Jorge Rodriguez

CYBERSEC CONTRACT AUDIT REPORT

HolderFinance

Introduction

During December of 2020, Holder Finance engaged CTDSec to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. Holder Finance provided CTDSec with access to their code repository and whitepaper. **On December 11, the development team solved all the incidents presented in the report and therefore it is safe to deploy.**

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bugfree status. The audit documentation is for discussion purposes only.

I always recommend having a bug bounty program opened to detect future bugs.

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Autoxify contract followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code files are considered in-scope for the review:

Escrow_single_file_no_import.Sol

Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

● Correctness of the protocol implementation [Result OK]

● User funds are secure on the blockchain and cannot be transferred without user permission **[Solved by DEV team]**

● Vulnerabilities within each component as well as secure interaction between the network components [Result OK]

● Correctly passing requests to the network core [Result OK]

● Data privacy, data leaking, and information integrity [Result OK]

● Susceptible to reentrancy attack **[Solved by DEV team]**

● Key management implementation: secure private key storage and proper management of encryption and signing keys [Result OK]

● Handling large volumes of network traffic [Result OK]

● Resistance to DDoS and similar attacks [Result OK]

● Aligning incentives with the rest of the network [Result OK]

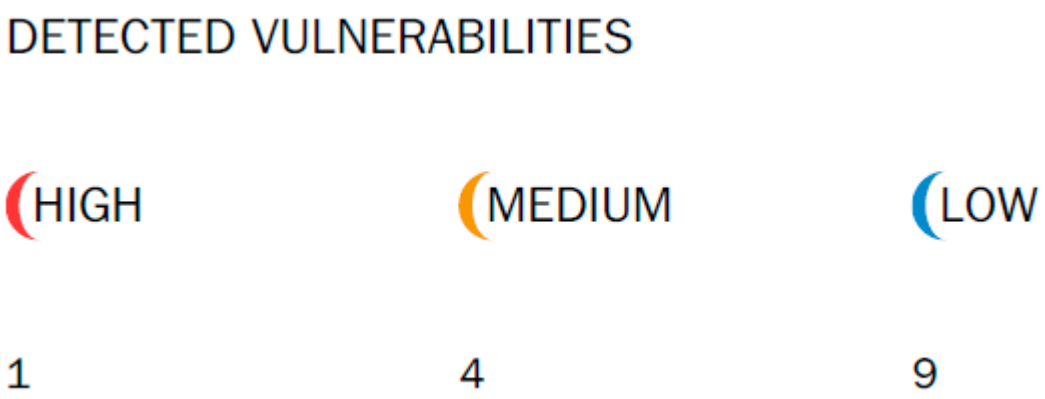
● Any attack that impacts funds, such as draining or manipulating of funds **[Solved by DEV team]**

● Mismanagement of funds via transactions [Result OK]

● Inappropriate permissions and excess authority [Result OK]

● Special token issuance model [Result OK]

Contract - Escrow_single_file_no_import.Sol



<u>All issues were solved</u> by the Development team for further information please read the last page (Summary of the audit).

ISSUES

HIGH- SWC-105

Any sender can withdraw Ether from the contract account.

Arbitrary senders other than the contract creator can profitably extract Ether from the contract account. Verify the business logic carefully and make sure that appropriate security controls are in place to prevent unexpected loss of funds.

Locations

```
281 function _transferToken(address _to, uint256 _amount, address _token) private {
282 if (_token == address(0)) {
283 _to.transfer(_amount);
284 } else {
285 ERC20 token = ERC20(_token);
```

MEDIUM SWC-107

Write to persistent state following external call.

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Locations


```
260
261 function _createDeal(address _walletOffer, address _tokenAddressOffer, uint256 _amountOffered, address _tokenAddressRequest, uint256 _amountRequest) private {
262 uint256 dealID = deals.push(Deal(0,_walletOffer, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest,address(0),now,0,DealStatus.ACTIVE)) -1;
263 deals[dealID].ID = dealID;
264 emit NewDeal(dealID, _walletOffer, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest);
```

MEDIUM - SWC-107

Read of persistent state following external call.

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the

callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Locations

```
261 function _createDeal(address _walletOffer, address _tokenAddressOffer, uint256 _amountOffered, address _tokenAddressRequest, uint256 _amountRequest) private {
262 uint256 dealID = deals.push(Deal(0,_walletOffer, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest,address(0),now,0,DealStatus.ACTIVE)) -1;
263 deals[dealID].ID = dealID;
264 emit NewDeal(dealID, _walletOffer, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest);
265 }
```

MEDIUM - SWC-107

Write to persistent state following external call.

The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the

callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Locations

```
261 function _createDeal(address _walletOffer, address _tokenAddressOffer, uint256 _amountOffered, address _tokenAddressRequest, uint256 _amountRequest) private {
262 uint256 dealID = deals.push(Deal(0,_walletOffer, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest,address(0),now,0,DealStatus.ACTIVE)) -1;
263 deals[dealID].ID = dealID;
264 emit NewDeal(dealID, _walletOffer, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest);
265 }
```

MEDIUM - SWC-113

Multiple calls are executed in the same transaction.

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Locations

```
281 function _transferToken(address _to, uint256 _amount, address _token) private {
282 if (_token == address(0)) {
283 _to.transfer(_amount);
284 } else {
285 ERC20 token = ERC20(_token);
```

LOW - SWC-103

A floating pragma is set.

The current pragma Solidity directive is ""^0.4.23"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Locations

```
1 pragma solidity ^0.4.23;
2 pragma experimental ABIEncoderV2;
```

LOW - SWC-107

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract.

Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in

place.

Locations

```
271 require(msg.value >= _amountOffered, "ETH: Balance is not enough");
272 } else {
273 require(token.allowance(msg.sender,address(this)) >= _amountOffered, "ERC20: Allowance not enough");
274 require(token.balanceOf(msg.sender) >= _amountOffered, "ERC20: Balance is not enough");
275 token.transferFrom(msg.sender, address(this), _amountOffered); // sposto i fondi all'interno dello smart contract
```

LOW - SWC-107

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract.

Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

Locations

```
273 require(token.allowance(msg.sender,address(this)) >= _amountOffered, "ERC20: Allowance not enough");
274 require(token.balanceOf(msg.sender) >= _amountOffered, "ERC20: Balance is not enough");
275 token.transferFrom(msg.sender, address(this), _amountOffered); // sposto i fondi all'interno dello smart contract
276 }
277 _createDeal(msg.sender, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest);
```

LOW - SWC-107

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract.

Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in

place.

Locations


```
284 } else {
285 ERC20 token = ERC20(_token);
286 token.transfer(_to, _amount);
287 }
288 }
```

LOW - SWC-113

Multiple calls are executed in the same transaction.

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Locations

```
272 } else {
273 require(token.allowance(msg.sender,address(this)) >= _amountOffered, "ERC20: Allowance not enough");
274 require(token.balanceOf(msg.sender) >= _amountOffered, "ERC20: Balance is not enough");
275 token.transferFrom(msg.sender, address(this), _amountOffered); // sposto i fondi all'interno dello smart contract
276 }
```

LOW - SWC-113

Multiple calls are executed in the same transaction.

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a

malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Locations

```
284 } else {
285 ERC20 token = ERC20(_token);
286 token.transfer(_to, _amount);
287 }
288 }
```

LOW - SWC-113

Multiple calls are executed in the same transaction.

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Locations

```
273 require(token.allowance(msg.sender,address(this)) >= _amountOffered, "ERC20: Allowance not enough");
274 require(token.balanceOf(msg.sender) >= _amountOffered, "ERC20: Balance is not enough");
275 token.transferFrom(msg.sender, address(this), _amountOffered); // sposto i fondi all'interno dello smart contract
276 }
277 _createDeal(msg.sender, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest);
```

LOW - SWC-123

Requirement violation.

A requirement was violated in a nested call and the call was reverted as a result. Make sure valid inputs are provided to the nested call (for instance, via passed arguments).

Locations

```
324 } else {
325 ERC20 token = ERC20(_tokenRequest);
326 require(token.allowance(msg.sender,address(this)) >= _amountRequest, "ERC20: Allowance not enough");
327 require(token.balanceOf(msg.sender) >= _amountRequest, "ERC20: Balance is not enough");
328 token.transferFrom(msg.sender, address(this), _amountRequest);
```



```
}

contract Escrow is Ownable {

using SafeMath for uint256;

/**********************************
* EVENTS ************************
**********************************/
event NewDeal(uint256 dealID, address _dealOwner, address _tokenOffer, uint256 _amountOffered, address _tokenRequest, uint256 _amountRequest);
event DealCanceled(uint256 dealID);
event DealAccepted(uint256 dealID, address _buyer, address _tokenRequest, uint256 _amountRequest);


/**********************************
* CONSTANTS *********************
**********************************/

/**********************************
* VARIABLES *********************
**********************************/
enum DealStatus { NA, ACTIVE, CLOSED, CANCELED }

struct Deal {
uint256 ID;
address walletOffer;
address tokenAddressOffer;
uint256 amountOffered;
address tokenAddressRequest;
uint256 amountRequest;
address walletBuyer;
uint256 creationDate;
uint256 statusDate;
DealStatus status; // 0 Not available, 1 Active, 2 Closed, 3 Canceled
}
```



```
226
227 Deal[] public deals;
228
229 /*********************************
230 * MODIFIER ***********************
231 *********************************/
232
233 modifier isDealOwner(uint256 _dealID, address _dealOwner) {
234 require(deals[_dealID].walletOffer == _dealOwner, "DEAL: you are not the owner");
235 _;
```

```
236 }
237
238 modifier isNotDealOwner(uint256 _dealID, address _dealOwner) {
239 require(deals[_dealID].walletOffer != _dealOwner, "DEAL: you are the owner");
240 _;
241 }
242
243 modifier isDealStatus(uint256 _dealID, DealStatus _dealStatus) {
244 require(deals[_dealID].status == _dealStatus, "DEAL: status is not valid");
245 _;
246 }
247
248 /*********************************
249 * COSTRUCTOR *********************
250 *********************************/
251 constructor() public {
252
253 }
254
255 /*********************************
256 * FUNCTION ***********************
257 *********************************/
258
259 // ETH 0x0000000000000000000000000000000000000000
260
261 function _createDeal(address _walletOffer, address _tokenAddressOffer, uint256 _amountOffered, address _tokenAddressRequest, uint256 _amountRequest) private {
262 uint256 dealID = deals.push(Deal(0,_walletOffer, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest,address(0),now,0,DealStatus.ACTIVE)) -1;
263 deals[dealID].ID = dealID;
264 emit NewDeal(dealID, _walletOffer, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest);
265 }
```



```
266
267 function createDeal(address _tokenAddressOffer, uint256 _amountOffered, address _tokenAddressRequest, uint256 _amountRequest) payable external {
268 ERC20 token = ERC20(_tokenAddressOffer);
269
270 if (_tokenAddressOffer == address(0)) {
271 require(msg.value >= _amountOffered, "ETH: Balance is not enough");
272 } else {
273 require(token.allowance(msg.sender,address(this)) >= _amountOffered, "ERC20: Allowance not enough");
274 require(token.balanceOf(msg.sender) >= _amountOffered, "ERC20: Balance is not enough");
275 token.transferFrom(msg.sender, address(this), _amountOffered); // sposto i fondi all'interno dello smart contract
276 }
277 _createDeal(msg.sender, _tokenAddressOffer, _amountOffered, _tokenAddressRequest, _amountRequest);
278
279 }
280
281 function _transferToken(address _to, uint256 _amount, address _token) private {
282 if (_token == address(0)) {
283 _to.transfer(_amount);
284 } else {
285 ERC20 token = ERC20(_token);
286 token.transfer(_to, _amount);
287 }
288 }
289
290 function _cancelDeal(uint256 _dealID) private {
291 deals[_dealID].status = DealStatus.CANCELED;
292 deals[_dealID].statusDate = now;
293
294 address _to = deals[_dealID].walletOffer;
295 address _token = deals[_dealID].tokenAddressOffer;
296 uint256 _amount = deals[_dealID].amountOffered;
297
298 // send back the funds to the Owner
```



```
299 _transferToken(_to, _amount, _token);
300 emit DealCanceled(_dealID);
301 }
302
303 function cancelDeal(uint256 _dealID) isDealOwner(_dealID, msg.sender) isDealStatus(_dealID, DealStatus.ACTIVE) external {
304 _cancelDeal(_dealID);
305 }
306
307 function revertDeal(uint256 _dealID) isDealStatus(_dealID, DealStatus.ACTIVE) onlyOwner() external {
308 _cancelDeal(_dealID);
309 }
310
311 function acceptDeal(uint256 _dealID) isNotDealOwner(_dealID, msg.sender) isDealStatus(_dealID, DealStatus.ACTIVE) payable external {
312
313 // retrieve data from the deal
314 address _offer = deals[_dealID].walletOffer;
315 address _tokenRequest = deals[_dealID].tokenAddressRequest;
316 uint256 _amountRequest = deals[_dealID].amountRequest;
317
318 address _buyer = msg.sender;
319 address _tokenOffered = deals[_dealID].tokenAddressOffer;
320 uint256 _amountOffered = deals[_dealID].amountOffered;
321
322 if (_tokenRequest == address(0)) {
323 require(msg.value >= _amountRequest, "ETH: Balance is not enough");
324 } else {
325 ERC20 token = ERC20(_tokenRequest);
326 require(token.allowance(msg.sender,address(this)) >= _amountRequest, "ERC20: Allowance not enough");
327 require(token.balanceOf(msg.sender) >= _amountRequest, "ERC20: Balance is not enough");
328 token.transferFrom(msg.sender, address(this), _amountRequest);
329 }
```



```
331  // update the deals array
332  deals[_dealID].status = DealStatus.CLOSED;
333  deals[_dealID].statusDate = now;
334  deals[_dealID].walletBuyer = msg.sender;
335
336  // close the deal, sending the funds to the right parties
337  _transferToken(_offer, _amountRequest, _tokenRequest);
338  _transferToken(_buyer, _amountOffered, _tokenOffered);
339
340  emit DealAccepted(_dealID, _buyer, _tokenRequest, _amountRequest);
341  }
342
343  // GETTER - current Blockchain Time
344  function currentTime() public view returns(uint256) {
345  return(now);
346  }
347
348  // GETTER - numbers of deals
349  function totalDeals() public view returns(uint256) {
350  return(deals.length);
351  }
352
353  }
```

LOW - SWC-134

Call with hardcoded gas amount.

The highlighted function call forwards a fixed amount of gas. This is discouraged as the gas cost of EVM instructions may change in the future, which could break this contract's

assumptions. If this was done to prevent reentrancy attacks, consider alternative methods such as the checks-effects-interactions pattern or reentrancy locks instead.

Locations


```
281 function _transferToken(address _to, uint256 _amount, address _token) private {
282 if (_token == address(0)) {
283 _to.transfer(_amount);
284 } else {
285 ERC20 token = ERC20(_token);
```


Summary of the Audit

Dev notes on how issues has been solved:

HIGH- SWC-105

Any sender can withdraw Ether from the contract account.

The private method _transferToken is executed by 3 external functions:

function cancelDeal(uint256 _dealID) isDealOwner(_dealID, msg.sender) isDealStatus(_dealID, DealStatus.ACTIVE) nonReentrant() external {...}

function revertDeal(uint256 _dealID) isDealStatus(_dealID, DealStatus.ACTIVE) onlyOwner() nonReentrant() external {...}

function acceptDeal(uint256 _dealID) isNotDealOwner(_dealID, msg.sender) isDealStatus(_dealID, DealStatus.ACTIVE) nonReentrant() payable external {..}

We applied specific security modifier in order to prevent unexpected loss of funds

MEDIUM - SWC-107

Write to persistent state following external call.

We accept the recommendation and we added a new library from OpenZeppelin

```
contract ReentrancyGuard {
    // Booleans are more expensive than uint256 or any type that takes up a full
    // word because each write operation emits an extra SLOAD to first read the
    // slot's contents, replace the bits taken up by the boolean, and then write
    // back. This is the compiler's defense against contract upgrades and
    // pointer aliasing, and it cannot be disabled.

    // The values being non-zero value makes deployment a bit more expensive,
    // but in exchange the refund on every call to nonReentrant will be lower in
    // amount. Since refunds are capped to a percentage of the total
    // transaction's gas, it is best to keep them low in cases like this one, to
    // increase the likelihood of the full refund coming into effect.
    uint256 private constant _NOT_ENTERED = 1;
    uint256 private constant _ENTERED = 2;

    uint256 private _status;

    constructor () internal {
        _status = _NOT_ENTERED;
    }

    /**
```



```
    * @dev Prevents a contract from calling itself, directly or indirectly.
    * Calling a `nonReentrant` function from another `nonReentrant`
    * function is not supported. It is possible to prevent this from happening
    * by making the `nonReentrant` function external, and make it call a
    * `private` function that does the actual work.
    */
   modifier nonReentrant() {
      // On the first call to nonReentrant, _notEntered will be true
      require(_status != _ENTERED, "ReentrancyGuard: reentrant call");

      // Any calls to nonReentrant after this point will fail
      _status = _ENTERED;

      _;

      // By storing the original value once again, a refund is triggered (see
      // https://eips.ethereum.org/EIPS/eip-2200)
      _status = _NOT_ENTERED;
   }
}
```

the nonReentrant() modifier is applied to all the external methods highlighted in this security audit.

After working with the development team **they fixed all critical issues and the contract is safe to deploy**.

