



SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Buffer

Prepared for:

Buffer Finance

Prepared by:

Sherlock

Lead Security Expert:

0x52

Dates Audited:

November 14 - November 20, 2022

Prepared on:

December 8, 2022

Introduction

Buffer Finance is a non-custodial, exotic options trading platform built to trade short-term price volatility and hedge risk of high-leverage positions.

Scope

The following contracts in the [Buffer-Protocol-v2 @ 83d85d9](#) repo are in scope.

- `BufferBinaryOptions.sol`
- `BufferBinaryPool.sol`
- `BufferRouter.sol`
- `OptionsConfig.sol`
- `ReferralStorage.sol`

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
5	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues



0x52
__141345__
Ch_301
KingNFT
bin2chen
hansfrieze
adriro
kaliberpoziomka
HonorLt
eierina
pashov

Bnke0x0
cccz
rvierdiev
Deivitto
jonatascm
supernova
dipp
aphak5010
0x4non
m_Rassska
sach1r0

minhtrng
0xheynacho
0xadrii
ak1
0x007
peanuts
0xcc
eyexploit
ctf_sec



Issue H-1: When private keeper mode is off users can queue orders with the wrong asset

Source: <https://github.com/sherlock-audit/2022-11-buffer-judging/issues/85>

Found by

kaliberpoziomka, KingNFT, hansfrieze, adriro, 0x52, bin2chen

Summary

After an order is initiated, it must be filled by calling `resolveQueuedTrades`. This function validates that the asset price has been signed but never validates that the asset being passed in matches the asset of the `queuedTrade`. When private keeper mode is off, which is the default state of the contract, this can be abused to cause huge loss of funds.

Vulnerability Detail

```
for (uint32 index = 0; index < params.length; index++) {
    OpenTradeParams memory currentParams = params[index];
    QueuedTrade memory queuedTrade = queuedTrades[
        currentParams.queueId
    ];
    bool isSignerVerified = _validateSigner(
        currentParams.timestamp,
        currentParams.asset,
        currentParams.price,
        currentParams.signature
    );
    // Silently fail if the signature doesn't match
    if (!isSignerVerified) {
        emit FailResolve(
            currentParams.queueId,
            "Router: Signature didn't match"
        );
        continue;
    }
    if (
        !queuedTrade.isQueued ||
        currentParams.timestamp != queuedTrade.queuedTime
    ) {
        // Trade has already been opened or cancelled or the timestamp is wrong.
        // So ignore this trade.
        continue;
    }
}
```



```

    }

    // If the opening time is much greater than the queue time then cancel the
    → trad
    if (block.timestamp - queuedTrade.queuedTime <= MAX_WAIT_TIME) {
        _openQueuedTrade(currentParams.queueId, currentParams.price);
    } else {
        _cancelQueuedTrade(currentParams.queueId);
        emit CancelTrade(
            queuedTrade.user,
            currentParams.queueId,
            "Wait time too high"
        );
    }

    // Track the next queueIndex to be processed for user
    userNextQueueIndexToProcess[queuedTrade.user] =
        queuedTrade.userQueueIndex +
        1;
}

```

`BufferRouter#resolveQueueTrades` never validates that the asset passed in for `params` is the same asset as the `queuedTrade`. It only validates that the price is the same, then passes the price and `queueId` to `_openQueuedTrade`:

```

function _openQueuedTrade(uint256 queueId, uint256 price) internal {
    QueuedTrade storage queuedTrade = queuedTrades[queueId];
    IBufferBinaryOptions optionsContract = IBufferBinaryOptions(
        queuedTrade.targetContract
    );

    bool isSlippageWithinRange = optionsContract.isStrikeValid(
        queuedTrade.slippage,
        price,
        queuedTrade.expectedStrike
    );

    if (!isSlippageWithinRange) {
        _cancelQueuedTrade(queueId);
        emit CancelTrade(
            queuedTrade.user,
            queueId,
            "Slippage limit exceeds"
        );
    }

    return;
}

```



```
...

optionParams.totalFee = revisedFee;
optionParams.strike = price;
optionParams.amount = amount;

uint256 optionId = optionsContract.createFromRouter(
    optionParams,
    isReferralValid
);
```

Inside `_openQueuedTrade` it checks that the price is within the slippage bounds of the order, cancelling if its not. Otherwise it uses the price to open an option. According to documentation, the same router will be used across a large number of assets/pools, which means the publisher for every asset is the same, given that router only has one publisher variable.

Examples:

Imagine two assets are listed that have close prices, asset A = \$0.95 and asset B = \$1. An adversary could create a call that expires in 10 minutes on asset B with 5% slippage, then immediately queue it with the price of asset A. \$0.95 is within the slippage bounds so it creates the option with a strike price of \$0.95. Since the price of asset B is actually \$1 the adversary will almost guaranteed make money, stealing funds from the LPs. This can be done back and forth between both pools until pools for both assets are drained.

In a similar scenario, if the price of the assets are very different, the adversary could use this to DOS another user by always calling queue with the wrong asset, causing the order to be cancelled.

Impact

Adversary can rug LPs and DOS other users

Code Snippet

<https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferRouter.sol#L136-L185>

Tool used

Manual Review



Recommendation

Pass the asset address through so the BufferBinaryOptions contract can validate it is being called with the correct asset

Discussion

0x00052

Fixed in [PR#2](#)

Changes look good. The asset pair is now stored in BufferBinaryOptions and BufferRouter directly reads it from there instead of relying on user/keeper to supply the correct asset.



Issue H-2: Design of BufferBinaryPool allows LPs to game option expiry

Source: <https://github.com/sherlock-audit/2022-11-buffer-judging/issues/82>

Found by

__141345__, 0x52

Summary

When an option is created, enough collateral is locked in BufferBinaryPool to cover a payout should it close ITM. As long as an LP isn't locked (trivially 10 minutes) and there is sufficient liquidity they can cash out their shares for underlying. The price and expiration of all options are public by design, meaning an LP can know with varying degrees of certainty if they will make or lose money from an option expiry. The result is that there will be a race to withdraw capital before any option expires ITM. LPs who make it out first won't lose any money, leaving all other LPs to hold the bags.

On the flip-side of this when there are large options expiring OTM, LPs will rush to stake their capital in the pool. This allows them to claim the payout while experiencing virtually zero risk, since they can immediately withdraw after 10 minutes.

Vulnerability Detail

See summary.

Impact

LPs can game option expiry at the expense of other LPs

Code Snippet

<https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferBinaryPool.sol#L124-L126>

Tool used

Manual Review



Recommendation

I strongly recommend an epoch based withdraw and deposit buffer to prevent a situation like this. Alternatively increasing lockupPeriod would be a quicker, less precise fix.

Discussion

bufferfinance

Yes we were planning to adjust the lockup accordingly.

bufferfinance

Doesn't need to be fixed. The admin will adjust the config accordingly.

0x00052

Fixed in [PR#9](#)

lockupPeriod is now set in constructor so that it is adjustable



Issue M-1: resolveQueuedTrades() ERC777 re-enter to steal funds

Source: <https://github.com/sherlock-audit/2022-11-buffer-judging/issues/130>

Found by

HonorLt, bin2chen, KingNFT

Summary

_openQueuedTrade() does not follow the “Checks Effects Interactions” principle and may lead to re-entry to steal the funds

https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html

Vulnerability Detail

The prerequisite is that tokenX is ERC777 e.g. “sushi”

1. resolveQueuedTrades() call _openQueuedTrade()
2. in _openQueuedTrade() call "tokenX.transfer(queuedTrade.user)" if (revisedFee < queuedTrade.totalFee) before set queuedTrade.isQueued = false;

```
function _openQueuedTrade(uint256 queueId, uint256 price) internal {  
    ...  
    if (revisedFee < queuedTrade.totalFee) {  
        tokenX.transfer( /**@audit call transfer , if ERC777 , can re-enter  
↳    ***/  
        queuedTrade.user,  
        queuedTrade.totalFee - revisedFee  
    );  
    }  
  
    queuedTrade.isQueued = false; /**@audit change state***/  
}
```

3.if ERC777 re-enter to #cancelQueuedTrade() to get tokenX back,it can close, because queuedTrade.isQueued still equal true 4. back to _openQueuedTrade() set queuedTrade.isQueued = false 5.so steal tokenX

Impact

if tokenX equal ERC777 can steal token



Code Snippet

<https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferRouter.sol#L350>

Tool used

Manual Review

Recommendation

follow “Checks Effects Interactions”

```
function _openQueuedTrade(uint256 queueId, uint256 price) internal {  
    ...  
+   queuedTrade.isQueued = false;  
    // Transfer the fee to the target options contract  
    IERC20 tokenX = IERC20(optionsContract.tokenX());  
    tokenX.transfer(queuedTrade.targetContract, revisedFee);  
  
-   queuedTrade.isQueued = false;  
    emit OpenTrade(queuedTrade.user, queueId, optionId);  
}
```

Discussion

0x00052

Fixed in [PR#8](#)

Changes look good. Trade is now removed from queue before sending user refund during option opening to avoid potential reentrancy. Canceling already removed trade before sending refund so no change needed there.



Issue M-2: The `_fee()` function is wrongly implemented in the code

Source: <https://github.com/sherlock-audit/2022-11-buffer-judging/issues/95>

Found by

Ch_301

Summary

`_fee()` function is wrongly implemented in the code so the protocol will get fewer fees and the trader will earn more

Vulnerability Detail

```
(uint256 unitFee, , ) = _fees(10**decimals(), settlementFeePercentage);  
amount = (newFee * 10**decimals()) / unitFee;
```

let's say we have: newFee 100 USDC USDC Decimals is 6 settlementFeePercentage is 20% ==> 200

The unitFee will be 520_000

amount = (100 * 1_000_000) / 520_000 amount = 192 USDC Which is supposed to be
amount = 160 USDC

Impact

The protocol will earn fees less than expected

Code Snippet

```
function checkParams(OptionParams calldata optionParams)  
    external  
    view  
    override  
    returns (  
        uint256 amount,  
        uint256 revisedFee,  
        bool isReferralValid  
    )  
{  
    require(  
        assetCategory != AssetCategory.Forex ||
```



```

        isInCreationWindow(optionParams.period),
        "030"
    );

    uint256 maxAmount = getMaxUtilization();

    // Calculate the max fee due to the max txn limit
    uint256 maxPerTxnFee = ((pool.availableBalance() *
        config.optionFeePerTxnLimitPercent()) / 100e2);
    uint256 newFee = min(optionParams.totalFee, maxPerTxnFee);

    // Calculate the amount here from the new fees
    uint256 settlementFeePercentage;
    (
        settlementFeePercentage,
        isReferralValid
    ) = _getSettlementFeePercentage(
        referral.codeOwner(optionParams.referralCode),
        optionParams.user,
        _getbaseSettlementFeePercentage(optionParams.isAbove),
        optionParams.traderNFTId
    );
    (uint256 unitFee, , ) = _fees(10**decimals(), settlementFeePercentage);
    amount = (newFee * 10**decimals()) / unitFee;

```

<https://github.com/bufferfinance/Buffer-Protocol-v2/blob/83d85d9b18f1a4d09c728adaa0dde4c37406dfed/contracts/core/BufferBinaryOptions.sol#L318-L353>

```

function _fees(uint256 amount, uint256 settlementFeePercentage)
    internal
    pure
    returns (
        uint256 total,
        uint256 settlementFee,
        uint256 premium
    )
{
    // Probability for ATM options will always be 0.5 due to which we can skip
    ↪ using BSM
    premium = amount / 2;
    settlementFee = (amount * settlementFeePercentage) / 1e4;
    total = settlementFee + premium;
}

```

<https://github.com/bufferfinance/Buffer-Protocol-v2/blob/83d85d9b18f1a4d09c72>



[8adaa0dde4c37406dfed/contracts/core/BufferBinaryOptions.sol#L424-L437](#)

Tool used

Manual Review

Recommendation

The `_fee()` function needs to calculate the fees in this way

```
total_fee = (5000 * amount) / (10000 - sf)
```

Discussion

0x00052

Fixed in [PR#22](#)

Changes look good. New math returns correct values. Validated for both unit fee and option amount.



Issue M-3: resolveQueuedTrades is intended to be non atomic but invalid signature can still cause entire transaction to revert

Source: <https://github.com/sherlock-audit/2022-11-buffer-judging/issues/84>

Found by

0x52

Summary

BufferRouter#resolveQueuedTrades and unlockOptions attempt to be non atomic (i.e. doesn't revert the transaction if one fails) but an invalid signature can still cause the entire transaction to revert, because the ECDSA.recover sub call in _validateSigner can still revert.

Vulnerability Detail

```
function _validateSigner(
    uint256 timestamp,
    address asset,
    uint256 price,
    bytes memory signature
) internal view returns (bool) {
    bytes32 digest = ECDSA.toEthSignedMessageHash(
        keccak256(abi.encodePacked(timestamp, asset, price))
    );
    address recoveredSigner = ECDSA.recover(digest, signature);
    return recoveredSigner == publisher;
}
```

_validateSigner can revert at the ECDSA.recover sub call breaking the intended non atomic nature of BufferRouter#resolveQueuedTrades and unlockOptions.

Impact

BufferRouter#resolveQueuedTrades and unlockOptions don't function as intended if signature is malformed

Code Snippet

<https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferRouter.sol#L260-L271>



Tool used

Manual Review

Recommendation

Use a try statement inside `_validateSigner` to avoid any reverts:

```
function _validateSigner(
    uint256 timestamp,
    address asset,
    uint256 price,
    bytes memory signature
) internal view returns (bool) {
    bytes32 digest = ECDSA.toEthSignedMessageHash(
        keccak256(abi.encodePacked(timestamp, asset, price))
    );
-   address recoveredSigner = ECDSA.recover(digest, signature);
+   try ECDSA.recover(digest, signature) returns (address recoveredSigner) {
+       return recoveredSigner == publisher;
+   } else {
+       return false;
+   }
}
```

Discussion

bufferfinance

The protocol has been tested against wrong signatures. https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/tests/test_router.py#L815

0x00052

Escalate for 10 USDC.

My submission is valid and sponsor's comment here is inaccurate. `ECDSA.recover` will revert in the `_throwError` subcall under quite a few conditions not covered by their tests, including signature of invalid length and signature that resolve to `address(0)`.

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/24d1bb668a1152528a6e6d71c2e285d227ed19d9/contracts/utils/cryptography/ECDSA.sol#L88-L92>

sherlock-admin

Escalate for 10 USDC.



My submission is valid and sponsor's comment here is inaccurate. ECDSA.recover will revert in the _throwError subcall under quite a few conditions not covered by their tests, including signature of invalid length and signature that resolve to address(0).

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/24d1bb668a1152528a6e6d71c2e285d227ed19d9/contracts/utils/cryptography/ECDSA.sol#L88-L92>

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment (**do not create a new comment**).

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

hrishibhat

Escalation accepted

Invalid signatures resolving to address(0) reverts _validateSigner

sherlock-admin

Escalation accepted

Invalid signatures resolving to address(0) reverts _validateSigner

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

bufferfinance

Will fix this

0x00052

Fixed in [PR#28](#)

Changes look good. ECDSA.recover changed to ECDSA.tryRecover to prevent any revert when recovering signatures



Issue M-4: Insufficient support for fee-on-transfer tokens

Source: <https://github.com/sherlock-audit/2022-11-buffer-judging/issues/76>

Found by

supernova, rvierdiev, eierina, cccz, KingNFT, dipp, __141345__, Bnke0x0, jonatascm, pashov, Deivitto

Summary

The `BufferBinaryPool.sol` and `BufferRouter.sol` do not support fee-on-transfer tokens. If `tokenX` is a fee-on-transfer token, tokens received from users could be less than the amount specified in the transfer.

Vulnerability Detail

The `initiateTrade` function in `BufferRouter.sol` receives tokens from the user with amount set to `initiateTrade`'s `totalFee` input. If `tokenX` is a fee-on-transfer token then the actual amount received by `BufferRouter.sol` is less than `totalFee`. When a trade is opened, the protocol will send a `settlementFee` to `settlementFeeDisbursalContract` and a premium to `BufferBinaryPool.sol`, where the `settlementFee` is calculated using the incorrect, inflated `totalFee` amount. When the `totalFee` is greater than the fee required the user is reimbursed the difference. Since the `settlementFee` is greater than it should be the user receives less reimbursement.

In `BufferBinaryPool.sol`'s `lock` function, the premium for the order is sent from the Options contract to the Pool. The `totalPremium` state variable would be updated incorrectly if fee-on-transfer tokens were used.

The `_provide` function in `BufferBinaryPool.sol` receives `tokenXAmount` of `tokenX` tokens from the user and calculates the amount of shares to mint using the `tokenXAmount`. If fee-on-transfer tokens are used then the user would receive more shares than they should.

Impact

The protocol and users could suffer a loss of funds.

Code Snippet

[BufferRouter.sol#L86-L90](#)

[BufferBinaryPool.sol#L161](#)



BufferBinaryPool.sol#L236-L240

Tool used

Manual Review

Recommendation

Consider checking the balance of the contract before and after token transfers and using instead of the amount specified in the contract.

Discussion

0x00052

Only an issue if project intends to support fee-on-transfer tokens as underlying

bufferfinance

Not supporting fee-on-transfer tokens for now.

bufferfinance

Buffer won't be supporting fee-on-transfer tokens. Thus we are not fixing it.



Issue M-5: Limited support to a specific subset of ERC20 tokens

Source: <https://github.com/sherlock-audit/2022-11-buffer-judging/issues/73>

Found by

m_Rassska, ctf_sec, ak1, cccz, Bnke0x0, 0x007, minhtrng, rvierdiiev, 0xadrii, hansfrieze, 0xheynacho, HonorLt, bin2chen, eierina, aphak5010, __141345__, pashov, eyexploit, 0xcc, peanuts, sach1r0, 0x4non, adriro, jonatascm, Deivitto

Summary

Buffer contest states 'any ERC20 supported', therefore it should take into account all the different ways of signalling success and failure. This is not the case, as all ERC20's `transfer()`, `transferFrom()`, and `approve()` functions are either not verified at all or verified for returning true. As a result, depending on the ERC20 token, some transfer errors may result in passing unnoticed, and/or some successful transfer may be treated as failed.

Currently the only supported ERC20 tokens are the ones that fulfill both the following requirements:

- always revert on failure;
- always returns boolean true on success.

An example of a very well known token that is not supported is Tether USD (USDT).

IMPORTANT This issue is not the same as reporting that "return value must be verified to be true" where the checks are missing! Indeed **such a simplistic report should be considered invalid** as it still does not solve all the problems but rather introduces others. See Vulnerability Details section for rationale.

Vulnerability Detail

Tokens have different ways of signalling success and failure, and this affect mostly `transfer()`, `transferFrom()` and `approve()` in ERC20 tokens. While some tokens revert upon failure, others consistently return boolean flags to indicate success or failure, and many others have mixed behaviours.

See below a snippet of the USDT Token contract compared to the 0x's ZRX Token contract where the USDT Token transfer function does not even return a boolean value, while the ZRX token consistently returns boolean value hence returning false on failure instead of reverting.

USDT Token snippet (no return value) from Etherscan



```

function transferFrom(address _from, address _to, uint _value) public
↳ onlyPayloadSize(3 * 32) {
    var _allowance = allowed[_from][msg.sender];

    // Check is not needed because sub(_allowance, _value) will already throw if
    ↳ this condition is not met
    // if (_value > _allowance) throw;

    uint fee = (_value.mul(basisPointsRate)).div(10000);
    if (fee > maximumFee) {
        fee = maximumFee;
    }
    if (_allowance < MAX_UINT) {
        allowed[_from][msg.sender] = _allowance.sub(_value);
    }
    uint sendAmount = _value.sub(fee);
    balances[_from] = balances[_from].sub(_value);
    balances[_to] = balances[_to].add(sendAmount);
    if (fee > 0) {
        balances[owner] = balances[owner].add(fee);
        Transfer(_from, owner, fee);
    }
    Transfer(_from, _to, sendAmount);
}

```

ZRX Token snippet (consistently true or false boolean result) from Etherscan

```

function transferFrom(address _from, address _to, uint _value) returns (bool) {
    if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
    ↳ balances[_to] + _value >= balances[_to]) {
        balances[_to] += _value;
        balances[_from] -= _value;
        allowed[_from][msg.sender] -= _value;
        Transfer(_from, _to, _value);
        return true;
    } else { return false; }
}

```

Impact

Given the different usages of token transfers in BufferBinaryOptions.sol, BufferBinaryPool.sol, and BufferRouter.sol, there can be 2 types of impacts depending on the ERC20 contract being traded.



Impact type 1

The ERC20 token being traded is one that consistently returns a boolean result in the case of success and failure like for example 0x's ZRX Token contract. Where the return value is currently not verified to be true (i.e.: #1, #2, #3, #4, #5, #6) the transfer may fail (e.g.: no tokens transferred due to insufficient balance) but the error would not be detected by the Buffer contracts.

Impact type 2

The ERC20 token being traded is one that do not return a boolean value like for example the well know Tether USD Token contract. Successful transfers would cause a revert in the Buffer contracts where the return value is verified to be true (i.e.: #1, #2, #3, #4) due to the token not returning boolean results.

Same is true for approve calls.

Code Snippet

<https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferRouter.sol#L86-L90> <https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferRouter.sol#L331>

<https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferRouter.sol#L335-L338> <https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferRouter.sol#L361-L364>

<https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferBinaryOptions.sol#L141> <https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferBinaryOptions.sol#L477>

<https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferBinaryPool.sol#L162> <https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferBinaryPool.sol#L205>

<https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferBinaryPool.sol#L241> <https://github.com/sherlock-audit/2022-11-buffer/blob/main/contracts/contracts/core/BufferBinaryPool.sol#L323>

Tool used

Manual Review

Recommendation

To handle most of these inconsistent behaviors across multiple tokens, either use OpenZeppelin's SafeERC20 library, or use a more reusable implementation (i.e. library) of the following intentionally explicit, descriptive example code for an ERC20 `transferFrom()` call that takes into account all the different ways of signalling



success and failure, and apply to all ERC20 transfer(), transferFrom(), approve() calls in the Buffer contracts.

```
IERC20 token = whatever_token;

(bool success, bytes memory returndata) =
↳ address(token).call(abi.encodeWithSelector(IERC20.transferFrom.selector,
↳ sender, recipient, amount));

// if success == false, without any doubts there was an error and callee reverted
require(success, "Transfer failed!");

// if success == true, we need to check whether we got a return value or not
↳ (like in the case of USDT)
if (returndata.length > 0) {
    // we got a return value, it must be a boolean and it should be true
    require(abi.decode(returndata, (bool)), "Transfer failed!");
} else {
    // since we got no return value it can be one of two cases:
    // 1. the transferFrom does not return a boolean and it did succeed
    // 2. the token address is not a contract address therefore call() always
↳ return success = true as per EVM design
    // To discriminate between 1 and 2, we need to check if the address actually
↳ points to a contract
    require(address(token).code.length > 0, "Not a token address!");
}
```

Discussion

0x00052

Fixed in [PR#18](#)

Changes look good. Using safeERC20 for ERC20 transfers

