Tunnl Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Hans

July 1, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

4.1 Overview

The `TunnlTwitterOffers` contract is a decentralized on-chain marketing platform where brands (advertisers) submit offers and Twitter creators accept them. The contract uses Chainlink Functions and Automation to manage offers, escrow funds, verify Twitter content, and handle payouts.

The funds from the advertiser are held in escrow until the Twitter content is verified by AI. Escrowed funds are only released to the creator after the content is successfully verified by AI and has been live for a sufficient duration.

4.2 Actors

- **Advertisers**: Submit offers and provide funds for escrow.
- **Creators**: Accept offers and submit Twitter content for verification.
- **Owner**: The contract deployer. Can add/remove admins.
- **Admins**: Backend handlers who interact with the contract to:
  - Accept an offer and submit relevant requests via Chainlink Functions.
  - Cancel the offer if needed.
  - Manually retry sending functions requests in case of failures.

4.3 Key Features

- **Offer Creation and Acceptance**: Advertisers make offers, and creators can accept them. Once accepted, the funds are locked in escrow.
- **Automated Verification**: Chainlink Functions are used to check Twitter content. The verification process can only give a Yes or No response.

- **Payouts**: After successful verification, funds are released to creators, but only once the offer duration has passed and the content is still live.
- **Admin Controls**: Admins (backend systems) accept offers and submit them for verification through Chainlink Automation.
- **Users**: The main users are Advertisers and Creators.
- **Admin and Owner**: There can be multiple Admins, but only one Owner.
- **Escrowed USDC**: The contract owner or admin collects USDC from the advertiser and holds it in escrow until the offer is completed.
- **Custody**: The protocol does not take control of user funds.
- **Advertiser Percentage Fee**: This is an extra fee added to the offer value, charged to the advertiser.
- **Creator Percentage Fee**: This fee is taken from the amount paid to the creator.

4.4 Offer States

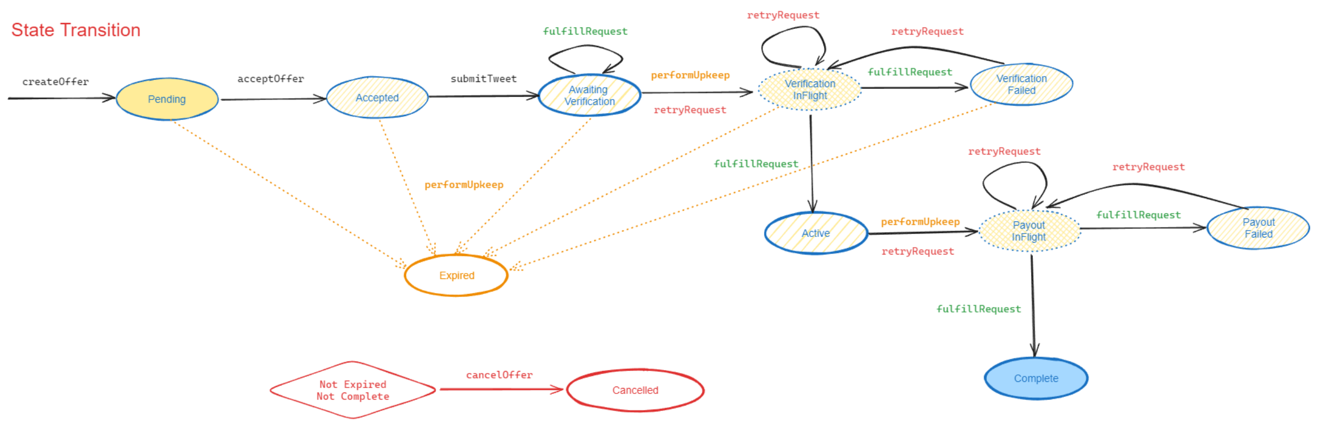

Figure 1: State Transition Diagram

5 Audit Scope

A single Solidity file `TunnlTwitterOffers.sol` is in scope for this audit.

6 Executive Summary

Over the course of 7 days, the Cyfrin team conducted an audit on the Tunnl smart contracts provided by Tunnl. In this period, a total of 9 issues were found.

Summary

Project Name	Tunnl
Repository	main
Commit	685cd12d70d9. . .
Audit Timeline	June 17th - June 25th
Methods	Manual Review

Issues Found

Critical Risk	1
High Risk	0
Medium Risk	5
Low Risk	0
Informational	3
Gas Optimizations	0
Total Issues	9

Summary of Findings

[C-1] Math error in creator payment calculation	Resolved
[M-1] An attacker can flood the protocol with false offers to cause DoS	Acknowledged
[M-2] Malicious creators can force advertisers to pay the flat fee by cancelling accepted offers	Acknowledged
[M-3] Using a wrong fee percentage [Self-Reported]	Resolved
[M-4] Incomplete implementation of state transition from Pending to Expired	Resolved
[M-5] Attackers can prevent users creating an offer by frontrunning	Acknowledged
[I-1] Some variable and argument names are misleading	Resolved
[I-2] Unnecessary duplicate state logic	Resolved
[I-3] Unnecessary use of OpenZeppelin's nonReentrant	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Math error in creator payment calculation

Description: The Tunnl protocol applies 3 kinds of fees.

- **Flat Fee**: The advertiser pays a flat fee and it is charged on acceptance of offers.
- **Advertiser Percentage Fee**: The advertiser percentage fee is the percentage fee that is charged to the advertiser. It is an extra amount added to the offer value.
- **Creator Percentage Fee**: The creator percentage fee is the percentage fee that is charged to the creator. It is charged based on the amount the creator is paid.

The advertisers create offers starting from a maximum offer amount $maxOfferAmount$ and the total amount including all posssible fees are calculated as $maxOfferAmount * (1 + advertiserFee)+flatFee$. This amount is stored in the `Offer::maxValueUsdc` and used in many places for accounting.

After a successful verification, `TunnlTwitterOffers::sendFunctionsRequest()` is called to get the actual amount that should be paid to the creator. This function calls a function `FunctionClient::_sendRequest()` with the request and the maximum offer amount is encoded in the request arguments. But in the calculation of `maxCreatorPayment`, `s_config.creatorFeePercentageBP` is used instead of `s_config.advertiserFeePercentageBP`.

```
        uint256 maxCreatorPayment = uint256((s_offers[offerId].maxValueUsdc -
        ↪   s_offers[offerId].flatFeeUsdc) * 10000)
            / uint256(10000 + s_config.creatorFeePercentageBP);//@audit-issue this should be
            ↪   10000+s_config.advertiserFeePercentageBP

        console.log("maxCreatorPayment: %s", maxCreatorPayment);//@audit-info

        bytes[] memory bytesArgs = new bytes[](4);
        bytesArgs[0] = abi.encode(offerId);
        bytesArgs[1] = abi.encodePacked(s_offers[offerId].creationDate);
        bytesArgs[2] = abi.encodePacked(maxCreatorPayment);//@audit-info maximum offer amount relayed
        bytesArgs[3] = abi.encodePacked(s_offers[offerId].offerDurationSeconds);
        req.setBytesArgs(bytesArgs);

        bytes32 requestId = _sendRequest(
            req.encodeCBOR(),
            s_config.functionsSubscriptionId,
            s_config.functionsCallbackGasLimit,
            s_config.functionsDonId
        );
```

The current implementation does not change the payment according to the performance of the content (e.g. likes, views, ...) and the relayed maximum amount is fully paid to the creator. Hence, the final payment to the creator is being wrong.

This error can lead to two kinds of problems.

- If `creatorFeePercentageBP<advertiserFeePercentageBP`, `maxCreatorPayment` becomes larger than the actual maximum offered amount and the payment distribution in the function `TunnlTwitterOffers::fulfillRequest` will revert with `Exceeds max` error.
- If `creatorFeePercentageBP>advertiserFeePercentageBP`, `maxCreatorPayment` becomes less than the actual maximum offered amount and the creator gets paid less amount.

Note that the current test suite belongs to the first case but the errors are not caught because request fulfillment is simulated with an artificial value rather than the actual value that is set by the nodes using the `calculatepayment.js` script.

```
    function test_PayOut() public {
        // Define the amount to be paid in USDC
        uint256 amountPaidUsdc = uint256(uint256(100e6));
        test_Verification_Success();
        // Warp time for payout and perform Chainlink automation
        vm.warp(block.timestamp + (1 weeks - 100));
        performUpkeep();
        // Fulfill request for payout  with PayOutamount
        mockFunctionsRouter.fulfill(
            address(tunnlTwitterOffers),
            functionsRequestIds[offerId],
            abi.encode(amountPaidUsdc),//@audit-info should be same to the maxCreatorPayout in the
            ↪  current implementation
            ""
        );

        // Assert balances after payout
        assertEq(mockUsdcToken.balanceOf(advertiser), 0);
        assertEq(mockUsdcToken.balanceOf(address(tunnlTwitterOffers)), (0));
        assertEq(
            mockUsdcToken.balanceOf(contentCreator),
            amountPaidUsdc - (amountPaidUsdc * tunnlTwitterOffers.getConfig().creatorFeePercentageBP) /
            ↪  10000
        );
    }
```

Impact: We evaluate the impact to be CRITICAL because the protocol will not function at all or the creator gets less amount than offered systematically.

Proof Of Concept: Because it is not easy to check the actual `maxCreatorPayment` value that is set as a request argument, we inserted a line to log its value in the function `TunnlTwitterOffers::sendFunctionsRequest()`.

```
console.log("maxCreatorPayment: %s", maxCreatorPayment);
```

Running the test case `test_PayOut()` outputs as below.

```
[PASS] test_PayOut() (gas: 531524)
Logs:
  100000000 110000000 <- Max offered amount and maxValueUsdc
  maxCreatorPayment: 102439024 <- This must be the same to the offered amount 100e6
  maxCreatorPayment: 102439024
```

Recommended Mitigation: Fix the `sendFunctionsRequest()` function as below. Note that the team reported another issue in using the fee percentage value from `s_config` instead of the `s_offers[offerId]` and the mitigiation below reflects that as well.

```
        uint256 maxCreatorPayment = uint256((s_offers[offerId].maxValueUsdc -
        ↪  s_offers[offerId].flatFeeUsdc) * 10000)//@audit-info why not use the percentage directly -
        ↪  (s_offers[offerId].maxValueUsdc - s_offers[offerId].flatFeeUsdc) * (10000 -
        ↪  s_config.creatorFeePercentageBP) / 10000
--             / uint256(10000 + s_config.creatorFeePercentageBP);
++            / uint256(10000 + s_offers[offerId].advertiserFeePercentageBP);
```

Tunnl: Fixed in PR 37.

Cyfrin: Verified.

7.2 Medium Risk

7.2.1 An attacker can flood the protocol with false offers to cause DoS

NOTE: This finding assumes the protocol fixed the other issue of incomplete implementation of state transition from Pending to Expired.

Description: The protocol maintains a set of all offers and they are managed by Chainlink automation functions : `checkUpkeep()` and `performUpkeep()`. The function `checkUpkeep()` iterates all offers in `s_offersToUpkeep` and checks the necessity of upkeeping.

```
TunnlTwitterOffers.sol
    function checkUpkeep(bytes calldata) external view override returns (bool upkeepNeeded, bytes memory performData) {
        bytes32[] memory offersToUpkeep = new bytes32[](0);
        for (uint256 i = 0; i < s_offersToUpkeep.length(); i++) {
            bytes32 offerId = s_offersToUpkeep.at(i);
            if (
                _needsVerificationRequest(s_offers[offerId].status, s_offers[offerId].dateToAttemptVerification) ||
                _needsExpiration(s_offers[offerId].status, s_offers[offerId].payoutDate, s_offers[offerId].acceptanceExpirationDate) ||
                _needsPayoutRequest(s_offers[offerId].status, s_offers[offerId].payoutDate)
            ) {
                offersToUpkeep = appendOfferToUpkeep(offersToUpkeep, offerId);
                if (offersToUpkeep.length == s_config.automationUpkeepBatchSize) {
                    return (true, abi.encode(offersToUpkeep));
                }
            }
        }
        return (offersToUpkeep.length > 0, abi.encode(offersToUpkeep));
    }
```

The ones that are necessary to upkeep are added to a local variable `offersToUpkeep` and it is returned once the number of offers to upkeep equals to the `s_config.automationUpkeepBatchSize`.

An offer is necessary to upkeep if it either needs verification / payout function request OR it is expired.

```
TunnlTwitterOffers.sol
    function _needsExpiration(Status offerStatus, uint32 payoutDate, uint32 acceptanceExpirationDate) internal view returns (bool) {
        return (
            (acceptanceExpirationDate < block.timestamp && offerStatus == Status.Pending) ||
            (payoutDate < block.timestamp &&
            (offerStatus == Status.Accepted ||
            offerStatus == Status.AwaitingVerification ||
            offerStatus == Status.VerificationInFlight ||
            offerStatus == Status.VerificationFailed))
        );
    }
```

As we can see in the above, an offer needs expiration if it is still in Pending and current time is later than `acceptanceExpirationDate`.

Now the problem is anyone can create any offers as long as they have enough allowance and there is no lower limit on the `maxPaymentUsdc` (maximum offer amount). So technically, anyone can create any number of offers with zero offer amount as long as they have a single allowance of `flatFeeUSDC`. These false offers will not be accepted by anyone because these will not bring any economical benefits to creators. After all, the offers will stay Pending in the `offersToUpkeep`. An attacker can exploit this by creating tons of false offers so that the Chainlink automation functions can not pick up the correct offer to handle (either verification or payout) and cause Denial-Of-Service.

Impact: We evaluate the impact to be Medium because it does not bring direct economical benefits to the attacker.

Proof Of Concept: The PoC below shows anyone can create any number of offers without approving significant amount.

```
    function test_CreateFalseOffers() public {
        uint32 acceptanceDurationSeconds = tunnlTwitterOffers.getConfig().minAcceptanceDurationSeconds;
        uint offerAmount = 0;
        uint maxValueUsdc = tunnlTwitterOffers.getConfig().flatFeeUsdc;

        vm.startPrank(advertiser);
        mockUsdcToken.mint(advertiser, maxValueUsdc);
        mockUsdcToken.approve(address(tunnlTwitterOffers), maxValueUsdc);//@audit-info single allowance
        ↪  of 5 USDC
        for(uint i = 100; i < 200; i++) {
            offerId = bytes32(i);
            tunnlTwitterOffers.createOffer(offerId, offerAmount, acceptanceDurationSeconds, 4 days);

            //@audit-info verifies the offer is created with the correct values
            assertEq(uint(getOffer(offerId).status), 0);
            assertEq(getOffer(offerId).flatFeeUsdc, tunnlTwitterOffers.getConfig().flatFeeUsdc);
            assertEq(getOffer(offerId).advertiserFeePercentageBP,
            ↪  tunnlTwitterOffers.getConfig().advertiserFeePercentageBP);
            assertEq(getOffer(offerId).creationDate, block.timestamp);
            assertEq(getOffer(offerId).acceptanceExpirationDate, block.timestamp +
            ↪  acceptanceDurationSeconds);
            assertEq(getOffer(offerId).advertiser, advertiser);
        }
        vm.stopPrank();
    }
```

Recommended Mitigation: There are several possible mitigations.

- Limit the number of offers that an address can hold at a time. This will prevent users to maintain bunch of offers with a small amount of allowance.
- Require the advertiser to pay flatFee on offer creation instead of offer acceptance and refund it if the offer is canceled or expired.

Tunnl: We first need to fix such that Pending offers are correctly added & removed from `s_offersToUpkeep`. We have decided not to address the DoS issue prior to the mainnet Beta launch as we think the risk of a competitor attempting to attack our product is very low within the first few weeks after launch. HOWEVER we will make a backlog task on our side to address this issue in our next iteration to prevent attacks from malicious competitors.

Cyfrin: Acknowledged.

7.2.2 Malicious creators can force advertisers to pay the flat fee by cancelling accepted offers

Description: The protocol charges a `flatFee` for all offers when they are accepted. Once accepted, the `flatFee` is sent to the protocol owner's address and is NOT refunded under any circumstances, including cancellation or expiration.

While this could be intentional, there is another issue related to malicious creators. After acceptance, if the content creator does nothing, the offer expires after `Offer.payoutDate` and the funds are refunded to the advertiser except for the flat fee. Or the content creator can call `cancelOffer()` to cancel the offer.

Although it is understood that there exists an off-chain backend mechanism to manage creators, it is desirable to have an on-chain escrow feature for the creators as well. This would require creators to put up some collateral when they accept an offer, which would reimburse the advertiser for their loss if the creator breaks the deal.

The current implementation can lead to situations where advertisers incur losses due to the non-refundable flat fee and potential malicious behavior by content creators. This could result in a lack of trust and reduced participation in the protocol.

Impact: We evaluate the overall impact to be Medium because the accepting offer is handled by admin.

Recommended Mitigation: Implement an on-chain escrow feature requiring creators to provide collateral when accepting an offer. This collateral would be used to reimburse advertisers in case the creator fails to fulfill their obligations.

Tunnl: For the Tunnl Beta Launch, we are accepting this “as designed”. Tunnl always has the option to manually reimburse Advertisers for this flat fee which is feasible for our beta given we will only have a few dozen initial users. Charging Creators is not an acceptable solution as this would now require Creators to send a transaction or enter some other payment method which we are trying to avoid given this would worsen the user experience for Creators. Currently, Creators only need to paste their wallet address into our UI and we handle the rest; we want to keep this super simple UX for Creators without them needing to do anything else.

HOWEVER: We will update this in the future by always charging Advertisers the flat fee for creating an offer without any refund mechanism to avoid confusion. We can simply charge a lower flat fee so this is more acceptable for Advertisers (say <$1). This can mitigate “spammy” offers too since Advertisers would always need to pay for each one they send. We will determine feasibility & final fee value based on gas prices on the Base blockchain during our beta launch.

Cyfrin: Acknowledged.

7.2.3 Using a wrong fee percentage [Self-Reported]

Description: In the middle of the audit, the Tunnl team spotted a bug in the function `sendFunctionsRequest()`. While the function calculates the `maxCreatorPayment`, it was using the `s_config.creatorFeePercentageBP` instead of `s_offers[offerId].creatorFeePercentageBP`. This could potentially affect the payment amount to the content creators when there is a change in the configuration.

Impact: We evaluate the impact to be Medium because of the low likelihood.

Tunnl: Fixed in PR 37.

Cyfrin: Verified.

7.2.4 Incomplete implementation of state transition from Pending to Expired

Description: All offers follow a strict state transition defined by the protocol. We extracted all possible state transitions based on the conditionals that define the possible states before changing into a new state.

Looking at the implementation of the function `performUpkeep()`, we can see that the developer assumed a possible state transition from `Pending` to `Expired`.

```
TunnlTwitterOffers.sol
    function performUpkeep(bytes calldata performData) external override {
        bytes32[] memory offersToUpkeep = abi.decode(performData, (bytes32[]));

        for (uint256 i = 0; i < offersToUpkeep.length; i++) {//@audit-info checks only the
↪ offersToUpkeep
            bytes32 offerId = offersToUpkeep[i];

            if (_needsVerificationRequest(s_offers[offerId].status,
↪ s_offers[offerId].dateToAttemptVerification)) {
                s_offers[offerId].status = Status.VerificationInFlight;
                sendFunctionsRequest(offerId, RequestType.Verification);
            } else if (_needsExpiration(s_offers[offerId].status, s_offers[offerId].payoutDate,
↪ s_offers[offerId].acceptanceExpirationDate)) {
                // Update status before making the external call
                Status previousStatus = s_offers[offerId].status;
                s_offers[offerId].status = Status.Expired;
                s_offersToUpkeep.remove(offerId);
                if (previousStatus == Status.Pending) {//@audit-info Pending -> Expired
```

```
                    // If expired BEFORE offer is accepted & funds have been locked in escrow, "revoke" the allowance by sending the advertiser's funds back to themselves
                    _revokeAllowanceForPendingOffer(offerId);
                } else {
                    // If expired AFTER offer is accepted & funds have been locked in escrow, release the funds in escrow back to the advertiser
                    uint256 amountToTransfer = s_offers[offerId].maxValueUsdc - s_offers[offerId].flatFeeUsdc;
                    usdcToken.safeTransfer(s_offers[offerId].advertiser, amountToTransfer);
                }
                emit OfferStatus(offerId, previousStatus, Status.Expired);
            } else if (_needsPayoutRequest(s_offers[offerId].status, s_offers[offerId].payoutDate)) {
                s_offers[offerId].status = Status.PayoutInFlight;
                sendFunctionsRequest(offerId, RequestType.Payout);
            }
        }
    }
```

The outer for loop is for `offersToUpkeep` that is supposed to be retrieved using a function `checkUpkeep()`. The function `checkUpkeep()` checks the offers in the set `s_offersToUpkeep` and an offer is added to this set ONLY when they are accepted. Hence, pending offers are not added to `s_offersToUpkeep` at all and the state transition from Pending to Expired does not happen.

Via the communication with the Tunnl team, it is confirmed that they intended to add Pending offers to the set `s_offersToUpkeep`.

Impact: We would evaluate the impact to be Medium because this finding does not incur financial loss or serious system dysfunctionality directly.

Recommended Mitigation: We recommend to add the Pending offers to the upkeep list and also to review other state transitions.

Tunnl: Fixed in PR 38.

Cyfrin: Verified.

7.2.5 Attackers can prevent users creating an offer by frontrunning

Description: The protocol allows anyone to create offers with an offer ID provided by the user. While this design gives users the flexibility to choose their offer ID, it also introduces a problem.

The `createOffer` function checks if an offer with the provided ID already exists and reverts if it does.

```
TunnlTwitterOffers.sol
    function createOffer(bytes32 offerId, uint256 maxPaymentUsdc, uint32 acceptanceDurationSeconds, uint32 _offerDurationSeconds) external {

        require(offerId != bytes32(0), "Invalid offerId");
        require(s_offers[offerId].creationDate == 0, "Offer already exists"); //@audit Can be abused to prevent offer creation
        require(acceptanceDurationSeconds >= s_config.minAcceptanceDurationSeconds, "AcceptanceDuration is too short");
        require(_offerDurationSeconds >= s_config.minOfferDurationSeconds, "OfferDuration is too short");
```

Malicious actors can abuse this and prevent normal users from creating offers.

Example:

- Alice, an attacker, decides to harm the protocol's reputation.

- Alice allows the `flatFee` amount of USDC to the protocol and monitors the mempool.
- A legitimate user, Bob, tries to create an offer with ID="BOB".
- Alice sends a front-running transaction to create a fake offer with the same ID.
- Alice's fake offer is created, and Bob's transaction reverts because an offer with the same ID already exists.

The impact is amplified for the following reasons:

- The protocol does not require actual funding but only an allowance for the creation of offers. Anyone can create any number of offers with a single allowance of the `flatFee` amount of USDC.
- Alice can choose to disrupt only "big" offers from reputable advertisers.

While the attack is not economically beneficial for the attacker, it is quite feasible for a possible competitor to run this kind of malicious activity. Additionally, the protocol will be deployed on Base(L2), where gas costs will be minor.

Impact: We evaluate the impact to be Medium because the attack does not directly affect users' funds and is not economically beneficial for the attacker.

Proof Of Concept:

```
    function test_FrontrunCreateOffer() public {
        address alice = address(0x10); // malicious actor
        address bob = address(0x20); // legitimate actor

        uint32 acceptanceDurationSeconds = tunnlTwitterOffers.getConfig().minAcceptanceDurationSeconds;
        uint offerAmount = 1e6;
        uint maxValueUsdc = tunnlTwitterOffers.getConfig().flatFeeUsdc + offerAmount;
        uint minAllowance = tunnlTwitterOffers.getConfig().flatFeeUsdc;

        // alice allows minimal amount of USDC to the contract
        vm.startPrank(alice);
        mockUsdcToken.mint(alice, minAllowance);
        mockUsdcToken.approve(address(tunnlTwitterOffers), minAllowance);
        vm.stopPrank();

        // bob intends to create an offer with id = 101010 (this is normally calculated offchain but
        ↪  here we just use an imaginary id)
        vm.startPrank(bob);
        mockUsdcToken.mint(bob, maxValueUsdc);
        mockUsdcToken.approve(address(tunnlTwitterOffers), maxValueUsdc);
        vm.stopPrank();

        offerId = bytes32(uint(0x101010));

        // alice front runs bob and creates an offer with the same id
        vm.startPrank(alice);
        tunnlTwitterOffers.createOffer(offerId, 0, acceptanceDurationSeconds, 4 days);
        vm.stopPrank();

        // now bob's offer creation reverts
        vm.startPrank(bob);
        vm.expectRevert("Offer already exists");
        tunnlTwitterOffers.createOffer(offerId, offerAmount, acceptanceDurationSeconds, 4 days);
        vm.stopPrank();
    }
```

Recommended Mitigation: We recommend adding a mechanism to verify the `msg.sender` with the `offerId` in `createOffer()`.

Tunnl: We have decided not to address this frontrunning issue prior to the mainnet Beta launch as we think the risk of a competitor attempting to attack our product is very low within the first few weeks after launch. HOWEVER:

we will make a backlog task on our side to address this issue in our next iteration to prevent attacks from malicious competitors. A potential solution would be to have the backend generate an approval signature which is given to the user and must be included in their `createOffer` transaction.

Cyfrin: Acknowledged. (The potential solution approach could resolve the issue.)

7.3 Informational

7.3.1 Some variable and argument names are misleading

Description: The use of misleading variable and argument names may result in incorrect implementation. It is recommended that these names be revised to enhance clarity and maintain code integrity.

- `maxPaymentUsdc` -> `maxOfferedAmountUSDC`

```
function createOffer(bytes32 offerId, uint256 maxPaymentUsdc, uint32 acceptanceDurationSeconds, uint32 _offerDurationSeconds)
```

- `Offer.maxValueUsdc` -> `Offer.totalPaymentFromAdvertiser`

```
uint256 maxValueUsdc; // Percentage fee + flat fee + maximum payment the content creator can receive in USDC
```

- `Offer.amountPaidUsdc` -> `Offer.netAmountPaidUsdc`

```
uint256 amountPaidUsdc; // The final amount paid to the content creator in USDC
```

Tunnl: Fixed in PR 38.

Cyfrin: Verified.

7.3.2 Unnecessary duplicate state logic

Description: The offer status is set two times to the same value in function `performUpkeep` and `sendFunctionsRequest`. Note that this kind of duplicate logic can cause a security issue in the future, e.g. the team might miss changing one of the two occurences.

```
TunnlTwitterOffers.sol - performUpkeep
            if (_needsVerificationRequest(s_offers[offerId].status, s_offers[offerId].dateToAttemptVerification)) {
                s_offers[offerId].status = Status.VerificationInFlight;
                sendFunctionsRequest(offerId, RequestType.Verification);

```

```
TunnlTwitterOffers.sol - sendFunctionsRequest
        s_offers[offerId].status = requestType == RequestType.Verification
            ? Status.VerificationInFlight
            : Status.PayoutInFlight;
        emit RequestSent(offerId, requestId, s_offers[offerId].status);
```

Tunnl: Fixed in PR 38.

Cyfrin: Verified.

7.3.3 Unnecessary use of OpenZeppelin's nonReentrant

Description: The contract `TunnlTwitterOffers` uses the OpenZeppelin's `nonReentrant` modifier in a function `retryRequests()`. But the implementation does not need this modifier. The function `retryRequests()` only makes calls to an internal function `sendFunctionsRequest()` and there is no external call in the process. Furthermore, the function `retryRequests()` is restricted by `onlyAdmins` modifier. We did further analysis and concluded that the protocol does not need the `nonReentrant` modifier as long as the payment token is fixed to USDC.

Tunnl: Fixed in PR 38.

Cyfrin: Verified.