Decentralized Vision LTD

Smart Contract Audit

Version 1.2

- *This document is confidential and may contain proprietary information and intellectual property of Electi Consulting Ltd.*
- *None of the information contained herein may be reproduced or disclosed under any circumstances without the express written permission of Electi Consulting Ltd.*

Table of Contents

Introduction

Electi (*Consultant*) was assigned to perform a smart contract audit on two smart contracts developed by Decentralized Vision (*Client*). Specifically, the smart contracts to be audited are PumaPayPullPayment.sol[1](***V1***) and PumaPayPullPaymentV2.sol[2](***V2***) found in the official github repository of PumaPay (commit [5eb99b1](5eb99b1)). The two aforementioned smart contracts are part of the PumaPay payment solution that enables businesses to implement advanced payment models based on the PMA token.

The audit was conducted against a list of known attacks which includes Reentrancy, Front-Running, Integer Over/UnderFlow, DoS with unexpected revert, Forcibly Sending Ether to a Contract as well as others. Significant importance has been given to manual code review to identify logic errors that could affect the business logic.

The scope of this audit is to perform the following:

1. Conduct a smart contract security audit against known threats, where applicable.
2. Identify potential flaws and vulnerabilities that can be exploited by malicious users.
3. Verify the behavior of the smart contract against the documentation.
4. Suggest potential improvements or best practices.

---

[1] https://github.com/pumapayio/smart-contracts/blob/master/contracts/PumaPayPullPayment.sol
[2] https://github.com/pumapayio/smart-contracts/blob/master/contracts/PumaPayPullPaymentV2.sol

Executive Summary

The outcome of the audit of the two smart contracts was:

- 1 informational
- 1 medium
- 2 high severity issues.

The exploitation of the issues identified is not straightforward and in some cases it requires certain assumptions. However, it must be noted that all the issues listed in this report are suggested to be fixed.

Overall, both smart contracts are well written, extensively tested and follow all the guidelines[3] for secure smart contract development.

[3] https://consensys.github.io/smart-contract-best-practices/

Issues

Reentrancy

Description:
This attack usually occurs when a contract sends ether to an external contract which contains malicious code in the fallback function. This piece of code executes functions in the vulnerable contract that are unexpected, hence the term "reentrance".

V1 contract interacts with external contracts in 6 different cases:

Function Name	Method	Description	Present in V2
addExecutor	_executor.transfer	Adds funds to newly registered executors' account.	Yes
addExecutor	owner().transfer	Checks the balance of the contract owner and send funds.	Yes
removeExecutor	owner().transfer	Checks the balance of the contract owner and send funds.	Yes
setRate	owner().transfer	Checks the balance of the contract owner and send funds.	No
registerPullPayment	msg.sender.transfer	Checks the balance of the executor and send funds if needed.	Yes
deletePullPayment	msg.sender.transfer	Checks the balance of the executor and send funds if needed.	Yes

Both **V1** and **V2** are not vulnerable to this attack since the built-in method **transfer()** is used to transfer funds. The transfer function only sends 2300 gas which is not enough to call another contract.

Notes:

```
175    function addExecutor(address payable _executor)
176    public
177    onlyOwner
178    isValidAddress(_executor)
179    executorDoesNotExists(_executor)
180    {
181        _executor.transfer(FUNDING_AMOUNT);
182        executors[_executor] = true;
183
184        if (isFundingNeeded(owner())) {
185            owner().transfer(FUNDING_AMOUNT);
186        }
187
188        emit LogExecutorAdded(_executor);
189    }
```

1. It is suggested, in general, that any changes to the state of the contract to happen before the interaction with external contracts. In this case line 182 should be placed before line 181.
2. Although in this case it might be considered as an "overkill", the use of mutexes is suggested. A mutex has the ability to lock the contract state until the external call is finished thus preventing reentrancy.

Business Impact	Likelihood	Severity
N/A	N/A	N/A

Arithmetic Over/Underflows

Description:

An arithmetic over/underflow occurs when an operation on a type causes it to go out of bounds. For example, adding 1 to a `uint8` type where its current value is 255 will result to 0 since 255 it is the maximum value it can hold(overflow). Similarly, subtracting 1 from a `uint8` type with current value 0 will result to 255(underflow).

V1

Variable	Type	Bounds Check (Upper/Lower)	User input	Vulnerable to Overflow/Underflow
initialPaymentAmountInCents	uint256	N/N	Yes	**No**
frequency	uint256	Y/Y	Yes	**No**
numberOfPayments	uint256	Y/Y	Yes	**No**
startTimestamp	uint256	Y/Y	Yes	**No**
nextPaymentTimestamp	uint256	N/Y	No	**No**
lastPaymentTimestamp	uint256	N/N	No	**No**
cancelTimestamp	uint256	N/N	No	**No**
amountInPMA	uint256	N/N	No	**Yes**

Notes:

In general, smart contracts should not assume that the data being passed as inputs are sanitized properly therefore, to the best extent, measures should be taken to prevent unexpected behaviour. In this smart contract, all the aforementioned variables are safe against over/underflows besides `amountInPMA`. Technically, this variable should be allowed to reach the maximum value of a `uint256` type, however it is quite unlikely that such a value can occur purposely. It is recommended that this value should be checked against a maximum bound that is reasonable in the context of the DAPP. Since this variable occurs from a complex arithmetic it is possible that an overflow might occur. Specifically, if the division value is significantly small (conversion rate), then the result of the calculation might overflow.

```
function calculatePMAFromFiat(uint256 _fiatAmountInCents, string memory
_currency)
internal
view
validConversionRate(_currency)
validAmount(_fiatAmountInCents)
returns (uint256) {
    return ONE_ETHER.mul(DECIMAL_FIXER).mul(_fiatAmountInCents).div
    (conversionRates[_currency]).div(FIAT_TO_CENT_FIXER);
}
```

Business Impact	**Likelihood**	**Severity**
Loss of tokens in case of an over/underflow	Low	Medium

<u>Resolution by PumaPay</u>

The fix for this issue checks if the inputs of the `calculatePMAFromFiat` are within the appropriate bounds and in conjunction with the improvement of the calculation the possibility of an over/underflow is eliminated.

```
function calculatePMAFromFiat(uint256 _fiatAmountInCents, string memory _currency)
internal
view
validConversionRate(_currency)
validAmount(_fiatAmountInCents)
returns (uint256) {
    return RATE_CALCULATION_NUMBER.mul(_fiatAmountInCents).div(conversionRates[_currency]);
```

V2

Variable	**Type**	**Bounds Check (Upper/Lower)**	**User input**	**Vulnerable to Overflow/Underflow**
initialPaymentAmountInCents	uint256	N/N	Yes	**No**
frequency	uint256	Y/Y	Yes	**No**
numberOfPayments	uint256	Y/Y	Yes	**No**
startTimestamp	uint256	Y/Y	Yes	**No**
nextPaymentTimestamp	uint256	N/Y	No	**No**
lastPaymentTimestamp	uint256	N/N	No	**No**
cancelTimestamp	uint256	N/N	No	**No**
amountInPMA	uint256	N/N	No	**Yes**
initialConversionRate	uint256	Y/Y	Yes	**No**

Notes:

The issue regarding `amountInPMA` found in **V1** is partially solved by checking the upper and lower bounds of the conversion rate. This check limits the values that can in the conversion rate but it does not eliminate completely the possibility of an over/underflow. The check against `initialConversionRate` is suggested to be modified to assert that the values stays within reasonable conversion rates. If a very low or very high conversion rate is entered by the executor (purposely or by mistake) there is risk of ending up with unexpected values assigned to `amountInPMA`.

```
    function calculatePMAFromFiat(uint256 _fiatAmountInCents, uint256 _conversionRate)
    internal
    pure
    validAmount(_fiatAmountInCents)
    validAmount(_conversionRate)
    returns (uint256) {
        return ONE_ETHER.mul(DECIMAL_FIXER).mul(_fiatAmountInCents).div(_conversionRate).div(FIAT_TO_CENT_FIXER);
    }
```

Business Impact	Likelihood	Severity
Invalid payment - Transfer of funds failure.	Low	Medium

Resolution by PumaPay

See **V1.**

Replay Attack in Pull Payment Execution

Description:

Both contracts **V1** and **V2** are vulnerable to a replay attack in function `executePullPayment`. The executor can replay the same pull payment as there is no state change in respect to that specific payment ID apart from the `nextPaymentTimestamp` being updated by adding the payment frequency. This allows a malicious user to execute the same pull payment multiple times in the future.

```
function executePullPayment(address _customer, bytes32 _paymentID)
public
paymentExists(_customer, msg.sender)
isValidPullPaymentExecutionRequest(_customer, msg.sender, _paymentID)
{
    uint256 amountInPMA;

    if (pullPayments[_customer][msg.sender].initialPaymentAmountInCents > 0) {
        amountInPMA = calculatePMAFromFiat(
            pullPayments[_customer][msg.sender].initialPaymentAmountInCents,
            pullPayments[_customer][msg.sender].currency
        );
        pullPayments[_customer][msg.sender].initialPaymentAmountInCents = 0;
    } else {
        amountInPMA = calculatePMAFromFiat(
            pullPayments[_customer][msg.sender].fiatAmountInCents,
            pullPayments[_customer][msg.sender].currency
        );

        pullPayments[_customer][msg.sender].nextPaymentTimestamp =
        pullPayments[_customer][msg.sender].nextPaymentTimestamp + pullPayments[_customer][msg.sender].frequency;
        pullPayments[_customer][msg.sender].numberOfPayments = pullPayments[_customer][msg.sender].numberOfPayments - 1;
    }

    pullPayments[_customer][msg.sender].lastPaymentTimestamp = now;
    token.transferFrom(
        _customer,
        pullPayments[_customer][msg.sender].treasuryAddress,
        amountInPMA
    );

    emit LogPullPaymentExecuted(
        _customer,
        pullPayments[_customer][msg.sender].paymentID,
        pullPayments[_customer][msg.sender].businessID,
        pullPayments[_customer][msg.sender].uniqueReferenceID
    );
}
```

Notes:

For this attack to succeed, it is required that the user has approved the appropriate amount of funds to be transferred. A potential change that mitigates the issue would be the following: For each unique transaction id (`uniqueReferenceID`), there should be a flag that indicates whether the payment has been processed. This check can be added in the `isValidPullPaymentExecutionRequest` modifier. The downside of this recommendation is that the smart contract should iterate over a list each time a pull payment execution is requested.

Business Impact	Likelihood	Severity
Malicious executors can extract funds from customers.	Low	High

Resolution by PumaPay

The fix[4] to this issue(both in **V1** and **V2)** introduces a new parameter(`_paymentNumber`) in the calling function which ensures that the same payment cannot be executed as the number of payments left(decreases in every payment) needs to match the payment number which is provided by the executor.

```
function executePullPayment(address _customerAddress, bytes32 _paymentID, uint256 _paymentNumber)
public
paymentExists(_customerAddress, msg.sender)
isValidPullPaymentExecutionRequest(_customerAddress, msg.sender, _paymentID, _paymentNumber)
```

```
modifier isValidPullPaymentExecutionRequest(address _customerAddress, address _pullPaymentExecutor, bytes32 _paymentID, uint256 _paymentNumber)
{
    require(pullPayments[_customerAddress][_pullPaymentExecutor].numberOfPayments == _paymentNumber,
        "Invalid pull payment execution request - Pull payment number of payment is invalid");
```

[4] https://github.com/pumapayio/smart-contracts/commit/c248be94bb00ac19eb0d53629a4698f86da3d3f9

Replay Attack in Pull Payment Registration

Description:

A replay attack similar to the previous one also exists, for both **V1** and **V2**, in the `registerPullPayment` function. Specifically, a malicious executor can register the same pull payment by replaying the signature acquired by the user during the payment registration. The function does not check whether the same transaction id has been registered in the past. Eventually, the malicious executor can execute multiple(same) pull payments that do not have their state changed.

For **V1**:

```
function registerPullPayment(
    uint8 v,
    bytes32 r,
    bytes32 s,
    bytes32[2] memory _ids, // [0] paymentID, [1] businessID
    address[3] memory _addresses, // [0] customer, [1] pull payment executor, [2] treasury wallet
    string memory _currency,
    string memory _uniqueReferenceID,
    uint256 _initialPaymentAmountInCents,
    uint256 _fiatAmountInCents,
    uint256 _frequency,
    uint256 _numberOfPayments,
    uint256 _startTimestamp
)
public
isExecutor()
{
    require(_ids[0].length > 0, "Payment ID is empty.");
    require(_ids[1].length > 0, "Business ID is empty.");
    require(bytes(_currency).length > 0, "Currency is empty.");
    require(bytes(_uniqueReferenceID).length > 0, "Unique Reference ID is empty.");
    require(_addresses[0] != address(0), "Customer Address is ZERO_ADDRESS.");
    require(_addresses[1] != address(0), "Beneficiary Address is ZERO_ADDRESS.");
    require(_addresses[2] != address(0), "Treasury Address is ZERO_ADDRESS.");
    require(_fiatAmountInCents > 0, "Payment amount in fiat is zero.");
    require(_frequency > 0, "Payment frequency is zero.");
    require(_frequency < OVERFLOW_LIMITER_NUMBER, "Payment frequency is higher thant the overflow limit.");
    require(_numberOfPayments > 0, "Payment number of payments is zero.");
    require(_numberOfPayments < OVERFLOW_LIMITER_NUMBER, "Payment number of payments is higher thant the overflow limit.");
    require(_startTimestamp > 0, "Payment start time is zero.");
    require(_startTimestamp < OVERFLOW_LIMITER_NUMBER, "Payment start time is higher thant the overflow limit.");
```

For **V2**:

```
function registerPullPayment(
    uint8 v,
    bytes32 r,
    bytes32 s,
    bytes32[4] memory _paymentDetails, // 0 paymentID, 1 businessID, 2 uniqueReferenceID, 3 paymentType
    address[3] memory _addresses, // 0 customer, 1 pull payment executor, 2 treasury
    uint256[3] memory _paymentAmounts, // 0 _initialConversionRate, 1 _fiatAmountInCents, 2 _initialPaymentAmountInCents
    uint256[4] memory _paymentTimestamps, // 0 _frequency, 1 _numberOfPayments, 2 _startTimestamp, 3 _trialPeriod
    string memory _currency
)
public
isExecutor()
isValidPaymentType(_paymentDetails[3])
{
    require(_paymentDetails[0].length > 0, "Payment ID is empty.");
    require(_paymentDetails[1].length > 0, "Business ID is empty.");
    require(_paymentDetails[2].length > 0, "Unique Reference ID is empty.");

    require(_addresses[0] != address(0), "Customer Address is ZERO_ADDRESS.");
    require(_addresses[1] != address(0), "Beneficiary Address is ZERO_ADDRESS.");
    require(_addresses[2] != address(0), "Treasury Address is ZERO_ADDRESS.");

    require(_paymentAmounts[0] > 0, "Initial conversion rate is zero.");
    require(_paymentAmounts[1] > 0, "Payment amount in fiat is zero.");
    require(_paymentTimestamps[0] > 0, "Payment frequency is zero.");
    require(_paymentTimestamps[1] > 0, "Payment number of payments is zero.");
    require(_paymentTimestamps[2] > 0, "Payment start time is zero.");

    require(_paymentAmounts[0] < OVERFLOW_LIMITER_NUMBER, "Initial conversion rate is higher thant the overflow limit.");
    require(_paymentAmounts[1] < OVERFLOW_LIMITER_NUMBER, "Payment amount in fiat is higher thant the overflow limit.");
    require(_paymentAmounts[2] < OVERFLOW_LIMITER_NUMBER, "Payment initial amount in fiat is higher thant the overflow limit.");
    require(_paymentTimestamps[0] < OVERFLOW_LIMITER_NUMBER, "Payment frequency is higher thant the overflow limit.");
    require(_paymentTimestamps[1] < OVERFLOW_LIMITER_NUMBER, "Payment number of payments is higher thant the overflow limit.");
    require(_paymentTimestamps[2] < OVERFLOW_LIMITER_NUMBER, "Payment start time is higher thant the overflow limit.");
    require(_paymentTimestamps[3] < OVERFLOW_LIMITER_NUMBER, "Payment trial period is higher thant the overflow limit.");

    require(bytes(_currency).length > 0, "Currency is empty");
```

Notes:

Similarly to the recommendations provided in the attack found in pull payment execution, the attack is mitigated by checking if the pull payment already exists.

Business Impact	Likelihood	Severity
Malicious executors can extract funds from customers.	Low	High

Resolution by PumaPay

In **V1**[5] the issue is fixed with the introduction of the function `doesPaymentExist`. This function checks if an existing pull payment already exists for a given pair (executor-client) therefore it prevents malicious executors from registering the same pull payment multiple times.

```
function doesPaymentExist(address _customerAddress, address _pullPaymentExecutor)
internal
view
returns (bool) {
    return (
    bytes(pullPayments[_customerAddress][_pullPaymentExecutor].currency).length > 0 &&
    pullPayments[_customerAddress][_pullPaymentExecutor].fiatAmountInCents > 0 &&
    pullPayments[_customerAddress][_pullPaymentExecutor].frequency > 0 &&
    pullPayments[_customerAddress][_pullPaymentExecutor].startTimestamp > 0 &&
    pullPayments[_customerAddress][_pullPaymentExecutor].numberOfPayments > 0 &&
    pullPayments[_customerAddress][_pullPaymentExecutor].nextPaymentTimestamp > 0
    );
}
```

In **V2** the issue is fixed by taking advantage of the `paymentIds` variable and requires that for a new registration there is no existing value assigned. This prevents malicious executors from registering the same pull payment.

```
modifier paymentExists(address _customerAddress, address _pullPaymentExecutor) {
    require(pullPayments[_customerAddress][_pullPaymentExecutor].paymentIds[0] != "", "Pull Payment does not exists.");
    _;
}
```

[5] https://github.com/pumapayio/smart-contracts/commit/c248be94bb00ac19eb0d53629a4698f86da3d3f9

Electi Consulting Ltd

Alexandros Hasikos

Head of R&D

01/08/2019