The bZx attacks explained

This article will examine in details what happened during the two transactions that exploited vulnerabilities to open under-collateralized positions in bZx, causing a loss of equity worth more than a million dollar in total.

It is a technical analysis. For information from the bZx developers, please see their official post-mortem and Twitter account.

First transaction

This is about the transaction 0xb5c8..., that was mined on Saturday 15th February 2020 at 01:38:57 UTC.

In that transaction an attacker used a flaw in bZx/Fulcrum to take an under-collateralized position, resulting in approximately 370k$ worth of profit for them, and approximately 620k$ of equity loss in the bZx lending pool.

Let's see why there was a vulnerability, and how it wasn't an oracle bug. We will source every claim with links to the original transaction.

High-level overview

To see what happened, we can use either Oko or EthDecoder. Both let us see the tree of all the calls made during that transaction. Beware: that transaction is fairly complex.

Here are the main calls that happened during it:

A. The attacker borrows 10000 ETH from DyDx.
B. The attacker sends 5500 ETH to Compound, and borrows 112 WBTC.
C. The attacker sends 1300 ETH to bZx to open a 5x short position for WBTC.
    - bZx internally converts 5637 ETH to 51 WBTC through a Kyber order routed to Uniswap (huge spread).
D. The attacker converts the 112 WBTC (borrowed at B.) to 6871 ETH on Uniswap (because the prices got skewed at C.)
E. The attacker sends back the 10000 ETH to DyDx.
F. The attacker ends up with 71 ETH, then do a little obfuscation dance (see below) and sends 65 ETH to the attacker originating EOA.

> **Note:**
>
> No, they didn't make 71 ETH of "pure arbitrage profit". They ended up the transaction with a Compound position having 5500 ETH of collateral and only 112 wBTC borrowed. This is around 350k$ worth of equity in Compound.

Why the transaction is suspicious

Before we dive into the details, a few things to note:

- The attacker-controlled address and contracts are new, and never interacted with bZx, Compound, or anything. So they obviously have zero balance everywhere.
- All the attacker-deployed contracts, and the address used to invoke the transactions are funded by 0x296e.... This address was funded by Tornado Cash (an Ethereum mixer), shortly before the attack. It seems like the attacker spent efforts on staying anonymous, so we cannot trace the funds further (or we would need some probabilistic / taint analysis).
- At the end of the transaction, the attacker contract creates another contract, sends 65 ETH to it only to immediately self-destruct it, so the money ends up to the EOA that the transaction origi-

nates from. This is a very contrived way of sending ETH to `tx.origin`. I'm not sure what the purpose of this is, but my best guess is it could be obfuscation, and/or a way to try to avoid frontrunning bots from frontrunning the attack transaction by making it harder to replay.

Most importantly, by quickly looking at it, you notice the origin account of the transaction starts with nothing, then borrows and moves a pile of cash, causes two huge Uniswap orders (in both directions) in the course of the same transaction, and ends up with 65 ETH. That definitely looks fishy.

Walkthrough of the transaction

We will now go over each of these actions to try clarifying what happened.

A. The DyDx instant borrow

How did they get enough liquidity to pull off their attack?

The `operate()` function of Dydx Solo contract is called by a second attacker contract 0x0d...: call here.

This single `operate()` call contains two successive actions:

- First, a `ActionType.Withdraw` of 10000 ETH, to the first attacker contract.
- Second, a `ActionType.Call` to the first attacker contract.

What happens here is that DyDx only checks if you have collateral when all the operations you wanted to do are finished. But if you do everything atomically you don't need a collateral!

Note that the whole exploit will happen inside of the `Call` action that's initiated from DyDx. The attacker is going to withdraw the funds they borrowed from DyDx, pull off the exploit, then put the funds back. At the end their account doesn't have have any debt, so there is no under-collateralization and DyDx doesn't revert the call.

This is the easy part that gets the 10k ETH needed to do the exploit.

B. The Compound borrow

Like the DyDx borrow, this is only a step used to convert the borrowed ETH to enough WBTC so they can pull off the attack.

This happens in two calls:

1. mint() to send the collateral to Compound.
2. borrow() to use the account to borrow the 112 WBTC.

For the curious, the actual WBTC transfer inside of `borrow()` happens here.

C. bZx position opening

This call opens a Fulcrum position, shorting ETH against WBTC with a x5 leverage. This position is on 1300 ETH (huge).

Internally, bZx uses Kyber to determine the mid-price for the tokens involved in the position (it averages the price from both directions: call 1, call 2). The prices it gets are all representing the correct market prices. As this tweet also points out, Uniswap is not used as a price feed.

The slippage risk

However, when you open a position like this, it needs to convert these 1300 ETH multiplied by the leverage, to WBTC, which becomes your collateral.

The conversion is sent through Kyber. Kyber queries each reserve, but no reserve seems to have enough liquidity to fulfill that order alone, except for Uniswap. So the order is routed to Uniswap.

For such a huge volume, going through Uniswap skews the price a lot: bZx sends 5637 ETH, receives 51 WBTC. That's 110 BTC/ETH where the normal price is closer to 36 BTC/ETH!

This is normally fine, as the position is overcollateralized by at least 20%, so you would need a slippage bigger than that to cause a problem. But here it was the case, so the slippage caused a loss that ate into the lending pool.

The bZx bug

However, this seems to be an intentional design: the code also makes sure that the caller account is fully collateralized after everything is finished. If it is not, the call should revert.

So if there is a huge loss caused by slippage, the caller would not have enough collateral and the call would revert. This makes sense, and other contracts like DyDx have a similar design (see above).

It is supposed to be enforced by this code that the position is still collateralized enough:

```
require ((
         loanDataBytes.length == 0 && // Kyber only
         sentAmounts[6] == sentAmounts[1]) || // newLoanAmount
   !OracleInterface(oracle).shouldLiquidate(
         loanOrder,
         loanPosition
   ),
   "unhealthy position"
);
```

But because of a logic bug, the first part of that condition is true and the `shouldLiquidate()` is never called (you can check in the trace). So when the call should have reverted, it didn't.

Lev Livnev has a more detailed writeup of the call stack that leads to that bug.

Effect on bZx pool

After that transaction, bZx has:

- +51 WBTC
- +1300 -5637 = -4337 ETH

So this transaction caused a loss of equity of around 620000$ in bZx.

This is an outside view. From the perspective of bZx, the attacker converted their 1300 ETH into 51 WBTC of collateral (bug), and also left 360 ETH as escrowed interest. You can refer to their official post-mortem to learn more about that and how it should affect the people who put loans in the pool.

D. The Uniswap arbitrage

At the previous step C., the attacker exploited a bug in bZx that caused it to trade a huge amount on Uniswap, at a 3x inflated price.

Because of the way Uniswap works, this caused a big price swing in the price of the WBTC pool. This distorted price can then be arbitraged back to the normal price, for a profit.

This is what they do: they arbitrage against Uniswap by selling the 112 WBTC they borrowed from Compound (step B.) on Uniswap. Because the Uniswap supply is all distorted, they are able to sell these 112 WBTC for 6871 ETH.

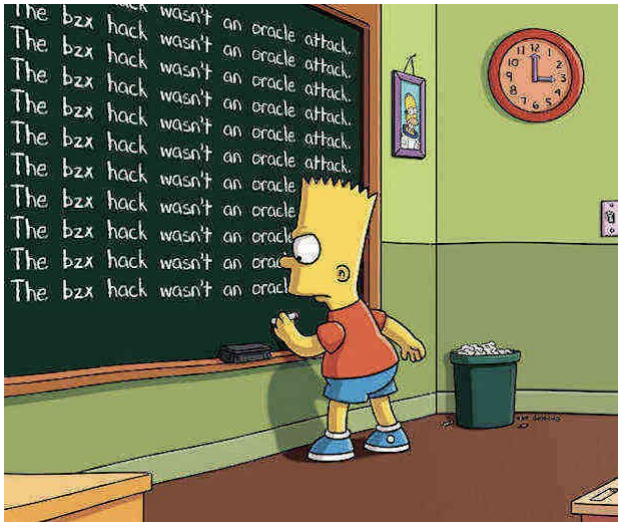
This is a price of 61 BTC/ETH: they are selling their 112 WBTC at twice the market price.

E & F. Settling everything

Now they have enough ETH to refund their borrowed DyDx Ethers, and they have 65 ETH of leftovers that they send back to the account that sent the transaction.

Again, note that they didn't make 65 ETH of profit. This is only the breadcrumbs, as the biggest chunk of the profit is in the Compound position that they opened.

Summary



As we have seen, DyDx and Compound are only here to get enough leverage. And it's the position that the attacker took on bZx that caused a huge Uniswap skewing that they then exploited.

Also note that the attacker only opened a position, and that's it. There were not fiddling with the Uniswap prices first, or anything like that.

It's the mere fact of opening their huge position that caused a leak of funds from bZx to Uniswap, that they exploited.

The aftermath

Attacker: repaying the Compound position

After that transaction, the first attacker smart contract ends up with a Compound account with more than 300k$ of equity, made of:

- 143000$ worth of collateralized ETH: +5500 ETH
- 109000$ worth of WBTC debt: -112 WBTC

However, they cannot withdraw their ETH directly, or their account would end up under-collateralized. So they need to buy WBTC on the market, pay off their debt and withdraw their ETH.

Guess what? That's exactly what they have been doing. Precisely two hours after their exploit transaction they started to buy WBTC and repay their loan.

This is an example transaction where they repay their loan. This feature is part or their smart contract.

It took them a bit less than two days to fully repay their positions. They sent all the fund to their EOA, which now has 1193 ETH.

A feast for the arbitrage bots

You probably noticed that the equity loss from bZx and the money the attacker made don't add up.

It happens that the attacker possibly didn't maximize their profit, and they left Uniswap completely unbalanced after their attack. A lot of bots then rushed to make a profit out of it.

Two examples:

- This arbitrage transaction made a profit of 315 ETH (see the balance change of the contract 0xb958...).

- This other transaction made a profit of 9 ETH.
- There is a lot more: you can see the list here (page 23).

Bonus findings

Even the attacker makes mistakes

When withdrawing money, they always have their contract create a temporary contract that self-destruct itself immediately (see above). And they specify the amount of Ether they want to withdraw.

See here for an example. However this example was their second try!

First they failed, because instead of passing 10.1×10^{18} Wei, they passed $10.1 \times 10^{18} \times 10^{18}$ (they multiplied the amount in Wei twice by 10^{18}). So obviously this was a ridiculously high amount and it didn't work.

This is a very small mistake without consequences, but it was interesting to see.

Possibly related contract

The self-destructing contracts that the attacker was using have an unknown function selector, for the function that triggers the self-destruct: `0xf2adf1cb`.

We don't know what's the original name. However we can find this contract that is from 2+ years ago, but seems to do something very similar, and has that same function selector! It's the only one on mainnet.

There is no way to tell if this was related, or if it's just a coincidence. But it's worth considering.

Second transaction

Let's now look at the second transaction 0x7628..., which happened on Tuesday 18th February 2020 at 03:13:58 UTC.

It caused the same effect as the first one, namely opening an under-collateralized position on bZx. However it uses a completely different method, and is more straightforward to understand.

High-level overview

Again, I recommend you use either Oko or EthDecoder.

Here are the main calls that happened during it:

A. The attacker borrows 7500 ETH from bZx (flash borrow)
B. The attacker repeatedly calls Kyber to convert 900 ETH to 155,994 sUSD (distorting the Kyber sUSD prices)
C. The attacker uses the Synthetix depot contract to convert 3518 ETH to 943,837 sUSD
D. The attacker borrows 6796 ETH on bZx, sending only 1,099,841 sUSD (oracle attack)
E. The attacker transfers back 7500 ETH to bZx to repay their flash loan

At the end the attacker ends up with 2378 ETH in their attack contract. They transfer it to their EOA shortly after.

Walkthrough of the transaction

Before you start, I recommend you take a look at the decompiled contract code, you can see the sequence of calls to be made hardcoded there.

Let's now go through each step of the transaction:

A. The bZx flash borrow

This step is comparable with the first step of the first exploit. Only it uses bZx instead of DyDx.

Again, the goal is to borrow enough money to be able to pull off the exploit, and again, the rest of the attacker activity happen inside of a callback to the attacker contract, initiated by bZx.

B. Distorting the Kyber prices

Kyber uses "reserves", which provides liquidity. For the ETH-sUSD pair there are two reserves:

- Uniswap.
- A Synthetix one, that implements a LiquidityConversionRates that automatically adjusts the price (conceptually similar to Uniswap).

A trade will necessarily either hit one of them (depending on which one gives the best price).

The attacker contract buys most of the sUSD liquidity available on both reserves. For that, they do 19 successive buys:

- 540 ETH to 92k sUSD, hitting the Uniswap Kyber reserve.
- 20 ETH to 5.2 sUSD, hitting the sUSD Kyber reserve.
- 20 ETH to 4.9 sUSD, hitting the sUSD Kyber reserve.
- 20 ETH to 4.7 sUSD, hitting the sUSD Kyber reserve.
- ...

You can see each trade getting a worse price. That's the attacker skewing the prices by eating all available liquidity.

We get from 270 ETH/sUSD (normal rate) to a price of 111 ETH/sUSD in Kyber.

C. Buying a lot of sUSD at a normal rate

This is done using the Synthetix Depot contract which has a lot of liquidity that you can access.

The attacker calls the exchangeEtherForSynths() function to exchange 6000 ETH to 943,837 sUSD.

The rate is 157 ETH/sUSD. It's a bad rate, but significantly better than the distorted rate that Kyber now returns (see above).

D. Borrowing ETH for sUSD on bZx (oracle attack)

This is where the oracle attack is executed.

I strongly recommend that you read this article from Sam Sun about oracle attacks, which explain how it works.

The idea is that bZx queries Kyber for the current ETH/sUSD rate, but now that the attacker distorted the market it will get an erroneous rate! This allows the attacker to borrow much more ETH than they could normally with this amount of sUSD, because bZx is fed a wrong price oracle.

To do it, they simply call the iETH contract's borrowTokenFromDeposit() function. They send 1,099,841 sUSD (they bought the bought from the Synthetix Depot, and some more while distorting the Kyber prices), and are able to borrow 6796 ETH.

Effect on bZx pool

We can compute that bZx sent them 1.7mm$ and received only 1.1mm$ worth of sUSD. That's an equity loss of around 600k$.

Wait, why did it work?

If you read the article about oracle attacks, you probably wonder how come the exploit worked. The article describes a very similar attack, and it was fixed.

Let's quote the last fix that was implemented after that disclosure:

> 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.

The check was implemented here:

```
require(
   spreadPercentage <= maxSpread,
   "bad price"
);
```

This means that bZx will check the price in both directions and look at the difference. However, here both reserves are Uniswap-like and both got their price manipulated.

So, yes, bZx was supposed to check for a small enough spread. But both these reserves do have a constant, small spread (that depends on their fees). So the checks did pass.

E. Settling the debt

Now that the attacker realized a profit in ETH, they can pay back the 7500 ETH, so the transaction can terminate correctly because the flash loan has been paid back.

Conclusion

This attack was much simpler than the first one, and this was indeed an oracle attack.

A few things to note:

Comparing both attacks

In the first attack, the attacker makes bZx do a bad trade by opening a leveraged position, and then profit off arbitraging it back. Here the attacker first skews the price, and then borrows ETH to trigger their oracle attack on bZx (no internal Kyber trade happens).

This is pretty different. Also note that in the second attack, the attacker could have pulled back the price that they skewed first for more profit. But if you look at the numbers you notice they spent 900 ETH to skew the price, compared to the 3518 ETH worth of sUSD that they bought from the Synthetix Depot and leveraged on bZx. Because the money they spent on skewing the price was only 10% of the amount they magically multiplied later, they didn't need to care.

Is it the same attacker?

A few things to note:

- the EOA of the second attacker was funded 2+ years ago from ShapeShift (whereas the first was funded shortly before from Tornado Cash).
- The second attacker didn't use newly-created contracts self-destruction to transfer money.
- The second attacker could well have used DyDx to get the liquidity they needed, like in the first attack. Instead they used a similar feature from bZx.
- The attack contracts used during both transactions are pretty different: The first has lots of functions taking parameters, has an error message... The second has two simple functions.

So this would point the attacker being different persons. But we can never be sure.

Previous: Analyzing suspicious smart contract vacuuming