Cross-Bridge Attack: Multichain Attack Analysis (January 2022)

Adrian Koegl, Alpha Kaba, Sally Wang

December 2022

Columbia University

E6998 Engineering Blockchain and Web3 Apps

Professor Yunfeng Yang

A. Introduction

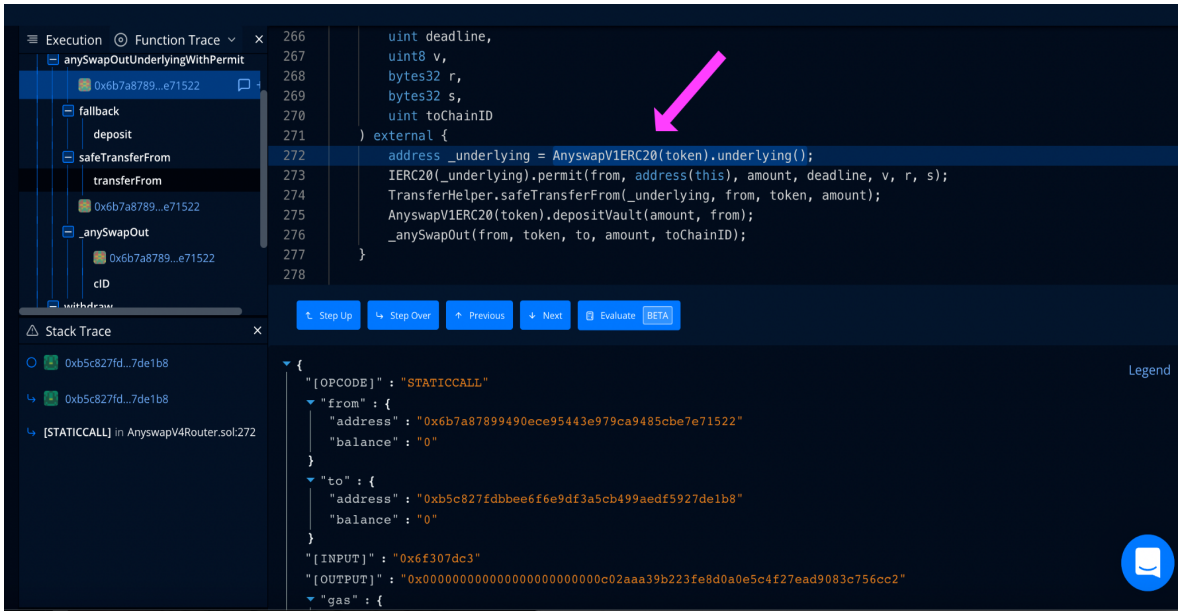
Multichain (formerly known as Anyswap) underwent a cross-attack bridge attack on its chain January 19th, 2022. The attacker sweeps about 1.4 $ million of dollars from the chain by exploiting a logical bug within the core router contract. Leading him to deposit funds into a malicious contract. This report illustrates the sophisticated attack using an online simulation, tenderly, and then proposes solutions to fix the buggy contract.

B. Understanding the attack

Prior to simulating the attack, let's first understand the logical bug that leads to the exploit in the first place. The bug that led to this attack was in the *anySwapOutUnderlyingWithPermit()* function within *anyswapRouterv4.sol*, the core router contract. This function allows a user to freely swap a token between any two chains. Additionally, it reduces fees by not sending the transaction to the blockchain by the unsuccessful attempt of using the ERC20 permit() function.

C. Attack replication

We will use Tenderly, a robust online debugging tool to track the critical bug in the *anyswapRouterv4.sol* smart contract. The simulation can be found here.



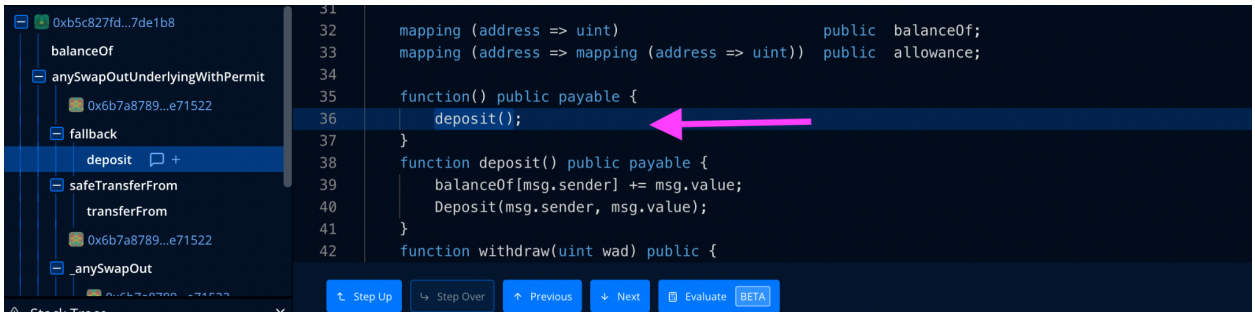
The attacker called *anySwapOutUnderlyingWithPermit()* with the below malicious *calldata*:


```
{
  "from": "0x3ee505ba316879d246a8fd2b3d7ee63b51b44fab",
  "token": "0xb5c827fdbbee6f6e9df3a5cb499aedf5927de1b8",
  "to": "0xb5c827fdbbee6f6e9df3a5cb499aedf5927de1b8",
  "amount": "308636644758370382903",
  "deadline": "10000000000000000000",
  "v": 0,
  "r":
"0x3000000000000000000000000000000000000000000000000000000000000000",
  "s":
"0x3000000000000000000000000000000000000000000000000000000000000000",
  "toChainID": "56"
}
```


The attack pursue as follows:

1. On line 272, *address_underlying = AnyswapV1ERC20(token).underlying()* unwraps the attacker's token from a *anyToken* to *Token*. Here, *Token* is a "DAI" and *anyToken* is *anyDAI*. However, the *Token* here is the masqued malicious contract of the attacker.

2. On line 273, *IERC20(_underlying).permit(from, address(this), amount, deadline, v, r, s)* attempts to call, the *underlying token* used by the attacker, wETH. However, this contract's token does not have a *permit()* function, thus, returns "fallback" function. The Multichain router's developer assumed that wETH's *permit()* function would verify the signature. Unfortunately, the danger of calling this unknown function returns a "fallback", which does nothing material in this case, then allowing the exploiter to pursue execute the execution of step 3.



3. Line 274, `TransferHelper.safeTransferFrom(_underlying, from, token, amount);`, assumes if we have reached this point, the signature provided by the attacker was verified, although in our case it was not as explained in part 2, hence should call the `safeTransferFrom()` function. This function allowed the attacker to swipe off `308636644758370382903 wei` to their malicious contract.

The last line of the function simply deals with the readjusting of the liquidity of the pool. Thus, the attacker successfully completes their exploit on *anyswapRouterv4.sol*.

Theoretically, it is worth mentioning that, the third step should not have approved the transfer as the attacker's signature never proved the victim's approval to access the latter's funds.

D. Debugging the core buggy contract

1. Per an analysis performed by ZenGo, a web3 company accessing the security of smart contracts, the *anySwapOutUnderlyingWithPermit()* was called for the very first time on January 19th, 2022. Thus, the first call ever made was by the exploiter of this contract. This means the function is indeed nonessential to the core router functionality *anyswapRouterv4.sol,* implemented 2 years prior. Hence, an easy fix would be to simply eliminate the function from the core contract.
2. In step 2 of the attack replica, we demonstrated that the wrapped token's contract, wETH, does not have a *permit()* function. Thus, ensuring that `IERC20(_underlying).permit()` first contains a permit function prior to calling the function would have eliminated the "fallback" to be called in the first place. And this could have been done by checking the encoded-abi of the subsidiary contract, wETH in our case.

E. Conclusion

The main takeaway from the analysis of this cross-bridge attack in the web3 ecosystem is that building a bi-contract abi-encoded checker would massively help to mitigate whether functions called from one contract surely exists in the one contract prior to deployment. However, the technical challenge of building such a bi-contract abi-encoded checker resides in the inconsistency in practice among developers. Some contracts directly call from API such as Openzeppelin, whereas others include a copy of the libraries used within the deployed contract itself. Hence, this

checker would be, ideally, a robust tool that consists of a manual copy-paste of encoded ABIs of the contracts involved, and then the checker finally synthetic its findings based on the abi provide.