e

CD SECURITY

AUDIT REPORT

PowerX
November 2024

Prepared by
0xruhum
klau5

Introduction

A time-boxed security review of the **PowerX** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **PowerX**

PowerX is a referral system for that enables multi-level marketing (MLM) structures. With a flexible commission structure participants are paid using both ERC20 and ETH tokens.

The first product using the referral system are the Wrapped Hydra Miners (WHydra) which represent time-bound mining operations. The protocol handles minting and claiming WHydra tokens as well as the automatic distribution of the commission generated through the user's payments to referrers and partners.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

review commit hash -
9aca6c8e888c3c923d3c7dbac6b7181ab7f9b0e55ff132349ee079df942ef4de78b71844f4d15e05

Scope

The following smart contracts were in scope of the audit:

- `contracts/*`

The following number of issues were found, categorized by their severity:

- Critical & High: 7 issues
- Medium: 4 issues
- Low & Info: 3 issues

Findings Summary

ID	Title	Severity
[H-01]	referrer ID in PowerX isn't collision resistant	High
[H-02]	User can redirect their commissions to themselves	High
[H-03]	Transferring a WHydra token to a new user will cause user's referrer to be set to the token's referrer	High
[H-04]	Attacker can sandwich swap in `WHydra.claimMint()`	High
[H-05]	Attacker can set the referrer for an unregistered user	High
[H-06]	Anyone can set the referral of top-level users or partners	High
[H-07]	Claim commission can be stolen by setting `token.mint.referrer` to a different account than `onboardedBy`	High
[M-01]	When the owner of WHydra NFT changes, claim commission can not be transferred to the WHydra referrer	Medium
[M-02]	`earlyEndMint` sends TitanX to wrong proxy, making early end impossible	Medium
[M-03]	`onboardedBy` is not separated by project, so referral commissions are paid even during the minting period of other projects in the future	Medium
[M-04]	Refund function may fail or refund incorrect amounts	Medium
[L-01]	Total Commission Rate Should Not Exceed 100%	Low
[I-01]	Check `mintPower` limit first	Informational
[I-02]	Users can use WHydra for fraud because claimed WHydra tokens can still be transferred	Informational

Detailed Findings

[H-01] referrer ID in PowerX isn't collision resistant

Severity

Impact: High

Likelihood: Medium

Description

In Referral.sol:1083 the `referrerId` is computed as a 10 character long hex string:

```
function _generateReferrerId(Set storage set, address user) private view
returns (ShortString id) {
    bytes32 hashed = keccak256(abi.encodePacked(set._referrals.length,
block.timestamp, address(this), user));
    id = _substring(_toHexString(hashed), 0, 10).toShortString();
}
```

The possible combinations for a 10-character hex string are $16^{10} = 2^{40}$. If an attacker would generate 1 million IDs, the chance of them finding an existing ID is: $1 - e^{-1,000,000^2/2^{41}} = 1 - e^{-0.4547} = 0.3654 = 36.54\%$

Having a colliding referrer ID will cause the existing user's referrer ID point to a new `Info` struct because `set._ids[]` is overridden for the given referrer ID in Referral.sol:1094:

```
function _getOrAddInfo(Set storage set, address user) private returns
(Info storage info) {
  uint256 position = set._users[user];

  if (position == 0) {
    ShortString id = _generateReferrerId(set, user);

    // Push new dataset directly to storage
    set._referrals.push();

    // Initialise
    Info storage newInfo = set._referrals[set._referrals.length - 1];
    newInfo.id = id;
    newInfo.addy = user;
    newInfo.onboardedBy = NULL_STRING;

    // Use position 0 to indicate `not in set`
    position = set._referrals.length;
    set._users[user] = position;
    set._ids[id] = position;

    emit NewUser(id, user);
  }

  return set._referrals[position - 1];
}
```

Recommendations

Increase the hex string size to allow for more IDs and revert if the calculated ID is known.

[H-02] User can redirect their commissions to themselves

Severity

Impact: Medium

Likelihood: High

Description

When a user uses PowerX for the first time, it will assign the user a referrer. The value can be chosen by the user themselves since it's passed in the function's params. An attacker can calculate their referrer ID *before* interacting with the PowerX contract and use that value as the `referrer`:

```
function _generateReferrerId(Set storage set, address user) private view
returns (ShortString id) {
  bytes32 hashed = keccak256(abi.encodePacked(set._referrals.length,
block.timestamp, address(this), user));
  id = _substring(_toHexString(hashed), 0, 10).toShortString();
}

function addCommission(
  Set storage set,
  address caller,
  address asset,
  uint256 amount,
  address user,
  string memory referrer
) internal returns (uint256 total) {
  ShortString referrerId = referrer.toShortString();

  Info storage userInfo = _getOrAddInfo(set, user);

  // Onboard user only once (do not allow user to modify tree-position)
  if (isValidId(set, referrerId) &&
ShortString.unwrap(userInfo.onboardedBy) ==
ShortString.unwrap(NULL_STRING)) {
    userInfo.onboardedBy = referrerId;
    set._tree[userInfo.id] = referrerId;

    emit OnboardedBy(userInfo.id, referrerId);
  }
```

```
    // Call helper function with applyCommission = true to modify storage
    return _calculateAndApplyCommission(set, caller, asset, amount,
  userInfo.id, userInfo.onboardedBy);
  }
```

The `onboardedBy` ID will point to their address causing a recursive loop in `Referral._calculateAndApplyCommission()`:

```
  function _calculateAndApplyCommission(
    Set storage set,
    address caller,
    address asset,
    uint256 amount,
    ShortString userId,
    ShortString onboardedBy
  ) internal returns (uint256 total) {
    // ...

    for (uint256 idx = 0; idx < set._levels[caller][asset].length; idx++) {
      uint256 level = set._levels[caller][asset][idx];

      if (isValidId(set, onboardedBy)) {
        Info storage mlmReferrerInfo = _getInfoFromId(set, onboardedBy);

        uint256 mlmOverride = mlmReferrerInfo.overrides[caller].get(asset);
        level = mlmOverride > 0 ? mlmOverride : level;

        uint256 commission = (amount * level) / Constants.BASIS;
        total += commission;

        mlmReferrerInfo.vault.addToVault(asset, commission);

        // the array index corresponds to the MLM level
        emit AddCommission(caller, mlmReferrerInfo.id, userId, idx, level,
  asset, commission);

        // Iterate
        onboardedBy = mlmReferrerInfo.onboardedBy;
      } else {
        break;
      }
    }
  }
```

Any commission generated by the user and reserved for the referrers will go to the user instead.

Because the referrer stored in the PowerX contract will be used for every interaction the attacker only needs to do that once. After that, they can freely interact with any kind of frontend while collecting the commission.

Recommendations

Prevent users from setting themselves as their own referral. Additionally, when `closedMint` is `false`, enforce that the `referrer` parameter must be empty.

[H-03] Transferring a WHydra token to a new user will cause user's referrer to be set to the token's referrer

Severity

Impact: High

Likelihood: Medium

Description

When a WHydra token is minted, a `referrer` will be stored in the token's data struct in WHydra.sol:280:

```
  function startMint(StartMintParams calldata params) external payable
returns (Token memory) {
    // ...

    // Start the mint
    Token storage token = _startMint(params.to, params.mintPower,
params.numOfDays);

    // Update token details
    token.mint.referrer = params.referrer.toShortString();

    // ...
    return token;
  }
```

Wrapped Hydra tokens are transferrable as they are standard ERC721 tokens. If the token is transferred to the a new user, i.e. a user that is not registered in the PowerX contract, there's the possibility that the user's referrer is set to the referrer stored in the token data.

In `claimMint()` it will pay a commission for the user with `referrer = token.mint.referrer`. If that user isn't registered, it will set their `onboardedBy` address to that referrer, see WHydra.sol:400 & Referral.sol:634:

```
function claimMint(ClaimMintParams calldata params) external {
  Token storage token = tokens[params.tokenId];

   // ...

  address nftOwner = _ownerOf(params.tokenId);
```

```
  PHydra proxy = PHydra(token.mint.proxy);
  // ...
  token.claimCommission.amount = _payCommission(address(hydra), minted,
nftOwner, token.mint.referrer.toString());
  // ...
}
```

```
function addCommission(
  Set storage set,
  address caller,
  address asset,
  uint256 amount,
  address user,
  string memory referrer
) internal returns (uint256 total) {
  ShortString referrerId = referrer.toShortString();

  Info storage userInfo = _getOrAddInfo(set, user);

  // Onboard user only once (do not allow user to modify tree-position)
  if (isValidId(set, referrerId) &&
ShortString.unwrap(userInfo.onboardedBy) ==
ShortString.unwrap(NULL_STRING)) {
    userInfo.onboardedBy = referrerId;
    set._tree[userInfo.id] = referrerId;

    emit OnboardedBy(userInfo.id, referrerId);
  }

  // Call helper function with applyCommission = true to modify storage
  return _calculateAndApplyCommission(set, caller, asset, amount,
userInfo.id, userInfo.onboardedBy);
}
```

Recommendations

The new user's referrer should be the previous token owner.

[H-04] Attacker can sandwich swap in `WHydra.claimMint()`

Severity

Impact: High

Likelihood: High

Description

`WHydra.claimMint()` is callable by anyone for the NFT's owner. In the function parameters, the caller specifies the swap's slippage, if the NFT owner's ETH is supposed to be refunded:

```
function claimMint(ClaimMintParams calldata params) external {
  Token storage token = tokens[params.tokenId];
  // ...
  address nftOwner = _ownerOf(params.tokenId);
  PHydra proxy = PHydra(token.mint.proxy);
  // ...
  // Process refunds if enabled
  if (token.doRefund && token.mintPayment.asset == Constants.WETH) {
    token.mintRefund.refund.asset = Constants.WETH;
    token.mintRefund.spent.asset = HYDRA;

    Swap.SwapResult memory result = Swap.refund(
      Swap.SwapParams({
        path: REFUND_WETH_PATH,
        slippage: params.slippage,
        twap: params.twap,
        amount: token.mintPayment.amount + token.mintCommission.amount,
        recipient: nftOwner,
        deadline: params.deadline
      })
    );

    token.mintRefund.refund.amount = result.received;
    token.mintRefund.spent.amount = result.spent;
  }
  // ...
}
```

Meaning, that anybody can execute a swap without any slippage protection. This will be abused by bots who claim the mint immediately to sandwich it.

Recommendations

Only the NFT's owner should be allowed to claim the mint.

[H-05] Attacker can set the referrer for an unregistered user

Severity

Impact: High

Likelihood: Medium

Description

An attacker is able to register and set the `onboardedBy` address for an unregistered user. That way they can frontrun the user's mint tx to set `onboardedBy` to their own address to steal the commission.

1. you register a new user with a new address. You set the `referrer` to your main address
2. you frontrun the user's tx where they would be registered and call `PowerX.moveAddress()` to move the user you created to the victim's address

```
// PowerX.sol
/**
 * @notice Moves a user's ID to a new address
 * @param to New address to associate with the user's ID
 */
function moveAddress(address to) external {
  _referral.moveAddress(msg.sender, to);
}
```

```
// Referral.sol
function moveAddress(Set storage set, address from, address to) internal {
  if (!isUser(set, from)) {
    revert NoUser();
  }
  if (isUser(set, to)) {
    revert AlreadyRegistered();
  }

  Info storage info = _getOrAddInfo(set, from);
  info.addy = to;

  set._users[to] = set._users[from];
  delete set._users[from];

  emit AddressMoved(info.id, from, to);
}
```

If the user isn't registered yet, `isUser() == false`, you can update `_users[to]`.

In `Referral._getOrAddInfo()` it will check whether `_users[address] == null`, if that's not the case, it returns the stored value:

```
function _getOrAddInfo(Set storage set, address user) private returns
(Info storage info) {
  uint256 position = set._users[user];

  if (position == 0) {
    ShortString id = _generateReferrerId(set, user);

    // Push new dataset directly to storage
    set._referrals.push();
```

```
    // Initialise
    Info storage newInfo = set._referrals[set._referrals.length - 1];
    newInfo.id = id;
    newInfo.addy = user;
    newInfo.onboardedBy = NULL_STRING;

    // Use position 0 to indicate `not in set`
    position = set._referrals.length;
    set._users[user] = position;
    set._ids[id] = position;

    emit NewUser(id, user);
  }

  return set._referrals[position - 1];
}
```

In our attack, that's the attacker's registered user.

Here's a PoC showcasing the attack:

```
// PowerX.ts
it("attack", async () => {
  const fixture = await loadFixture(deployIsolatedFixture);
  const { powerX, user, genesis, accessManager, others } = fixture;
  const partner = others[0];
  const alice = others[1];

  // this is the attacker's own address
  // we need to add it to generate a referrer ID
  await powerX.connect(genesis).addUser(partner);
  expect(await powerX.isUser(partner)).to.equal(true);

  let partnerView = await powerX["getView(address,address[])"](partner,
[]);
  const id = ShortString.toString(partnerView.id);

  // Now, the attacker, creates a completely new user entry by for
example, paying a commmission
  // THey set the new user's onboardedBy to be the partner's ID
  await powerX.connect(genesis).enable(user, Constants.ADDRESS_ZERO,
true);
  await powerX
    .connect(user)
    .addCommission(Constants.ADDRESS_ZERO, ethers.parseEther("1"), user,
id);

  let userView = await powerX["getView(address,address[])"](partner, []);
  expect(ShortString.toString(userView.id), id);

  // Now, an honest user wants to join the system. They are not registered
```

```
yet.
  // we simulate that through the admin creating a new user entry
  // Normally, they would call `WHydra.startMint()`

  // The attacker frontruns the user's tx, to move the address of their
new user entry
  // to the user's own address. That way they populate the user's data
beforehand.
  // They set it to their own onboardedBy address.
  await powerX.connect(user).moveAddress(alice.address);
  await powerX.connect(genesis).addUser(alice);

  let aliceView = await powerX["getView(address,address[])"](alice, []);
  expect(ShortString.toString(aliceView.onboardedBy)).to.equal(id);
});
```

Recommendations

When a user is already registered, if the `referrer` parameter set by `WHydra.startMint` is different from `userInfo.onboardedBy`, revert the transaction to allow the user to recognize that they have been attacked. The user can then take countermeasures such as using a different account or requesting the administrator to change their referral.

[H-06] Anyone can set the referral of top-level users or partners

Severity

Impact: Medium

Likelihood: High

Description

While the WHydra minting is still open (when `closedMint` is `false`), users can buy tokens without a referral. Users who minted during the minting period are top-level nodes, so their `onboardedBy` should normally be empty. Also, since partners are set through `setPartnerAllocation`, their `onboardedBy` can be empty.

In `Referral.addCommission`, when a user makes their first purchase, the user's information is initialized and `onboardedBy` is registered. However, if `onboardedBy` is empty, it can be set later. In other words, by calling `Referral.addCommission` again, `onboardedBy` can be reinitialized.

```
  function _getOrAddInfo(Set storage set, address user) private returns
(Info storage info) {
    uint256 position = set._users[user];

    if (position == 0) {
```

```
      ShortString id = _generateReferrerId(set, user);

      // Push new dataset directly to storage
      set._referrals.push();

      // Initialise
      Info storage newInfo = set._referrals[set._referrals.length - 1];
      newInfo.id = id;
      newInfo.addy = user;
@>    newInfo.onboardedBy = NULL_STRING;

      // Use position 0 to indicate `not in set`
      position = set._referrals.length;
      set._users[user] = position;
      set._ids[id] = position;

      emit NewUser(id, user);
    }

    return set._referrals[position - 1];
  }

  function addCommission(
    Set storage set,
    address caller,
    address asset,
    uint256 amount,
    address user,
    string memory referrer
  ) internal returns (uint256 total) {
    ShortString referrerId = referrer.toShortString();

@>  Info storage userInfo = _getOrAddInfo(set, user);

    // Onboard user only once (do not allow user to modify tree-position)
@>  if (isValidId(set, referrerId) &&
ShortString.unwrap(userInfo.onboardedBy) ==
ShortString.unwrap(NULL_STRING)) {
@>    userInfo.onboardedBy = referrerId;
      set._tree[userInfo.id] = referrerId;

      emit OnboardedBy(userInfo.id, referrerId);
    }

    // Call helper function with applyCommission = true to modify storage
    return _calculateAndApplyCommission(set, caller, asset, amount,
userInfo.id, userInfo.onboardedBy);
  }
```

The problems that can arise from this are as follows.

1. An attacker can forcibly designate them as the `onboardedBy` of a top-level user or partner. Since there is no restriction on setting `param.to` in the `Referral.addCommission` function, an attacker can forcibly buy tokens for a top-level user. So they can set the top-level user's `onboardedBy` to the attacker through the `referrer` parameter. This allows the attacker to receive commissions from the top-level user or their child users.
2. A top-level user or partner can register themselves in their own `onboardedBy`. By registering themselves as their own referral, they can receive additional commissions that they shouldn't normally receive.
3. When `closedMint` is `true`, a valid `referrer` parameter must be provided to issue tokens. If a top-level user wants to issue additional tokens after the minting period has ended, they will no longer be a top-level user unless they register themselves as their own referral.
4. The top-level user and partner's referral can be set at a later time, which also introduces the problem of creating loops in the referral tree. If a referral is looped from user C → B → A → C, then when user B buys tokens and level commissions are 3%, 2%, and 1%, A will receive 3%, C will receive 2%, and B will receive 1% as commission.

Recommendations

Allow `onboardedBy` to be set only when an account is first created. In `Referral.addCommission`, instead of determining if it's a newly registered account by checking if `onboardedBy` is `NULL_STRING`, return whether the account was created from the return value of `_getOrAddInfo` and use that for determination.

[H-07] Claim commission can be stolen by setting `token.mint.referrer` to a different account than `onboardedBy`

Severity

Impact: Medium

Likelihood: High

Description

When calling `WHydra.startMint` to issue tokens, `params.referrer` is set as `token.mint.referrer`. `token.mint.referrer` is a referral who gets the claim commission.

```
  function startMint(StartMintParams calldata params) external payable
returns (Token memory) {
    ...

    if (params.payWithETH) {
      ...

      // Calculate commission
@>    commission = _payCommission(address(0), result.spent, params.to,
```

```
params.referrer);

      // Refund remaining ETH
      uint256 remaining = msg.value - commission - result.spent;
      if (remaining > 0) {
        Address.sendValue(payable(msg.sender), remaining);
      }
    } else {
      ...
      }
    }

    // Start the mint
    Token storage token = _startMint(params.to, params.mintPower,
params.numOfDays);

    // Update token details
@>  token.mint.referrer = params.referrer.toShortString();
    token.doRefund = params.doRefund;

    token.mintCommission.asset = params.payWithETH ? Constants.WETH :
TITANX;
    token.mintCommission.amount = commission;

    token.mintPayment.asset = params.payWithETH ? Constants.WETH : TITANX;
    token.mintPayment.amount = payment;

    return token;
  }

  function claimMint(ClaimMintParams calldata params) external {
    ...

    // Pay claim commission
    token.claimCommission.asset = HYDRA;
@>  token.claimCommission.amount = _payCommission(address(hydra), minted,
nftOwner, token.mint.referrer.toString());

    ...
  }
```

However, if the user buying the token is not a new user, the minting commission is given to `userInfo.onboardedBy` instead of `param.referrer`. This means that the referral receiving the minting commission and the referral receiving the claim commission can be set differently.

```
  function addCommission(
    Set storage set,
    address caller,
    address asset,
    uint256 amount,
    address user,
```

```
@>  string memory referrer
  ) internal returns (uint256 total) {
    ShortString referrerId = referrer.toShortString();

    Info storage userInfo = _getOrAddInfo(set, user);

    // Onboard user only once (do not allow user to modify tree-position)
    if (isValidId(set, referrerId) &&
ShortString.unwrap(userInfo.onboardedBy) ==
ShortString.unwrap(NULL_STRING)) {
      userInfo.onboardedBy = referrerId;
      set._tree[userInfo.id] = referrerId;

      emit OnboardedBy(userInfo.id, referrerId);
    }

    // Call helper function with applyCommission = true to modify storage
@>  return _calculateAndApplyCommission(set, caller, asset, amount,
userInfo.id, userInfo.onboardedBy);
  }
```

It's more accurate to set the commission receiver based on `userInfo.onboardedBy`, because a user can get the claim commission back if the a user sets `params.referrer` differently.

Recommendations

When setting `token.mint.referrer`, use `userInfo.onboardedBy` retrieved through `PowerX.getView`.

[M-01] When the owner of WHydra NFT changes, claim commission can not be transferred to the WHydra referrer

Severity

Impact: Medium

Likelihood: Medium

Description

WHydra owners can call `claimMint` to mint Hydra. The claim commission should go to the `token.mint.referrer` set at the time of minting WHydra.

```
  function claimMint(ClaimMintParams calldata params) external {
    ...

    // Pay claim commission
```

```
    token.claimCommission.asset = HYDRA;
@>  token.claimCommission.amount = _payCommission(address(hydra), minted,
nftOwner, token.mint.referrer.toString());

    ...
  }
```

Since WHydra is an NFT, it can be freely transferred. Even if WHydra is transferred, `token.mint.referrer` does not change, but because the owner of the NFT has changed, `userInfo.onboardedBy` in PowerX changes.

Commission distribution occurs in `Referral.addCommission`, and if the user's `userInfo.onboardedBy` is set, it uses `userInfo.onboardedBy` instead of the `referrer` received as a parameter. In other words, instead of `token.mint.referrer` set at the time of minting WHydra, the commission is transferred to the `userInfo.onboardedBy` of the new NFT owner.

```
  function addCommission(
    Set storage set,
    address caller,
    address asset,
    uint256 amount,
    address user,
@>  string memory referrer
  ) internal returns (uint256 total) {
    ShortString referrerId = referrer.toShortString();

    Info storage userInfo = _getOrAddInfo(set, user);

    // Onboard user only once (do not allow user to modify tree-position)
@>  if (isValidId(set, referrerId) &&
ShortString.unwrap(userInfo.onboardedBy) ==
ShortString.unwrap(NULL_STRING)) {
      userInfo.onboardedBy = referrerId;
      set._tree[userInfo.id] = referrerId;

      emit OnboardedBy(userInfo.id, referrerId);
    }

    // Call helper function with applyCommission = true to modify storage
@>  return _calculateAndApplyCommission(set, caller, asset, amount,
userInfo.id, userInfo.onboardedBy);
  }
```

Setting a separate `token.mint.referrer` for WHydra seems to be intended to give commission to the referral set at the time of minting, even if the NFT has been transferred. Therefore, the commission should be given to `token.mint.referrer` instead of `userInfo.onboardedBy`.

Recommendations

When distributing commission in `Referral.addCommission`, transfer the commission to the `referrer` received as a parameter. If this conflicts with the existing implementation, another approach could be to add a `useParam` parameter to `Referral.addCommission` and only ignore `userInfo.onboardedBy` when this parameter is `true`.

[M-02] `earlyEndMint` sends TitanX to wrong proxy, making early end impossible

Severity

Impact: Medium

Likelihood: Medium

Description

The early end mint feature allows users to mint Hydra earlier by paying an additional 50% of the cost. To do this, users need to deposit TitanX tokens to the old proxy used when calling `startMint`, and then call `proxy.earlyEndMint`.

However, in the `earlyEndMint` function, TitanX is sent to the `activeProxy`. Since the proxy changes after creating 1000 startMint requests, the `activeProxy` may not be the old proxy. As TitanX is not transferred to the old proxy, `proxy.earlyEndMint` will fail.

```
    function earlyEndMint(EarlyEndParams calldata params) external payable {
      ...

      // Handle payment
      if (params.payWithETH) {
        // Swap ETH to TitanX
        Swap.SwapResult memory result = Swap.buy(
          Swap.SwapParams({
            path: BUY_TITANX_PATH,
            slippage: params.slippage,
            twap: params.twap,
            amount: earlyEndCost,
@>          recipient: address(activeProxy),
            deadline: params.deadline
          }),
          true
        );

        // Track spendings in WETH
        token.earlyEndPayment.asset = Constants.WETH;
        token.earlyEndPayment.amount = result.spent;

        // Refund any remaining ETH
        uint256 remaining = msg.value - result.spent;
        if (remaining > 0) {
```

```
          Address.sendValue(payable(msg.sender), remaining);
        }
      } else {
        // Transfer TitanX for early-end cost
@>      titanX.safeTransferFrom(msg.sender, address(activeProxy),
earlyEndCost);

        // Track spendings in TitanX
        token.earlyEndPayment.asset = TITANX;
        token.earlyEndPayment.amount = earlyEndCost;
      }

      // Execute early-end on proxy
@> PHydra proxy = PHydra(token.mint.proxy);
@>  uint256 minted = proxy.earlyEndMint(token.mint.id, address(this));

      ...
    }
```

Recommendations

Send TitanX to `token.mint.proxy` instead of `activeProxy`.

[M-03] `onboardedBy` is not separated by project, so referral commissions are paid even during the minting period of other projects in the future

Severity

Impact: Medium

Likelihood: Medium

Description

WHydra is one of the projects connected to PowerX, and in the future, multiple projects similar to WHydra will be integrated with PowerX. For projects similar to WHydra, it is expected that only partner commissions will be received during the minting period.

However, PowerX manages user referrals globally rather than by project. Therefore, once a user has set a referral, they must pay referral commissions even when buying tokens during the minting period of other projects.

```
    function addCommission(
      Set storage set,
      address caller,
      address asset,
```

```
    uint256 amount,
    address user,
    string memory referrer
  ) internal returns (uint256 total) {
    ShortString referrerId = referrer.toShortString();

    Info storage userInfo = _getOrAddInfo(set, user);

    // Onboard user only once (do not allow user to modify tree-position)
    if (isValidId(set, referrerId) &&
ShortString.unwrap(userInfo.onboardedBy) ==
ShortString.unwrap(NULL_STRING)) {
      userInfo.onboardedBy = referrerId;
      set._tree[userInfo.id] = referrerId;

      emit OnboardedBy(userInfo.id, referrerId);
    }

    // Call helper function with applyCommission = true to modify storage
@>  return _calculateAndApplyCommission(set, caller, asset, amount,
userInfo.id, userInfo.onboardedBy);
  }
```

Recommendations

Manage referrals on a per-project basis. Alternatively, add a function to pay partner commissions but not level commissions to PowerX so that only partner commissions are paid during the minting period.

[M-04] Refund function may fail or refund incorrect amounts

Severity

Impact: Medium

Likelihood: Medium

Description

In the`Swap.refund` function, when the contract holds less tokens than the maximum required amount, it only uses the available tokens. `Swap.refund`'s `params.amount` represents the amount of tokens desired to receive at the end. For WHydra, this means the amount of WETH you want to receive.

The `estimateMinimumOutputAmount` function calculates the minimum output token amount that can be received for a given input token amount. This function expects `params.amount` to be the input token amount. However, when calling `estimateMinimumOutputAmount`, it uses the same `params` received at the `Swap.refund`, so `params.amount` actually represents the output token amount.

Consequently, when `estimateMinimumOutputAmount` is called with incorrect token amounts, it will set incorrect `amountOutMinimum` parameters when requesting swap to UniswapV3, which may cause the swap to fail, or swap an incorrect amount of tokens for the refund.

```
  function refund(SwapParams memory params) external returns (SwapResult
memory result) {
    PathDecoder.Hop[] memory hops = PathDecoder.decode(params.path);
    if (hops[hops.length - 1].tokenOut != Constants.WETH) revert
InvalidSwapPath();

    IERC20 paymentToken = IERC20(hops[0].tokenIn);
    uint256 amountInMax = estimateMaximumInputAmount(params);
    uint256 balance = paymentToken.balanceOf(address(this));

    bytes[] memory inputs = new bytes[](2);
    inputs[1] = abi.encode(MSG_SENDER, 0);

    uint256 ethBefore = address(this).balance;
    bool isExactInput = balance < amountInMax;

    _approveForSwap(paymentToken, isExactInput ? balance : amountInMax);
    inputs[0] = isExactInput
@>    ? abi.encode(ADDRESS_THIS, balance,
estimateMinimumOutputAmount(params), params.path, SOURCE_MSG_SENDER)
      : abi.encode(ADDRESS_THIS, params.amount, amountInMax,
PathDecoder.encodeReverse(hops), SOURCE_MSG_SENDER);

    IUniversalRouter(Constants.UNIVERSAL_ROUTER).execute(
      abi.encodePacked(isExactInput ? SWAP_EXACT_IN : SWAP_EXACT_OUT,
UNWRAP_WETH),
      inputs,
      params.deadline
    );

    result.spent = isExactInput ? balance : balance -
paymentToken.balanceOf(address(this));
    result.received = address(this).balance - ethBefore;
    Address.sendValue(payable(params.recipient), result.received);
  }

  function estimateMinimumOutputAmount(SwapParams memory params) public
view returns (uint256) {
    PathDecoder.Hop[] memory hops = PathDecoder.decode(params.path);
@>  uint256 amount = params.amount;

    for (uint256 i = 0; i < hops.length; i++) {
@>    (amount, ) = getQuote(hops[i].tokenIn, hops[i].tokenOut,
hops[i].fee, params.twap, amount);
    }

    return (amount * (Constants.BASIS - params.slippage)) /
```

```
Constants.BASIS;
  }
```

Recommendations

When calling `estimateMinimumOutputAmount`, `params.amount` should be changed to `paymentToken.balanceOf(address(this))`.

[L-01] Total Commission Rate Should Not Exceed 100%

Severity

Impact: Medium

Likelihood: Low

Description

The sum of all commission rates, including partners and referrals, should be enforced to not exceed 100%. During `startMint`, since users are charged the minting cost and commission, theoretically it won't fail even if commission rates exceed 100%. However, in `claimMint`, since it receives a fixed amount of HydraX and distributes it according to each party's ratio, if the sum of commission rates is set to exceed 100%, it will always fail due to insufficient HydraX.

Recommendations

When updating partner allocations or referral commission rates, ensure the sum of all commission rates does not exceed 100%.

[I-01] Check `mintPower` limit first

Description

When executing `Hydra.startMint`, the transaction is reverted if `mintPower` exceeds `MAX_MINT_POWER_CAP`. By checking this first in `WHydra.startMint`, we can revert the transaction earlier, thereby reducing gas costs when incorrect `mintPower` values are passed.

Recommendations

Revert in `WHydra.startMint` when `mintPower > MAX_MINT_POWER_CAP`.

[I-02] Users can use WHydra for fraud because claimed WHydra tokens can still be transferred

Description

WHydra NFTs are no longer valuable once Hydra is minted by calling `claimMint`. Since the WHydra tokens is neither blocked nor burned after claim, they could be used for fraud.

Consider a scenario where an external user attempts to purchase an unclaimed(valuable) WHydra from an NFT marketplace. If the WHydra owner(attacker) front-runs this by calling `claimMint`, the buyer ends up purchasing a WHydra with no value.

Recommendations

This can be prevented by blocking the transfer of WHydra that no longer has value, or by burning it when claimed.