DECENTRALIZED INTELLIGENCE

Audit Report

Produced for

BX Digital

0.1 Introduction

This report provides a thorough and detailed examination of the *DeliveryVersusPayment* (DVP) smart contract system, which harnesses smart contract capabilities to facilitate secure, transparent, and efficient asset exchanges within financial transactions. The system is meticulously crafted to elevate transaction transparency, velocity, and dependability while proactively addressing and mitigating associated risks.

At the core of the system lie the *DeliveryVersusPayment* contracts, available in two distinct iterations, serving as its foundational bedrock. These contracts deliver a spectrum of services to clients seeking to execute ERC20-standard token deliveries with utmost security. Moreover, the *DeliveryVersusPayment* contracts incorporate features allowing clients to retract orders prior to confirmation, thereby enhancing system flexibility.

Throughout this report, we present our findings with clarity, brevity, and quantitative precision, ensuring our clients gain a comprehensive understanding of potential system vulnerabilities and recommendations for further fortification.

0.1.1 Scope

The audit and fuzzing was performed on the following one solidity file:

- `contracts/DeliveryVersusPaymentV2.sol`
  sha256: f3f81c9968ab2c7e8772e92bc32e552b37196470eea5bb389afb6e5ae7eb094e

0.1.2 Excluded from scope

Files not specified in the scope section — particularly test files, scripts, external dependencies, and configuration files — are not included in the scope.

0.2 Automated Findings

The client's smart contract(s) were analyzed using the Slither[1] static analysis tool. The tool reported a total of 1 contracts in the source files and 25 contracts in the dependencies. No assembly code was detected in the contracts.

The analysis revealed 1 informational issues and 1 low severity issues.

The following table summarizes the key statistics for each contract:

Name	functions	ERCs	Features
DeliveryVersusPaymentV2	51	ERC165	Receive token, Send token, Delegatecall, Upgradeable

[1]https://github.com/crytic/slither

0.2.1 Issues Detected

The Slither analysis reported the following issues:

Arbitrary `from` address in transferFrom (Low severity, not valid)

The `createTradeAndLockTokens` function in `DeliveryVersusPaymentV2` contracts uses an arbitrary `from` address in the `transferFrom` function call, which may lead to a control leak if the token implements ERC777[2] standard or has any hooks calling the asset holder when doing transfers. However, we do not considered this a valid issue because the `trades` mapping could prevent reentrancy attacks, and the function has strict access control.

Local Variable Shadowing (Informational)

The `increaseAllowance` and `decreaseAllowance` functions in the `AssetToken` contract have local variables named `owner` that shadow the `owner` function from the `Ownable` contract. This is considered an informational issue and does not appear to pose a security risk.

In summary, the automated analysis did not reveal critical issues that would pose a significant risk to the functionality or security of the smart contracts. The reported issues related to reentrancy may be mitigated by the design of the contracts.

0.3 Fuzzing Report

The fuzzing process was initially planned to be conducted over a period of 72 hours. However, the exhaustive search of a single oracle completed within 1 hour (47 minutes to be exact). The fuzzer utilized the initialization script, shown below, to set up the necessary state for the fuzzing campaign. This script (cf. Listing 1) simulated the presence of a confirmed or cancelled order, enabling the fuzzer to test the contract under various scenarios.

```
contract DvpScript is Script {
    AssetToken token;
    Proxy proxy;
    DeliveryVersusPaymentV2 dvp;

    uint256 AMOUNT = 10 ether;
    address public sender = 0x23618e81E3f5cdF7f54C3d65f7FBc0aBf5B21E8f;
    address public receiver = 0xa0Ee7A142d267C1f36714E4a8F75612F20a79720;

    function setUp() public {
        token = new AssetToken();
        dvp = new DeliveryVersusPaymentV2();
        proxy = new Proxy(address(dvp));
```

[2] https://eips.ethereum.org/EIPS/eip-777

```
        console.log("Token address", address(token));
        console.log("Dvp address", address(dvp));
        console.log("Proxy address", address(proxy));

        // Init the dvp contract
        proxy.initialize();

        // give seller 100 ether tokens: prepare for the test
        token.mint(sender, AMOUNT * 2);

        // Seller approves the dvp contract
        uint256 deployerPrivateKey = vm.envUint("SELLER_PRIVATE_KEY");
        vm.startBroadcast(deployerPrivateKey);
        token.approve(address(proxy), type(uint256).max);
        vm.stopBroadcast();
    }

    function run() public {
        // 1. Test confirmPayment()
        //    1.1 Sender approves the dvp contract,
        //    1.2 proxy's oracle executes createTradeAndLockTokens(),
        //    1.3 proxy's oracle executes confirmPayment(),
        //    1.4 Sender decreases 10 ether, Receiver gets 10 ether.
        toTestConfirmPayment();
        // 2. Test confirmPayment()
        //    2.1 Sender approves the dvp contract,
        //    2.2 proxy's oracle executes createTradeAndLockTokens(),
        //    2.3 proxy's oracle executes cancelTrade(),
        //    2.4 Sender cancels the order and gets 10 ether back,
            Receiver gets 0 ether.
        toTestCancelTrade();
    }

    function toTestConfirmPayment() public {
        bytes32 tradeId = bytes32(uint256(1));
        address assetAddress = address(token);
        uint64 timestamp;
        bytes32 theHash = keccak256(abi.encode(tradeId, sender, receiver,
             assetAddress, AMOUNT, timestamp));
        proxy.createTradeAndLockTokens(theHash, sender, assetAddress,
            AMOUNT);

        proxy.confirmPayment(theHash, tradeId, sender, receiver,
            assetAddress, AMOUNT, timestamp);

    }

    function toTestCancelTrade() public {
        bytes32 tradeId = bytes32(uint256(2));
        address assetAddress = address(token);
        uint64 timestamp;
        bytes32 theHash = keccak256(abi.encode(tradeId, sender, receiver,
             assetAddress, AMOUNT, timestamp));
        proxy.createTradeAndLockTokens(theHash, sender, assetAddress,
            AMOUNT);

        proxy.cancelTrade(theHash, tradeId, sender, receiver,
            assetAddress, AMOUNT, timestamp);
    }
}
```

Listing 1: Fuzzing setup script.

The script creates instances of the `AssetToken`, `DeliveryVersusPaymentV2`, and `Proxy` contracts. It initializes the DVP contract, mints tokens for the seller, and approves the DVP contract to spend the seller's tokens.

The fuzzer yielded the following key statistics in Table 1:

Metric	**Count**
Total Test Cases	40,814
Valid Test Cases	20,470
Successfully Executed	14,144
ABI Interface Coverage	7/7

Table 1: Fuzzing output statistics.

The fuzzing campaign generated a total of 40,814 test cases, out of which 20,470 (50.15%) were valid. These valid test cases were then executed, resulting in 14,144 (69.10%) being successfully executed.

The fuzzer achieved complete path coverage of the contract's ABI interface, testing all 7 functions (by path we mean a combination of actions, not including all possible parameters). This comprehensive coverage ensures that all accessible functions were thoroughly exercised during the fuzzing process.

With a token leak oracle, the fuzzer did not identify any fund losses or security vulnerabilities in the contract under the given test cases. This indicates that the contract demonstrated robustness and security against these generated inputs.

However, it is important to note that while the absence of identified vulnerabilities is a positive indicator, it does not guarantee the absence of all potential issues because only the token leak oracle is employed. Further testing and analysis, such as manual code review, may be necessary to establish a higher level of confidence in the contract's security.

We therefore chose to extend our analysis to an additional four man-days of manual auditing. Please find its results in the following.

0.4 Manual Auditing Finding Summary

The following encapsulates our discoveries subsequent to a manual examination of the DVP contract deployment. The contractual framework leverages oracles to administer project oversight, yet it is imperative to acknowledge the latent susceptibility to manipulation should these oracles engage in malicious conduct. Such actions pose the potential to instigate a Denial of Service (DoS) incident attributable to the inadequacies within the system's verification protocols. Further, given that the AssetToken is not constrained, it has the potential to represent any ERC20 token. This introduces a risk of triggering Denial of Service attacks or token locking when the AssetToken represents slightly non-standard ERC20 tokens. Nonetheless, commendable efficiency is demonstrated by the contract system in its judicious conservation of gas resources while concurrently preserving the integrity of its routine operations.

In totality, the contracts manifest commendable attributes of design and engineering. However, it is imperative to address areas necessitating refinement, notably the identification of one medium-severity vulnerability, one low-severity vulnerability, and one informational concern.

ID	Severity	Title	Category	Status
PVE-001	Medium	AssetAddress can be variable balance token	Security Features	Open
PVE-002	Low	Malicious oracle can manipulate the trade order	Security Features	Open
PVE-003	Info	Front-running possibility when the oracle is malicious	Business Logics	Open

Table 2: Summary of our findings following a manual investigation.

0.5 Detailed Results

0.5.1 AssetAddress can be a variable balance token

Summary[3]

The *DeliveryVersusPayment* smart contract system allows clients to deliver any ERC20-compliant token as the *assetToken*. However, this flexibility may introduce potential risks when dealing with tokens that have special functionalities. In particular, the use of variable balance tokens or fee-on-transfer tokens may lead to issues such as the locking of funds, loss of funds, and Denial of Service attacks.

Details

[3]See reference https://github.com/code-423n4/2022-09-vtvl-findings/issues/278

In the provided contract, the *assetToken*, which clients of the project may want to deliver, can be any token that implements the ERC20 standard interface. In some cases, the *assetToken* contract may contain special functionality, such as an inflation method, which may lead to three potential issues:

1. The variable balance tokens can cause the lock of funds.
2. The variable balance tokens can lead to loss of funds and Denial of Service for the confirmation or cancelation of an order.
3. The fee-on-transfer tokens can lead to loss of funds and Denial of Service for the confirmation or cancelation of an order.

Regarding the first potential issue 1, we assume that an *inflation token* is a variable balance token that earns interests over time and directly inflates the owners' balances. Such tokens include for example \$stETH[4] and \$USD+[5]. When the seller enters the market, our *inflation token*'s balance will increase over time. When we call *confirmPayment()* after *createTradeAndLockTokens()*, it will lock 100 tokens in the *dvp* since 100 seconds have passed. Under the above scenario, we can therefore conclude, that the funds locked can no longer be withdrawn from the contract.

```solidity
    function test_rebaseToken_inflation_lockingOfFund() public {
        bytes32 tradeId;
        address assetAddress = address(inflation_token);
        uint64 timestamp;
        bytes32 theHash = keccak256(abi.encode(tradeId, seller, buyer, assetAddress, AMOUNT, timestamp));
        dvp.createTradeAndLockTokens(theHash, seller, assetAddress, AMOUNT);

        // When the dvp receives the inflation_token at the beginning, the balance is `AMOUNT`
        assertEq(inflation_token.balanceOf(address(dvp)), AMOUNT);
        vm.warp(block.timestamp + 100); // pass 100 seconds
        // However, the dvp's inflation_token balance is increasing by the time.
        assertEq(inflation_token.balanceOf(address(dvp)), AMOUNT + 100);

        dvp.confirmPayment(theHash, tradeId, seller, buyer, assetAddress, AMOUNT, timestamp);
        // We can only withdraw `AMOUNT` inflation_token, left 100 in the dvp, leading to locking of fund.
        assertEq(inflation_token.balanceOf(address(dvp)), 100);
        // The seller can get `AMOUNT` inflation_token normally
        assertEq(inflation_token.balanceOf(address(buyer)), AMOUNT);
    }
```

Listing 2: Variable balance token causing a fund lock up.

[4] Lido Liquid staked Ether 2.0: https://etherscan.io/token/0xae7ab96520de3a18e5e111b5eaab095312d7fe84
[5] Overnight USD+: https://optimistic.etherscan.io/token/0x73cb180bf0521828d8849bc8cf2b920918e23032

Regarding the second potential issue 2, we assume that a *deflation_token* is a balance variable token that the owners' balances will reduce over time, as an example, $PRX token on Ethereum[6]. When we enter the market, our *deflation_token*'s balance will decrease over time. When we call *confirmPayment()* after *createTradeAndLockTokens()*, the transaction will revert because the *dvp* 's balance is less than AMOUNT: AMOUNT - 100 < AMOUNT. Under this second scenario, we can therefore conclude, that the order can neither be confirmed nor cancelled, resulting in a loss of funds and a DoS situation.

```
function test_rebaseToken_deflation_DoS() public {
    bytes32 tradeId;
    address assetAddress = address(deflation_token);
    uint64 timestamp;
    bytes32 theHash = keccak256(abi.encode(tradeId, seller, buyer,
        assetAddress, AMOUNT, timestamp));
    dvp.createTradeAndLockTokens(theHash, seller, assetAddress,
        AMOUNT);

    // When the dvp receives the deflation_token at the beginning,
        the balance is `AMOUNT`
    assertEq(deflation_token.balanceOf(address(dvp)), AMOUNT);
    vm.warp(block.timestamp + 100); // pass 100 seconds
    // However, the dvp's deflation_token balance is decreasing by
        the time.
    assertEq(deflation_token.balanceOf(address(dvp)), AMOUNT - 100);

    // The next action will revert: the dvp's deflation_token balance
         is `AMOUNT - 100` while
    // that it wants to transfer `AMOUNT` defaltion_token. So it
        leads to DoS.
    vm.expectRevert();
    dvp.confirmPayment(theHash, tradeId, seller, buyer, assetAddress,
         AMOUNT, timestamp);
}
```

Listing 3: Variable balance token causing fund loss and Denial of Service.

Regarding the third potential issue 3, when the token is a fee-on-transfer token (tokens that deduct a fee during the transfer), the transaction will revert, causing a DoS issue. The *$XCHF token*[7] is for example such a token on the ETH network, and the *$ELEPHANT token*[8] is another instance on the BSC network. In the PoC (cf. Listing 4), we fork from the BSC block height 35196717. When we call the *confirmPayment()* function after *createTradeAndLockTokens()*, the transaction will revert because *dvp* 's balance is less than AMOUNT: 7.2 < 8.0. Under this third scenario, we can therefore conclude, that the order can neither be confirmed nor cancelled, resulting in a loss of funds and a DoS situation.

[6] PRX token: https://etherscan.io/token/0xe8847d2fa66d0d1f4a77221cae1e47d8d59cf7d7
[7] bitcoinsuisse $XCHF token: https://etherscan.io/token/0xb4272071ecadd69d933adcd19ca99fe80664fc08
[8] $ELEPHANT token: https://bscscan.com/address/0xD96EC811359BFD94D2dfe2A3Bd8DA68BF262Be1A

```
    function test_elephant_DoS() public {
        // IElephant public elephant = IElephant(0
            xD96EC811359BFD94D2dfe2A3Bd8DA68BF262Be1A);
        bytes32 tradeId;
        address assetAddress = address(elephant);
        uint64 timestamp;
        bytes32 theHash = keccak256(abi.encode(tradeId, seller, buyer,
            assetAddress, AMOUNT, timestamp));
        // Seller wants to sell for AMOUNT=8.000000000
        emit log_named_decimal_uint("seller transfer amount", AMOUNT,
            elephant.decimals());
        dvp.createTradeAndLockTokens(theHash, seller, assetAddress,
            AMOUNT);
        // But the dvp only receives 7.200000000
        emit log_named_decimal_uint("dvp receive amount", elephant.
            balanceOf(address(dvp)), elephant.decimals());

        // Revert: dvp's balance is less than `AMOUNT`,
            7.200000000<8.000000000
        vm.expectRevert("SafeMath: subtraction overflow");
        dvp.confirmPayment(theHash, tradeId, seller, buyer, assetAddress,
             AMOUNT, timestamp);
    }
```

Listing 4: Fee-on-transfer tokens causing fund loss and Denial of Service.

Recommendation

We would like to provide two suggestions to potentially address this issue.

1. Limit the *assetAddress* to a whitelist of approved tokens. The whitelist should be managed by the contract maintainer, who can add or remove tokens as needed.

```
// Adding the following codes
mapping(address => bool) isTokenWhiteListed;
function AddTokenWhiteList(asset address) onlyRole(MAINTAINER_ROLE)
    {
     isTokenWhiteListed[asset] = true;
}

function RemoveTokenWhiteList(asset address) onlyRole(
    MAINTAINER_ROLE) {
     isTokenWhiteListed[asset] = false;
}

// in function 'createTradeAndLockTokens', Asset should be in white
    list
require(isTokenWhiteListed[assetAddress], "Asset token not
    whitelisted");
```

Listing 5: Potential token whitelist fix for issue 0.5.1.

2. Add a balance check before and after the *safeTransferFrom* function to ensure that the *dvp* contract receives the expected amount of tokens. This helps to

prevent issues with fee-on-transfer tokens, which may deduct a fee during the transfer and cause DoS problem.

```
function createTradeAndLockTokens(
    bytes32 tradeHash,
    address seller,
    address assetAddress,
    uint256 amount
) external onlyRole(TRADE_ORACLE_ROLE) {

...
uint balanceBefore = IERC20(assetAddress).balanceOf(address(this));
// line 142
IERC20(assetAddress).safeTransferFrom(seller, address(this), amount)
    ;
require(amount ==  IERC20(assetAddress).balanceOf(address(this)) -
    balanceBefore, "Asset transferFrom verification failed");
...
}
```

Listing 6: Potential balance check fix for issue 0.5.1.

0.5.2 A malicious oracle can manipulate the trade order

Summary

We identified a vulnerability in the `createTradeAndLockTokens` function, allowing a malicious entity with `TRADE_ORACLE_ROLE` privileges to initiate trades with arbitrarily manipulated token amounts. This is facilitated by the absence of a verification process for the `tradeHash` against the actual trade amount, leading to potential unauthorized asset transfers and contract asset depletion.

Details

The `createTradeAndLockTokens` function lacks a mechanism to ensure that the `tradeHash` accurately represents the trade details such as the token amount. This omission allows a TRADE_ORACLE to submit any `tradeHash`, which may not correspond to the actual trade parameters and could represent an arbitrarily large `amount`. Consequently, a manipulated `tradeHash` can be used to establish fictitious trades in the contract's records.

This vulnerability is critical in the `confirmPayment` function, where the integrity of `tradeHash` is pivotal. A malicious oracle could confirm an altered trade with an arbitrarily large `amount`, directing tokens to an unauthorized address and possibly depleting the contract's assets.

The vulnerability poses a risk due to the potential for complete asset depletion enabled by a single compromised or rogue oracle, undermining the trust model of the contract.

```
  // Line 126 in DeliveryVersusPaymentV2.go
```

```
function createTradeAndLockTokens(
  bytes32 tradeHash,
  address seller,
  address assetAddress,
  uint256 amount
) external onlyRole(TRADE_ORACLE_ROLE) {
  // Check if the trade already exists
  TradeStatus currentTradeStatus = trades[tradeHash];
  if (currentTradeStatus != TradeStatus.NONE) {
    revert TradeStatusCheckError(tradeHash, currentTradeStatus,
        TradeStatus.NONE);
  }

  trades[tradeHash] = TradeStatus.ESCROW;

  // transfer tokens from seller to DvP (fails if allowance is
      insufficient)
  IERC20(assetAddress).safeTransferFrom(seller, address(this), amount);

  emit TradeCreatedAndTokensLocked(tradeHash, seller, assetAddress,
      amount);
}
```

Listing 7: Lack of input validation.

Recommendation

To mitigate the vulnerability, we recommend modifying the function `createTradeAndLockTokens` and `verifyDataHash` to (cf. Lisiting 8). We introduce a new formula for `tradeHash`: `keccak256(abi.encode(seller, assetAddress, amount, keccak256(abi.encode(tradeId, buyer, timestamp))))`. This may ensure that the hash accurately reflects and securely encapsulates key trade parameters while hiding information such as `tradeId` and `buyer` when creating a trade. The verification step prevents unauthorized transactions by compromised oracles, potentially enhancing contract security. For your convenience we have assessed the impact on gas costs of such change in Table 3.

```
function verifyDataHash_Fixed(
  bytes32 tradeHash,
  bytes32 tradeId,
  address seller,
  address buyer,
  address assetAddress,
  uint256 amount,
  uint64 timestamp
) internal pure {
  bytes32 calculatedHashValue = keccak256(
    abi.encode(seller, assetAddress, amount, keccak256(abi.encode(
        tradeId, buyer, timestamp)))
  );
  if (tradeHash != calculatedHashValue) {
    revert IncorrectTradeHash(tradeHash, calculatedHashValue);
  }
}

function createTradeAndLockTokens_Fixed(
  bytes32 tradeHash,
  address seller,
  address assetAddress,
  uint256 amount,
  bytes32 checkHash
) external onlyRole(TRADE_ORACLE_ROLE) {
  bytes32 calculatedHashValue = keccak256(
    abi.encode(seller, assetAddress, amount, checkHash)
  );
  if (tradeHash != calculatedHashValue) {
    revert IncorrectTradeHash(tradeHash, calculatedHashValue);
  }
  ...
}
```

Listing 8: Potential fix for issue 0.5.2.

Function	Gas cost
createTradeAndLockTokens	89,226
createTradeAndLockTokensFixed	90,010 (784 ↑)
verifyDataHash	1,080
verifyDataHashFixed	1,220 (140 ↑)

Table 3: Gas consumption comparison before and after suggested fix of issue 0.5.2.

0.5.3 Front-running possibility when the oracle is malicious

Summary

If the oracle is malicious, *cancelTrade()* and *confirmPayment()* can be front-run.

Details

All oracles can read the parameters when an oracles call *cancelTrade()* or *confirmPayment()*, an oracle may increase the gas price and use the same parameters to perform front-running.

For example, when the oracle1 wants to call *cancelTrade()* to cancel the trade order, oracle2 observes the canceling transaction, copies the parameters executes the *confirmPayment()* transaction with a higher gas price.

Recommendation

Rewrite the verification logic into two parts: one for cancellation verification and the other for confirmation verification.

0.6 Conclusion

In conclusion, this report has summarized features and potential security issues of the DeliveryVersusPayment smart contract system, which leverages smart contract capabilities to facilitate secure, transparent, and efficient asset exchanges within financial transactions.

The core of the system, the *DeliveryVersusPayment* contracts, deliver a spectrum of services to clients seeking to execute ERC20-standard token deliveries with utmost security. The contracts also incorporate features allowing clients to cancel orders before confirmation.

However, our manual auditing process has identified several potential threats, notably one medium-severity vulnerability, one low-severity vulnerability, and one informational concern. The medium-severity vulnerability pertains to the potential risks introduced by variable balance tokens and fee-on-transfer tokens, which may lead to issues such as the locking of funds, loss of funds, and Denial of Service attacks. The low-severity vulnerability highlights the possibility of trade order manipulation by a malicious oracle due to insufficient verification mechanisms. The informational concern points out the potential for front-running when the oracle is malicious. Both the second and the third issue necessity a trustworthy oracle role.

To address these vulnerabilities and enhance the robustness of the DVP system, we have provided detailed potential recommendations. These include implementing a whitelist of approved tokens, adding balance checks to prevent issues with fee-on-transfer tokens, modifying the trade hash verification process to ensure the integrity of trade parameters, and rewriting the verification logic to mitigate front-running risks.

It is crucial to emphasize that while our fuzzer with the token leak oracle did not identify any vulnerabilities under the given test cases, we would recommend the purchase of additional fuzzing oracles for a more in-depth investigation.

Disclaimer