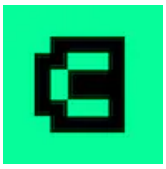

Canto contest
Findings & Analysis Report

2023-03-31

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 smart contract system written in Solidity. The audit contest took place between November 23—November 28 2022.

Wardens

37 Wardens contributed reports to the Canto contest:

1. 0xSmartContract
2. 0xhacksmithh
3. AkshaySrivastav
4. Awesome
5. Beepidibop
6. Deivitto
7. DijkstraDev
8. Dinesh11G
9. Englave
10. JC
11. Jeiwan
12. Josiah
13. Rahoz
14. RaymondFam
15. ReyAdmirado
16. Ruhum
17. SaeedAlipoor01988

18. Sathish9098
19. Tricko
20. abiih
21. aphak5010
22. cccz
23. chaduke
24. chrisdior4
25. cryptonue
26. exolorkistis
27. gzeon
28. hihen
29. joestakey
30. keccak123
31. martin
32. nicobevi
33. oyc_109
34. peritoflores
35. ronnyx2017
36. rotcivegaf
37. sorrynotsorry

This contest was judged by berndartmueller.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 5 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 3 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 13 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 26 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 contest repository, and is composed of 6 smart contracts written in the Solidity programming language and includes 264 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 (2)

[H-01] User can redirect fees by using a proxy contract

Submitted by Ruhum, *also found by* ronnyx2017 *and* hihen

https://github.com/code-423n4/2022-11-canto/blob/main/CIP-001/src/Turnstile.sol#L86-L101

https://github.com/code-423n4/2022-11-canto/blob/main/Canto/x/csr/keeper/evm_hooks.go#L51

Impact

For any given tx, the fees are sent to its recipient (`To`). Anybody can register an address using the Turnstile contract. Thus, a user is able to create a proxy contract with which they execute other smart contracts. That way, the fees are sent to their own contract instead of the actual application they are using. People who use smart contract wallets don't even have to bother with setting up a proxy structure. They just add their own wallet to the Turnstile contract.

Also, there might be a possibility of someone setting up a proxy for high-usage contracts where the fees are sent back to the caller. So for contract X, we create X' which calls X for the caller. Since X' is the recipient of the tx, it gets the gas refund. To incentivize the user to use X' instead of X, X' sends a percentage of the refund to the caller. The feasibility both technically and economically depends on the contract that is attacked. But, theoretically, it's possible.

The incentive to take it for yourself instead of giving it to the app is pretty high. Since this causes a loss of funds for the app I rate it as HIGH.

Proof of Concept

Registering an address is permissionless:

```
function register(address _recipient) public onlyUnregistere
    address smartContract = msg.sender;

    if (_recipient == address(0)) revert InvalidRecipient();

    tokenId = _tokenIdTracker.current();
    _mint(_recipient, tokenId);
    _tokenIdTracker.increment();

    emit Register(smartContract, _recipient, tokenId);

    feeRecipient[smartContract] = NftData({
        tokenId: tokenId,
        registered: true
    });
}
```

Fees are sent to the recipient of the tx:

```
func (h Hooks) PostTxProcessing(ctx sdk.Context, msg core.Messag
        // Check if the csr module has been enabled
        params := h.k.GetParams(ctx)
        if !params.EnableCsr {
                return nil
        }

        // Check and process turnstile events if applicable
        h.processEvents(ctx, receipt)

        contract := msg.To()
        if contract == nil {
                return nil
        }

        // ...
```

Recommended Mitigation Steps

It's pretty difficult to fix this properly. The ideal solution is to distribute fees according to each contract's gas usage. That will be a little more complicated to implement. Also, you have to keep an eye on whether it incentivizes developers to make their contracts less efficient. Another solution is to make this feature permissioned so that only select contracts are allowed to participate. For example, you could say that an address has to be triggered X amount of times before it is eligible for gas refunds.

tkkwon1998 (Canto) acknowledged and commented:

> We acknowledge this as true, but it's a drawback that was discussed during the design of CSR.

[H-02] A registered contract won't earn fees if `_recipient` is a fresh address

Submitted by **Jeiwan**, *also found by* **ronnyx2017**

Users might become victims of a false positive: if they use a fresh account as an NFT recipient during contract registration, the transaction won't revert, but the registered contract will never earn fees for the token holder. And since a contract can be

registered only once, there won't be a way for affected users to re-register contracts and start earning fees. This can affect both bigger and smaller projects that register their contracts with the Turnstile contract: the only condition for the bug to happen is that the recipient address that's used during registration is a fresh address (i.e. an address that hasn't been used yet).

Proof of Concept

The `register` function allows the calling contract to specify the address that will receive the freshly minted NFT (Turnstile.sol#L86):

```
function register(address _recipient) public onlyUnregistered re
    address smartContract = msg.sender;

    if (_recipient == address(0)) revert InvalidRecipient();

    tokenId = _tokenIdTracker.current();
    _mint(_recipient, tokenId);
    _tokenIdTracker.increment();

    emit Register(smartContract, _recipient, tokenId);

    feeRecipient[smartContract] = NftData({
        tokenId: tokenId,
        registered: true
    });
}
```

A recipient address can be any address besides the zero address. However, on the consensus layer, there's a stricter requirement (event_handler.go#L31-L33): a recipient address cannot be a *fresh account*, that is an address that:

- hasn't ever received native coins;
- hasn't ever sent a transaction;
- hasn't ever had contract code.

While, on the application layer, calling the `register` function with a fresh address will succeed, on the consensus layer a contract won't be registered.

When a `Register` event is processed on the consensus layer, there's a check that requires that the recipient address is an *existing account* in the state database (event_handler.go#L31-L33):

```
// Check that the receiver account  exists in the evm store
if acct := k.evmKeeper.GetAccount(ctx, event.Recipient); acct ==
  return sdkerrors.Wrapf(ErrNonexistentAcct, "EventHandler::Regi
}
```

If the recipient account doesn't exist, the function will return, but the register transaction won't revert (errors during the events processing doesn't result in a revert: evm_hooks.go#L123-L132, evm_hooks.go#L49).

The `GetAccount` function above returns `nil` when an address doesn't exist in the state database. To see this, we need to unwind the `GetAccount` execution:

1. The `GetAccount` is called on an `evmKeeper` (event_handler.go#L31):

```
if acct := k.evmKeeper.GetAccount(ctx, event.Recipient); acct == nil {
  return sdkerrors.Wrapf(ErrNonexistentAcct, "EventHandler::RegisterEve
}
```

2. `evmKeeper` is set during the CSR Keeper initialization (keeper.go#L27):

```
func NewKeeper(
  cdc codec.BinaryCodec,
  storeKey sdk.StoreKey,
  ps paramtypes.Subspace,
  accountKeeper types.AccountKeeper,
  evmKeeper types.EVMKeeper,
  bankKeeper types.BankKeeper,
  FeeCollectorName string,
) Keeper {
  // set KeyTable if it has not already been set
  if !ps.HasKeyTable() {
    ps = ps.WithKeyTable(types.ParamKeyTable())
  }

  return Keeper{
    storeKey:         storeKey,
```

```
      cdc:              cdc,
      paramstore:       ps,
      accountKeeper:    accountKeeper,
      evmKeeper:        evmKeeper,
      bankKeeper:       bankKeeper,
      FeeCollectorName: FeeCollectorName,
    }
  }
```

3. The CSR Keeper is initialized during the main app initialization (app.go#L473-L478), this is also when the EVM Keeper is initialized (app.go#L409-L413):

```
app.EvmKeeper = evmkeeper.NewKeeper(
  appCodec, keys[evmtypes.StoreKey], tkeys[evmtypes.TransientKey], app.
  app.AccountKeeper, app.BankKeeper, &stakingKeeper, app.FeeMarketKeepe
  tracer,
)
```

4. The EVM Keeper is implemented and imported from Ethermint (keeper.go#L67);
5. Here's the `GetAccount` function (statedb.go#L25):

```
func (k *Keeper) GetAccount(ctx sdk.Context, addr common.Address) *stat
  acct := k.GetAccountWithoutBalance(ctx, addr)
  if acct == nil {
    return nil
  }

  acct.Balance = k.GetBalance(ctx, addr)
  return acct
}
```

6. The `GetAccountWithoutBalance` function calls `GetAccount` on `accountKeeper` (keeper.go#L255-L258):

```
acct := k.accountKeeper.GetAccount(ctx, cosmosAddr)
if acct == nil {
  return nil
}
```

7. The Account Keeper is implemented in the Cosmos SDK (account.go#L41-L49):

```
func (ak AccountKeeper) GetAccount(ctx sdk.Context, addr sdk.AccAddress
  store := ctx.KVStore(ak.storeKey)
  bz := store.Get(types.AddressStoreKey(addr))
  if bz == nil {
    return nil
  }

  return ak.decodeAccount(bz)
}
```

8. It basically reads an account from the store passed in the context object (context.go#L280-L282);
9. In the Account Keeper, there's also `SetAccount` function (account.go#L72), and it's called in Ethermint by the EVM Keeper (statedb.go#L126);
10. The EVM Keeper's `SetAccount` is called when transaction changes are committed to the state database (statedb.go#L449);
11. The state database is a set of state objects, where keys are account addresses and values are accounts themselves;
12. The `getOrNewStateObject` function initializes new state objects (statedb.go#L221-L227);
13. `getOrNewStateObject` is only called by these functions: `AddBalance`, `SubBalance`, `SetNonce`, `SetCode`, `SetState` (statedb.go#L290-L328).

Thus, a new account object in the state database is only created when an address receives native coins, sends a transaction (which increases the nonce), or when contract code is deployed at it.

Example Exploit Scenario

1. Alice deploys a smart contract that attracts a lot of users.
2. Alice registers the contract in Turnstile. As a recipient contract for the NFT, Alice decides to use a dedicated address that hasn't been used for anything else before (hasn't received coins, hasn't sent a transaction, etc.).
3. The `register` function call succeeds and Alice's contract gets registered in Turnstile.
4. However, due to the "only existing recipient account" check on the consensus layer, Alice's contract wasn't registered on the consensus layer and doesn't earn fees.

5. Since `register` and `assign` can only be called once (due to the `onlyUnregistered` modifier), Alice cannot re-register her contract. She can transfer the NFT to a different address, however this won't make the contract registered on the consensus layer and the owner of the NFT will never receive fees.

Recommended Mitigation Steps

Consider removing the "only existing recipient account" check in the `RegisterEvent` handler since it creates a discrepancy between the application and the consensus layers. Otherwise, if it's mandatory that receiver addresses are not fresh, consider returning an error in the `PostTxProcessing` hook (which will revert a transaction) if there was an error during events processing.

tkkwon1998 (Canto) confirmed and commented:

> This is certainly an issue as there is a mismatch in checks between app and consensus layers.

Medium Risk Findings (3)

[M-01] evm_hooks ignores some important errors

Submitted by **hihen**

https://github.com/code-423n4/2022-11-canto/blob/2733fdd1bee73a6871c6243f92a007a0b80e4c61/Canto/x/csr/keeper/evm_hooks.go#L49

https://github.com/code-423n4/2022-11-canto/blob/2733fdd1bee73a6871c6243f92a007a0b80e4c61/Canto/x/csr/keeper/evm_hooks.go#L101-L135

Impact

Some contracts and some Turnstile tokens (NFTs) will not be able to receive CSR fees forever.

Proof of Concept

In evm*hooks.go, the [PostTxProcessing](*https://github.com/code-423n4/2022-11-canto/blob/2733fdd1bee73a6871c6243f92a007a0b80e4c61/Canto/x/csr/keeper/evmhooks.go#L49) will call `h.processEvents(ctx, receipt)` to handle `Register` and `Assign` events from Turnstile contract first:

```
h.processEvents(ctx, receipt)
```

Notice that the `processEvents` function does not return any error.

However, it is possible for **processEvents** to encounter an error:

```
func (h Hooks) processEvents(ctx sdk.Context, receipt *ethtypes.
  ...
  for _, log := range receipt.Logs {
    ...
    if log.Address == turnstileAddress {
      ...
      switch event.Name {
      case types.TurnstileEventRegister:
        err = h.k.RegisterEvent(ctx, log.Data)
      case types.TurnstileEventUpdate:
        err = h.k.UpdateEvent(ctx, log.Data)
      }
      if err != nil {
        h.k.Logger(ctx).Error(err.Error())
        return
      }
    }
  }
}
```

According to the above implementation of `processEvents`, it will process all the events emitted by the transaction one by one. If one of them encounters an error, it will return directly without any error, and any subsequent unprocessed events will be ignored.

Suppose we have a transaction containing the following events (by contract calls):

1. Register C1 with token1
2. Register C2 with token2
3. Assign C3 with token1

If `RegisterEvent()` returns an error when handling the first event, then all of the events will not be handled because `processEvents()` will return after logging the error.

And `PostTxProcessing()` continues to execute normally because it is unaware of the error.

According to the current implementation of `RegisterEvent()` and `UpdateEvent`, they are both easy to encounter an error. Like `register()` using a recipient that doesn't exist yet.

As a result, none of the C1, C2, C3 contracts will be able to receive any CSR fee because they are not recorded in csr store.

Contracts C1, C2, C3 will never be able to register for CSR because they are marked registered in Turnstile contract (evm store) and will be reverted by `onlyUnregistered` when calling `register()` or `assign()`.

And all other contracts calling `assign(token1)` or `assign(token2)` will enter the same state as C1/C2/C3, because the `assign()` will succeed in Turnstile contract but fail in `UpdateEvent()` (because the store can not find token1 or token2):

```
// Check if the NFT that is being updated exists in the CSR st
nftID := event.TokenId.Uint64()
csr, found := k.GetCSR(ctx, nftID)
if !found {
      return sdkerrors.Wrapf(ErrNFTNotFound, "EventHandler::Up
}
```

Tools Used

VS Code

Recommended Mitigation Steps

`processEvents()` should return the error it encounters, and `PostTxProcessing()` should return that error too.

tkkwon1998 (Canto) confirmed and commented:

> If there are multiple TXs registering contracts, but one of them fails, all of them will fail without much of a warning, so issue confirmed. Although, instead of returning the error to revert the TX, the error should be logged and the for loop inside of processEvents should be allowed to continue.

[M-02] `PostTxProcessing` can revert user transactions not interacting with Turnstile

Submitted by Jeiwan, also found by hihen

https://github.com/code-423n4/2022-11-canto/blob/2733fdd1bee73a6871c6243f92a007a0b80e4c61/Canto/x/csr/keeper/evm_hooks.go#L63

https://github.com/code-423n4/2022-11-canto/blob/2733fdd1bee73a6871c6243f92a007a0b80e4c61/Canto/x/csr/keeper/evm_hooks.go#L75

https://github.com/code-423n4/2022-11-canto/blob/2733fdd1bee73a6871c6243f92a007a0b80e4c61/Canto/x/csr/keeper/evm_hooks.go#L81

https://github.com/code-423n4/2022-11-canto/blob/2733fdd1bee73a6871c6243f92a007a0b80e4c61/Canto/x/csr/keeper/evm_hooks.go#L88

Impact

Any transaction, even those that don't interact with the Turnstile contract, can be reverted by the `PostTxProcessing` hook if there was a CSR specific error. Thus, the CSR module can impair the behavior of smart contracts not related to the module.

Proof of Concept

The `PostTxProcessing` is used by the keeper to register contracts with the CSR module and distribute gas fees to registered contracts (evm_hooks.go#L41). The hook can return an error while handling CSR specific operations:

- Reading a CSR object from the storage (evm_hooks.go#L61-L64);
- Sending gas fees to the module (evm_hooks.go#L73-L76);
- Reading the address of Turnstile (evm_hooks.go#L79-L82);
- Calling the `distributeFees` function (evm_hooks.go#L85-L89).

In case any of these operations fails, the whole transaction will be reverted (state_transition.go#L272-L278):

```
if err = k.PostTxProcessing(tmpCtx, msg, receipt); err != nil {
  // If hooks return error, revert the whole tx.
  res.VmError = types.ErrPostTxProcessing.Error()
  k.Logger(ctx).Error("tx post processing failed", "error", err)
```

One example of when the hook can revert in normal circumstances is when the fees to be distributed are 0, which can be caused by a combination of low gas usage of a transaction, a small CSR share, and rounding (the fees are a share of the gas spent to execute a transaction: evm_hooks.go#L66-L70). In this case, the `distributeFees` call will revert and will cause the whole transaction to be reverted as well (evm_hooks.go#L86-L89, Turnstile.sol#L149).

Recommended Mitigation Steps

In the `PostTxProcessing` hook, consider always logging errors and returning `nil` to avoid impairing user transactions. Also, consider logging a fatal error and exiting when the module cannot function due to an error.

tkkwon1998 (Canto) confirmed and commented:

> Error should not be returned, but rather logged and continued to an eventual nil return.

[M-03] There is no re-register or re-assign function

Submitted by sorrynotsorry

https://github.com/code-423n4/2022-11-canto/blob/2733fdd1bee73a6871c6243f92a007a0b80e4c61/CIP-001/src/Turnstile.sol#L86-L101

https://github.com/code-423n4/2022-11-canto/blob/2733fdd1bee73a6871c6243f92a007a0b80e4c61/CIP-001/src/Turnstile.sol#L107-L120

Impact

There is no re-register or re-assign option for the smart contracts.

Let's assume a smart contract is registered either through the `register()` function with a new NFT minted or the `assign()` function to an existing NFT.

However, if somehow, the NFT is burned by the owner or transferred to another owner either by an approval or compromised tx, there is no option to re-register for these contracts which create gas fees but might not get a fee distribution in return.

And if the NFT is burned or transferred to another owner, the smart contracts will lose the fees generated if not previously withdrawn.

Proof of Concept

`register` function;

```
        function register(address _recipient) public onlyUnregistere
            address smartContract = msg.sender;

            if (_recipient == address(0)) revert InvalidRecipient();

            tokenId = _tokenIdTracker.current();
            _mint(_recipient, tokenId);
            _tokenIdTracker.increment();

            emit Register(smartContract, _recipient, tokenId);
```

```
            feeRecipient[smartContract] = NftData({
                tokenId: tokenId,
                registered: true
            });
        }
```

[Permalink]

`assign` function;

```
        function assign(uint256 _tokenId) public onlyUnregistered re
            address smartContract = msg.sender;

            if (!_exists(_tokenId)) revert InvalidTokenId();

            emit Assign(smartContract, _tokenId);

            feeRecipient[smartContract] = NftData({
                tokenId: _tokenId,
                registered: true
            });

            return _tokenId;
        }
```

[Permalink]

Recommended Mitigation Steps

The team might consider adding an option to validate historical registrations and re-register those contracts accordingly.

[tkkwon1998 (Canto) acknowledged and commented]:

> Currently there is no way to re-assign or re-register. This is a known limitation, and will be made extremely clear to all devs registering their contracts.

Low Risk and Non-Critical Issues

For this contest, 13 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: Josiah, Deivitto, peritoflores, cryptonue, keccak123, RaymondFam, rotcivegaf, aphak5010, gzeon, martin, cccz, *and* joestakey *.*

Low Risk Issues Summary

Number	Issues Details	Context
[L-01]	`onlyOwner` privileges can cause damage of the project in a possible privateKey exploit	1
[L-02]	Use `safeTransferOwnership` instead of `transferOwnership` function	1
[L-03]	Owner can renounce Ownership	1
[L-04]	Critical Address Changes Should Use Two-step Procedure	1
[L-05]	Missing Re-Entrancy Guard to `withdraw` function	1
[L-06]	No Check if OnErc721Received is implemented	1

Total: 6 issues

[L-01] Very critical `onlyOwner` privileges can cause damage of the project in a possible privateKey exploit

https://github.com/code-423n4/2022-11-canto/blob/main/CIP-001/src/Turnstile.sol#L148

Typically, the contract's `onlyOwner` is the account that deploys the contract. As a result, the `onlyOwner` is able to perform certain privileged activities.

However, `onlyOwner` privileges are numerous and there is no timelock structure in the process of using these privileges.

The onlyOwner is assumed to be an EOA, since the documents do not provide information on whether the `onlyOwner` will be a multisign structure.

In parallel with the private key thefts of the project onlyOwners, which have increased recently, this vulnerability has been stated as Medium.

Similar vulnerability;

Private keys stolen:

Hackers have stolen cryptocurrency worth around €552 million from a blockchain project linked to the popular online game Axie Infinity, in one of the largest cryptocurrency heists on record. Security issue: PrivateKey of the project officer was stolen: https://www.euronews.com/next/2022/03/30/blockchain-network-ronin-hit-by-552-million-crypto-heist

Proof of Concept

`onlyonlyOwner` powers;

```
Canto/contracts/turnstile.sol:
  146
  147:     /// @notice Distributes collected fees to the smart c
  148:     /// @param _tokenId NFT that earned fees
  149:     function distributeFees(uint256 _tokenId) public only
  150          if (msg.value == 0) revert NothingToDistribute();
```

`onlyOwner` can also make this functionality unavailable by using the renounceOwnership property.

Recommended Mitigation Steps

1. A timelock contract should be added to use `onlyOwner` privileges. In this way, users can be warned in case of a possible security weakness.
2. `onlyOwner` can be a Multisign wallet and this part is specified in the documentation.

[L-02] Use `safeTransferOwnership` instead of `transferOwnership` function

Turnstile.sol#L4

`transferOwnership` function is used to change Ownership from `Ownable.sol` .

Use a 2 structure transferOwnership which is safer.

`safeTransferOwnership` is more secure due to 2-stage ownership transfer.

Recommended Mitigation Steps

Use `Ownable2Step.sol` [Ownable2Step.sol]

[L-03] Owner can renounce Ownership

[Turnstile.sol#L4]

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.

The Openzeppelin's Ownable used in this project contract implements renounceOwnership. This can represent a certain risk if the ownership is renounced for any other reason than by design. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

`onlyOwner` functions:

```
Canto/contracts/turnstile.sol:
  146
  147:     /// @notice Distributes collected fees to the smart c
  148:     /// @param _tokenId NFT that earned fees
  149:     function distributeFees(uint256 _tokenId) public only
  150          if (msg.value == 0) revert NothingToDistribute();
```

Recommended Mitigation Steps

We recommend either reimplementing the function to disable it or to clearly specify if it is part of the contract design.

[L-04] Critical Address Changes Should Use Two-step Procedure

The critical procedures should be a two-step process.

Canto/contracts/turnstile.sol:

```
86      /// @return tokenId of the ownership NFT that collect
87:     function register(address _recipient) public onlyUnre
88:         address smartContract = msg.sender;
89:
90:         if (_recipient == address(0)) revert InvalidRecip
91:
92:         tokenId = _tokenIdTracker.current();
93:         _mint(_recipient, tokenId);
94:         _tokenIdTracker.increment();
95:
96:         emit Register(smartContract, _recipient, tokenId)
97:
98:         feeRecipient[smartContract] = NftData({
99:             tokenId: tokenId,
100:            registered: true
101:        });
102:    }
```

Recommended Mitigation Steps

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding a two-step procedure on the critical functions.

[L-05] Missing Re-Entrancy Guard to `withdraw` function

Turnstile.sol#L141

Ether transfer is done with `Address.sendValue(_recipient, _amount);` in the `withdraw` function, OZ's Address.sol library is used

Canto/contracts/turnstile.sol:

```
127     /// @return amount of fees withdrawn
128:    function withdraw(uint256 _tokenId, address payable _
129:        public
130:        onlyNftOwner(_tokenId)
```

```
131:         returns (uint256)
132:     {
133:         uint256 earnedFees = balances[_tokenId];
134:
135:         if (earnedFees == 0 || _amount == 0) revert Nothi
136:         if (_amount > earnedFees) _amount = earnedFees;
137:
138:         balances[_tokenId] = earnedFees - _amount;
139:
140:         emit Withdraw(_tokenId, _recipient, _amount);
141:
142:         Address.sendValue(_recipient, _amount);
143:
144:         return _amount;
145:     }
```

There is this warning in OZ's Address.sol library. Accordingly, he used the Check-Effect-Interaction pattern in the project:

```
* IMPORTANT: because control is transferred to `recipient`, care
     * taken to not create reentrancy vulnerabilities. Consider
     * {ReentrancyGuard} or the
     * https://solidity.readthedocs.io/en/v0.5.11/security-cons
     */
```

It would be best practice to use re-entrancy Guard for reasons such as complicated dangers such as view Re-Entrancy that emerged in the last period and the possibility of expanding the project and its integration with other contracts:

https://twitter.com/lmount_/status/1577307245276459011?t=XSvtlkum8nuzI8ZM7lfq-A&s=19

https://github.com/euler-xyz/euler-contracts/commit/91adeee39daf8ece00584b6f7ec3e60a1d226bc9

[L-06] No Check if OnErc721Received is implemented

`Turnstile.register()` will mint a NFT . When minting a NFT, the function does not check if a receiving contract implements `onERC721Received()` .

`_recipient` can be contract address.

The intention behind this function is to check if the address receiving the NFT. If it is a contract, implements `onERC721Received()`. Thus, there is no check whether the receiving address supports ERC-721 tokens and position could be not transferrable in some cases.

Following shows that `_mint` is used instead of `_safeMint`:

```
Canto/contracts/turnstile.sol:
   86      /// @return tokenId of the ownership NFT that collect
   87:     function register(address _recipient) public onlyUnre
   88:         address smartContract = msg.sender;
   89:
   90:         if (_recipient == address(0)) revert InvalidRecip
   91:
   92:         tokenId = _tokenIdTracker.current();
   93:         _mint(_recipient, tokenId);
   94:         _tokenIdTracker.increment();
   95:
   96:         emit Register(smartContract, _recipient, tokenId)
   97:
   98:         feeRecipient[smartContract] = NftData({
   99:             tokenId: tokenId,
  100:             registered: true
  101:         });
  102:     }
```

Recommended Mitigation Steps

Consider using `_safeMint` instead of `_mint`.

Non-Critical Issues Summary

Number	Issues Details	Context
[N-01]	For modern and more readable code; update import usages	1
[N-02]	NatSpec is missing	1
[N-03]	No same value input control	1

Number	Issues Details	Context
[N-04]	`0 address` check for `asset`	1
[N-05]	`Function writing` that does not comply with the `Solidity Style Guide`	1
[N-06]	Add a timelock to critical functions	1
[N-07]	Take advantage of Custom Error's return value property	1
[N-08]	Using Vulnerable Version of Openzeppelin	1
[N-09]	Import only the parts you use	1

Total: 9 issues

[N-01] For modern and more readable code; update import usages

Solidity code is also cleaner in another way that might not be noticeable: the struct Point. We were importing it previously with global import but not using it. The Point struct `polluted the source code` with an unnecessary object we were not using because we did not need it.

This was breaking the rule of modularity and modular programming: `only import what you need` . Specific imports with curly braces allow us to apply this rule better.

Recommended Mitigation Steps

```
import {contract1 , contract2} from "filename.sol";
```

A good example from the ArtGobblers project:

```
import {Owned} from "solmate/auth/Owned.sol";
import {ERC721} from "solmate/tokens/ERC721.sol";
import {LibString} from "solmate/utils/LibString.sol";
import {MerkleProofLib} from "solmate/utils/MerkleProofLib.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib
import {ERC1155, ERC1155TokenReceiver} from "solmate/tokens/ERC1
import {toWadUnsafe, toDaysWadUnsafe} from "solmate/utils/Signed
```

[N-02] NatSpec is missing

NatSpec is missing for the following functions, constructor and modifier:

```
CIP-001/src/Turnstile.sol:
  18
  19:     struct NftData {
  20:         uint256 tokenId;
  21:         bool registered;
  22:     }
  23:
  24:     Counters.Counter private _tokenIdTracker;

  32:     event Register(address smartContract, address recipien
  33:     event Assign(address smartContract, uint256 tokenId);
  34:     event Withdraw(uint256 tokenId, address recipient, uin
  35:     event DistributeFees(uint256 tokenId, uint256 feeAmoun

  61:     constructor() ERC721("Turnstile", "Turnstile") {}
```

[N-03] No same value input control

```
CIP-001/src/Turnstile.sol:
   89      /// @return tokenId of the ownership NFT that collect
   90:     function register(address _recipient) public onlyUnre
   91:         address smartContract = msg.sender;
   92:
   93:         if (_recipient == address(0)) revert InvalidRecip
   94:
   95:         tokenId = _tokenIdTracker.current();
   96:         _mint(_recipient, tokenId);
   97:         _tokenIdTracker.increment();
   98:
   99:         emit Register(smartContract, _recipient, tokenId)
  100:
  101:         feeRecipient[smartContract] = NftData({
  102:             tokenId: tokenId,
  103:             registered: true
  104:         });
  105:     }
```

[N-04] 0 address check for asset

```
CIP-001/src/Turnstile.sol:
  144
  145:         Address.sendValue(_recipient, _amount);
```

Also check of the address to protect the code from 0x0 address problem just in case. This is best practice or instead of suggesting that they verify address `!= 0x0`, you could add some good NatSpec comments explaining what is valid and what is invalid and what are the implications of accidentally using an invalid address.

Recommended Mitigation Steps

Like this; `if (_recipient == address(0)) revert ADDRESS_ZERO();`

[N-05] Function writing that does not comply with the Solidity Style Guide

Order of Functions; ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. But there are contracts in the project that do not comply with this.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html

Functions should be grouped according to their visibility and ordered:

constructor

receive function (if exists)

fallback function (if exists)

external

public

internal

private

Within a grouping, place the view and pure functions last.

[N-06] Add a timelock to critical functions

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate (less risk of a malicious owner making a sandwich attack on a user).

Consider adding a timelock to:

https://github.com/code-423n4/2022-11-canto/blob/main/CIP-001/src/Turnstile.sol#L12 Turnstile.sol#L12

For the imported contracts containing critical functions (TransferOwnerShip, RenounOwnerShip etc.), add timelock to them.

[N-07] Take advantage of Custom Error's return value property

An important feature of Custom Error is that values such as address, tokenID, msg.value can be written inside the `()` sign, this kind of approach provides a serious advantage in debugging and examining the revert details of dapps such as tenderly.

For Example:

```
CIP-001/src/Turnstile.sol:
  110       /// @return tokenId of the ownership NFT that collect
  111:      function assign(uint256 _tokenId) public onlyUnregiste
  112:          address smartContract = msg.sender;
  113:
- 114:          if (!_exists(_tokenId)) revert InvalidTokenId();
+ 114:          if (!_exists(_tokenId)) revert InvalidTokenId(_to
```

[N-08] Using Vulnerable Version of OpenZeppelin

The package.json configuration file says that the project is using 4.7.0 of OpenZeppelin which has a vulnerability.

Although there is no security vulnerability covering the project, it is recommended to use the latest version 4.7.3.

```
CIP-001/lib/openzeppelin-contracts/package.json:
  1: {
  2:   "name": "openzeppelin-solidity",
  3:   "description": "Secure Smart Contract library for Solidit
  4:   "version": "4.7.0",
```

[N-09] Import only the parts you use

Instead of importing all of the imported contracts, import only the parts you use. This both eliminates a security risk and is a better method for transparency for those who read and audit the contract.

```
Canto/x/csr/keeper/csr.go:
   2
   3: import (
   4:   "encoding/binary"
   5:
   6:   "github.com/Canto-Network/Canto/v2/x/csr/types"
   7:   "github.com/cosmos/cosmos-sdk/store/prefix"
   8:   sdk "github.com/cosmos/cosmos-sdk/types"
   9:   "github.com/ethereum/go-ethereum/common"
  10: )
```

berndartmueller (judge) commented:

> Great report!

Gas Optimizations

For this contest, 26 reports were submitted by wardens detailing gas optimizations. The **report highlighted below** by **Tricko** received the top score from the judge.

The following wardens also submitted reports: JC, Deivitto, peritoflores, Awesome, oyc_109, abiih, RaymondFam, rotcivegaf, aphak5010, gzeon, DijkstraDev, Dinesh11G, AkshaySrivastav, Englave, martin, Rahoz, SaeedAlipoor01988, exolorkistis, ReyAdmirado, Sathish9098, Beepidibop, chaduke, nicobevi, chrisdior4, *and* 0xhacksmithh .

[G-01] Remove `registered` flag.

Besides `isRegistered` function, `registered` is never used. The flag is only 1bit, but it allocates an entire extra storage slot for every call to `register` or `assign`. `SSTORE` opcode cost up to 20000 gas for uninitialized slots like these.

Removing this flag reduces the total storage needed by 50% for each allocation on `feeRecipient` mapping, resulting in gas reductions for `register` and `assign` functions.

	avg. gas (before modification)	avg. gas (after modification)	gas diff
assign	44884	23688	-21196 (-47.2%)
register	157540	137738	-19802 (-12.7%)
total	202242	161426	-40998 (-20.3%)

Modifications

If we start `TokenId` count from 1, we can check if the NFT is registered by checking if `feeRecipient[_smartContract] != 0`. This allows us to remove the `registered` flag and simplify the code as shown below.

```
diff --git a/Turnstile.sol.orig b/Turnstile.sol.mod
index b8c216a..c02c467 100644
--- a/Turnstile.sol.orig
+++ b/Turnstile.sol.mod
@@ -12,15 +12,10 @@ import "openzeppelin/utils/Counters.sol";
 contract Turnstile is Ownable, ERC721Enumerable {
     using Counters for Counters.Counter;

-    struct NftData {
-        uint256 tokenId;
-        bool registered;
-    }
```

```
-
     Counters.Counter private _tokenIdTracker;

     /// @notice maps smart contract address to tokenId
-    mapping(address => NftData) public feeRecipient;
+    mapping(address => uint256) public feeRecipient;

     /// @notice maps tokenId to fees earned
     mapping(uint256 => uint256) public balances;
@@ -54,7 +49,9 @@ contract Turnstile is Ownable, ERC721Enumerable
         _;
     }

-    constructor() ERC721("Turnstile", "Turnstile") {}
+    constructor() ERC721("Turnstile", "Turnstile") {
+        _tokenIdTracker.increment();
+    }

     /// @notice Returns current value of counter used to tokenId
     /// @return current counter value
@@ -68,14 +65,14 @@ contract Turnstile is Ownable, ERC721Enumerable
     function getTokenId(address _smartContract) external view re
         if (!isRegistered(_smartContract)) revert Unregistered(

-        return feeRecipient[_smartContract].tokenId;
+        return feeRecipient[_smartContract];
     }

     /// @notice Returns true if smart contract is registered to
     /// @param _smartContract address of the smart contract
     /// @return true if smart contract is registered to collect
     function isRegistered(address _smartContract) public view re
-        return feeRecipient[_smartContract].registered;
+        return feeRecipient[_smartContract] != 0;
     }

     /// @notice Mints ownership NFT that allows the owner to co
@@ -94,10 +91,7 @@ contract Turnstile is Ownable, ERC721Enumerable

         emit Register(smartContract, _recipient, tokenId);

-        feeRecipient[smartContract] = NftData({
-            tokenId: tokenId,
-            registered: true
-        });
+        feeRecipient[smartContract] = tokenId;
```

```
    }

    /// @notice Assigns smart contract to existing NFT. That NF
@@ -111,10 +105,7 @@ contract Turnstile is Ownable, ERC721Enumera

        emit Assign(smartContract, _tokenId);

-       feeRecipient[smartContract] = NftData({
-           tokenId: _tokenId,
-           registered: true
-       });
+       feeRecipient[smartContract] = _tokenId;

        return _tokenId;
    }
```

[G-02] Use `ERC721` instead of `ERC721Enumerable`

Using `ERC721Enumerable` is known to incur large gas overheads, as the Turnstile contract already manage the `TokenId` by itself (using `Counters.Counter`) and does not use any of the extra functionality present in `ERC721Enumerable` (e.g `totalSupply()`, `tokenByIndex(index)`, `tokenOfOwnerByIndex(owner, index)`). We can safely change to vanilla `ERC721`.

	avg. gas (before modification)	avg. gas (after modification)	gas diff	
register	157540	72209	-85331 (-54.2%)	

Modifications

```
diff --git a/Turnstile.sol.orig b/Turnstile.sol.mod
index b8c216a..7f580b0 100644
--- a/Turnstile.sol.orig
+++ b/Turnstile.sol.mod
@@ -2,14 +2,14 @@
 pragma solidity 0.8.17;

 import "openzeppelin/access/Ownable.sol";
-import "openzeppelin/token/ERC721/extensions/ERC721Enumerable.s
+import "openzeppelin/token/ERC721/ERC721.sol";
 import "openzeppelin/utils/Counters.sol";
```

```
 /// @notice Implementation of CIP-001 https://github.com/Canto-
 /// @dev Every contract is responsible to register itself in the
 ///      If contract is using proxy pattern, it's possible to re
 ///      Recipient withdraws fees by calling `withdraw(uint256,a
-contract Turnstile is Ownable, ERC721Enumerable {
+contract Turnstile is Ownable, ERC721 {
     using Counters for Counters.Counter;
```

[G-03] Leave 1 wei in `balance` after `withdraw`

Every time that a user withdraws all their balance, their storage slot on the `balance` mapping is freed, refunding gas to the caller. However the next call to `distributeFees` on this same user balance would require the storage slot to be re-initialized, which incurs high gas cost.

This effect is dependent on users mainly withdrawing their full balances, which seems the most probable behaviour, as there is no incentive to keep any balance left on the Turnstile contract. Considering this user behaviour, modifying the `withdraw` function to leave 1 wei in storage can result in a net reduction of gas.

	avg. gas (before modification)	avg. gas (after modification)	gas diff
withdraw	10142	12682	+2590 (+25.5%)
distributeFees	22472	4562	-17910 (-79.7%)
total	32614	17244	-15370 (-42.1%)

* These values are from a simulation of 10 sucessive (full `withdraw` followed by `distributeFees`) sequences.

Note that while there is a net reduction of gas, it is heavily skewed to the contract owner (who is the sole caller of `distributeFees`). While users would be affected negatively as the `withdraw` gas cost would be increased.

Modification

```
diff --git a/Turnstile.sol.orig b/Turnstile.sol.mod
```

```
index b8c216a..72c6e7c 100644
--- a/Turnstile.sol.orig
+++ b/Turnstile.sol.mod
@@ -132,7 +132,7 @@ contract Turnstile is Ownable, ERC721Enumeral
         uint256 earnedFees = balances[_tokenId];

         if (earnedFees == 0 || _amount == 0) revert NothingToWi
-        if (_amount > earnedFees) _amount = earnedFees;
+        if (_amount >= earnedFees) _amount = earnedFees - 1;

         balances[_tokenId] = earnedFees - _amount;
```

Gas Optimizations Conclusion

Combining all the modifications described above, we obtain the following total gas diffs.

	avg. gas (before modifications)	avg. gas (after modifications)	gas diff
assign	44884	23688	-21196 (-47.2%)
register	157540	50339	-107141 (-68.0%)
withdraw	10142	12682	+2590 (+25.5%)
distributeFees	22472	4562	-17910 (-79.7%)
total	235038	91271	-143767 (-61.2%)

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