bZx Hack Full Disclosure (With Detailed Profit Analysis)

PeckShield

On 02/15, we have provided a transaction-level recap on the bZx hack that recently captures various headlines in DeFi-related tweets and media. There are quite a few misunderstandings circulating around about the nature of this particular hack. We emphasize that this is not an oracle attack. Instead, it is a clever arbitrage execution, which did exploit a bug in bZx smart contract implementation to allow for the leakage of supposedly-locked bZx funds to Uniswap and further absorb the leaked funds into a Compound position. In this blog, we'd like to provide a full disclosure of the hack with an in-depth profit analysis, just as promised in our previous blog.

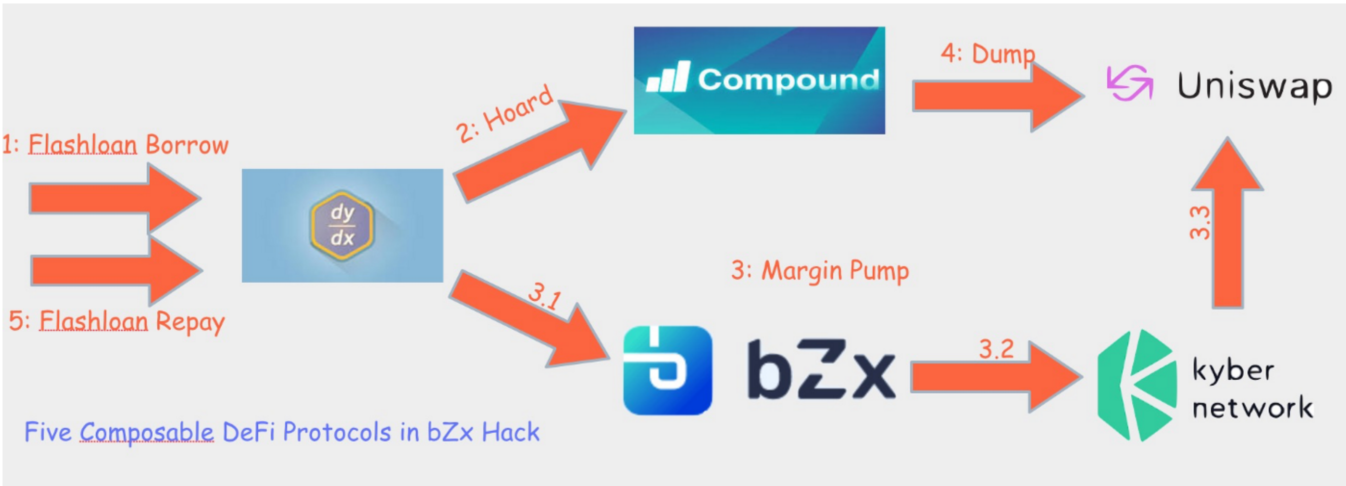

Figure: Five Arbitrage Steps in bZx Hack

Five Exploitation Steps For Arbitrage

The culprit transaction is 0xb5c8bd9430b6cc87a0e2fe110ece6bf527fa4f170a4bc8cd032f768fc5219838, which was mined at 2020–02–15 01:38:57 +UTC at the block height #9484688. As shown in the above figure, this attack can be separated into five distinct steps: **Flashloan Borrow**, **Hoard**, **Margin Pump**, **Dump**, **Flashloan Repay**. In the following, we examine each specific step.

1: Flashloan Borrow. This step basically takes advantage of the dYdX flashloan feature to borrow 10,000 ETH. This part is already known and we will not go into the details.

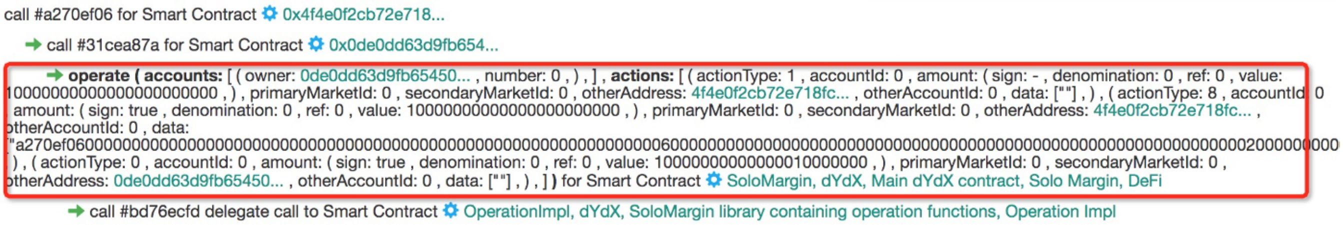

Figure 1: Flashloan Borrowing From dYdX

After this step, we notice the attacker has the following asset breakdown. There is no gain yet.

Protocol	Amount	Asset	Type
dYdX	-10,000	ETH/WETH	Debt

Accounts	Amount	Asset	Type
-	+10,000	ETH/WETH	Balance

2: Hoard. With the borrowed flashloan, the attacker deposits 5500 ETH into Compound as collateral to borrow 112 WBTC. This is a normal Compound operation and this hoarded WBTC is to be dumped in Step 4.

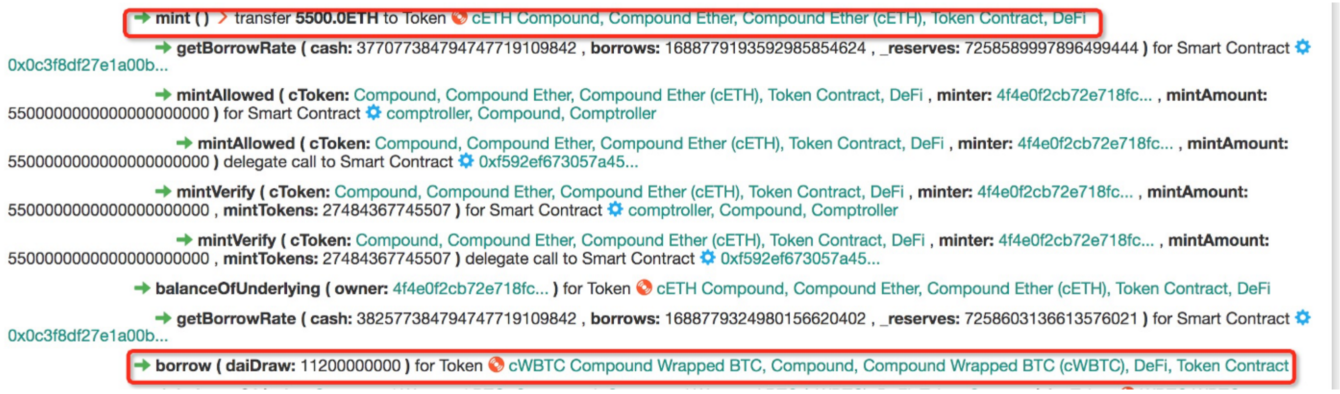
→ mint ( ) > transfer 5500.0ETH to Token cETH Compound, Compound Ether, Compound Ether (cETH), Token Contract, DeFi
→ getBorrowRate (cash: 377077384794747719109842 , borrows: 1688779193592985854624 , _reserves: 7258589997896499444) for Smart Contract 0x0c3f8df27e1a00b...
→ mintAllowed (cToken: Compound, Compound Ether, Compound Ether (cETH), Token Contract, DeFi , minter: 4f4e0f2cb72e718fc... , mintAmount: 5500000000000000000000) for Smart Contract comptroller, Compound, Comptroller
→ mintAllowed (cToken: Compound, Compound Ether, Compound Ether (cETH), Token Contract, DeFi , minter: 4f4e0f2cb72e718fc... , mintAmount: 5500000000000000000000) delegate call to Smart Contract 0xf592ef673057a45...
→ mintVerify (cToken: Compound, Compound Ether, Compound Ether (cETH), Token Contract, DeFi , minter: 4f4e0f2cb72e718fc... , mintAmount: 5500000000000000000000 , mintTokens: 27484367745507) for Smart Contract comptroller, Compound, Comptroller
→ mintVerify (cToken: Compound, Compound Ether, Compound Ether (cETH), Token Contract, DeFi , minter: 4f4e0f2cb72e718fc... , mintAmount: 5500000000000000000000 , mintTokens: 27484367745507) delegate call to Smart Contract 0xf592ef673057a45...
→ balanceOfUnderlying (owner: 4f4e0f2cb72e718fc...) for Token cETH Compound, Compound Ether, Compound Ether (cETH), Token Contract, DeFi
→ getBorrowRate (cash: 382577384794747719109842 , borrows: 1688779324980156620402 , _reserves: 7258603136613576021) for Smart Contract 0x0c3f8df27e1a00b...
→ borrow (daiDraw: 11200000000) for Token cWBTC Compound Wrapped BTC, Compound, Compound Wrapped BTC (cWBTC), DeFi, Token Contract

Figure 2: WBTC Hoarding From Compound

After this step, we notice the following changes regarding the attacker-controlled assets. Apparently, there is still no gain yet.

Protocol	Amount	Asset	Type
dYdX	-10,000	ETH/WETH	Debt
Compound	+5,500	ETH/WETH	Collateral
Compound	-112	WBTC	Debt

Accounts	Amount	Asset	Type
-	+4,500	ETH/WETH	Balance
-	+112	WBTC	Balance

3: Margin Pump. After hoarding, this step takes advantage of the bZx margin trade feature to short ETH in favor of WBTC (i.e., sETHwBTCx5). In particular, the attacker deposits 1300 ETH and calls bZx margin trading function, i.e., `mintWithEther` (that cascadingly invokes `marginTradeFromDeposit`). The margin trading function leverages KyberSwap to swap the borrowed 5637.623762 ETH for 51.345576 WBTC in return. Notice that it is 5x borrow to short ETH. The swap essentially drives up the conversion rate of 1 WBTC to around 109.8 WETH, roughly triple the normal conversion rate (~38.5 WETH/WBTC).

Specifically, to complete this trade, bZx forwards the order to KyberSwap, which then essentially consults its reserves and finds the best rate. It turns out to be the KyberUniswap reserve. This step essentially drives the WBTC price up in Uniswap three times higher.

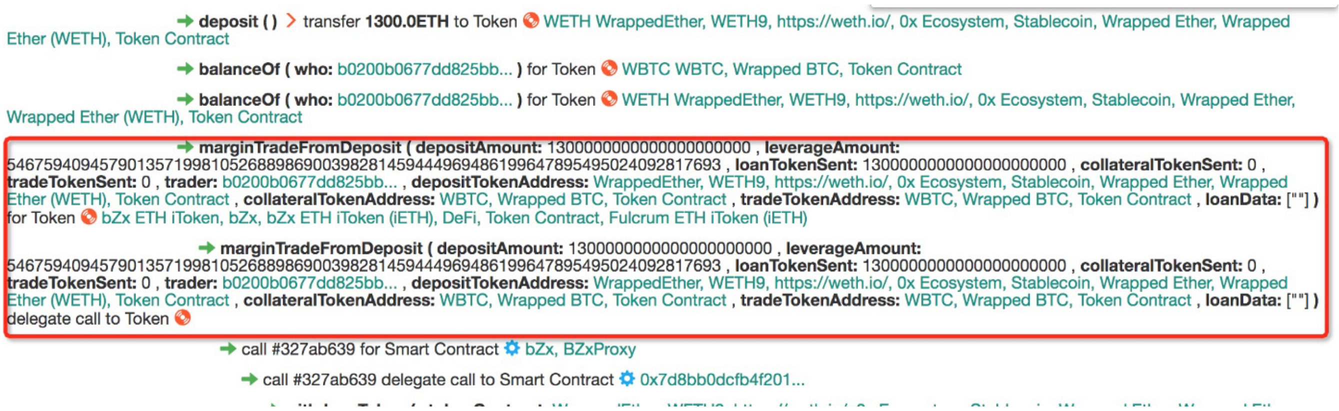

Figure 3: Margin Pumping With bZx (and Kyber + Uniswap)

It should be noted that this step should be thwarted by the built-in sanity check, which verifies the position will not go default after the swap. However, this check did not kick in when the attack occurs and we examine the details later in the smart contract bug section.

After this step, we notice the following changes regarding the attacker-controlled assets. Still, there is no gain yet after this step.

Protocol	Amount	Asset	Type
dYdX	-10,000	ETH/WETH	Debt
Compound	+5,500	ETH/WETH	Collateral
Compound	-112	WBTC	Debt
bZx	+1,300	ETH/WETH	Collateral
bZx	-5,637	ETH/WETH	Debt
bZx	+51	WBTC	Collateral

Accounts	Amount	Asset	Type
-	+3,200	ETH/WETH	Balance
-	+112	WBTC	Balance

4: Dump. With the spiked WBTC price in Uniswap, the attacker sells the Compound-borrowed 112 WBTC back for WETH in Uniswap.

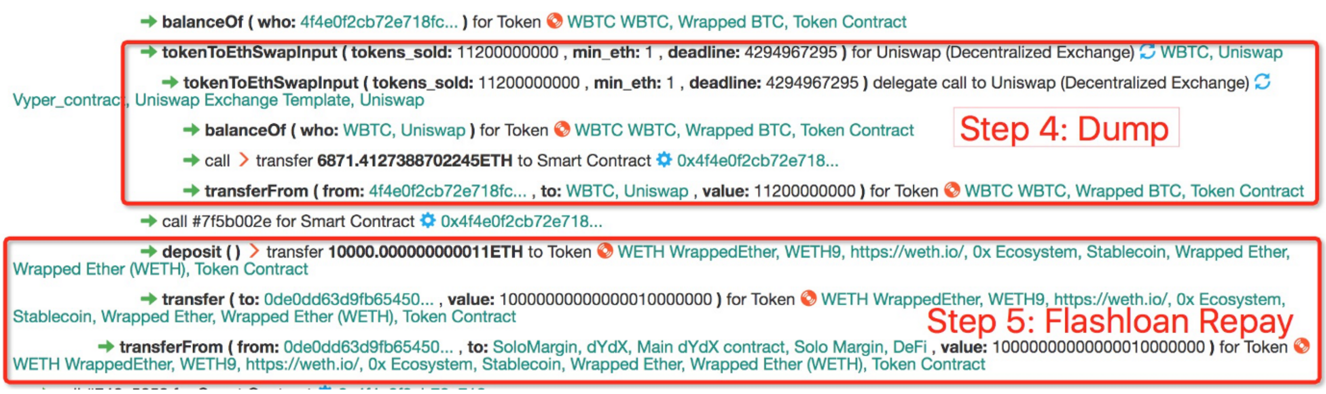

Figure 4: WBTC Dumping With Uniswap

This dump step leads to the net of 6871.4127388702245 ETH in return with the overall conversation rate of 1WBTC=61.4 WETH. After this step, the attacker observes substantial profits with the following asset breakdown.

Protocol	Amount	Asset	Type
dYdX	-10,000	ETH/WETH	Debt
Compound	+5,500	ETH/WETH	Collateral
Compound	-112	WBTC	Debt
bZx	+1,300	ETH/WETH	Collateral
bZx	-5,637	ETH/WETH	Debt
bZx	+51	WBTC	Collateral

Accounts	Amount	Asset	Type
-	+3,200	ETH/WETH	Balance
-	+6,871	ETH/WETH	Balance

5: Flashloan Repay. With the netted 6871.4127388702245 ETH from the dumped 112 WBTC, the attacker repays the flashloan 10000.000000000011ETH back to dYdX, thus completing the flashloan.

We re-calculate the following asset breakdown after this step. It turns out that the attacker gains the 71ETH arbitrage profit, plus the two positions, one in Compound (+5,500WETH/-112WBTC) and another in bZx (-4,337WETH/+51WBTC). The Compound position is very profitable while the bZx position is in default state. Apparently, right after the exploit, the attacker starts to arrange the payment of Compound debt (112BTC) to claim the collateral (5,500WETH). For the bZx position, since it is already in default, the attacker shows no futher interest.

Protocol	Amount	Asset	Type
Compound	+5,500	ETH/WETH	Collateral
Compound	-112	WBTC	Debt
bZx	+1,300	ETH/WETH	Collateral
bZx	-5,637	ETH/WETH	Debt
bZx	+51	WBTC	Collateral

Accounts	Amount	Asset	Type
-	+71	ETH/WETH	Balance

Considering the average market price of 1WBTC=38.5WETH (or 1WETH=0.025BTC), the attacker can get 112 WBTC with ~4,300 ETH. As a result, the attacker gains 71 WETH + 5,500 WETH — 4,300 ETH = 1,271 ETH, roughly $355,880 (assuming the ETH price of $280).

bZx Smart Contract Bug

The magic under the hood is the fact how the Uniswap WBTC/ETH was manipulated up to 61.4 for profit. As mentioned in Step 3, the WBTC/ETH price was even pumped up to 109.8 when the normal market price was at only around 38. In other words, there is an intentional huge price slippage triggered for exploitation. However, such a huge price slippage should cause the bZx position not fully collateralized. But why the under-collateralized position will be allowed in the first place, which naturally leads to the discovery of a hidden bug in the bZx smart contract implementation.

In particular, the **margin pump** started from the function, `marginTradeFromDeposit()`.

```
823         loanOrderHash = _borrowTokenAndUse(
824             leverageAmount,
825             [
826                 trader,
827                 collateralTokenAddress,     // collateralTokenAddress
828                 tradeTokenAddress,          // tradeTokenAddress
829                 trader                      // receiver
830             ],
831             [
832                 0,                      // interestRate (found later)
833                 amount,                 // amount of deposit
834                 0,                      // interestInitialAmount (interest is calculated based on fixed-term loan)
835                 loanTokenSent,
836                 collateralTokenSent,
837                 tradeTokenSent,
838                 0
839             ],
840             true,                       // amountIsADeposit
841             loanDataBytes
842         );
```

Figure 5: marginTradeFromDeposit()

As shown in Figure 5, `marginTradeFromDeposit()` invokes `_borrowTokenAndUse()` with the fourth parameter set as `true` in line 840.

```
1327     function _borrowTokenAndUse(
1328         uint256 leverageAmount,
1329         address[4] memory sentAddresses,
1330         uint256[7] memory sentAmounts,
1331         bool amountIsADeposit,
1332         bytes memory loanDataBytes)
1333         internal
1334         returns (bytes32 loanOrderHash)
1335     {
1336         require(sentAmounts[1] != 0, "21"); // amount
1337 
1338         loanOrderHash = loanOrderHashes[leverageAmount];
1339         require(loanOrderHash != 0, "22");
1340 
1341         _settleInterest();
1342 
1343         LoanData memory loanOrder = loanOrderData[loanOrderHash];
1344         bool useFixedInterestModel = loanOrder.maxDurationUnixTimestampSec == 0;
1345         //sentAmounts[7] = loanOrder.marginPremiumAmount;
1346 
1347         if (amountIsADeposit) {
1348             (sentAmounts[1], sentAmounts[0]) = _getBorrowAmountAndRate( // borrowAmount, interestRate
1349                 loanOrderHash,
1350                 sentAmounts[1], // amount
```

Figure 6: _borrowTokenAndUse()

Inside `_borrowTokenAndUse()`, `_getBorrowAmountAndRate()` is invoked in line 1348 when `amountIsADeposit` is `true`. The returned `borrowAmount` would be stored in `sentAmounts[1]`.

```
1354            // update for borrowAmount
1355            sentAmounts[6] = sentAmounts[1]; // borrowAmount
1356        } else {
1357            // amount is borrow amount
1358            sentAmounts[0] = _nextBorrowInterestRate2( // interestRate
1359                sentAmounts[1], // amount
1360                _totalAssetSupply(0),
1361                useFixedInterestModel
1362            );
1363        }
1364
1365        if (sentAddresses[2] == address(0)) { // tradeTokenAddress
1366            // tradeTokenSent is ignored if trade token isn't specified
1367            sentAmounts[5] = 0;
1368        }
1369
1370        uint256 borrowAmount = _borrowTokenAndUseFinal(
1371            loanOrderHash,
1372            sentAddresses,
1373            sentAmounts,
1374            loanDataBytes
1375        );
```

Figure 7: _borrowTokenAndUse()

Also in _borrowTokenAndUse(), `sentAmounts[6]` is filled with the value of `sentAmounts[1]` in line 1355 in the case of `amountIsADeposit == true` (we'll see this later). Later on, _borrowTokenAndUseFinal() is called in line 1370.

```
1414        sentAmounts[1] = IBZx(bZxContract).takeOrderFromiToken.value(msgValue)( // borrowAmount
1415            loanOrderHash,
1416            sentAddresses,
1417            sentAmounts,
1418            loanDataBytes
1419        );
```

Figure 8: _borrowTokenAndUseFinal()

In line 1414, `_borrowTokenAndUseFinal()` calls `takeOrderFromiToken()` through the IBZx interface such that the transaction flows into the `bZxContract`.

```
145         require ((
146                 loanDataBytes.length == 0 && // Kyber only
147                 sentAmounts[6] == sentAmounts[1]) || // newLoanAmount
148             !OracleInterface(oracle).shouldLiquidate(
149                 loanOrder,
150                 loanPosition
151             ),
152             "unhealthy position"
153         );
```

Figure 9: bZxContract::takeOrderFromiToken()

Here comes the interesting part. In line 145–153, there's a `require()` call to check whether the position is **healthy** or **unhealthy**. Unfortunately, in the case `loadDataBytes.length == 0 && sentAmounts[6] == sentAmounts[1]`, the sanity check `bZxOracle::shoudLiquidate()` would be skipped. That's exactly the condition that the exploit triggered to avoid the sanity check.

```
500     function shouldLiquidate(
501         BZxObjects.LoanOrder memory loanOrder,
502         BZxObjects.LoanPosition memory loanPosition)
503         public
504         view
505         returns (bool)
506     {
507         return (
508             getCurrentMarginAmount(
509                 loanOrder.loanTokenAddress,
510                 loanPosition.positionTokenAddressFilled,
511                 loanPosition.collateralTokenAddressFilled,
512                 loanPosition.loanTokenAmountFilled,
513                 loanPosition.positionTokenAmountFilled,
514                 loanPosition.collateralTokenAmountFilled) <= loanOrder.maintenanceMarginAmount
515             );
516     }
```

Figure 10: bZxOracle::shouldLiquidate()

If we take a look into `bZxOracle::shouldLiquidate()`, the check `getCurrentMarginAmount() <= loanOrder.maintenanceMarginAmount` in line 514 would do the job by catching the **margin pump** step and thus preventing this attack.

Here we'd also like to thank Bloxy for the wonderful tools we used to generate some of the diagrams in this article.

About us

PeckShield Inc. is an industry leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem. For any business or media inquiries (including the need for smart contract auditing), please contact us at telegram, twitter, or email.