Origin Protocol

Security Assessment

November 28, 2018

Prepared For:
Josh Fraser | Origin Protocol
josh@originprotocol.com

Prepared By:
Robert Tonic | *Trail of Bits*
robert.tonic@trailofbits.com

Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Josselin Feist | *Trail of Bits*
josselin@trailofbits.com

Changelog:

November 28th, 2018:	Initial report delivered
September 13th, 2019:	Added Appendix D with retest results

Executive Summary

Between November 13 and November 28, Trail of Bits assessed Origin Protocol's `OriginToken`, `Marketplace`, and migration contracts. Two engineers conducted this assessment over the course of four person-weeks.

The assessment focused on the independent operation of the `OriginToken` and `Marketplace` systems, as well as their components which interacted with each other. Additionally, the upgrade pattern for the `OriginToken` contract was reviewed for scenarios which could cause problems in the `Marketplace` system.

The assessment's first week was spent familiarizing with the Origin Protocol smart contracts and the associated components. This entailed building the components and learning the expected use of each. Once the systems were built and base knowledge was established, manual review of the `Marketplace` and `OriginToken` contracts took place.

The second week was spent developing property tests for the `OriginToken` contract, as well as continuing manual review of the `Marketplace` and `OriginToken` migration contracts. The migration contract was reviewed for issues which could arise before, during, or after an `OriginToken` migration, and their impacts on `Marketplace` operations.

High-severity issues pertained to `Marketplace` arbitrary code execution and system malfunction after contract migration. Issues classified as low- or medium-severity generally pertained to the `Marketplace` contract, where fund loss or theft was possible. Finally, informational-severity issues involved race conditions and pausable contracts.

The discovered vulnerabilities within the `Marketplace`, `OriginToken`, and `TokenMigration` contracts are typical for these types of contracts. The `Marketplace` contract contains a significant number of unique function implementations, which led to proportionately more findings compared to the `OriginToken` contract. The `Marketplace` contract would benefit significantly from unit testing to ensure funds are not trapped or otherwise inaccessible through expected contract operations. On the other hand, the `OriginToken` contract is mostly a standard `ERC20` contract which heavily depends on the `OpenZeppelin` implementation of the standard. The notable implementation of `approveAndCallWithSender` introduces added complexity, but review has not yielded any immediate concerns with this implementation and its intended interaction with the `Marketplace` contract. The primary area of concern regarding the `OriginToken` is its migration strategy, as the current implementation will break `Marketplace` operations.

We believe that all findings related to the `Marketplace` contract should be remediated, and unit tests should be developed to further test currency-related operations. Redeployment of the `Marketplace` contract will be necessary for these remediations. Additionally, further development and testing of the migration pattern for both the `OriginToken` and `Marketplace` contracts should take place, with special emphasis on the operations which depend on each other. Once these steps are taken, we strongly recommend another assessment before a live production system is released.

Update: On September 13, 2019 Trail of Bits reviewed the fixes proposed by Origin Protocol of the issues presented in this report. The fixes were either fixed or their risk was accepted. The detailed fixes log is present in Appendix D.

Project Dashboard

Application Summary

Name	Origin Protocol
Version	4b31657825523962434c6d4e4c61feb0bb1ee518
Type	NodeJS, Solidity
Platforms	Ethereum

Engagement Summary

Dates	November 13 to 28, 2018
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	4	■■■■
Total Medium-Severity Issues	4	■■■■
Total Low-Severity Issues	1	■
Total Informational-Severity Issues	2	■■
Total	11	

Category Breakdown

Access Controls	3	■■■
Timing	2	■■
Undefined Behavior	1	■
Patching	3	■■■
Data Validation	2	■■
Total	11	

Engagement Goals

The engagement was scoped to provide a security assessment of Origin Protocol's smart contracts. This included an `ERC20` token implementation and its migration contract, and a `Marketplace` contract.

Specifically, we sought to answer the following questions:

- Is there any way for an unauthorized user to access user or `Marketplace` escrow funds?
- Could funds become trapped?
- Is there any way to prevent a listing from being finalized?
- Is there any property broken during or after a contract migration occurs?

Coverage

This review included the `OriginToken` contract and its migration contract, along with a `Marketplace` contract. The master branch at commit `4b31...e518` was used during the assessment.

The `Marketplace` contract was reviewed for flaws related to buyer and seller operations such as creating, deleting, and modifying offers and listings. The dispute workflow was also assessed to ensure correctness and vulnerability to manipulation by a malicious buyer or seller.

The `OriginToken` contract was reviewed for flaws related to the `ERC20` implementation, as well as the implemented `ERC827` `approveAndCallWithSender` function which was added to perform `Marketplace` contract operations dependent on `OriginTokens`. An Echidna test harness was also developed to assist testing of the `OriginToken` contract properties.

Recommendations Summary

Short Term

❑ **Define a method of flagging a listing as withdrawn.** By defining this flag, duplicate withdrawals can be prevented if checked beforehand.

❑ **Encourage executing dispute transactions with a high gas price.** This encourages miner prioritization of the dispute transaction.

❑ **Ensure the affiliate address is not `0x0` or the `Marketplace` contract address.** This will help prevent errors on `transfer`, and prevent the `Marketplace` from receiving funds it cannot withdraw.

❑ **Ensure documentation makes `OriginToken` users aware of the `ERC20` `approve` race condition.** Use of the `increaseApproval` and `decreaseApproval` functions should be suggested.

❑ **Implement a method for a seller or arbitrator to revoke an offer after it has been accepted without external contract calls.** Removing this dependence will help prevent situations where a malicious contract could break the dispute or finalization process.

❑ **Refund the deposit when an offer is finalized by the seller.** This will prevent deposit funds being lost in the `Marketplace`.

❑ **Do not depend on the listing to withdraw an offer.** A user should be able to withdraw an offer regardless of the status of the listing.

❑ **Add safeguards to `TokenMigration.`** Check that both contracts are paused during a migration.

Long Term

❑ **Consider allowing a window of time for disputes to be submitted.** This will prevent a buyer from finalizing before the seller can dispute an accepted offer.

❑ **Ensure the affiliate address is under active control and is aware of the affiliate status of the offer.** Implement a function requiring the affiliate to claim his or her commission.

❑ **Consider if `approveAndCallWithSender` should be modified to use `increaseApproval` and `decreaseApproval`.** This will reduce exposure to the `approve` race condition in the `ERC20` standard.

❑ **Prevent arbitrary `ERC20` contracts from being used as currency.** Consider using a whitelist of allowed contract addresses.

❑ **Decouple the listing and offer withdrawal logic.** This could prevent issues where dependant logic leads to broken offer or listing states.

❑ **Ensure the `Marketplace` contract can be paused.** This will ensure safety and consistency in the event of malicious activity or a required upgrade.

❑ **Use unit testing to identify situations which lead to trapped funds.** Ensure that no token is left once all listings are finalized or canceled.

❑ **Create tests to simulate migrations in different scenarios.** This can be used to simulate situations such as when one or both token contracts are left unpaused.

❑ **Perform an external audit of a contract prior its deployment.** Deploying a contract before it has been audited could result in contract abuse or costly re-deployment operations.

Findings Summary

#	Title	Type	Severity
1	Marketplace OGN balance is drainable through withdrawListing	Access Controls	High
2	Disputes are front-runnable by a buyer	Timing	Medium
3	Remote code execution through arbitrary ERC20 implementation	Undefined Behavior	High
4	ERC20 approve race conditions	Timing	Informational
5	Marketplace contract can trap funds if the whitelist is disabled	Access Controls	Medium
6	OriginToken contract migration breaks Marketplace offer references	Patching	High
7	Withdrawn listing prevents seller from withdrawing submitted offers	Data Validation	Low
8	Seller finalization of an offer with an affiliate and commission results in trapped funds	Access Controls	Medium
9	OriginToken migration while unpaused leads to inconsistent state	Patching	Medium
10	Marketplace cannot be paused	Patching	Informational
11	Tokens with external code execution can lead to theft of tokens through reentrancy	Data Validation	High

1. Marketplace OGN balance is drainable through withdrawListing

Severity: High
Difficulty: Low
Type: Access Controls
Finding ID: TOB-Origin-001
Target: `origin-contracts/contracts/marketplace/v00/Marketplace.sol`

Description

The `Marketplace` contract allows for the submission and withdrawal of listings by users. When submitting a listing, an initial deposit may be made in `OGN`, and a deposit manager address is assigned.

```
function _createListing(
    address _seller,
    bytes32 _ipfsHash,  // IPFS JSON with details, pricing, availability
    uint _deposit,      // Deposit in Origin Token
    address _depositManager // Address of listing depositManager
)
    private
{
    /* require(_deposit > 0); // Listings must deposit some amount of Origin Token */
    require(_depositManager != 0x0, "Must specify depositManager");

    listings.push(Listing({
        seller: _seller,
        deposit: _deposit,
        depositManager: _depositManager
    }));
    if (_deposit > 0) {
        tokenAddr.transferFrom(_seller, this, _deposit); // Transfer Origin Token
    }
    emit ListingCreated(_seller, listings.length - 1, _ipfsHash);
}
```

Figure 1: The internal `_createListing` *function, used to create a listing by public methods.*

When a listing is subsequently withdrawn by the listing's deposit manager, the amount of deposited `OGN` is transferred from the `Marketplace`'s account to an account of the deposit manager's choosing:

```
function withdrawListing(uint listingID, address _target, bytes32 _ipfsHash) public {
    Listing storage listing = listings[listingID];
    require(msg.sender == listing.depositManager, "Must be depositManager");
    require(_target != 0x0, "No target");
    tokenAddr.transfer(_target, listing.deposit); // Send deposit to target
    emit ListingWithdrawn(_target, listingID, _ipfsHash);
}
```

Figure 2: The public `withdrawListing` *function, used to withdraw a listing.*

Because there is no check on whether a listing has been withdrawn previously, the withdrawal by the deposit manager can be repeated, draining the `Marketplace` `OGN` account balance.

Exploit Scenario

Bob has an account with 20 `OGN`. The `Marketplace` contract has 100 `OGN` associated to its address. Bob approves the `Marketplace` contract to manage 20 of his account's `OGN`. Bob then creates a listing with a deposit of 20 `OGN`, and assigns himself as the listing deposit manager. Bob subsequently withdraws the same listing 6 times into his account, resulting in Bob recovering the initially deposited 20 `OGN`, and draining the `Marketplace` contract's original 100 `OGN`.

Recommendation

Define a method of flagging a listing as withdrawn. Check this flag is not set before performing a withdrawal.

2. Disputes are front-runnable by a buyer

Severity: Medium Difficulty: High
Type: Timing Finding ID: TOB-Origin-002
Target: `origin-contracts/contracts/marketplace/v00/Marketplace.sol`

Description
When a listing's offer has been accepted, there is a period of time in which only the buyer may finalize the offer (finalization window). However, either the buyer or the seller has the ability to initiate a dispute during the finalization window.

```
function dispute(uint listingID, uint offerID, bytes32 _ipfsHash) public {
    Listing storage listing = listings[listingID];
    Offer storage offer = offers[listingID][offerID];
    require(
        msg.sender == offer.buyer || msg.sender == listing.seller,
        "Must be seller or buyer"
    );
    require(offer.status == 2, "status != accepted");
    require(now <= offer.finalizes, "Already finalized");
    offer.status = 3; // Set status to "Disputed"
    emit OfferDisputed(msg.sender, listingID, offerID, _ipfsHash);
}
```

Figure 1: The `dispute` *function, allowing a buyer or seller to submit a dispute regarding an accepted offer.*

Because the buyer is the only one able to finalize an offer during the finalization window, a buyer could trick a seller into accepting an offer.

```
function finalize(uint listingID, uint offerID, bytes32 _ipfsHash) public {
    Listing storage listing = listings[listingID];
    Offer storage offer = offers[listingID][offerID];
    if (now <= offer.finalizes) { // Only buyer can finalize before finalization window
        require(
            msg.sender == offer.buyer,
            "Only buyer can finalize"
        );
    } else { // Allow both seller and buyer to finalize if finalization window has passed
        require(
            msg.sender == offer.buyer || msg.sender == listing.seller,
            "Seller or buyer must finalize"
        );
    }
    ...
}
```

Figure 2: The `finalize` *function, defining the finalization window which allows only a buyer to finalize a listing with an offer.*

The buyer then sees that the seller has submitted a transaction to dispute the accepted offer, and subsequently submits a transaction to finalize the offer with a higher gas price, resulting in the finalization transaction being mined before the dispute.

Exploit Scenario
Alice has a listing on the Origin market. Bob submits an offer to Alice's listing, and tricks Alice into accepting the offer. Alice realizes she was tricked, and attempts to submit a dispute transaction. Bob observes Alice's dispute transaction and submits a finalization transaction with a higher gas price so that it is processed first, invalidating Alice's dispute.

Recommendation
Short term, set disputes' submission at a high gas price to encourage miner prioritization of the transaction.

Long term, consider allowing a window of time for disputes to be submitted after an offer has been accepted.

3. Remote code execution through arbitrary ERC20 implementation

Severity: High
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-Origin-003
Target: `origin-contracts/contracts/marketplace/v00/Marketplace.sol`

Description
When a user creates an offer, the address to a deployed implementation of the `ERC20` interface is accepted as the `_currency` parameter.

```
function makeOffer(
    uint listingID,
    bytes32 _ipfsHash,   // IPFS hash containing offer data
    uint _finalizes,     // Timestamp an accepted offer will finalize
    address _affiliate,  // Address to send any required commission to
    uint256 _commission, // Amount of commission to send in Origin Token if offer finalizes
    uint _value,         // Offer amount in ERC20 or Eth
    ERC20 _currency,     // ERC20 token address or 0x0 for Eth
    address _arbitrator  // Escrow arbitrator
)
    public
    payable
{
...

    offers[listingID].push(Offer({
        status: 1,
        buyer: msg.sender,
        finalizes: _finalizes,
        affiliate: _affiliate,
        commission: _commission,
        currency: _currency,
        value: _value,
        arbitrator: _arbitrator,
        refund: 0
    }));

    if (address(_currency) == 0x0) { // Listing is in ETH
        require(msg.value == _value, "ETH value doesn't match offer");
    } else { // Listing is in ERC20
        require(msg.value == 0, "ETH would be lost");
        require(
            _currency.transferFrom(msg.sender, this, _value),
            "transferFrom failed"
        );
    }

    emit OfferCreated(msg.sender, listingID, offers[listingID].length-1, _ipfsHash);
}
```
Figure 1: The public `makeOffer` function, used for creating an offer for a listing.

Subsequently, the `_currency` parameter is stored in the listing struct, and used as the method of payment for the offer.

```
function paySeller(uint listingID, uint offerID) private {
    Listing storage listing = listings[listingID];
    Offer storage offer = offers[listingID][offerID];
    uint value = offer.value - offer.refund;

    if (address(offer.currency) == 0x0) {
        require(offer.buyer.send(offer.refund), "ETH refund failed");
        require(listing.seller.send(value), "ETH send failed");
    } else {
        require(
            offer.currency.transfer(offer.buyer, offer.refund),
            "Refund failed"
        );
        require(
            offer.currency.transfer(listing.seller, value),
            "Transfer failed"
        );
    }
}
```

Figure 2: The `paySeller` function, used to transfer the `ERC20` tokens offered to the seller.

A malicious `ERC20` implementation could prevent an offer from finalizing, since the logic for finalization relies on a successful `transfer` call.

Exploit Scenario

Alice creates a listing on the `Marketplace`. Bob then submits an offer using a custom `MaliciousERC20` contract. Alice accepts Bob's offer. Bob does not finalize the offer. Alice waits for the finalization period to expire to finalize the offer. Bob uses the `MaliciousERC20` contract to revert on every call to `transfer`, resulting in an offer that can't be finalized.

Recommendation

Short term, implement a method for either a seller or arbitrator to revoke an offer after it has been accepted, without performing calls to the `_currency` contract.

Long term, prevent arbitrary `ERC20` contracts from being used for the currency method. Consider using a whitelist of allowed contract addresses.

4. ERC20 approve race conditions

Severity: Informational Difficulty: High
Type: Timing Finding ID: TOB-Origin-004
Target: `origin-contracts/contracts/token/OriginToken.sol`

Description
Origin conforms to the `ERC20` token standard, which contains an unavoidable race condition. This race condition is only exploitable by sophisticated attackers, but could result in loss of funds for Origin users.

The `ERC20` standard requires two functions, `approve` and `transferFrom`, which allow users to designate other trusted parties to spend funds on their behalf. Calls to any Ethereum function, including these, are visible to third parties prior to confirmation on-chain. A sophisticated attacker can front-run them and insert their own transactions to occur before the observed calls.

The `approve` function is defined to take an address and an amount, and set that address's allowance to the specified amount. Then, that address can call `transferFrom` and move up to their allowance of tokens as if they were the owners. However, `approve` is specified to be idempotent. It sets the approval to a new value regardless of its prior value; it doesn't modify the allowance.

Exploit Scenario
Alice, a non-malicious user, has previously approved Bob, a malicious actor, for 100 `OGN`. She wishes to increase his approval to 150. Bob observes the `approve(bob, 150)` transaction prior to its confirmation and front-runs it with a `transferFrom(alice, bob, 100)`. Then, as soon as the new approval is in, his allowance is set to 150 and he can call `transferFrom(alice, bob, 150)`. Alice believes she's setting Bob's allowance to 150, and he can only spend 150 tokens. Due to the race condition, Bob can spend 250 `OGN`.

Recommendation
Short term, ensure documentation makes users aware of this issue and that they may use the `increaseApproval` and `decreaseApproval` functions inherited from the OpenZeppelin token contracts.

Long term, consider if `approveAndCallWithSender` should be modified to use `increaseApproval` and `decreaseApproval` as well.

5. Marketplace contract can trap funds if the whitelist is disabled

Severity: Medium
Difficulty: Medium
Type: Access Controls
Finding ID: TOB-Origin-005
Target: `origin-contracts/contracts/marketplace/v00/Marketplace.sol`

Description

The `Marketplace` contract protects listings from receiving offers from unauthorized affiliates through the use of an affiliate whitelist. The `makeOffer` function checks to see if the provided affiliate is `0x0` to prevent trapping tokens in the `Marketplace` contract.

```
function makeOffer(
    uint listingID,
    bytes32 _ipfsHash,   // IPFS hash containing offer data
    uint _finalizes,     // Timestamp an accepted offer will finalize
    address _affiliate,  // Address to send any required commission to
    uint256 _commission, // Amount of commission to send in Origin Token if offer finalizes
    uint _value,         // Offer amount in ERC20 or Eth
    ERC20 _currency,     // ERC20 token address or 0x0 for Eth
    address _arbitrator  // Escrow arbitrator
)
    public
    payable
{
    bool affiliateWhitelistDisabled = allowedAffiliates[address(this)];
    require(
        affiliateWhitelistDisabled || allowedAffiliates[_affiliate],
        "Affiliate not allowed"
    );

    if (_affiliate == 0x0) {
        // Avoid commission tokens being trapped in marketplace contract.
        require(_commission == 0, "commission requires affiliate");
    }
...
```

Figure 1: The `makeOffer` function's affiliate whitelist logic.

However, if the whitelist is disabled, the `Marketplace` contract address could be provided as the affiliate address, thus the commission is distributed to the `Marketplace` contract upon offer finalization.

```
function payCommission(uint listingID, uint offerID) private {
    Offer storage offer = offers[listingID][offerID];
    if (offer.affiliate != 0x0) {
        require(
            tokenAddr.transfer(offer.affiliate, offer.commission),
            "Commission transfer failed"
        );
    }
}
```

Figure 2: The `payCommission` *function, with the check to see if the affiliate is 0x0.*

Exploit Scenario
Alice creates a listing using the `Marketplace` contract, which does not have an active whitelist. Bob then submits an offer for Alice's listing, including the `Marketplace` contract address as the affiliate address. Alice accepts the offer, which Bob finalizes. Upon finalization, the commission `OGN` is distributed to the `Marketplace` contract balance.

Recommendation
Short term, ensure the affiliate address is not `0x0`, nor the address of the `Marketplace` contract.

Long term, ensure the affiliate address is under active control and is aware of the affiliate status of the offer. Consider implementing an `acceptCommission` or similar function, requiring the affiliate to claim their commission.

6. OriginToken contract migration breaks Marketplace offer references

Severity: High
Difficulty: Low
Type: Patching
Finding ID: TOB-Origin-006
Target: `origin-contracts/contracts/token/TokenMigration.sol`

Description

The `OriginToken` has a `TokenMigration` contract which performs a migration of balances from the previous version of the `OriginToken` to the newly deployed contract. During the migration, users are unable to interact with the previously deployed `OriginToken`; the contract is in a `paused` state. While users may resume operations on the newly deployed `OriginToken` after migration, the `Marketplace` contract does not update the references to the previously deployed `OriginToken` address to the newly deployed address, resulting in broken functionality for listings and offers.

After migration, the previous `OriginToken` contract remains in a paused state. This breaks the `Marketplace` listing and offer operations since no entity can perform operations involving the previous--paused--`OriginToken` contract. If the previous contract is subsequently unpaused, the listings and offers can be finalized, but the operations would be performed on the previous `OriginToken`, which is no longer relevant.

Exploit Scenario

Alice creates a listing with a deposit in `OriginToken` on the `Marketplace` contract. Charlie submits an offer for Alice's listing using `OriginToken` as the currency. The `OriginToken` contract is subsequently upgraded using the `TokenMigration` contract. The upgrade prevents the listing from completing, thus Charlie's offer funds and Alice's deposit are unrecoverable.

Recommendation

Short term, the `Marketplace` contract should have a mechanism for updating the contract address being used for listing deposits and offer currency. Validation should occur to ensure offers and listings are completable after this update.

Long term, develop and test a migration strategy for both the `OriginToken` and `Marketplace` contracts.

7. Withdrawn listing prevents seller from withdrawing submitted offers

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-Origin-007
Target: `origin-contracts/contracts/marketplace/v00/Marketplace.sol`

Description

According to GitHub issue 930 and related to the earlier implementations of the `withdrawListing` function, when withdrawing a listing, the listing should be deleted from the array of `listings`, similar to how offers are deleted after withdrawal in the `withdrawOffer` function.

```
function withdrawListing(uint listingID, address _target, bytes32 _ipfsHash) public {
    Listing storage listing = listings[listingID];
    require(msg.sender == listing.arbitrator);
    require(_target != 0x0);
    tokenAddr.transfer(_target, listing.deposit); // Send deposit to target
    delete listings[listingID]; // Remove data to get some gas back
    emit ListingWithdrawn(_target, listingID, _ipfsHash);
}
```

Figure 1: The previous `withdrawListing` *function definition on commit* `c714...9245`

If the listing data is deleted in this way the `listings` array will return an invalid value when referenced. This causes logic to fail when the seller attempts to withdraw an offer. This applies similarly to any other function which references a `listing` attribute. In particular, the `finalize` and `executeRuling` can also result in funds being lost in the `Marketplace`.

```
function withdrawOffer(uint listingID, uint offerID, bytes32 _ipfsHash) public {
    Listing storage listing = listings[listingID];
    Offer storage offer = offers[listingID][offerID];
    require(
        msg.sender == offer.buyer || msg.sender == listing.seller,
        "Restricted to buyer or seller"
    );
    require(offer.status == 1, "status != created");
    refundBuyer(listingID, offerID);
    emit OfferWithdrawn(msg.sender, listingID, offerID, _ipfsHash);
    delete offers[listingID][offerID];
}
```

Figure 2: The `withdrawOffer` *function, showing the dependence on the listings index.*

Exploit Scenario

Charlie creates a listing on the `Marketplace` contract. Alice submits an offer to Charlie's listing. Her offer funds are transferred to the `Marketplace` contract address as an escrow account. Charlie subsequently withdraws his listing. Due to the deletion of the listing from the `listings` array, Charlie may no longer withdraw Alice's offer and refund her funds.

Recommendation
Short term, do not depend on the listing to withdraw an offer. A user should be able to withdraw an offer regardless of the listing's status.

Long term, decouple the listing and offer withdrawal logic.

8. Seller finalization of an offer with an affiliate and commission results in trapped funds

Severity: Medium   Difficulty: Low
Type: Access Controls   Finding ID: TOB-Origin-008
Target: `origin-contracts/contracts/marketplace/v00/Marketplace.sol`

Description
When a buyer submits a listing offer with a commission value and it is subsequently accepted by the seller, the commission is subtracted from the listing's deposit, leaving the commission escrowed in the `Marketplace` contract balance. Upon finalization of the offer by the buyer, the affiliated listed on the offer receives the allocated commission from the `Marketplace`'s balance.

```
function acceptOffer(uint listingID, uint offerID, bytes32 _ipfsHash) public {
    Listing storage listing = listings[listingID];
    Offer storage offer = offers[listingID][offerID];
    ...
    listing.deposit -= offer.commission; // Accepting an offer puts Origin Token into escrow
    ...
}
```

Figure 1: An excerpt of the `acceptOffer` function, highlighting the commission reduction from the deposit.

However, if the finalization period passes and the seller finalizes the offer in place of the buyer, the affiliate does not receive the commission. This is intended behavior. However, the commission is trapped in the `Marketplace`'s balance. There's no way to return the commission to the listing's deposit pool.

```
function finalize(uint listingID, uint offerID, bytes32 _ipfsHash) public {
    Listing storage listing = listings[listingID];
    Offer storage offer = offers[listingID][offerID];
    ...
    paySeller(listingID, offerID); // Pay seller
    if (msg.sender == offer.buyer) { // Only pay commission if buyer is finalizing
        payCommission(listingID, offerID);
    }
    ...
}
```

Figure 2: An excerpt of the `finalize` function, highlighting the lack of commission withdrawal upon seller finalization.

Exploit Scenario
Alice creates a listing on the `Marketplace`. Bob submits an offer for Alice's listing with an affiliate and commission, which is subsequently accepted. Bob does not finalize within the

finalization window, allowing Alice to finalize the offer. Because Alice finalized the offer, the commission is never paid for the offer. The commission cannot be retrieved from the `Marketplace`’s account balance.

Recommendation
Short term, refund the listing deposit upon offer finalization by the seller.

Long term, use unit testing to ensure that no token is left once all listings are finalized or canceled.

9. OriginToken migration while unpaused leads to inconsistent state

Severity: Medium Difficulty: Medium
Type: Patching Finding ID: TOB-Origin-009
Target: `origin-contracts/contracts/token/{OriginToken,TokenMigration}.sol`

Description
Comments in the `TokenMigration` source code indicate a migration should only performed while both source and destination tokens contracts are in a paused state. However, the migration contract does not check or enforce this condition which may lead to inconsistencies in `OriginToken` balances after the migration.

```
/**
 * @title Migrates balances from one token contract to another
 * @dev Migrates all balances from one token contract to another. Both contracts
 * must be pausable (to prevent changes during migration), and the target
 * contract must support minting tokens.
 */
contract TokenMigration is Ownable {
```

Figure 1: Comments from `TokenMigration.sol` *describing* `OriginToken` *state requirements for migration*

Exploit Scenario
Origin initiates a migration of `OriginToken` balances using the `TokenMigration` contract but forgets to pause the original `OriginToken` contract. After Alice’s balance of `OGN` has been migrated but before Bob’s has, she transfers 100 `OGN` to Bob. When Bob’s balance is eventually migrated, this will result in 100 extra `OGN` being minted and the `totalSupply` will no longer match the true total amount of `OGN` in circulation.

Recommendation
Short term, add safeguards to `TokenMigration.sol` to check that both contracts are paused during migration.

Long term, create tests to simulate migrations in different scenarios, such as when one or both token contracts are left unpaused.

10. Marketplace cannot be Paused

Severity: Informational
Difficulty: Low
Type: Patching
Finding ID: TOB-Origin-010
Target: `origin-contracts/contracts/marketplace/v00/Marketplace.sol`

Description
In the event of malicious activity or required upgrade, there is currently no way to pause the `Marketplace` contract.

Exploit Scenario
Bob is using Alice's `Marketplace` v0 contract. Alice decides to deploy a new version of the `Marketplace`, v1. Bob is still able to trade on the `Marketplace` v0 due to Alice's inability to pause the contract.

Recommendation
Ensure the `Marketplace` can be paused safely in the event of malicious activity or a required upgrade.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code Quality

Code-quality findings are included to detail findings which are not considered to be immediate security concerns, but could lead to the introduction of vulnerabilities in the future.

contracts/marketplace/v00/Marketplace.sol

- General
    - Repeated `require` logic should be moved into function modifiers.
    - Similar `require()` revert messages should be generalized to prevent confusion.
    - Ensure function return types are explicitly stated for all functions.
    - Consider a more pull-based design for fund transfers. Streamlining could result in confusing movement of funds for buyers and sellers.
- Specific
    - Move the affiliate whitelist logic in the `makeOffer` function into its own modifier. This makes the logic re-usable, and reduces logic irrelevant to the process of making an offer from the `makeOffer` function.
    - Consider using `SafeMath` for numeric operations. Native numeric operations could lead to value over- and under-flows.
    - Line 111 of `_createListing` includes a commented-out line of code which requires the deposit to be greater than zero. This should be removed if it is now irrelevant. Commented code should not be committed.
    - Ensure return values are checked for external `ERC20` calls. Not doing so could result in failed calls silently passing as successful.

contracts/token/TokenMigration.sol

- General
    - Lack of approval migration will require users to explicitly approve the `Marketplace` contract to complete listings and offers, even if the `Marketplace` allows updating of the offer `currency` address.

C. Automated ERC20 Property Testing

Trail of Bits used Echidna, our property-based testing framework, to find security issues and logic errors in the Solidity components of Origin Protocol. During the assessment we used Echidna to seek a sequence of transactions that would break important properties in the `OriginToken` contract, such as transfers between addresses not on the transactor whitelist, as well as standard `ERC20` properties, such as transferring unauthorized funds or a normal user minting new `OGN`. This suite of tests resulted in no new findings.

The suite of test files includes:

- `origin_success_noowner.sol`
- `origin_success_owner.sol`
- `origin_throw_noowner_paused.sol`
- `origin_throw_noowner_unpaused.sol`
- `origin_throw_owner_unpaused.sol`
- `origin_sucess.yaml`
- `origin_throw.yaml`

To run the tests, first ensure Echidna is installed either locally or as a docker image. Execute the following command with each Echidna test file and corresponding configuration file (e.g., files starting with `origin_success` use `origin_success.yaml` as the config file).

An example invocation, using the docker version of Echidna:

```
$ docker run -v `pwd`:/src trailofbits/echidna echidna-test
/src/origin_throw_noowner_paused.sol TEST --config /src/origin_throw.yaml
```

D. Fix Log

On September 13, 2019 Trail of Bits reviewed the fixes proposed by Origin Protocol of the issues presented in this report. The fixes were included in V01_Marketplace.sol and bd6026033. Origin Protocol deployed the fixed version on mainet at 0x698ff47b84837d3971118a369c570172ee7e54c2.

The Origin protocol addressed or accepted all discovered issues in their codebase as a result of the assessment. Of those issues, 4 were remediated and 6 were risk accepted. Some of the issues for which the risk is accepted rely on off-chain mitigations, that were not reviewed by Trail of Bits. We reviewed each of the fixes to understand the strength of the correctness of the proposed remediation.

Additionally, Trail of Bits identifies a new issue in the codebase: TOB-Origin-011. This issue is a reentrancy that can occur if a token with external code execution capacity (such as an ERC223 token) is used. This issue was fixed in bd6026033.

ID	Title	Severity	Status
01	Marketplace OGN balance is drainable through withdrawListing	High	Fixed
02	Disputes are front-runnable by a buyer	Medium	Risk Accepted
03	Remote code execution through arbitrary ERC20 implementation	High	Risk Accepted
04	ERC20 approve race conditions	Informational	Fixed
05	Marketplace contract can trap funds if the whitelist is disabled	Medium	Risk Accepted
06	OriginToken contract migration breaks Marketplace offer references	High	Risk Accepted
07	Withdrawn listing prevents seller from withdrawing submitted offers	Low	Fixed
08	Seller finalization of an offer with an affiliate and commission results in trapped funds	Medium	Fixed
09	OriginToken migration while unpaused leads to inconsistent state	Medium	Risk Accepted

10	Marketplace cannot be paused	Informational	Risk Accepted
11	Tokens with external code execution can lead to theft of tokens through reentrancy	High	Fixed

Detailed Fix Log

Finding 1: Marketplace OGN balance is drainable through withdrawListing
Fixed.

Finding 2: Disputes are front-runnable by a buyer
Risk Accepted.
The Origin Protocol team stated that:

> *The buyer doesn't gain anything by doing this, so there is no need to fix.*

Finding 3: Remote code execution through arbitrary ERC20 implementation
Risk Accepted.
The Origin Protocol team stated that:

> *A whitelist of allowed ERC20 tokens will be implemented at the client Layer.*

In addition to trapping the contract, Trail of Bits recommends to be careful with the capacity for an attacker to mint GasToken through the market..

Finding 4: ERC20 approve race conditions
Fixed.
The Origin Protocol team added user documentation to ensure they are aware of the race condition.

Finding 5: Marketplace contract can trap funds if the whitelist is disabled
Risk Accepted.
The Origin Protocol team stated that:

> *This is only possible when bypassing the Origin client layer.*

Finding 6: OriginToken contract migration breaks Marketplace offer references
Not fixed.
The Origin Protocol team stated that:

> *There seem to be no dangling references to the old token contract once marketplace.setTokenAddr() is called with the new contract address. However the issue exists for offers made with an ERC20 which is subsequently paused or upgraded, so an*

> *effort to reach out to affected users would have to be taken were this to ever happen. This is not just true of OGN as offers can be made with any ERC20 token.*

The issue affects also a migration on the currency token. If the token is migrated while offers were created but not finalized, the on-going offers will not reflect the change of the token address.

Finding 7: Withdrawn listing prevents seller from withdrawing submitted offers
Fixed.
The Origin Protocol team stated that:

> *This referenced an earlier version of the code and is not valid for the deployed version.*

Finding 8: Seller finalization of an offer with an affiliate and commission results in trapped funds
Fixed
Line 317 of V01_Marketplace.sol

Finding 9: OriginToken migration while unpaused leads to inconsistent state
Risk Accepted.
The Origin Protocol team stated that:

> *Origin will ensure token is paused in this scenario.*

Finding 10: Marketplace cannot be paused
Risk Accepted.
The Origin Protocol team stated that:

> *Currently, we don't intend for the marketplace contract to be pausable*

Finding 11: Tokens with external code execution can lead to theft of tokens through reentrancy
Fixed in bd6026033.

Trail of Bits noted that reentrancies in the fixed version might still lead to generating incorrect event orders, which could confuse off-chain tools.

11. Tokens with external code execution can lead to theft of tokens through reentrancy

Severity: High
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-Origin-011
Target: `V01_Marketplace.sol`

Description

`Marketplace` does not follow the Checks-Effects-Interactions pattern for its interaction with the currency and the `tokenAddr` tokens. As a result, an attacker can drain the contract funds through a reentrancy.

All the interactions with the currency contracts are done before changing the state of `MarketPlace`. For example, if the buyer wants to refund its offer, the tokens will be sent before the order is deleted:

```
function refundBuyer(uint listingID, uint offerID) private {
    Offer storage offer = offers[listingID][offerID];
    if (address(offer.currency) == 0x0) {
        require(offer.buyer.send(offer.value), "ETH refund failed");
    } else {
        require(
            offer.currency.transfer(offer.buyer, offer.value),
            "Refund failed"
        );
    }
}
```

Figure 1: V01_Marketplace.sol#L376-L386

```
function withdrawOffer(uint listingID, uint offerID, bytes32 _ipfsHash) public {
    Listing storage listing = listings[listingID];
    Offer storage offer = offers[listingID][offerID];
    require(
        msg.sender == offer.buyer || msg.sender == listing.seller,
        "Restricted to buyer or seller"
    );
    require(offer.status == 1, "status != created");
    refundBuyer(listingID, offerID);
    emit OfferWithdrawn(msg.sender, listingID, offerID, _ipfsHash);
```

```
        delete offers[listingID][offerID];
```

Figure 2: V01_Marketplace.sol#L263-L274

If the currency has external code execution capacity (for example an ERC223 token), an attacker can be able to re-entrer the `withdrawOffer` function to withdraw more funds that deposed.

The situation is similar for the other interactions with the currency contract, including:

- `finalize` -> `paySeller`
- `executeRuling` -> `paySeller`
- `executeRuling` -> `refundBuyer`

Moreover, the interactions with the `tokenAddr` follow the same pattern and are vulnerable, including:

- `withdrawListing` (as shown in the Fix Log)
- `Finalize` -> `payCommission`

Exploit Scenario
Bob's token is an ERC223 listed in the MarketPlace. The MarketPlace contains $10,000 of Bob's token. Eve creates a listing with $1,000 deposit. Eve withdraws the listing and uses the reentrancy the drain the $10,000 tokens.

Recommendation
Short term, ensure all the external contract interactions follow the Checks-Effects-Interactions pattern.

Long term, run continuously Slither or Crytic on the codebase. The issue is detected by Slither.