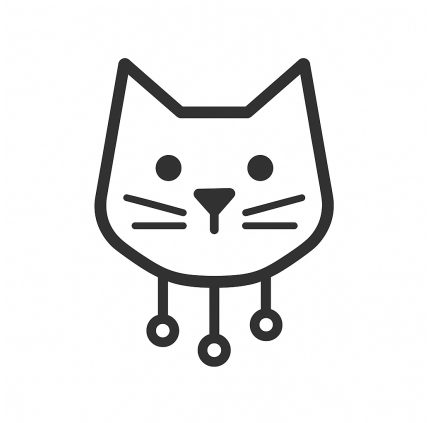
Bify Audit Report

Radcipher

July 19, 2025



Bify Audit Report

Version 1.0

Radcipher

Audited by: saikumar279,ParthMandale,0zKillua,frescoio,ghodarod95

July 16th 2026 - July 19th

July 19th, 2025

Bify Audit Report

Radcipher

July 19, 2025

Prepared by: [Radcipher]

Table of Contents

Protocol Summary

BIFY is an AI-powered marketplace operating through BIFY.IO, designed to integrate NFTs and tokenized real-world assets into a secure, intelligent, and efficient trading ecosystem. By leveraging predictive analytics, automation, and on-chain verification, BIFY enhances asset curation, investment decisions, and accessibility for users.

Disclaimer

Radcipher team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Likelihood	Impact		
	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

- Review commit hash : 99f5e5d6e2000220b528c97cbe65769955a7d687

Scope

```
./bify-contract/

- BifyLaunchpad.sol
- BifyLaunchpadCore.sol
- BifyLaunchpadLibrary.sol
- BifyLaunchpadPhase.sol
- BifyLaunchpadPhaseCore.sol
- BifyLaunchpadQuery.sol
- BifyLaunchpadQueryBase.sol
- BifyMarketplace.sol
- BifyNFT.sol
- BifyTokenPayment.sol
- IWhitelistManagerExtended.sol
- MarketplaceQuery.sol
- MarketplaceQueryLibrary.sol
- NFTCollection.sol
- WhitelistManagerExtended.sol
- WhitelistManagerFactory.sol

./bify-contract-storage/

- BifyLaunchpadStorage.sol

./bify-contract-libraries/

- BifyCollectionFactory.sol
- BifyCollectionRegistry.sol
- BifyPaymentManager.sol
- BifyQueryLibrary.sol
- BifyWhitelistManager.sol
- MarketplaceValidation.sol
```

Executive Summary

Radcipher carried out a security assessment of the bify-contract repository. Our review uncovered 13 high-risk, 11 medium-risk, and 4 low-risk vulnerabilities. We strongly advise addressing these findings prior to any mainnet deployment. Given the number and nature of the issues - and the likelihood that further fixes or adjustments will arise during remediation — we highly recommend scheduling a follow-up audit after the fixes are applied to ensure the protocol's security across all stages.

Issues found

Severity	Number of issues found
High	13
Medium	11
Low	4
Info	1
Gas	1
Total	30

About

Radcipher is a comprehensive Web3 security firm made up of passionate smart contract researchers, tokenomics experts, and professionals deeply versed in the logic and economic principles behind blockchain protocols. Beyond identifying and addressing critical vulnerabilities in DeFi and NFT projects, we bring a holistic approach combining technical rigor with economic insight. Our team is dedicated to delivering thorough, high-quality security audits that not only focus on bugs but also on the soundness of the underlying protocol logic and tokenomics design. While absolute security can never be guaranteed, we commit to applying our full expertise and meticulous attention to detail to help protect and strengthen your project's foundation. Discover more about our work Radcipher website or connect with us on X @Radcipher.

Disclaimer

No smart contract security review can ensure the complete elimination of vulnerabilities. These reviews are limited by time, resources, and expertise, aiming to identify as many issues as feasible within those constraints. We cannot promise total security following the audit, nor can we guarantee that all problems will be uncovered. We strongly advise implementing ongoing security measures such as additional audits, bug bounty programs, and live on-chain monitoring.

Introduction

Radcipher performed smart contract security audit of the **bify-contract** , concentrating on the security aspects of its smart contract implementation.

About Bify

The platform enhances curation, investment decisions, and accessibility through predictive analytics, automation, and on-chain ownership verification. With the NFT market projected to surpass $230 billion and real-world assets (RWAs) expected to exceed $16 trillion by 2030, BIFY taps into this massive opportunity by offering a seamless, AI-driven marketplace for both digital and physical assets. Focused on innovation,

security, and transparency, BIFY bridges the gap between digital assets, RWAs, and the broader crypto ecosystem—serving artists, collectors, and investors alike.

Risk Classification

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

Impact Definitions

- **High** – Results in significant material loss to the protocol or causes substantial harm to a large group of users.
- **Medium** – Causes moderate material loss to the protocol or moderately affects a group of users.
- **Low** – Leads to minor material loss or impacts a small number of users.

Recommended Actions Based on Severity

- **High** – Must be resolved prior to deployment.
- **Medium** – Recommended to fix.
- **Low / Informational** – Optional fix.

Findings

Disclamer

All reported vulnerabilities were promptly acknowledged and fixed by the team after disclosure.

[H-1] Failure to Register Launchpad Collections Due to Improper Ownership Handling

Description

The `BifyLaunchpad` contract delegates collection creation to the `coreContract`, which internally calls `_registerWithMarketplace()` to register the collection with the marketplace. However, the `registerLaunchpadCollection()` function in the marketplace contract is protected by the `onlyOwner` modifier. Since the `coreContract` is not the owner of the marketplace, the call always fails. Due to the use of a `try/catch` block, this failure is silently ignored, resulting in unregistered collections.

Flow: `BifyLaunchpad::createCollection`

- `coreContract.createCollectionWithCreator`
- `BifyLaunchpadCore::_registerWithMarketplace()`
- `BifyMarketplace::registerLaunchpadCollection`
- Fails due to onlyOwner — `BifyLaunchpadCore` is not the owner of `BifyMarketplace`

`BifyLaunchpad::createCollectionWithWhitelist`

- `coreContract.createCollectionWithWhitelist`

- `BifyLaunchpadCore::_registerWithMarketplace()`
- `BifyMarketplace::registerLaunchpadCollection`
- Fails due to onlyOwner — `BifyLaunchpadCore` is not the owner of `BifyMarketplace`

Impact

Collections created through the launchpad will **not be registered** in the marketplace contract. This can break fee routing, listing visibility, and other marketplace-integrated operations such as **Fee Collection for the platform**. Since the failure is silent, users and developers may incorrectly assume the collection has been registered successfully.

Recommendations

- Replace the `onlyOwner` restriction on `registerLaunchpadCollection()` with a role-based modifier (e.g., `onlyCoreContract` or `onlyOperator`).
- Alternatively, make the `coreContract` the owner of the marketplace or grant it ownership privileges.
- Log a failure event inside the `catch` block to improve observability and help identify failed registrations during audits or testing.

[H-2] Missing Marketplace Registration in BifyLaunchpad createCollectionWithWhitelistAndRevealStrategy

Description

The `BifyLaunchpad` contract defines a function
`BifyLaunchpad::createCollectionWithWhitelistAndRevealStrategy`
to deploy whitelist-enabled collections with reveal strategy support.
This function internally calls
`coreContract.registerWhitelistCollection`
to handle registration and whitelist data storage.

However, unlike `createCollection()` (which internally invokes `_registerWithMarketplace()` via the `coreContract.createCollection()` flow), this function **does not** invoke `BifyLaunchpadCore::_registerWithMarketplace()`. As a result, collections created via this flow are **not registered with the marketplace**, leading to inconsistencies in listing, fee routing, and discoverability.

Relevant function calls: - `BifyLaunchpad.createCollectionWithWhitelistAndRevealStrategy()`
→ deploys collection via `BifyCollectionFactory.createWhitelistCollection(...)`
→ calls `coreContract.registerWhitelistCollection(...)`
→ **does not** call `_registerWithMarketplace(...)`

This leads to a situation where the collection is stored in the core storage but never added to the marketplace's registry, which may affect visibility and interactions.

Impact

- Collections created via this function do **not get added** to the `launchpadCollections` registry in the marketplace.
- As a result, the marketplace does **not recognize** these collections as launchpad-originated, and any transaction involving them will **not trigger platform fee collection** logic.

Recommendations

- Put the `_registerWithMarketplace()` logic into the `coreContract.registerWhitelistCollection()` function so that **all** launchpad-originated collections (regardless of the creation path) are consistently

registered with the marketplace.

[H-3] Incorrect Whitelist Manager Ownership Assignment in WhitelistManagerFactory::createWhitelistManager

Description

The `BifyLaunchpad` contract exposes the `createCollectionWithWhitelist()` function, which in turn calls `coreContract.createCollectionWithWhitelist(...)` and passes the original sender as `_creator`. If `_enableAdvancedWhitelist` is `true`, the core contract proceeds to deploy a `WhitelistManagerExtended` via the `WhitelistManagerFactory`.

```
// Inside CoreContract.createCollectionWithWhitelist
if (_enableAdvancedWhitelist) {
    address factory = _getStorage().whitelistManagerFactory();
    require(factory != address(0), "Factory not set");
    WhitelistManagerFactory whitelistFactory = WhitelistManagerFactory(factory);
    whitelistContract = whitelistFactory.createWhitelistManager(_whitelistName);
}
```

In `WhitelistManagerFactory.createWhitelistManager`, the following logic is used:

```
WhitelistManagerExtended manager = new WhitelistManagerExtended(_name);
manager.transferOwnership(msg.sender); // msg.sender is the core contract
```

Because msg.sender in this context is the core contract, the deployed whitelist manager's ownership is set to the core contract itself, instead of the actual creator (_creator). As a result, the collection creator has no ownership privileges over the WhitelistManagerExtended contract.

This prevents the creator from using any `onlyOwner`-protected functions like `createTier`, `updateTier`, `addToDirectWhitelist`, `batchAddToDirectWhitelist`, `trackMint`, `updateName`.

Impact

- The collection creator cannot manage or update the whitelist manager.
- All onlyOwner functionalities inside the whitelist manager become inaccessible to the user, effectively locking them out of whitelist control.

Recommendations

Recommendation ->

```
-   whitelistContract =
- whitelistFactory.createWhitelistManager(_whitelistName);
+   whitelistContract =
+whitelistFactory.createWhitelistManager(_whitelistName, _creator);
```

WhitelistManagerFactory ->

```
-   function createWhitelistManager(string memory _name)
-external returns (address managerAddress) {
+   function createWhitelistManager(string memory _name, address _creator)
+external returns (address managerAddress) {
        WhitelistManagerExtended manager = new WhitelistManagerExtended(_name);

-        manager.transferOwnership(msg.sender);
```

```
+         manager.transferOwnership(_creator);
          managerAddress = address(manager);

-         creatorManagers[msg.sender].push(managerAddress);
+         creatorManagers[_creator].push(managerAddress);
      allManagers.push(managerAddress);

-         emit WhitelistManagerCreated(msg.sender, managerAddress, _name);
+         emit WhitelistManagerCreated(msg.sender, managerAddress, _name);
          return managerAddress;
      }
```

[H-4] Misaligned Ownership Access Between BifyLaunchpad and BifyLaunchpadCore Causes System-wide Reverts

Description

The `BifyLaunchpad` contract exposes multiple externally callable functions that attempt to interact with the `CoreContract` (i.e., `BifyLaunchpadCore`). However, these delegated calls consistently revert due to restrictive ownership checks on the CoreContract, where onlyOwner modifiers or hardcoded msg.sender verifications reject calls originating from BifyLaunchpad.

The impacted functions in `BifyLaunchpad` include:

- updateRoyaltyInfo(…)
- trackMint(…)
- updatePlatformFee(…)
- updateBifyFee(…)
- updateFeeRecipient(…)
- toggleBifyPayment(…)
- pause(), unpause()
- emergencyWithdraw(), emergencyWithdrawToken(…)
- setWhitelistManagerFactory(…)

Each of these functions attempts to call their corresponding counterparts in `CoreContract`:

Because `BifyLaunchpad` is not set as the owner nor granted any special role in CoreContract, all these delegated calls revert.

Impact

- Users or platform admins calling functions on BifyLaunchpad will experience consistent transaction reverts.
- Operational features like mint tracking, royalty updates, fee configuration, and emergency functions are entirely unusable through the launchpad frontend.

Recommendation

To ensure functional delegation from BifyLaunchpad to CoreContract:

- Either assign `BifyLaunchpad` as the `owner()` of `CoreContract`, or
- Implement role-based access control in CoreContract (e.g., a `hasRole(LAUNCHPAD_ROLE, msg.sender)` pattern), and grant the role to BifyLaunchpad.

[H-5] Arbitrary Royalty Overwrite During Auction Creation Undermines Creator Royalties

Description

The protocol allows creators to set a default royalty percentage when creating an NFT collection via the launchpad flow, but this royalty enforcement is bypassed during auction listing by subsequent token holders.

When a collection is created via:

`BifyLaunchpad::createCollection` → `BifyLaunchpadCore::createCollectionWithCreator` → `BifyCollectionFactory::createNFTCollection`

A `_royaltyPercentage` is passed and stored as the default royalty for the collection. This presumably reflects the creator's expected earnings from secondary sales.

However, during resale, any user (not necessarily the original creator) who owns a token can create an auction via:

`BifyMarketplace::createAuctionWithToken(...)`

This function accepts a new royaltyPercentage as a parameter: `royaltyPercentage`

Note: Above i took only `BifyMarketplace::createAuctionWithToken` this is protocol wide design flaw where listings are made through `BifyMarketplace`. There are no checks to ensure that this value honors or matches the originally defined royalty during collection creation. The only validation performed is that the new royalty lies within globally set boundaries:

```
require(
    royaltyPercentage >= minRoyaltyPercentage &&
    royaltyPercentage <= maxRoyaltyPercentage,
    "Invalid royalty percentage"
);
```

Thus, a secondary seller can undercut the original creator by specifying a lower royalty effectively overriding the intended default.

Impact

- Creator royalties are bypassed, reducing their expected income on secondary sales.
- Violates user expectations from NFT collection configuration, especially for creators relying on royalties for compensation.

Recommendation

To enforce consistent royalty logic: - During createAuctionWithToken, fetch the default royalty from the collection metadata or registry, and override any user input if the caller is not the original creator.

[H-6] Incorrect Marketplace Fee Recipient Assignment in NFT Collection Constructor

Description

In the `NFTCollection` contract constructor, the `marketplaceFeeRecipient` intended to receive platform-level marketplace fees is incorrectly assigned to the `_creator`:

```
constructor(
        string memory _name,
```

```
        string memory _symbol,
        uint256 _maxSupply,
        uint96 _royaltyPercentage,
        string memory _baseURI,
        uint256 _mintStartTime,
        uint256 _mintEndTime,
        uint256 _mintPrice,
        uint256 _maxMintsPerWallet,
        address _creator,
        RevealStrategy _revealStrategy
    ) ERC721(_name, _symbol) {
.............
        _setDefaultRoyalty(_creator, _royaltyPercentage);

        if (_creator != msg.sender) {
            transferOwnership(_creator);
        }

        marketplaceFee = 250;
@>        marketplaceFeeRecipient = _creator;

.....
    }
```

This assignment occurs when the contract is deployed via:

- BifyLaunchpad::createCollection →
- coreContract.createCollectionWithCreator →
- BifyCollectionFactory.createNFTCollection →
- NFTCollection constructor

This sets the `_creator` as both the collection owner and the `marketplaceFeeRecipient`.

However, as confirmed by the team, the correct recipient of marketplace fees should be the **Bify marketplace owner**, not the NFT collection creator.

Impact

- Marketplace revenue is completely diverted from the Bify protocol to the individual collection creator.
- All fees collected from primary minting go to the wrong party, resulting in financial losses for the platform.

Recommendations

Assign `marketplaceFeeRecipient` to the platform `(Bify) owner` instead of the `_creator`.

[H-7] emergencyWithdraw , emergencyWithdrawToken and emergencyWithdrawTokens Can Drain User Funds Mid-Auction

Description

Both the `BifyLaunchpadCore` and `BifyMarketplace` contracts implement `emergencyWithdraw` and `emergencyWithdrawToken(s)` functions that allow the contract owner to withdraw all ETH or tokens held in the contract at any time:

```
// BifyLaunchpadCore
function emergencyWithdraw() external onlyOwner { ... }
function emergencyWithdrawToken(address _token) external onlyOwner { ... }

// BifyMarketplace
function emergencyWithdraw() external onlyOwner { ... }
function emergencyWithdrawTokens() external onlyOwner { ... }
```

While intended for emergencies, these functions can be abused by a malicious or compromised owner to steal funds deposited by users including active auction bid amounts or sale proceeds.

In the context of `BifyMarketplace`, ETH from bids and NFTs are held in escrow. The owner can call emergencyWithdraw() at any time, draining all ETH and token balances, even while auctions are still active, with no restrictions or time locks.

Impact

- High severity due to potential loss of all user funds.
- Platform becomes unsafe and untrustworthy: users may lose NFTs or ETH mid-auction.

Recommendations

- Restrict usage of emergencyWithdraw and emergencyWithdrawToken(s) functions:
  - Add whenPaused modifier or require protocol to be in emergency mode.
  - Implement a time delay before execution (e.g., 24–48 hours).
- Use a MultiSig Wallet for these emergency actions to avoid single point of failure or abuse.

[H-8] First Lister Treated as NFT Creator Incorrect Royalty Attribution

Description

The current marketplace logic incorrectly assigns the royalty recipient by treating the first person who lists an NFT on the platform as its creator.

In the `BifyMarketplace::createAuction` function of `BifyMarketplace`, the following line assigns the NFT's "creator":

```
function createAuction(
        address nftContract,
        uint256 tokenId,
        uint256 reservePrice,
        uint256 buyNowPrice,
        uint256 startTime,
        uint256 duration,
        AssetType assetType,
        uint256 royaltyPercentage,
        bytes32 category
    ) external whenNotPaused nonReentrant returns (uint256 auctionId) {
.....................................
        IERC721(nftContract).transferFrom(msg.sender, address(this), tokenId);

        if (tokenCreators[nftContract][tokenId] == address(0)) {
            tokenCreators[nftContract][tokenId] = msg.sender;
        }
.....................................
```

In the `BifyMarketplace::createAuctionWithToken` function of `BifyMarketplace`, the following line assigns the NFT's "creator":

```
function createAuctionWithToken(
        address nftContract,
        uint256 tokenId,
        uint256 reservePrice,
        uint256 buyNowPrice,
        uint256 startTime,
        uint256 duration,
        AssetType assetType,
        uint256 royaltyPercentage,
        bytes32 category
    ) external whenNotPaused nonReentrant returns (uint256 auctionId) {
.....................................

        IERC721(nftContract).transferFrom(msg.sender, address(this), tokenId);

        if (tokenCreators[nftContract][tokenId] == address(0)) {
            tokenCreators[nftContract][tokenId] = msg.sender;
        }
.....................................
```

This logic permanently stores the first seller on the marketplace as the creator for that specific NFT. All future sales will distribute royalties to this address even if they are not the actual creator of the collection.

Real-world Scenario:

1. John creates the collection
2. Alice mints the nft from collection oreither buys it from somewhere and lists an NFT on the marketplace and becomes the recorded "creator".
3. Bob buys it and later re-auctions it.
4. Royalties from Bob's future sale go to Alice, even though Alice may have just resold it and didn't originally is the creator.
5. Worse, Alice could have bought this NFT elsewhere (like OpenSea), and upon listing it here, permanently hijacks all royalties.

Impact

- Royalties are paid to the wrong person (potentially maliciously).
- True NFT creators may never receive royalties.

Recommendation

Instead of assigning the creator as the first person to list or auction the NFT,Fetch the actual creator of the NFT collection and assign that address as the creator.

[H-9] Malicious NFT creator can change the marketplaceFeeRecipient address to its own address

Description

The `setMarketplaceFeeRecipient` function in the `NFTCollection` contract allows the collection owner (NFT creator) to change the marketplace fee recipient address at any time after deployment. This creates a vulnerability where a malicious creator can redirect all marketplace fees to their own address. The

function only validates that the new recipient is not the zero address and has restrictions of `onlyOwner` which in this case is the creator himself, making it possible to execute this vulnerability.

`NFTCollection` contract ->

```
    function setMarketplaceFeeRecipient(address _feeRecipient) external onlyOwner {
        require(_feeRecipient != address(0), "Invalid fee recipient");
        marketplaceFeeRecipient = _feeRecipient;
    }
```

Impact

Loss of Funds to `marketplaceFeeRecipient` operators as they lose expected fee revenue when creators redirect fees to themselves.

Recommendation

Remove Dynamic Fee Recipient Updates: Eliminate the setMarketplaceFeeRecipient function entirely, the one set in the constructor should be immutable.

[H-10] Malicious NFT Creator Can Eliminate Marketplace Fees

Description

The `setMarketplaceFee` function in the NFTCollection contract allows the collection owner (NFT creator) to modify the marketplace fee percentage at any time, including setting it to 0%. While the function includes an upper bound check (maximum 10%), it lacks a minimum fee requirement (which is 2.5% in constructor). This enables malicious creators to reduce the fee to 0% to capture the entire sale proceeds that should be shared with the marketplace platform.

```
    function setMarketplaceFee(uint256 _fee) external onlyOwner {
        require(_fee <= 1000, "Fee cannot exceed 10%");
        marketplaceFee = _fee;
    }
```

Impact

Marketplace platforms lose all expected fee revenue when creators set fees to 0% and therefore at the time of withdraw user will get 100% back without any fees being subtracted.

Recommendations

Implement Minimum Fee Threshold: Add a lower bound check (e.g., require(_fee >= 250, “Fee cannot be less than 2.5%”)) to ensure the marketplace always receives minimum compensation.

[H-11] Incorrect Ownership Check Renders Phase Management Functions Unusable

Description

The `BifyLaunchpad.sol` contract calls for phase management to a `BifyLaunchpadPhase.sol` contract. However, the target functions in the Phase contract, such as `disableWhitelistPhase`, `disablePublicPhase`, and `configureAdvancedWhitelist`, are protected by an onlyOwner modifier.

When `BifyLaunchpad.sol` calls these functions, the `msg.sender` within the Phase contract is the `BifyLaunchpad.sol` contract's address, which is not the owner. This mismatch causes all calls to these critical functions to revert.

Impact

Denial of Service to `storageContract.isCollectionCreator(_collection, msg.sender` ie. the creator of the NFT will himself not be able to interact with these functions.

Recommendation

Make an Authorized modifier in `BifyLaunchpadPhase.sol` and make `BifyLaunchpad.sol` contract to be authorized and later implement that authorized modifier on function `disableWhitelistPhase`, `disablePublicPhase`, and `configureAdvancedWhitelist`. Remove the onlyOwner Modifier.

[H-12] Flawed Authorization in trackMint Function Prevents Mint Tracking

Description

The mint tracking mechanism is broken due to a flawed authorization model across `BifyLaunchpad.sol` and `BifyLaunchpadCore.sol`

The trackMint function in `BifyLaunchpad.sol` calls `coreContract.trackMint`. This causes the `msg.sender` in the `core` contract to be the `BifyLaunchpad` contract itself, which is neither the collection creator nor the owner, leading to an inevitable revert.

```
function trackMint(
    address _collection,
    address _minter,
    uint256 _quantity
) external {
    // @audit BUG: This check will always fail
    require(msg.sender == _collection || msg.sender == owner(), "Unauthorized");
    _getStorage().incrementTotalMints(_collection, _quantity);
    _getStorage().incrementUserMints(_collection, _minter, _quantity);
}
```

Impact

Denial of Service to the creator of NFT, as the `trackMint` functionality is completely unusable, as all calls will revert.

Recommendation

The `core` contract trackMint function needs to give authorization to the `BifyLaunchpad` contract to be able to call this function.

[H-13] Royalty Fee Cap Can Be Bypassed After Collection Creation

Description

The `updateRoyaltyInfo` function in `BifyLaunchpadCore.sol` allows a collection creator to update the royalty percentage after the collection has been created. However, this function fails to enforce the 10%

(1000 basis points) maximum royalty fee that is checked during the initial collection creation via `validateCollectionParams`. This omission creates a loophole where a creator can deploy a collection with a compliant royalty fee and then later update it to an exorbitant percentage, bypassing the platform's intended limits.

```
function updateRoyaltyInfo(
    address _collection,
    uint96 _royaltyPercentage
) external {
    // ... authorization checks ...

    // @audit BUG: Missing check for _royaltyPercentage <= 1000
    // This allows creators to set royalties > 10%
    NFTCollection(payable(_collection)).setDefaultRoyalty(
        creator,
        _royaltyPercentage
    );
}
```

Impact

Creators can set excessively high royalty fees (e.g. 100%), which is unfair to secondary market buyers and sellers who may not be aware of the change.

Recommendation:

Enforce Royalty Cap on Updates:

```
function updateRoyaltyInfo(
    address _collection,
    uint96 _royaltyPercentage
) external {
    // ... authorization checks ...

    // Add the missing validation
    require(_royaltyPercentage <= 1000, "Royalty cannot exceed 10%");

    NFTCollection(payable(_collection)).setDefaultRoyalty(
        creator,
        _royaltyPercentage
    );
}
```

[M-1] Overpayment Not Refunded in NFTCollection::mint() and NFTCollection::whitelistMint()

Description

In the `NFTCollection` contract, the `mint()` and `whitelistMint()` functions fail to refund excess ETH if the caller sends more than the required minting cost (`msg.value > price * quantity`).

Affected functions:

- mint(uint256 _quantity)
- whitelistMint(uint256 _quantity, bytes32[] calldata _merkleProof, uint256 _tierId)

In both functions:

- The check ensures msg.value >= price * quantity
- However, no logic is present to return the overpaid ETH back to the user

This leads to user-side overpayment without recourse. Even though the functions don't revert, this omission may result in unintended user losses and poor UX.

Impact

- User funds may be lost permanently if they accidentally overpay

Recommendation

Add refund logic to return excess tokens to the user in both `NFTCollection::mint()` and `NFTCollection::whitelistMint()` functions.

[M-2] Misleading BifyMarketplace::setMaxBid Function Without Actual Auto-Bidding Logic

Description

The `setMaxBid()` function in the `BifyMarketplace` contract allows users to register a `maxBidAmount` for an auction, which suggests support for automatic bidding (i.e., the system will bid on their behalf up to the max). However:

- The contract does not use this value anywhere else.
- No tokens is escrowed or reserved when setting the max bid.
- There is no logic to auto-bid when a user is outbid.
- The event emitted may give users a false sense of active participation in the auction.

This could lead to **user confusion** or **missed auction opportunities**, where users think their bid will auto-adjust but no actual bids are made.

Impact

- Users may incorrectly assume they are participating in the auction and lose out due to no bids being placed.

Recommendations

- Implement auto-bidding logic.
- Or Remove the setMaxBid function entirely.

[M-3] Reusable Signature in mintNFT() Allows NFT Spam to Creator

Description

The `mintNFT()` function in the `BifyNFT` contract relies on a signature from a trusted backend (backendSigner) to authorize NFT creation. However, the signature is generated over static data:

```
bytes32 messageHash = keccak256(
    abi.encodePacked(
        uri,
```

```
            creator,
            royaltyPercentage,
            uint8(assetType),
            category,
            block.chainid
        )
    );
```

This hash does not include any user-specific or time-sensitive data such as the caller's address or a deadline. As a result, anyone not just the intended recipient can reuse a valid signature to mint the same NFT to the same creator address repeatedly.

This enables infinite NFT minting spam to the creator, degrading the platform's integrity and user experience. An attacker could mint hundreds of NFTs without any restriction as long as the signature remains valid.

Impact

- Due to abuse potential: attackers can mint unlimited NFTs with the same URI and metadata to the same creator.
- Trust issue for creators receiving NFTs they didn't authorize.

Recommendations

To prevent signature reuse and unauthorized minting: - Include the `msg.sender` (caller) and a `deadline` in the messageHash

[M-4] Missing Minimum Bid Increment Enables Micro-Bidding & Auction Griefing

Description

The auction logic in the `BifyMarketplace` contract does not enforce a minimum bid increment. The only requirement is that each new bid be strictly greater than the current highest bid. This means even a 1 wei increase is sufficient to become the new highest bidder.

- This can be used to force bidders to bid more.
- Additionally, since the auction is extended by 10 minutes for any bid placed within the last 10 minutes, a malicious actor or bot can repeatedly place 1 wei bids in rapid succession, causing auctions to be extended indefinitely potentially for days.

These behaviors lead to unfair bidding mechanics and can be systematically abused by bots to disrupt auctions and frustrate legitimate participants.

Affected Functions:

- `BifyMarketplace::placeBid`
- `BifyMarketplace::placeBidWithToken`

Impact

- Enables auction denial-of-service via micro-bid spam and indefinite extension.
- Frustrates genuine bidders, potentially driving them away due to prolonged auctions and unnecessary spend.

Recommendations

- Introduce a minimum bid increment rule—either a percentage or flat value.

```
require(bidAmount >= auction.highestBid + MIN_BID_INCREMENT, "Bid increment too low");
```

- Apply this logic in both placeBid and placeBidWithToken functions to ensure fair competition and prevent griefing abuse.

[M-5] NFTCollection::reveal Can Be Called Prematurely or Missed for Certain Strategies Inconsistent Reveal Logic

Description

The `NFTCollection` contract defines a flexible reveal system via the `RevealStrategy` enum:

```
enum RevealStrategy {
    STANDARD,
    BLIND_BOX,
    RANDOMIZED,
    DYNAMIC
}
```

However, the current implementation of the `NFTCollection::reveal` function is overly permissive and can lead to logic inconsistencies or unintended reveal states, depending on the strategy:

```
function reveal() external onlyOwner {
    require(!revealed, "Already revealed");
    revealed = true;
    emit RevealStatusChanged(true);
}
```

RANDOMIZED Strategy - Missing Seed Check

- If strategy is `RANDOMIZED` and `reveal()` is called before the seed is set, the NFTs will appear as revealed (revealed = true) even though token IDs are not properly shuffled via setRandomSeed().
- This breaks metadata consistency and defeats the purpose of random reveal logic.

Impact

- Incorrect metadata reveal behavior for randomized NFTs (could allow early reveal or break reveal fairness).

Recommendations

- Restrict `reveal()` when strategy is `RANDOMIZED`.

[M-6] DirectWhitelist users can mint within tier after tier is expired

Description

In `WhitelistManagerExtended` the creator is able to create tiers for users to mint within, but each tier has a time window. There is inconsistency in the enforcement of this time window - users that are directly whitelisted via directWhitelist mapping will be able to bypass this tier time window and mint within a tier even after the time window has expired. In `isWhitelisted` there are 2 whitelist user options, thus 2 seperate blocks of logic. One for if the user is in the `directWhtielist` mapping and the other for when they are not. Regardless of the user being in the `directWhitelist` or not, the same tier is where the mint is taking place. Thus, the same tier time window restrictions should be checked - but the time window is not checked only for the directWhitelist users :

```
function isWhitelisted(
    address _user,
    uint256 _tierId,
    bytes32[] calldata _merkleProof
) external view returns (bool) {

    // @audit missing time enforcement for direct whitelists,
    // should have it. `validateMintEligibility` has it
    if (directWhitelist[_user] != TierLevel.Invalid) {
        uint256 userTierLevel = uint256(directWhitelist[_user]);

        return userTierLevel >= _tierId + 1;
    }

    if (_tierId < tierCount) {
        Tier storage tier = tiers[_tierId];
        if (
            tier.active &&
            block.timestamp >= tier.startTime &&
            block.timestamp <= tier.endTime
        ) {
            bytes32 leaf = keccak256(abi.encodePacked(_user, _tierId));
            return MerkleProof.verify(_merkleProof, tier.merkleRoot, leaf);
        }
    }

    return false;
}
```

This can be verified as a mistake as the function validateMinteligibility which is responsible for determining if a user is eligible for a whitelist mint, enforces the time window of the tier for directWhitelist users as well - and only returns true that they are eligible for a mint, not only if they are directWhitelisted but also, if the tier they are whitelisted for is still within the time window.

```
function validateMintEligibility(
    address _user,
    uint256 _tierId,
    uint256 _quantity,
    bytes32[] calldata _proof
) external view returns (bool canMint, uint256 price) {
    require(_tierId < tierCount, "Invalid tier ID");

    Tier storage tier = tiers[_tierId];

    (bool isActive, ) = isTierActive(_tierId);
    if (!isActive) {
        return (false, 0);
    }

    bool isUserWhitelisted = false;

    TierLevel directTier = directWhitelist[_user];
    if (directTier != TierLevel.Invalid) {
        isUserWhitelisted = uint256(directTier) >= _tierId + 1;
    }
```

The check is done REGARDLESS of the user being in the `directWhitelist` or not, and is the primary check for the tier - as the function will return false that a user is not eligible for a mint in that tier if the tier time window has passed. The check for the time window of the tier is done in isTierActive. Take note that the same _tierId is used for later determining the directWhitelist status of the user

Impact

Unfair advantage to users who are in the `directWhitelist` mapping - allowing them to mint new nft's at times when there is no competition and no one else is able to mint.

Also, direct bypass of the time window constraint of each tier

Recommendations

Add the enforcement of the tier time window for `directWhitelist` users by using `isTierActive`

[M-7] CreateCollectionWithWhitelistAndRevealStrategy will create wrong indexing

Description

The flow of the function `createCollectionWithWhitelistAndRevealStrategy` increments the `collectionCount` after adding the newly deployed creation to the `allCreations[]` array via `addCollection` but it will create an empty index as the `collectionCount` is moved forward but the new collection is never set for that index via calling `setCollectionByIndex` . The flow from `BifyLaunchpad` ends with registering the new collection in the core contract via:

```
   function registerWhitelistCollection(
        address _collectionAddress,
        address _creator,
        string memory _name,
        string memory _symbol,
        uint256 _maxSupply,
        uint96 _royaltyFee,
        bytes32 _category,
        bool _enableAdvancedWhitelist,
        string memory _whitelistName
    ) external {
....

        // Update storage contract
        _getStorage().setCollectionData(_collectionAddress, data);
        _getStorage().setRegisteredCollection(_collectionAddress, true);
        _getStorage().addCreatorCollection(_creator, _collectionAddress);
        _getStorage().addCollection(_collectionAddress);

        // Add to category if specified
        if (_category != bytes32(0)) {
            _getStorage().addCollectionToCategory(
                _category,
                _collectionAddress
            );
            _getStorage().incrementCategoryCount(_category);
        }
```

```
->      // @audit NEVER CALLS `setCollectionByIndex` but it increments
        // the collectionCount - All further creations will be at wrong index
        // Increment collection count
->        _getStorage().incrementCollectionCount();
```

But the registerWhitelistCollection function increments the collection count without ever actually adding the new creation to the collectionByIndex array via `setColllctionByIndex` which should be called :

```
    function setCollectionByIndex(
        uint256 index,
        address collection
    ) external onlyAuthorized {
        collectionByIndex[index] = collection;
```

Impact

All indexing after this function executes will be in the wrong state, and any further querying will produce the wrong information

The `registerWhitelistCollection` function, which increments but never calls `setCollection-ByIndex` - leading to a new collection being created and added to all of the storage contract variables for query, except for the `setCollectionByIndex`. This will lead to an empty index as the `collection-Count` will be incremented without ever setting the new collection for the current index.

More over, this is a persistent issue, as there will be a new empty index every time this function is executed - creating major discrepancies in the query system.

The issue is also within the `BifyLaunchpadCore::createCollectionWithWhitelist` function as well, the same core issue as described above for the `createCollectionWithWhitelistAndReveal-Strategy`

Recommendation

Call `setCollectionByIndex` for the new collection at the current index before incrementing the `col-lectionCount`.

[M-8] trackMint function never called within mint flow - Indexing and storage contract will be wrong

Description

The `BifyLaunchpadCore` contract and `BifyLaunchpad` are expected to update the storage contract to have accurate indexing of total minted for a collection and total unique holders for a collection. This is handled via calling `trackMint` in either : `BifyLaunchpad` (which calls `BifyLaunchpad-Core::trackMint`) or directly in `BiflaunchpadCore`. As you can see here, it is expected to be called by the NFTCollection contract - which makes sense as the mints will happen there:

```
     * @notice Track a mint event for indexing purposes
     * @param _collection Collection address
     * @param _minter Minter address
     * @param _quantity Number of NFTs minted
     * @param _isWhitelistMint Whether this was a whitelist mint
     */
    function trackMint(
        address _collection,
```

```
        address _minter,
        uint256 _quantity,
        bool _isWhitelistMint
    ) external {
        require(
 ->           msg.sender == _collection || msg.sender == owner(),
            "Unauthorized"
        );
```

But the 2 mint functions in the `NFTCollection` do not accurately do this, they do not call `trackMint` and never accurately update the `BifyLaunchpadStorage` contract :

```
    function mint(uint256 _quantity) external payable nonReentrant
    {_mintBatch(msg.sender, _quantity);
 ->       // @audit FINDING : MISSING !! Should call `BifyLaunchpadCore.trackMint` -
            //public mints are never tracked nor storage updated
    }
```

This function completely ignores the need to call `trackMint` and the storage contract is never updated with the new mints Now, the whitelistMint function has 2 use cases: 1 when it is using a whitelistManager and when it is not. Either way, for both scenarios, the trackMint function is never called and the storage never accurately updated.

```
    function whitelistMint() {
              if (isExternalWhitelist) {
  -> whitelistManager.trackMint(msg.sender, _tierId, _quantity);
            // @audit FINDING : MISSING !! Should call `BifyLaunchpadCore.trackMint` -
            // whitelist mints  are never tracked nor storage updated
            // @audit whitelistManager trackMint does not update storage,
            // just the tier mapping in whitelistManagerExtended contract } else {
_mintBatch(msg.sender, _quantity);
            // @audit FINDING : MISSING !! Should call `BifyLaunchpadCore.trackMint` -
            // whitelist mints are never tracked nor storage updated
```

Impact

The storage contract will always have the wrong storage information, which will lead to an inconsistent and inaccurate state at all times.

Recommendation

At the end of each `mint` flow, correctly update the storage contract by also calling `trackMint` in either the `BifyLaunchpadCore` or `BifyLaunchpad` contract

[M-9] Missing Parameter Validation in Collection Creation Function may bring bad state

Description

The `createCollectionWithWhitelist function inBifyLaunchpadCore.sollacks proper parameter validation, failing to call`BifyLaunchpadLibrary.validateCollectionParams()‘ to verify input parameters. This allows malicious creators to bypass critical validation checks and create collections with parameters that can bring bad state.

```
require(_maxSupply > 0, "Supply must be > 0");
require(_royaltyPercentage <= 1000, "Royalty cannot exceed 10%");
```

```
require(bytes(_name).length > 0, "Name cannot be empty");
require(bytes(_symbol).length > 0, "Symbol cannot be empty");
```

Impact

- Excessive Royalty Fees: Creators can set royalty fees above 10% (1000 basis points), extracting unfair value from secondary sales

Recommendations

In function `createCollectionWithWhitelist` itself implement these checks.

```
require(_maxSupply > 0, "Supply must be > 0");
require(_royaltyPercentage <= 1000, "Royalty cannot exceed 10%");
require(bytes(_name).length > 0, "Name cannot be empty");
require(bytes(_symbol).length > 0, "Symbol cannot be empty");
```

[M-10] Incorrect Whitelist Tier Logic Prevents Valid Users from Minting (DoS)

Description

The whitelist tier validation logic in `WhitelistManagerExtended.sol` contains a mathematical error that prevents legitimate whitelisted users from minting NFTs. The issue occurs in the `isWhitelisted` function where the tier comparison uses an incorrect formula `userTierLevel >= _tierId + 1`, causing valid users with tier IDs 3 to be incorrectly returned with False.

Here the comparison of user's `uint256 _tierId` is been done with user's `uint256(directWhitelist[_user]);` which can only return four values: 0, 1, 2, 3 as they are Enum.

```
mapping(address => TierLevel) public directWhitelist;`

    enum TierLevel {
        Invalid,
        Tier1,
        Tier2,
        Tier3
    }
```

Vulnerable Code in NFTCollection.sol:

```
function whitelistMint() {
    require(whitelistManager.isWhitelisted(msg.sender, _tierId, _merkleProof),
    "Not whitelisted for tier");
    // ... rest of minting logic
}
```

Buggy Logic in WhitelistManagerExtended.sol:

```
function isWhitelisted(
    address _user,
    uint256 _tierId,
    bytes32[] calldata _merkleProof
) external view returns (bool) {
    if (directWhitelist[_user] != TierLevel.Invalid) {
        uint256 userTierLevel = uint256(directWhitelist[_user]);
        // @audit BUG: Incorrect comparison logic
```

```
        return userTierLevel >= _tierId + 1;
    }
}
```

It will be returning false in all the condition where `_tierId` of the honest whitlisted user is equal to or greater than 3 ie. (`_tierId => 3`)

Impact

Denial of Service: Legitimate whitelisted users cannot mint their allocated NFTs

Recommendations

Correct the tier validation logic in function `isWhitelisted`

[M-11] Manual Exchange Rate may Prone to Manipulation and Market Volatility

Description

The `BifyTokenPayment.sol` contract relies on a manually set exchange rate (`ethToBifyRate`) to determine the price of BIFY tokens in terms of ETH. This rate is initialized in the constructor and can be changed at any time by the contract owner without any restrictions. This centralized approach creates a significant risk, as the rate does not automatically reflect real-time market conditions and can be arbitrarily manipulated.

```
// Rate is set manually in the constructor
constructor(address _bifyToken) {
    bifyToken = IERC20(_bifyToken);
    ethToBifyRate = _initialRate; // Example: This is a fixed value
}

// Owner can change the rate at will
function setEthToBifyRate(uint256 _rate) external onlyOwner {
    ethToBifyRate = _rate;
    emit EthToBifyRateUpdated(_rate);
}
```

Impact

The owner can set a self-serving exchange rate, causing users to either overpay for items or allowing insiders to purchase assets at a steep discount.

Recommendation

Integrate a Decentralized Price Oracle and Replace the manual rate.

[L-1] NFTCollection::ownerMint Bypasses maxMintsPerWallet Restriction

Description

The `NFTCollection::mint()` function in the `NFTCollection` contract properly enforces the `maxMintsPerWallet` limit by tracking mints using `addressMintCount`. However, the ownerMint()

function does not update this counter when minting tokens to a user address:

```
function ownerMint(
    address _recipient,
    uint256 _quantity
) external onlyOwner nonReentrant {
    require(nextTokenId + _quantity - 1 <= maxSupply, "Exceeds max supply");

    _mintBatch(_recipient, _quantity);
}
```

This omission allows the contract owner to mint unlimited tokens to any address, bypassing the wallet mint cap. A recipient can thus hold more than the allowed maximum, undermining fair distribution and violating user expectations of capped mints.

Impact

- Circumvents distribution limits: Users could receive more NFTs than allowed via owner-minting.
- Unfair advantage: May allow team wallets or insiders to accumulate more tokens than public users.

Recommendation

- Update the `NFTCollection::ownerMint()` function to optionally enforce the wallet mint limit.

[L-2] Whitelist Cannot Be Disabled Once Enabled

Description

In the `NFTCollection` contract, the `setWhitelistConfig()` function enables the whitelist minting phase by setting the `whitelistEnabled` flag to true and configuring Merkle tree parameters:

```
function setWhitelistConfig(
        bytes32 _merkleRoot,
        uint256 _startTime,
        uint256 _endTime,
        uint256 _price,
        uint256 _maxPerWallet
    ) external onlyOwner {
        require(_startTime < _endTime, "Invalid time window");

        whitelistMerkleRoot = _merkleRoot;
        whitelistMintStartTime = _startTime;
        whitelistMintEndTime = _endTime;
        whitelistMintPrice = _price;
        whitelistMaxMintsPerWallet = _maxPerWallet;
@>        whitelistEnabled = true;

        emit WhitelistUpdated(_merkleRoot);
        emit WhitelistConfigUpdated(
            _startTime,
            _endTime,
            _price,
            _maxPerWallet
        );
    }
```

However, there is no function to disable the whitelist after it has been enabled. If the Merkle root is leaked, misconfigured, or expired, it creates a risk where unauthorized users may exploit whitelist minting and there's no way to immediately pause or turn off the whitelist phase.

This makes the contract inflexible and vulnerable to potential misuses or configuration errors.

Impact

No mitigation mechanism if Merkle root or proofs are leaked or invalid.

Recommendation

Add a function to allow the contract owner to disable the whitelist

[L-3] NFT Cannot Be Burned Internal _burn() Function Unused

Description

The `BifyNFT` contract defines an internal `_burn()` function that overrides the base implementations from ERC721 and ERC721URIStorage. However, this `_burn()` function is never invoked anywhere in the contract—neither internally nor externally. As a result:

- There is no mechanism for burning NFTs, even by the contract itself.
- Once an NFT is minted, it becomes permanent and non-removable from the supply.
- This limits the flexibility of the NFT lifecycle creators or users cannot invalidate or remove NFTs, even if needed due to spam, invalid data, regulatory takedowns, etc.

```
function _burn(
    uint256 tokenId
) internal override(ERC721, ERC721URIStorage) {
    super._burn(tokenId);

    // Clear royalty information
    _resetTokenRoyalty(tokenId);
}
```

Impact

- NFTs once minted are irreversible and unburnable.
- Leads to NFT spam accumulation, especially since mintNFT() can be abused as described in other issues.

Recommendations

Implement and expose a controlled NFT burn mechanism. Either: - Add a public/external burnNFT(uint256 tokenId) function, gated appropriately (e.g., only creator or approved):

```
function burnNFT(uint256 tokenId) external {
    require(_isApprovedOrOwner(msg.sender, tokenId), "Not authorized to burn");
    _burn(tokenId);
}
```

- Or integrate _burn() into other flows like blacklisting, ownership transfer denial, or spam moderation.

[L-4] Improper Access Control : Any user can deploy WhitelistMangerExtended instances

Description

The `WhitelistManagerFactory` contract allows any external user to create their own `WhitelistManagerExtended` instance via the `createWhitelistManager()` function. Since this function is public and lacks access control, unauthorized users can deploy their own whitelist manager contracts and gain full ownership of them.

```
// @audit MISSING : access control, anyone can call this function
// and deploy new instance
    function createWhitelistManager(
        string memory _name
    ) external returns (address managerAddress) {
        WhitelistManagerExtended manager = new WhitelistManagerExtended(_name);
```

If the main minting contract is configured to dynamically reference an external whitelist manager (based on whitelistMerkleRoot being interpreted as an address), an attacker could:

Deploy a malicious `WhitelistManagerExtended`

Configure it to falsely validate themselves as whitelisted Trigger the whitelistMint() function to mint NFTs without proper authorization or payment .

Impact

Users can deploy their own malicious `WhitelistManagerExtended` instances, but the rest of the system will severely limit the extent of the impact that this can have

Recommendations

Add the `Authorized` modifier for legit addresses only.

[I-1] Unsafe ERC721 Transfer in buyFixedPriceWithToken Function

Description

The `buyFixedPriceWithtoken` function in `BifyMarketPlace` uses the `transferfrom` for ERC721 token transfer, which is unsafe when transferring to a smart contract addres or EOA. This could lead to a permanent loss of NFT if transferred to contract that cannot handle `ERC721`

```
 function buyFixedPriceWithToken(
        uint256 listingId
    ) external nonReentrant whenNotPaused {

//...code

//@audit  unsafe transfer
IERC721(listing.nftContract).transferFrom(
            address(this),
            msg.sender,
            listing.tokenId
        );
```

Impact

This could lead to a permanent loss of NFT if transferred to contract that cannot handle `ERC721`

Recommendation:

Having `_checkOnERC721Received` check or a `safeTransfer` lib can mitigate the issue

[G-1] Returning Full Contract Type in _getStorage() Increases Bytecode and Gas Costs

Description In BifyLaunchpadPhase.sol, the _getStorage() function returns the full contract type BifyLaunchpadStorage, which leads to increased bytecode size and higher deployment gas costs. This happens because the compiler includes all public and inherited functions of BifyLaunchpadStorage when returning the full contract type.

```
function _getStorage()
    internal
    view
    override
    returns (BifyLaunchpadStorage)
{
    return BifyLaunchpadStorage(storageContract);
}
```

Returning the full contract type tightly couples the contract to the implementation, making it less modular and harder to upgrade or reuse components. Instead, using an interface reference would ensure only the required external function signatures are compiled in.

Impact

- Increases contract bytecode size
- Raises deployment and interaction gas costs

Recommendations Replace the return type `BifyLaunchpadStorage` with an interface type like `IBifyLaunchpadStorage`