project: Bond Issue
Version 1.0

Contents

Disclosure

Disclaimer and Scope

A code review represents a snapshot in time, and the findings and recommendations presented in this report reflect the information gathered during the assessment period. It is important to note that any modifications made outside of this timeframe will not be captured in this report.

While diligent efforts have been made to uncover potential vulnerabilities, it is essential to recognize that this assessment may not uncover all potential security issues in the protocol.

It is imperative to understand that the findings and recommendations provided in this audit report should not be construed as investment advice.

Furthermore, it is strongly recommended that projects consider undergoing multiple independent audits and/or participating in bug bounty programs to increase their protocol security.

Please be aware that the scope of this security audit does not extend to the compiler layer, such as the UPLC code generated by the compiler or any areas beyond the audited code.

The scope of the audit did not include additional creation of unit testing or property-based testing of the contracts.

Assessment overview

From January 16th, 2024 to April 16th, 2024, Danogo engaged Anastasia Labs to evaluate and conduct a security assessment of its Bond Issue protocol. All code revision was performed following industry best practices.

Phases of code auditing activities include the following:

- Planning – Customer goals are gathered.
- Discovery – Perform code review to identify potential vulnerabilities, weak areas, and exploits.
- Attack – Confirm potential vulnerabilities through testing and perform additional discovery upon new access.
- Reporting – Document all found vulnerabilities.

The engineering team has also conducted a comprehensive review of protocol optimization strategies.

Each issue was logged and labeled with its corresponding severity level, making it easier for our audit team to manage and tackle each vulnerability.

Assessment components

Manual revision

Our manual code auditing is focused on a wide range of attack vectors, including but not limited to.

- UTXO Value Size Spam (Token Dust Attack)
- Large Datum or Unbounded Protocol Datum
- EUTXO Concurrency DoS
- Unauthorized Data modification
- Multisig PK Attack
- Infinite Mint
- Incorrect Parameterized Scripts
- Other Redeemer
- Other Token Name
- Arbitrary UTXO Datum
- Unbounded protocol value
- Foreign UTXO tokens
- Double or Multiple satisfaction
- Locked Ada
- Locked non Ada values
- Missing UTXO authentication
- UTXO contention

Executive summary

- Danogo Bonds allow Cardano users to enhance their yield by offering staking liquidity to borrowers participating in events like ISPOs, Catalyst Voting or strengthening their Staking Pools.
- A Danogo Staking Bond is a loan agreement for non-custodial lending of staking rights to Borrower. A smart contract locks lender's ADA and allows a borrower to attach their stake key for a specified duration as long as conditions are met (interest paid)
- Danogo provides an intuitive interface that connects lenders and borrowers through various Danogo Bonds issued. Borrowers quickly set up their loans, with flexible initial conditions : ranging from loan terms, loan volumes to loan interest rates. Lenders can also quickly provide Liquidity by specifying requested yield and maturity date or loan duration. .
- Danogo also provides a method to help borrowers or lenders optimize their benefits by finding the most suitable liquidity offers /loans that match their needs. Danogo Bond holders can wait till maturity to redeem their ADA or sell their Bond on the Bond Exchange to receive instant liquidity.

Code base

Repository

https://github.com/Danogo2023/bond-issue

Commit

f49864c750cfb0a4d7f5ed98f92c58bf1b6cd791

Files audited

SHA256 Checksum	Files
0c2d837893df453e0949b72451757de5 a1f479bd38b3dfcada75cf7015e0c3da	lib/borrow_position/change_stake_key.ak
7471a981841e469c69c7b13965d49832 5e951292084a8edfd0d7cb2996f138d3	lib/borrow_position/pay_interest.ak
99ce48179f012f5e53538c99fd9ce515f0 6eb175da58710886e76c6f48bce1f2	lib/borrow_position/redeem_bond.ak
7ced0e0c79e408e296bbb70014b16cfa1 78eb81c9488625939eb17d5d0df699f	lib/borrow_position/redeem_fee.ak
02ca91eec90d6888d7e939707d918530 c865475dfbb7aa36b011f925fb6dd38c	lib/borrow_request/bond_create.ak
bf06b996d58c3a95d80b154c3fca5bad6 e1dccff17ea7a85f711aef2ad3f6d27	lib/borrow_request/request_nft.ak
99943fb16de234d83bb523e12ba19d84 ad1123f38313739128c4bd719b77efbe	lib/borrow_request/request_update.ak
0b52fe0c54e17a2b63a43209107e4dc35 9b0c6b0270aaed91470ac70f0c3e79e	lib/issue_bond/utils.ak
37da87088e439e6fd5c49bff7e281c2333 b2e37674a4f9c66f447b6c8ca7ae8f	validators/bond_issue.ak

Severity Classification

- **Critical**: This vulnerability has the potential to result in significant financial losses to the protocol. They often enable attackers to directly steal assets from contracts or users, or permanently lock funds within the contract.
- **Major**: Can lead to damage to the user or protocol, although the impact may be restricted to specific functionalities or temporal control. Attackers exploiting major vulnerabilities may cause harm or disrupt certain aspects of the protocol.
- **Medium**: May not directly result in financial losses, but they can temporarily impair the protocol's functionality. Examples include susceptibility to front-running attacks, which can undermine the integrity of transactions.
- **Minor**: Minor vulnerabilities do not typically result in financial losses or significant harm to users or the protocol. The attack vector may be inconsequential or the attacker's incentive to exploit it may be minimal.
- **Informational**: These findings do not pose immediate financial risks. These may include protocol optimizations, code style recommendations, alignment with naming conventions, overall contract design suggestions, and documentation discrepancies between the code and protocol specifications.

Finding severity ratings

The following table defines levels of severity and score range that are used throughout the document to assess vulnerability and risk impact.

	Level	Severity	Findings
🟥	5	Critical	0
🟥	4	Major	2
🟧	3	Medium	1
🟨	2	Minor	1
🟦	1	Informational	0

Findings

ID-401 Bond nfts can be minted with any name

	Level	Severity	Status
🟥	4	Major	Resolved

Attack vector

1. The attacker prepares a fake request nft policy (for example a function that always returns true): `req_nft_fake`. He also decides on which (possibly previously minted) token name to use: `target_name`.
2. The attacker preparea a fake request validator script `req_nft_fake` from which he can spend UTxOs (for example the "always true" function), he also creates a UTxO `fake_req_utxo` on this address that has a valid request datum and an nft token minted via the `req_nft_fake` minting policy.
3. The attacking transaction is constructed as:
   - **inputs:**
     (a) The `fake_req_utxo`.
   - **minting policies:**
     (a) 'Bond-nft' with
       - redeemer:

```
BondCreate(
  out_ref:   <any value>
  req_skh:   req_fake
  bond_skh:  <real bond skh>
  pid:       req_nft_fake
)
```

       - minted values:

```
(Bond_pid, <target_name>, 1)
```

   - **outputs:**
     (a) Bond: a valid bond at the real bond script address with `token_name` set to `target_name`.

Root cause

The `bond_nft` minting policy accepts any script hash and any policy id that is communicated to it through the `BondCreate` redeemer. Malicious users can exploit this and create duplicate bonds with the same token name that has the authentication token (bond nft) that the real one has, but with any bond details that adheres to the minimum set of restrictions that are enforced by the platform:

```
and {
  (value.quantity_of(o.value, nft_pid, obond_dt.token_name) == 1)?,
  //
  dict.has_key(mint_bond_nfts_valid, obond_dt.token_name)?,
  //
  (obond_dt.duration >= cfg.platform.min_duration)?,
  (obond_dt.buffer >= cfg.platform.min_buffer)?,
  (obond_dt.fee >= cfg.platform.fee)?,
  (obond_dt.bond_amount > 0)?,
  (obond_dt.start == epoch_curr - cfg.epoch.epoch_boundary_as_epoch)?,
}
```

This exploit can be used to manipulate the prices of legitimate bonds, confuse users, frontend and/or backend systems to behave unexpectedly, since it explicitly breaks the documented assumption that "Bond NFTs are unique on the protocol."

Recommendation

The bond token name should be solely derived from an output reference that is spent in the minting transaction. In the bond nft policy the token name should be validated as:

1. The output with the reference that is used to generate the token name is spent.
2. The datum of the output bond UTxO has the correct token name set.
3. Each output reference is only used once in bond generation (even if multiple are minted in the same transaction).

Resolution

Resolved in commit 44163a6c03ae4c8d4686060d411c15e8704e9616.

ID-402 Real looking fake bond can be created with arbitrary borrower

	Level	Severity	Status
🟥	4	Major	Resolved

Description

Similar to vulnerability ID-401, the `req_pid` and `req_skh` values are read from the untrusted redeemer, which allows a malicious user of the protocol to inject any request data that passes the platform validation while carrying fake information. Also, this approach allows bypassing validity checks that are enforced by the request script. This allows creating bonds with arbitrary values in the `borrower` field and arbitrary number of bond tokens minted, the only logic that is enforced that the amount of bond tokens minted must be equal to the `bond_amount` field of the generated bond. These "fake" bonds can carry almost any Ada value, hence there is no guarantee that they can be redeeemed for the expected amount of Ada, hence allowing confusing/scamming traders of the bond, while creating the illusion of that the protocol and specific other users of the protocol are not behaving honestly, by not maintaining enough coverage of their issued bonds.

Attack vector

1. The attacker prepares the following:
   - `fake_req_validator`: a fake request validator scripts, which always allows spending for the attacker
   - `fake_req_policy`: a fake minting policy script, which always allows minting for the attacker
   - `fake_borrower`: the attacker decides what borrower value to use
   - `mint_amount`: the attacker decides how many bond tokens to mint
2. The attacker creates a `fake_request_utxo`, such that:
   - the address corresponds to the `fake_req_validator` script hash
   - the value contains a token (`fake_req_policy`, `fake_borrower`, 1)
   - the datum is a request datum that passes the platform validation
   - the value contains enough Ada to cover the interest to be paid
3. The attacker calculates the `bond_name` the same way as the code `hash_out_ref_salt(fake_request_utxo.output_reference, fake_borrower)` would do.
4. The attacking transaction is constructed as:

- **inputs:**
  (a) The constructed `fake_request_utxo` with any redeemer that the fake script accepts.
- **minting policies:**
  (a) `Bond-nft` with
    - redeemer:
```
BondCreate(
  out_ref:  reference to fake_request_utxo
  req_skh:  script hash of fake_request_validator
  bond_skh: real bond_skh
  pid:      script hash of fake_request_policy
)
```
    - minted values:
```
(Bond_nft, bond_name, 1)
```
  (b) `Bond-token` with
    - redeemer:
```
BondCreate(
  out_ref:  reference to fake_request_utxo,
  req_skh:  script hash of fake_request_validator,
  bond_skh: any script hash
  pid:      any policy id
)
```
    - minted values:
```
(Bond_token, bond_name, mint_amount)
```
- **outputs:**
  (a) Bond UTxO with:
    - the real bond script address
    - the value containing the minted `bond_nft`
    - the value containing any Ada value that satisfies the minimum Ada requirements for the chain
    - a bond datum that satisfies these criteria:

```
(obond_dt.duration    >= cfg.platform.min_duration),
(obond_dt.buffer      >= cfg.platform.min_buffer),
(obond_dt.fee         >= cfg.platform.fee),
(obond_dt.bond_amount >  0),
(obond_dt.start       == epoch_curr - cfg.epoch.
   epoch_boundary_as_epoch),
(obond_dt.bond_symbol == bond_pid),
(obond_dt.token_name  == bond_name),
(obond_dt.borrower    == fake_borrower),
(obond_dt.bond_amount == mint_amount)
```

(b) A UTxO at the address of the attacker's wallet containing the minted bond tokens, while also retrieving the Ada that was used to create the fake request.

Result

The attacker minted real bond tokens with a constructed a bond UTxO, which look real, but has no coverage. Without analyzing the on chain creation transaction it is hard to discover that the bond tokens cannot be redeemed for the expected amounts of Ada.

Recommendation

It is not recommended to accept values read from untrusted sources (a redemmer in this case) as a guarantee that some validity checks (that are implemented in another script) are performed.

The bond coverage amount (principal + interest) should be checked in the bond token minting policy. Furthermore, it is recommended to change the deployment order of the protocol and use multi-validators for minting policies and validators that are associated. This way it is easier to avoid circular dependencies between the validator functions and to avoid having to rely on untrusted values communicated through redeemers.

Resolution

Resolved in commit becc92bca8bb22f04b343abfcede404fb2307f8b.

ID-301 Borrower nfts can be minted with any names

	Level	Severity	Status
🟨	3	Medium	Resolved

Attack vector

Assumptions

The attacker has the amount of Ada required to generate two requests.

Attacking transaction

1. The attacker prepares a fake request nft policy (for example a function that always returns true): `Policy_fake`. He also decides on which (possibly previously minted) token name to use: `target_name`. He can optionally also prepare a fake request validator from which he can collect the Ada spent on the requests easily: `V_fake`.

2. The attacking transaction is constructed as:

   - **inputs:**

     (a) Any UTxO `U` from the wallet of the attacker that has the required amount of Ada.

   - **minting policies:**

     (a) `Borrower-nft` with

       – redeemer:

```
RequestCreate(
  out_ref: <reference of U>,
  req_skh: V_fake,
  pid:     Policy_fake
)
```

       – minted values:

```
(Borrower_pid, <borrower_valid>, 1)
(Borrower_pid, target_name     , 1)
```

     (b) `Policy_fake` with

       – minted values:

```
(Policy_fake, <borrower_valid>, 1)
(Policy_fake, target_name     , 1)
```

- **outputs:**
  (a) Request 1: a valid request at address `V_fake` with borrower: `<borrower_valid>`.
  (b) Request 2: a valid request at address `V_fake` with borrower: `target_name`.
  (c) A UTxO at the address of the attacker's wallet containing both the minted borrower nfts.

Root cause

The names of the minted borrower nfts are not validated exactly, their validity is only checked by the amount minted:

```
let mint_borrower_nfts_valid = dict.filter(mint_borrower_nfts,
  fn(_n, q) { q == 1 }
)
```

Recommendation

It is recommended to only allow minting one borrower nft per transaction and the name to be also checked, for example:

```
let mint_borrower_nfts_valid = dict.filter(mint_borrower_nfts,
    fn(n, q) { and(n == borrower_valid, q == 1) }
  )
```

Resolution

Resolved in commit fd74db9b8b26f1ed651179ca2141f440ff69534d.

ID-201 Aiken compilation error: using opaque type in the datum

	Level	Severity	Status
🟨	2	Minor	Resolved

Description

The Aiken language has a feature called "opaque types", which allows library developers to mark types as opaque, which disallows the possibility for users of the type to construct or deconstruct values without using the functions provided by the library. This allows enforcing invariants that all instances of this value inherently should posess. One such type is the `Value` implemented in the Aiken standard library. The explicit invariant that is enforced is: "a Value will never contain a zero quantity of a particular token"

This property is specifically important, newer versions of the Aiken compiler enforce that these types should not be constructed from untrusted sources without actually checking that the invariant holds. Some datum types that are implemented in the separate Daken library, which is a dependency of the bond-issue library are using fields of type ‘Value‘, which is not recommended, since the invariant in this case can be easily broken. Newer Aiken compilers report this as an error.

Recommendation

It is recommended to use the type `List<(PolicyId, List<(AssetName, Int)>)>` instead of `Value` in datum and redeemer fields.

Resolution

Resolved in commit hash `1595c935dc8e0159b783e6a1ebeb583440ca3156`.