ENIGMA DARK

Securing the Shadows

Security Review

Flaunch v1.1 Protocol Upgrade

March 21, 2025

Contents

Summary

Enigma Dark
Enigma Dark is a web3 security firm leveraging the best talent in the space to secure all kinds of blockchain protocols and decentralized apps. Our team comprises experts who have honed their skills at some of the best auditing companies in the industry. With a proven track record as highly skilled white-hats, they bring a wealth of experience and a deep understanding of the technology and the ecosystem.

Learn more about us at enigmadark.com

Flaunch v1.1 Protocol Upgrade
The ƒlaunch protocol is a launchpad platform built on Uniswap v4, incorporating advanced mechanics for token launch and trading. It emphasizes sustainability in token economies by introducing features such as Progressive Bid Walls and decentralized revenue-sharing models. These mechanisms aim to create a fair and transparent environment for participants, balancing incentives for developers and traders to prioritize long-term ecosystem stability over short-term speculation.

Engagement Overview

Over the course of 1.6 weeks, beginning March 21 2025, the Enigma Dark team conducted a security review of the Flaunch v1.1 Protocol Upgrade project. The review was performed by two Lead Security Researchers: vnmrtz & 0xWeiss.

The following repositories were reviewed at the specified commits:

Repository	Commit
flaunchgg-contracts	`diff 5794f46...cb74abd`

Risk Classification

Severity	Description
Critical	Vulnerabilities that lead to a loss of a significant portion of funds of the system.
High	Exploitable, causing loss or manipulation of assets or data.
Medium	Risk of future exploits that may or may not impact the smart contract execution.
Low	Minor code errors that may or may not impact the smart contract execution.
Informational	Non-critical observations or suggestions for improving code quality, readability, or best practices.

Vulnerability Summary

Severity	Count	Fixed	Acknowledged
Critical	1	1	0
High	1	1	0
Medium	3	3	0
Low	6	6	0
Informational	5	2	3

Findings

Index	Issue Title	Status
C-01	Creator can steal all the fees from the protocol	Fixed
H-01	Anyone can cause the protocol to not be able to claim fees	Fixed
M-01	Consider moving `TreasuryManager` initialization to the factory	Fixed
M-02	Protocol and creator can mutually grief each other	Fixed
M-03	Airdrop is incorrectly allocated to the treasury	Fixed
L-01	Incorrect Rounding Direction in Fee Calculation	Fixed
L-02	Consider making `STALE_TIME_WINDOW` configurable	Fixed
L-03	Missing BidWall staleness check in `AnyPositionManager`	Fixed
L-04	`addAirdrop` allows for an ending smaller than current timestamp	Fixed
L-05	Airdrop allocations will be miscalculated	Fixed
L-06	`checkAirdropEligibility` does not check whether the airdrop is active	Fixed
I-01	Missing Safety Checks in core protocol contract constructors and setters	Acknowledged
I-02	Simplify Role Management by Replacing AccessControl with Custom Mappings	Acknowledged
I-03	Unnecessary Try/Catch in creator retrieval	Fixed
I-04	Missing fallback base URI on `AnyFlaunch`	Fixed
I-05	Ownable is not needed anymore	Acknowledged

Detailed Findings

Critical Risk

C-01 - Creator can steal all the fees from the protocol

Severity: Critical Risk

Context:

- RevenueManager.sol#L97

Technical Details:
The `claim` function allows for anyone to withdraw fees and send them to the protocol recipient address and the creator address:

```
    function claim() public tokenExists returns (uint creatorAmount_, uint
protocolAmount_) {
        // Withdraw fees earned from the ERC721, unwrapping into ETH
        flaunchToken.flaunch.positionManager().withdrawFees(address(this),
true);

        // Discover the balance held following our fee withdrawal. We don't
just want to include
        // the fees withdrawn, but also any other ETH resting in the
manager.
        uint balance = payable(address(this)).balance;

        // If no ETH balance has been withdrawn, just return zero values
early
        if (balance == 0) {
            return (creatorAmount_, protocolAmount_); //ok
        }

        // If we have a protocol recipient, then we need to split the
revenue. If no protocol
        // recipient is set, then the creator receives the full amount.
        creatorAmount_ = balance;
        if (protocolRecipient != address(0)) {
            // Split the amount of revenue between protocol and end-owner
creator, avoiding dust
            protocolAmount_ = balance * protocolFee / MAX_PROTOCOL_FEE;
            creatorAmount_ -= protocolAmount_;
        }

        // Disperse our revenue between the two parties, without validating
```

```
receipt as we
        // don't want either call to prevent the other receiving fees.
        (bool creatorSuccess,) = payable(creator).call{value:
creatorAmount_}('');
        if (creatorSuccess) { creatorTotalClaim += creatorAmount_; }
        bool protocolSuccess;
        if (protocolAmount_ != 0) {
            (protocolSuccess,) = payable(protocolRecipient).call{value:
protocolAmount_}('');
            if (protocolSuccess) { protocolTotalClaim += protocolAmount_; }
        }

        emit RevenueClaimed(
            creator, creatorAmount_, creatorSuccess,
            protocolRecipient, protocolAmount_, protocolSuccess
        );
    }
```

The creator can re-enter in the callback function and call again `claim()`:

```
    (bool creatorSuccess,) = payable(creator).call{value: creatorAmount_}
('');
    if (creatorSuccess) { creatorTotalClaim += creatorAmount_; }
```

Therefore the fees from the `protocolRecipient` will not be sent and the remaining balance will be sent to the creator again.

```
    if (protocolAmount_ != 0) {
        (protocolSuccess,) = payable(protocolRecipient).call{value:
protocolAmount_}('');
        if (protocolSuccess) { protocolTotalClaim += protocolAmount_; }
    }
```

Impact:
The creator can steal the fees from the protocol recipient.

Recommendation:
Add two different functions to claim fees, one for the creator and one for the protocol

Developer Response:
Fixed at commit `7c245e5`.

High Risk

H-01 - Anyone can cause the protocol to not be able to claim fees

Severity: High Risk

Context:

- RevenueManager.sol#L97

Technical Details:
The `claim` function allows for anyone to withdraw fees and send them to the creator address:

```
function claim() public tokenExists returns (uint creatorAmount_, uint
protocolAmount_) {
        // Withdraw fees earned from the ERC721, unwrapping into ETH
        flaunchToken.flaunch.positionManager().withdrawFees(address(this),
true);

        // Discover the balance held following our fee withdrawal. We don't
just want to include
        // the fees withdrawn, but also any other ETH resting in the
manager.
        uint balance = payable(address(this)).balance;

        // If no ETH balance has been withdrawn, just return zero values
early
        if (balance == 0) {
            return (creatorAmount_, protocolAmount_); //ok
        }

        // If we have a protocol recipient, then we need to split the
revenue. If no protocol
        // recipient is set, then the creator receives the full amount.
        creatorAmount_ = balance;
        if (protocolRecipient != address(0)) {
            // Split the amount of revenue between protocol and end-owner
creator, avoiding dust
            protocolAmount_ = balance * protocolFee / MAX_PROTOCOL_FEE;
            creatorAmount_ -= protocolAmount_;
        }

        // Disperse our revenue between the two parties, without validating
receipt as we
        // don't want either call to prevent the other receiving fees.
```

```
        (bool creatorSuccess,) = payable(creator).call{value:
creatorAmount_}('');
        if (creatorSuccess) { creatorTotalClaim += creatorAmount_; }
        bool protocolSuccess;
        if (protocolAmount_ != 0) {
            (protocolSuccess,) = payable(protocolRecipient).call{value:
protocolAmount_}('');
            if (protocolSuccess) { protocolTotalClaim += protocolAmount_; }
        }

        emit RevenueClaimed(
            creator, creatorAmount_, creatorSuccess,
            protocolRecipient, protocolAmount_, protocolSuccess
        );
    }
```

By forwarding a specific amount of gas that is enough to execute the entire function with the last 1/64 of gas forwarded, the call that sends value to `protocolRecipient` can fail if the 63/64 (as specified in eip-150) of the remaining gas would not be enough to execute the callback.

This can be pre-calculated by the caller of the function and forwarded to execute the call.

Impact:
Anyone can cause the protocol to not be able to claim fees.

Recommendation:
Add two different functions to claim fees, one for the creator and one for the protocol

Developer Response:
Fixed at commit `7c245e5`.

Medium Risk

M-01 - Consider moving `TreasuryManager` initialization to the factory

Severity: Medium Risk

Context.

- TreasuryManagerFactory.sol#L50-L63
- FlaunchZap.sol#L265-L272

Technical Details:
The `TreasuryManager` contract requires and `initialize` function call to set critical parameters such as `_flaunchToken` , `_owner` and custom `_data` . For proxy deployments, it is important that these initializations are performed atomically (bundled with proxy deployment).

In this case the `TreasuryMangerFactory` responsible for deploying the manager does not invoke the `initialize` function, leaving this responsibility to the `FlaunchZap` contract. This creates a risk in cases where the `deployManager` function is called directly from the factory without involving the zap, as it allows an attacker to potentially backrun a deployment and frontrun the initialization, taking control of the manager.

Impact:
Medium. An attacker could exploit this gap in atomicity by backrunning a factory manager-deployment without initialization and then executing an initialization call, effectively taking control of the manager.

Recommendation:
Move the initialization of the Manager from the `FlaunchZap` into the `TreasuryManagerFactory` contract, ensuring that all deployments and initializations are handled atomically within the factory itself.

Developer Response:
Fixed at commit `50e23da` .

M-02 - Protocol and creator can mutually grief each other

Severity: Medium Risk

Context:

- RevenueManager.sol#L97

Technical Details:
The `claim` function allows for anyone to withdraw fees and send them to the creator address and the creator address:

```
function claim() public tokenExists returns (uint creatorAmount_, uint
protocolAmount_) {
        // Withdraw fees earned from the ERC721, unwrapping into ETH
        flaunchToken.flaunch.positionManager().withdrawFees(address(this),
true);

        // Discover the balance held following our fee withdrawal. We don't
just want to include
        // the fees withdrawn, but also any other ETH resting in the
manager.
        uint balance = payable(address(this)).balance;

        // If no ETH balance has been withdrawn, just return zero values
early
        if (balance == 0) {
            return (creatorAmount_, protocolAmount_); //ok
        }

        // If we have a protocol recipient, then we need to split the
revenue. If no protocol
        // recipient is set, then the creator receives the full amount.
        creatorAmount_ = balance;
        if (protocolRecipient != address(0)) {
            // Split the amount of revenue between protocol and end-owner
creator, avoiding dust
            protocolAmount_ = balance * protocolFee / MAX_PROTOCOL_FEE;
            creatorAmount_ -= protocolAmount_;
        }

        // Disperse our revenue between the two parties, without validating
receipt as we
        // don't want either call to prevent the other receiving fees.
        (bool creatorSuccess,) = payable(creator).call{value:
creatorAmount_}('');
        if (creatorSuccess) { creatorTotalClaim += creatorAmount_; }
        bool protocolSuccess;
        if (protocolAmount_ != 0) {
            (protocolSuccess,) = payable(protocolRecipient).call{value:
protocolAmount_}('');
            if (protocolSuccess) { protocolTotalClaim += protocolAmount_; }
        }

        emit RevenueClaimed(
```

```
            creator, creatorAmount_, creatorSuccess,
            protocolRecipient, protocolAmount_, protocolSuccess
        );
    }
```

Both, the creator and the protocol recipient are able to spend forwarded gas from the caller in their external calls. This allows both addresses to grief the caller and make them spend close to the gas limit.

Impact:
Gas griefing.

Recommendation:
Add two different functions to claim fees, one for the creator and one for the protocol.

Developer Response:
Fixed at commit `7c245e5`.

M-03 - Airdrop is incorrectly allocated to the treasury

Severity: Medium Risk

Context:

- SnapshotAirdrop.sol#L67

Technical Details:
In the SnapshotAirdrop contract, which intends to allows creators add an airdrop for their memecoin, when calculating the not elegible amount for the airdrop, currently it subtracts the balance of the position manager and the balance of the pool manager:

```
  // Take snapshot of the balances
        uint totalSupply = Memecoin(_memecoin).totalSupply();
        uint notEligibleSupply =
Memecoin(_memecoin).balanceOf(address(positionManager)) +

Memecoin(_memecoin).balanceOf(address(positionManager.poolManager()));
```

While that's done correctly, it misses to subtract the balance in the treasury contract of that memecoin.

Impact:
Airdrop will be allocated to the treasury while it should be excluded.

Recommendation:
When subtracting the addresses that are not elegible, do subtract also the treasury address:

```
  // Take snapshot of the balances
        uint totalSupply = Memecoin(_memecoin).totalSupply();
        uint notEligibleSupply =
Memecoin(_memecoin).balanceOf(address(positionManager)) +

Memecoin(_memecoin).balanceOf(address(positionManager.poolManager()))
+  + address memecoinTreasury = Memecoin(_memecoin).treasury();
```

Developer Response:
Fixed at commit `2a467f3`.

Low Risk

L-01 - Incorrect Rounding Direction in Fee Calculation

Severity: Low Risk

Context:

- RevenueManager.sol#L115

Technical Details:
The protocol fee calculation uses incorrect rounding, causing the Protocol Fee Amount to be rounded down. As a general rule, protocol fee calculations are usually rounded up on DeFi protocols.

Recommendation:
Adjust the rounding logic to ensure protocol fees are rounded up, aligning with industry best practices.

Proposed Fix:

```
if (protocolRecipient != address(0)) {
    // Split the amount of revenue between protocol and end-owner creator,
avoiding dust
    protocolAmount_ = (balance * protocolFee + MAX_PROTOCOL_FEE - 1) /
MAX_PROTOCOL_FEE;
    creatorAmount_ -= protocolAmount_;
}
```

Developer Response:
Fixed at commit `6f3ea0e`.

L-02 - Consider making `STALE_TIME_WINDOW` configurable

Severity: Low Risk

Context:

- BidWall.sol#L90

Technical Details:
Flaunch v1.1 introduces a new staleness check for BidWall updates, designed to ensure that outdated BidWalls are repositioned closer to the current price tick. This helps maintain liquidity and improves the likelihood of users selling into the BidWall, especially for low market cap or low volume tokens. Currently, the staleness threshold is hardcoded to 7

days. As this is a new feature, the fixed duration may not be optimal across all market conditions or use cases. A static value limits flexibility and adaptability as the protocol is tested in production.

Impact:
Low. The feature functions as intended, but the fixed 7 day staleness window is not configurable. This could hinder the responsiveness of the system in adapting to real-world usage or different market environments. Seven days may prove either too long or too short as usage patterns emerge.

Recommendation:
Make the BidWall staleness threshold configurable, allowing protocol admins to fine-tune the duration based on observed performance and user behaviour.

Developer Response:
Fixed at commit `88b0a36`.

L-03 - Missing BidWall staleness check in `AnyPositionManager`

Severity: Low Risk

Context:

- AnyPositionManager.sol#L349
- PositionManager.sol#L580-L586

Technical Details:
In the Flaunch v1.1 upgrade, a staleness check has been added to the `PositionManager` contract to automatically update the bidWall position when it becomes outdated. This ensures better liquidity access and more efficient price discovery. However, this logic is missing in the `AnyPositionManager` variant. As a result, low-liquidity (small cap) memecoins may suffer from outdated bidWalls, making it harder for users to sell into them at a favorable tick.

Impact:
Missing feature parity with core PositionManager, leading to impaired usability for tokens with low trading volume.

Recommendation:
Implement the same staleness check and bidWall position update logic from `PositionManager` within the `AnyPositionManager`'s beforeSwap hook.

Developer Response:
Fixed at commit `02ad5e5`.

L-04 - `addAirdrop` allows for an ending smaller than current timestamp

Severity: Low Risk

Context:

- MerkleAirdrop.sol#L59

Technical Details:
The `addAirdrop` function creates an airdrop which has an end time, declared as `airdropEndTime`:

```
 function addAirdrop(
        address _creator,
        uint _airdropIndex,
        address _token,
        uint _amount,
        uint _airdropEndTime,
        bytes32 _merkleRoot,
        string calldata _merkleDataIPFSHash
    ) external payable override(IMerkleAirdrop) onlyApprovedAirdropCreators
{
        // Validate that the airdrop is configurated as expected
        if (_airdropIndex != airdropsCount[_creator]) revert
InvalidAirdropIndex();
        if (_airdropData[_creator][_airdropIndex].merkleRoot != bytes32(0))
revert AirdropAlreadyExists();

        // Pull in the tokens from the sender
        uint amount = _pullTokens(_token, _amount);

        // Create our airdrop struct
        _airdropData[_creator][_airdropIndex] = AirdropData({
            token: _token,
            airdropEndTime: _airdropEndTime,
            amountLeft: amount,
            merkleRoot: _merkleRoot,
            merkleDataIPFSHash: _merkleDataIPFSHash
        });

        unchecked {
            ++airdropsCount[_creator];
        }

        emit NewAirdrop(_creator, _airdropIndex, _token, amount,
_airdropEndTime);
    }
```

Currently, it allows for the end time to be in the past, which should not be possible.

Impact:
Expired aidrops can be created

Recommendation:
Add the following check:

```
 function addAirdrop(
        address _creator,
        uint _airdropIndex,
        address _token,
        uint _amount,
        uint _airdropEndTime,
        bytes32 _merkleRoot,
        string calldata _merkleDataIPFSHash
    ) external payable override(IMerkleAirdrop) onlyApprovedAirdropCreators
{
        // Validate that the airdrop is configurated as expected
        if (_airdropIndex != airdropsCount[_creator]) revert
InvalidAirdropIndex();
        if (_airdropData[_creator][_airdropIndex].merkleRoot != bytes32(0))
revert AirdropAlreadyExists();
+     if (_airdropEndTime <= block.timestamp) revert AirdropExpired();
        // Pull in the tokens from the sender
        uint amount = _pullTokens(_token, _amount);

        // Create our airdrop struct
        _airdropData[_creator][_airdropIndex] = AirdropData({
            token: _token,
            airdropEndTime: _airdropEndTime,
            amountLeft: amount,
            merkleRoot: _merkleRoot,
            merkleDataIPFSHash: _merkleDataIPFSHash
        });

        unchecked {
            ++airdropsCount[_creator];
        }

        emit NewAirdrop(_creator, _airdropIndex, _token, amount,
_airdropEndTime);
    }
```

Developer Response:
Fixed at commit `9da8585`.

L-05 - Airdrop allocations will be miscalculated

Severity: Low Risk

Context:

- SnapshotAirdrop.sol#L76

Technical Details:
When calling `addAirdrop`, the protocol uses `block.timestamp` as the cut-off for airdrop elegibility:

```
  _airdropData[_memecoin][airdropIndex] = AirdropData({
            creator: _creator,
            token: _token,
            totalTokensToAirdrop: amount,
            memecoinHoldersTimestamp: block.timestamp,
            eligibleSupplySnapshot: totalSupply - notEligibleSupply,
            airdropEndTime: _airdropEndTime,
            amountLeft: amount
        });
```

This is because the use of `block.number` inside the `ERC20VotesUpgradeable` contract has been overriden to use `block.timestamp`.

There are certain functions like `CLOCK_MODE` that still return `block.number` even if using `block.timestamp` as a way of calculating votes:

```
 function CLOCK_MODE() public view virtual override returns (string memory)
{
        // Check that the clock was not modified
        require(clock() == block.number, "ERC20Votes: broken clock mode");
        return "mode=blocknumber&from=default";
    }
```

Impact:
Incorrect return of checkpoint modes, could affect integrations

Recommendation:
Override the `CLOCK_MODE` and return `block.timestamp`

Developer Response:
Fixed at commit `e718b5a`.

L-06 - `checkAirdropEligibility` does not check whether the airdrop is active

Severity: Low Risk

Context:

- SnapshotAirdrop.sol#L187

Technical Details:
The `checkAirdropEligibility` function allows to check whether a user is eligible for a specific airdrop and what amount they can claim:

```
   function checkAirdropEligibility(address _memecoin, uint _airdropIndex,
address _user) external view override(ISnapshotAirdrop) returns (uint
claimableAmount) {

        AirdropData storage airdrop = _airdropData[_memecoin]
[_airdropIndex];
        uint claimantBalanceSnapshot =
Memecoin(_memecoin).getPastVotes(_user, airdrop.memecoinHoldersTimestamp);
require(timepoint < clock(), "ERC20Votes: future lookup"); so it should be
blocks no seconds

        claimableAmount = FullMath.mulDiv(airdrop.totalTokensToAirdrop,
claimantBalanceSnapshot, airdrop.eligibleSupplySnapshot);
    }
```

The function does not factor the state when the airdrop is inactive, as it means that at that specific time the user can really claim 0.

Impact:
`checkAirdropEligibility` will return a faulty state when the airdrop is not active

Recommendation:

Check that the airdrop is active, if not, return 0 as the claimable amount:

```
   function checkAirdropEligibility(address _memecoin, uint _airdropIndex,
address _user) external view override(ISnapshotAirdrop) returns (uint
claimableAmount) {
+ if !isAirdropActive(_memecoin, _airdropIndex) return 0;
        AirdropData storage airdrop = _airdropData[_memecoin]
[_airdropIndex];
        uint claimantBalanceSnapshot =
Memecoin(_memecoin).getPastVotes(_user, airdrop.memecoinHoldersTimestamp);
require(timepoint < clock(), "ERC20Votes: future lookup"); so it should be
blocks no seconds

        claimableAmount = FullMath.mulDiv(airdrop.totalTokensToAirdrop,
claimantBalanceSnapshot, airdrop.eligibleSupplySnapshot);
    }
```

Developer Response:
Fixed at commit `f12bc07`.

Informational

I-01 - Missing Safety Checks in core protocol contract constructors and setters

Severity: Informational

Context:

- PositionManager.sol#L190-L218
- AnyPositionManager.sol#L153-L177
- Flaunch.sol#L110-L128
- AnyFlaunch.sol#L81-L84
- FlaunchZap.sol#L117-L133
- BuyBackAndBurnFlay.sol#L238-L240

Technical Details:
Currently, protocol contracts such as BidWall, PositionsManagers, and Flaunch do not include essential safety validations. These missing checks include, but are not limited to, ensuring non-zero addresses and validating minimum/maximum bounds for critical parameters like escrow duration and fee values.

Given the immutable nature of these contracts once deployed, it is highly recommended to incorporate these safeguards directly into the contract logic. While deployment scripts currently enforce some constraints, relying solely on external tooling increases the risk of misconfigurations or unintended deployments.

Recommendation:
Introduce appropriate safety checks within constructors, setters, and initializer functions to enforce parameter validity and mitigate potential deployment risks.

Developer Response:
Acknowledged and partially implemented. Will include these in the next deployment.

I-02 - Simplify Role Management by Replacing AccessControl with Custom Mappings

Severity: Informational

Context:

- BidWall.sol#L6
- FairLaunch.sol#L4
- TreasuryManagerFactory.sol#L7

Technical Details:
The `FairLaunch`, `BidWall`, and `TreasuryManagerFactory` contracts inherit `AccessControl` from OpenZeppelin. However, each contract only utilizes a single role:
• `BidWall` and `FairLaunch` use the `POSITION_MANAGER` role.
• `TreasuryManagerFactory` uses the `FLAUNCH` role.

Given this limited usage, the full `AccessControl` implementation may be unnecessarily complex and could increase the potential attack surface for role-restricted functions.

Recommendation:
Replace the AccessControl inheritance with simple mappings to manage roles more efficiently:

- Use a `mapping(address => bool) positionManager` in BidWall and FairLaunch.
- Use a `mapping(address => bool) flauncContracts` in TreasuryManagerFactory.

Implement appropriate setter and getter functions to manage these mappings securely.

This approach simplifies the access control logic and reduces overall contract complexity and potential vulnerabilities.

Additional Note:
The HypeFeeCalculator contract currently relies on the `FairLaunch` contract's `hasRole` function to check for valid position managers. This will need to be updated to use the new custom role-checking logic (e.g., a public `isPositionManager(address)` function) in `FairLaunch`.

Developer Response:
Acknowledged and agreed this would be a better approach. However we stuck to using AccessControl to depend on the existing library. We will likely revisit this in a future deployment.

I-03 - Unnecessary Try/Catch in creator retrieval

Severity: Informational

Context:

- AnyFlaunch.sol#L184

Technical Details:
The `creator` function in the `AnyFlaunch` contract retrieves the owner of an NFT linked to a memecoin by calling the public `ownerOf` function. However, `ownerOf` reverts if the given tokenId does not exist. To handle this, the implementation currently wraps the call in a `try/catch block`. This workaround is unnecessary, as the internal `_ownerOf` function from ERC721 can be used instead, it returns the owner or address zero without reverting.

Recommendation:
Replace the `try/catch` around `ownerOf` with a direct call to `_ownerOf` for a simpler and more efficient implementation.

Developer Response:
Fixed at commit `ea6cf2e`.

I-04 - Missing fallback base URI on `AnyFlaunch`

Severity: Informational

Context:

- AnyFlaunch.sol#L149-L154
- Flaunch.sol#L234-L237

Technical Details:
The default `Flaunch` contract includes a fallback mechanism that returns a memecoin URI when the NFT base URI is empty. In contrast, the `AnyFlaunch` variant lacks this functionality and will instead return an empty string if no base URI is set.

Recommendation:
To ensure consistent behavior, consider either:

1. Implementing a similar fallback mechanism in AnyFlaunch, or
2. Preventing the configuration of empty base URIs via the constructor and setter functions.

Developer Response:
Fixed at commit `142ab82`. Since the token will be bridged in, the ERC20 may not have a tokenURI. Because of this, the fallback logic is intentionally excluded from the view function. However, we have added checks to prevent empty strings where applicable.

I-05 - Ownable is not needed anymore

Severity: Informational

Context:

- BidWall.sol#L36

Technical Details:
The `BidWall` contract has both contracts inherited to manage the access control. All of the required functionalities can be taken care of, just by using the `AccessControl` and referring to the owner as the `DEFAULT_ADMIN_ROLE`:

```
contract BidWall is AccessControl, Ownable {
```

Impact:
Considerably higher deploying costs than needed and redundancy in inheritance.

Recommendation:
Remove `Ownable` from inheritance.

Developer Response:
Acknowledged.

Disclaimer

This report does not endorse or critique any specific project or team. It does not assess the economic value or viability of any product or asset developed by parties engaging Enigma Dark for security assessments. We do not provide warranties regarding the bug-free nature of analyzed technology or make judgments on its business model, proprietors, or legal compliance.

This report is not intended for investment decisions or project participation guidance. Enigma Dark aims to improve code quality and mitigate risks associated with blockchain technology and cryptographic tokens through rigorous assessments.

Blockchain technology and cryptographic assets inherently involve significant risks. Each entity is responsible for conducting their own due diligence and maintaining security measures. Our assessments aim to reduce vulnerabilities but do not guarantee the security or functionality of the technologies analyzed.

This security engagement does not guarantee against a hack. It is a review of the codebase during a specific period of time. Enigma Dark makes no warranties regarding the security of the code and does not warrant that the code is free from defects. By deploying or using the code, the project and users of the contracts agree to use the code at their own risk. Any modifications to the code will require a new security review.