33AUDITS & CO.

Poply Audit Report

Introduction

A security review of the Poply protocol was done, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits & Co with Mahdi Karimi and Alireza-Razavi as the Security Researchers.

Disclaimer

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

About 33 Audits & Company

33Audits is an independent smart contract security researcher company and development group. We conduct audits as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @33audits.

About Poply

This audit is being performed on the core protocol contracts for Poply. The contracts in scope include `PoplyMarketplace`, `PoplyMarketplaceProxy`, `NftDrop`, and `NftDropFactory`. The Poply protocol is an NFT marketplace and drop system that allows users to create, list, buy, sell, and make offers on NFTs. The protocol includes platform fees, royalty distribution, allowlist functionality, and batch transfer capabilities.

Repository: https://github.com/0xpoply/contract

Scope: `PoplyMarketplace`, `PoplyMarketplaceProxy`, `NftDrop`, `NftDropFactory`

Severity Definitions

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

Impact - The technical, economic, and reputation damage from a successful attack

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

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Findings Summary

ID	Title	Severity	Status
[M-01]	Unrestricted `withdraw()` Function Can Drain Funds Needed for Offer Settlements	Medium	Fixed
[L-01]	NFTs Cannot Be Sold in Marketplace When Royalty Recipient Cannot Receive ETH	Low	Fixed
[L-02]	`buyItem` Does Not Refund Excess Funds Sent by Buyer	Low	Fixed
[L-03]	No Cap on Combined Platform Fee and Royalty — Total Fees Can Exceed 100%	Low	Fixed
[L-04]	`_baseTokenURI` Storage Leak Before Reveal Due to Blockchain Transparency	Low	Fixed
[L-05]	`isEligibleToMint` Returns Invalid Data When No Merkle Root Is Set for ClaimCondition	Low	Fixed

Medium

[M-01] Unrestricted `withdraw()` Function Can Drain Funds Needed for Offer Settlements

Description

The `PoplyMarketplace` contract includes a `withdraw(address _to)` function that allows the owner to withdraw all ETH held by the contract without any limitation or accounting for pending obligations.

The contract holds buyer/offeror funds in escrow for offers, and uses those funds to pay sellers when offers are accepted. If the owner calls `withdraw()` and drains the balance, the contract may no longer have enough funds to settle accepted offers or refund cancelled offers.

This is a centralization risk where using this function blocks canceling or accepting all of the ongoing offers and locks offerors' funds.

Affected Code:

```
// PoplyMarketplace.sol
function withdraw(address _to) external nonReentrant onlyOwner {
    (bool success, ) = payable(_to).call{value: address(this).balance}
("");
    require(success, "Transfer failed!");
}
```

This function withdraws **all** ETH held by the contract without tracking liabilities (e.g., total locked offer amounts).

Steps to Reproduce:

1. A buyer creates multiple offers, locking ETH in the contract.
2. The owner calls `withdraw()` to transfer the entire contract balance to their address.
3. A seller attempts to accept one of the offers:
    - The contract has no funds to pay the seller.
    - Transaction reverts.
4. Similarly, offerors cannot cancel their offers and retrieve their funds.

Impact

- **High Impact**: Offerors' funds become permanently locked in the contract if withdrawn by owner
- **Medium Likelihood**: Requires owner to inadvertently or maliciously drain contract balance

Recommendations

Track locked offer amounts and restrict withdrawals to only excess funds:

```
uint256 public totalLockedOffers;

function makeOffer(...) external payable {
    totalLockedOffers += msg.value;
    ...
}

function cancelOffer(uint256 _offerId) external {
    totalLockedOffers -= offers[_offerId].price;
    ...
}

function acceptOffer(uint256 _offerId, ...) external {
    totalLockedOffers -= offers[_offerId].price;
    ...
}

function withdraw(address _to) external onlyOwner {
    uint256 available = address(this).balance - totalLockedOffers;
    require(available > 0, "No excess funds available");
    (bool success, ) = payable(_to).call{value: available}("");
    require(success, "Transfer failed!");
}
```

This ensures only excess funds can be withdrawn, preserving escrowed funds for offers and sellers.

Status

Fixed - The developer has added tracking for total locked offers and restricted the `withdraw()` function to only allow withdrawal of excess funds not locked in pending offers.

Fix commit: d815627

Low

[L-01] NFTs Cannot Be Sold in Marketplace When Royalty Recipient Cannot Receive ETH

Description

NFTs cannot be sold in the marketplace if the royalty recipient address is set to a contract without a `receive`/`fallback` function that can accept ETH, or if the royalty recipient contract has been self-destructed.

In both cases, the call to transfer royalty payment will return `false`:

```
// PoplyMarketplace.sol - _sendRoyalty function
if (royaltyRecipient != address(0) && royaltyPayment > 0) {
    (bool sent, ) = payable(royaltyRecipient).call{value: royaltyPayment}
("");
    require(sent, "Royalty payment failed");  // @audit transaction
reverts
}
```

The `require` statement enforces that the marketplace must fund the royalty recipient address, which may prevent NFTs from being sold at the marketplace if the recipient cannot receive ETH.

Impact

- **Medium Impact**: NFTs may not be sold in the marketplace and transactions revert
- **High Likelihood**: Can occur in legitimate scenarios where creators use multi-sig or contract wallets without receive functions

Recommendations

While checking the `sent` flag after a call is a good and common pattern, in this edge case scenario it should be handled more gracefully. Instead of reverting, the contract could:

1. Remove the strict `require` check and allow the sale to proceed even if royalty payment fails:

```
if (royaltyRecipient != address(0) && royaltyPayment > 0) {
    (bool sent, ) = payable(royaltyRecipient).call{value: royaltyPayment}
("");
    // Log the failure but don't revert
    if (!sent) {
```

```
            emit RoyaltyPaymentFailed(royaltyRecipient, royaltyPayment);
        }
    }
```

2. Alternatively, implement a withdrawal pattern where failed royalty payments are accumulated for later withdrawal by the recipient.

Status

Fixed - The developer has modified the royalty payment logic to gracefully handle failed payouts without reverting the entire transaction.

Fix commit: 99e1c18

[L-02] `buyItem` Does Not Refund Excess Funds Sent by Buyer

Description

The `buyItem` function processes payments based on `msg.value` rather than the current listing price. This results in the following issues:

1. If a buyer sends more ETH than the current listing price, the entire `msg.value` is transferred to the seller and fees are calculated on the full amount.
2. There is no mechanism to refund the excess amount to the buyer.

Additionally, this creates a race condition scenario:

- If the seller updates the listing price to a lower value right before the buyer's transaction confirms, the buyer — expecting the old price — may send a higher amount.
- Because the contract transfers based on `msg.value` directly, the seller receives an overpayment, and the buyer loses funds.

Affected Code:

```
// PoplyMarketplace.sol
function buyItem(address _nftAddress, uint256 _tokenID)
    external
    payable
    nonReentrant
    isListed(_nftAddress, _tokenID)
    notOwner(_nftAddress, _tokenID, msg.sender)
{
    Listing memory listing = listings[_nftAddress][_tokenID];
    if (msg.value < listing.price) {
        revert NftMarketplace__PriceNotMet(_nftAddress, _tokenID,
msg.value);
    }

    // ...NFT transfer...
```

```
    uint256 royaltyAmount = _sendRoyalty(_nftAddress, _tokenID,
msg.value);
    uint256 amountToPay = msg.value - ((msg.value * platformFee) /
MAX_BPS) - royaltyAmount;
    (bool success, ) = payable(listing.seller).call{value: amountToPay}
("");
    require(success, "Could not pay the money to seller!");

    emit ItemBought(listing.seller, _nftAddress, _tokenID, listing.price);
}
```

This behavior deviates from standard marketplace expectations where the buyer pays exactly the listing price, and any overpayment is refunded.

Impact

- **Medium Impact**: Buyers can lose excess funds sent beyond listing price
- **Medium Likelihood**: Can occur through user error or race conditions during price updates

Recommendations

Use the current listing price for payment settlement and refund excess:

```
function buyItem(address _nftAddress, uint256 _tokenID)
    external
    payable
    nonReentrant
    isListed(_nftAddress, _tokenID)
    notOwner(_nftAddress, _tokenID, msg.sender)
{
    Listing memory listing = listings[_nftAddress][_tokenID];
    require(msg.value >= listing.price, "Insufficient payment");

    uint256 excess = msg.value - listing.price;

    // ...NFT transfer...

    // Calculate fees and royalty based on listing.price, not msg.value
    uint256 royaltyAmount = _sendRoyalty(_nftAddress, _tokenID,
listing.price);
    uint256 amountToPay = listing.price - ((listing.price * platformFee) /
MAX_BPS) - royaltyAmount;

    // Transfer exactly the listing price to seller
    payable(listing.seller).transfer(amountToPay);

    // Refund excess back to buyer if any
    if (excess > 0) {
        payable(msg.sender).transfer(excess);
    }
```

```
    emit ItemBought(listing.seller, _nftAddress, _tokenID, listing.price);
}
```

This ensures buyers are never overcharged and prevents sellers from receiving unintended overpayments.

Status

Fixed - The developer has modified the `buyItem` function to charge the exact listing price and refund any excess payment to the buyer.

Fix commit: 1361066

[L-03] No Cap on Combined Platform Fee and Royalty — Total Fees Can Exceed 100%

Description

The contract allows configuration of both a platform fee and a royalty fee on sales. However, there is no restriction or validation ensuring that the combined total of these fees does not exceed 100% of the transaction amount.

For example, if the platform fee is set to 90% and the royalty fee to 15%, the combined total would be 105%, exceeding the full transaction amount. This would cause the transaction to revert or break settlement logic, leaving no funds (or insufficient funds) for the seller.

Although such a configuration may seem unlikely in normal operation, it is theoretically possible and represents a misconfiguration risk.

Affected Code:

```
// PoplyMarketplace.sol
uint32 public platformFee;

// addCollectionRoyalty allows royalty up to 10%
function addCollectionRoyalty(address _nftAddress, address _creator,
uint32 _royalty)
    external
    onlyOwner
{
    require(_royalty <= 1000, "Royalty exceeds 10%");  // 10% = 1000 bps
    royalties[_nftAddress] = collectionRoyalty(_creator, _royalty);
    emit CollectionRoyaltyAdded(_nftAddress, _creator, _royalty);
}

// But platformFee can be set up to 100% (10000 bps)
function initialize(uint32 _fee) public initializer {
    require(_fee <= MAX_BPS, "Fee too high");  // MAX_BPS = 10000
    platformFee = _fee;
    totalOffers = 1;
}
```

There is a cap on royalty (10%) but the platform fee could be set as high as 100%, making the combined fees exceed 100%.

Impact

- **Medium Impact**: Misconfiguration could break marketplace functionality
- **Low Likelihood**: Requires owner to set platform fee above 90%

Recommendations

Implement a validation check when setting the platform fee or royalty to ensure:

```
platformFee + maxPossibleRoyalty <= 10000  // 100% in basis points
```

For example:

```
function initialize(uint32 _fee) public initializer {
    // Ensure platform fee + max royalty (10%) doesn't exceed 100%
    require(_fee <= 9000, "Fee too high - must leave room for royalty");
// 90% max
    platformFee = _fee;
    totalOffers = 1;
}

function setPlatformFee(uint32 _newFee) external onlyOwner {
    require(_newFee <= 9000, "Fee too high - must leave room for
royalty");
    platformFee = _newFee;
}
```

This will prevent configurations that exceed the total transaction amount and ensure consistent payout logic.

Status

Fixed - The developer has implemented caps on platform fee and added validation to ensure combined fees never exceed 100%.

Fix commit: 665e6d9

[L-04] `_baseTokenURI` Storage Leak Before Reveal Due to Blockchain Transparency

Description

The `_baseTokenURI` state variable in the `NftDrop` contract, although declared as `private`, can be read directly from blockchain storage (e.g., using `web3.eth.getStorageAt`) before the `revealed` flag is set to `true`. This allows anyone to access the base URI of NFT metadata before the intended reveal time.

This bypasses the `revealed` check in the `baseTokenURI()` function, which is intended to prevent access to the metadata URI before the reveal.

Affected Code:

```
// NftDrop.sol
string private _baseTokenURI;
bool public revealed;

function setBaseURI(string memory _newBaseURI) public onlyOwner notLocked
{
    _baseTokenURI = _newBaseURI;
}

function baseTokenURI() external view returns (string memory) {
    if (!revealed) revert NftDrop__NotRevealed();
    return _baseTokenURI;
}

function tokenURI(uint256 tokenId) public view virtual override returns
(string memory) {
    if (!_exists(tokenId)) revert NftDrop__ERC721NonexistentToken();
    if (!revealed) {
        return unrevealedURI;
    }
    string memory tokenIdInString = Strings.toString(tokenId);
    string memory baseURI = _baseURI();
    return bytes(baseURI).length > 0 ? string(abi.encodePacked(baseURI,
tokenIdInString)) : "";
}
```

Impact

- **Medium Impact**: Allows anyone to access the base URI of NFT metadata before the intended reveal time
- **High Likelihood**: Any user with blockchain knowledge can read storage slots

This undermines the reveal mechanism, which is often used for:

- Fairness in NFT distributions
- Building anticipation and excitement
- Preventing sniping based on rarity before public reveal

Recommendations

Implement a commit-reveal scheme where the owner commits a hash of the base URI, then reveals it later:

```
bytes32 public baseURICommitment;

function commitBaseURI(bytes32 _commitment) external onlyOwner notLocked {
    require(!revealed, "Already revealed");
    baseURICommitment = _commitment;
}

function revealBaseURI(string memory _baseURI, bytes32 _salt) external
onlyOwner notLocked {
    require(!revealed, "Already revealed");
    require(baseURICommitment != bytes32(0), "No commitment set");
    require(keccak256(abi.encodePacked(_baseURI, _salt)) ==
baseURICommitment, "Invalid reveal");

    _baseTokenURI = _baseURI;
    revealed = true;
    emit Revealed(_baseURI);
}
```

Alternatively:

- Store the base URI on IPFS and only store the IPFS hash on-chain after reveal
- Use encryption and only reveal the decryption key after the reveal period

Status

Fixed - The developer has implemented a commit-reveal scheme to prevent pre-reveal base URI leakage.

Fix commit: 9083f79

[L-05] `isEligibleToMint` Returns Invalid Data When No Merkle Root Is Set for ClaimCondition

Description

The `isEligibleToMint` function always returns `true` if no merkle root is set for a `ClaimCondition`, which is inconsistent with the actual minting validation in `_validateMint`.

Affected Code:

```
// NftDrop.sol
function isEligibleToMint(address _wallet, uint256 _conditionId,
AllowlistProof calldata _allowlistProof)
    public
    view
    returns (bool)
{
    uint256 startConditionIndex = claimCondition.currentStartId;
    uint256 conditionCount = claimCondition.count;
```

```
    if (_conditionId < startConditionIndex || _conditionId >=
startConditionIndex + conditionCount) {
        return false; // Invalid stage index
    }

    ClaimCondition memory currentClaimPhase =
claimCondition.conditions[_conditionId];

    if (currentClaimPhase.merkleRoot == bytes32(0)) return true; // @audit
always returns true

    return MerkleProof.verify(_allowlistProof.proof,
currentClaimPhase.merkleRoot, keccak256(abi.encodePacked(_wallet,
_allowlistProof.quantityLimitPerWallet)));
}
```

The function always returns `true` if no merkle root is set, but doesn't check:

- `maxSupply`
- If the wallet exceeded `maxPerWallet`
- `supplyClaimed` against `maxMint`

This causes inconsistency between `isEligibleToMint` and `_validateMint`.

Impact

- **Low Impact**: Users check their eligibility and receive incorrect data
- **High Likelihood**: Occurs whenever no merkle root is set

User Impact Scenario:

1. User checks their eligibility for current condition
2. The function incorrectly returns `true`
3. User runs a transaction
4. Transaction reverts due to actual validation failures
5. User loses gas fees

Recommendations

The function should actually check all eligibility requirements when no merkle root is set:

```
function isEligibleToMint(address _wallet, uint256 _conditionId,
AllowlistProof calldata _allowlistProof)
    public
    view
    returns (bool)
{
    uint256 startConditionIndex = claimCondition.currentStartId;
    uint256 conditionCount = claimCondition.count;
```

```
    if (_conditionId < startConditionIndex || _conditionId >=
startConditionIndex + conditionCount) {
        return false; // Invalid stage index
    }

    ClaimCondition memory currentClaimPhase =
claimCondition.conditions[_conditionId];

    if (currentClaimPhase.merkleRoot == bytes32(0)) {
        uint256 supplyClaimedByWallet =
claimCondition.supplyClaimedByWallet[_conditionId][_wallet];

        bool isEligible =
            (maxSupply > _totalMinted()) &&
            (currentClaimPhase.supplyClaimed < currentClaimPhase.maxMint)
&&
            (supplyClaimedByWallet < currentClaimPhase.maxPerWallet);
        return isEligible;
    }

    return MerkleProof.verify(_allowlistProof.proof,
currentClaimPhase.merkleRoot, keccak256(abi.encodePacked(_wallet,
_allowlistProof.quantityLimitPerWallet)));
}
```

Alternatively, provide a separate `isEligibleToMintFull` function that performs comprehensive checks including supply, timing, and per-wallet limits.

Status

Fixed - The developer has clarified that `isEligibleToMint` is intentionally an allowlist-only check, and added a new `isEligibleToMintFull` function that provides comprehensive eligibility validation aligned with `_validateMint` logic.

Fix commit: 39eb4df

Conclusion

This security audit of the Poply protocol identified several areas for improvement across both the marketplace and NFT drop contracts. All identified issues have been promptly addressed and resolved by the development team. The protocol demonstrates excellent responsiveness to security concerns and maintains good security practices overall.

Key improvements implemented include:

- Escrow protection for pending marketplace offers
- Graceful handling of failed royalty payments
- Buyer protection through excess payment refunds
- Fee cap enforcement to prevent misconfigurations
- Commit-reveal scheme for NFT metadata protection

- Enhanced eligibility checking for NFT minting

The development team's commitment to addressing these findings demonstrates a strong security-focused approach to smart contract development.

This report was prepared by 33Audits & Co and represents our independent security assessment of the Poply protocol smart contracts.