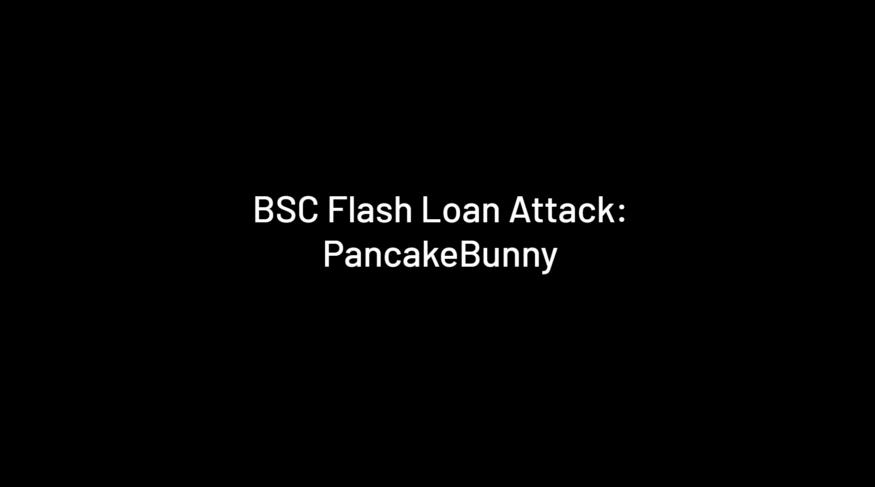
**BSC Flash Loan Attack: PancakeBunny**

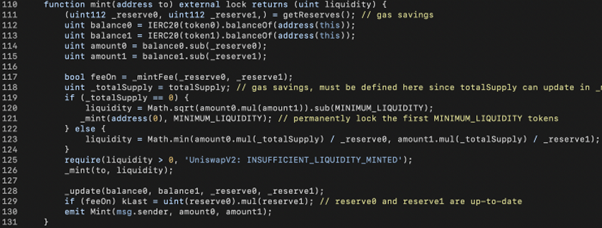


In May 2021, we witnessed multiple hacks targeting BSC DeFi products. In particular, a loophole related to reward minting in the yield aggregator, PancakeBunny, was exploited to mint ~7M BUNNY tokens from nothing, leading to a $45M financial loss. After the hack, three forked projects — AutoShark, Merlin Labs, and PancakeHunny — were exploited using similar techniques. Below we dig into loophole and give a step-by-step account of the exploit by**reproducing the attack** against PancakeBunny.

**Hidden Attack Surface: balanceOf()**

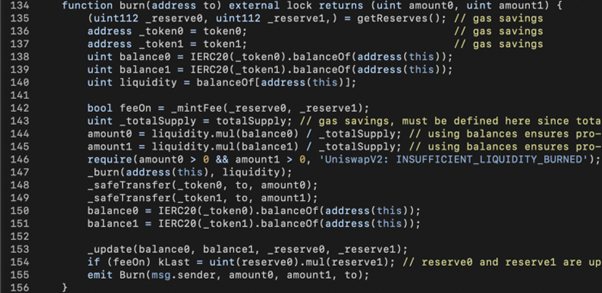
Many people believe that composability is crucial to the success of DeFi. Token contracts (e.g., ERC20s) play an essential role on the bottom layer of DeFi legos. However, developers may overlook some uncontrollable and unpredictable conditions when integrating ERC20s into their DeFi projects. For example, you can’t predict when and how many tokens you will receive when you retrieve the current token balance. This uncertainty creates a hidden attack surface.

In many cases, smart contracts reference the balances of ERC20s in their business logic. For example, when a user deposits some XYZ tokens into the smart contract, XYZ.balanceOf() is invoked to check how much money is received. If you’re familiar with the Uniswap codebase, you probably know that the UniswapV2Pair contract has many balanceOf() calls.



In the code snippet, UniswapV2Pair.mint() uses the current balances (balance0, balance1) and the book-keeping data (amount0, amount1) to derive the amounts deposited by the user(amount0, amount1). However, if a bad actor transfers some assets (token1 or token2) right before the mint() call, the victim would provide more liquidity than expected, i.e., more LP tokens are minted. If the rewards are calculated based on the amount of LP tokens, the bad actor can profit when the rewards exceed the expenses.

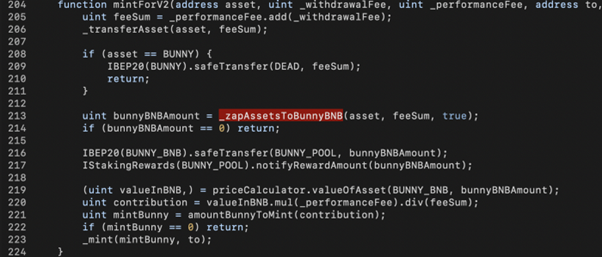
The UniswapV2Pair.burn() has a similar risk. The mint() function caller might jeopardize himself without a thorough understanding of the risks involved. This is what happened in the case of PancakeBunny.



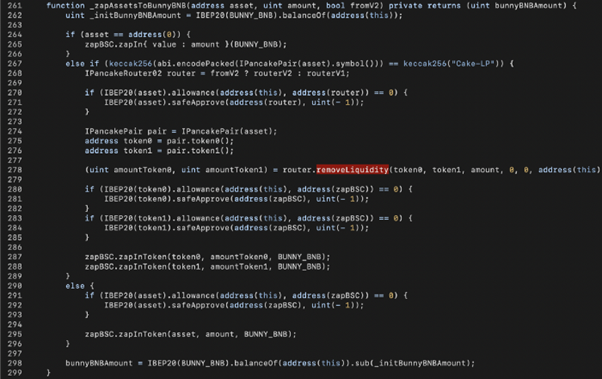
In the code snippet above, line 140 retrieves the balance of LP token via balanceOf() and stores it into liquidity. In lines 144–145, the portion of total LP tokens owned by UniswapV2Pair (i.e., liquidity out of \_totalSupply) is used to derive (amount0, amount1) with the current balances (balance0, balance1) of the two assets (i.e., token0 and token1). Later on, (amount0, amount1) of the two assets are transferred to the address in lines 148–149.

Here, a bad actor could manipulate (balance0, balance1) and the liquidity by sending some token0+token1 or the LP token into the UniswapV2Pair contract right before the mint() function is invoked to make the caller get more token0+token1 out. We’ll walk you through the PancakeBunny source code and show you how the bad actor can profit from doing this.

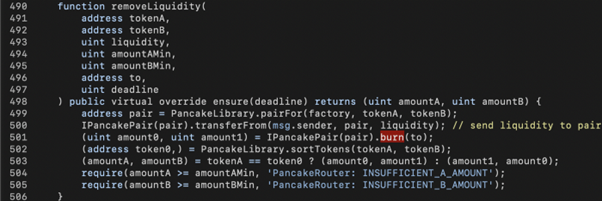
**Loophole Analysis: BunnyMinterV2**



In the PancakeBunny source code, the BunnyMinterV2.mintForV2() function is in charge of minting BUNNY tokens as rewards. Specifically, the amount to be minted (i.e., mintBunny)is derived from the input parameters: *\_withdrawalFees*, and *\_performanceFee*. The computation is related to three functions: *\_zapAssetsToBunnyBNB()*(line 213), *priceCalculator.valueOfAsset()* (line 219) and*amountBunnyToMint()* (line 221). Since the bad actor can mint a large amount of BUNNY, the problem lies in one of the three functions mentioned above.



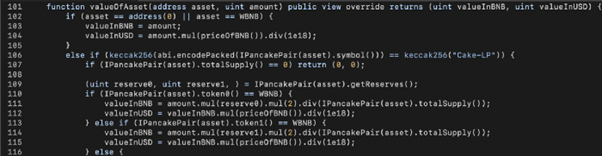
Let’s start from the *\_zapAssetsToBunnyBNB*() function. When the passed-in asset is a Cake-LP (line 267), a certain amount of LP tokens is used to remove liquidity and take (amountToken0, amountToken1) of (token0, token1) from the liquidity pool (line 278). With the help of the zapBSC contract, those assets are swapped for BUNNY-BNB LP tokens (lines 287–288). A corresponding amount of BUNNY-BNB LP tokens is then returned to the caller (line 298). Here, we have a problem. **Does the amount match the amount of LP tokens you assume to be burned?**



In the implementation of PancakeV2Router.removeLiquidity(), liquidity of LP tokens (amount in zapAssetsToBunnyBNB()) would be sent to the PancakePair contract (line 500) and PancakePair.burn() would be invoked. If the current LP token balance of PancakePair is greater than 0, the actual amount to be burned would be greater than amount, which indirectly increases the BUNNY amount to be minted.

**Another issue in \_zapAssetsToBunnyBNB() is the zapBSC.zapInToken() call.** The logic behind this is to exchange the two assets collected by the removeLiquidity() into BUNNY-BNB LP tokens. Since zapBSC swaps assets through PancakeSwap, the bad actor could use flash loans to manipulate the amount of swapped BUNNY-BNB.

Back to BunnyMinterV2.mintForV2(), the bunnyBNBAmount returned by zapAssetsToBunnyBNB() would be passed into priceCalculator.valueOfAsset() to quote the value based on BNB (i.e., vauleInBNB), similar to an oracle mechanism.



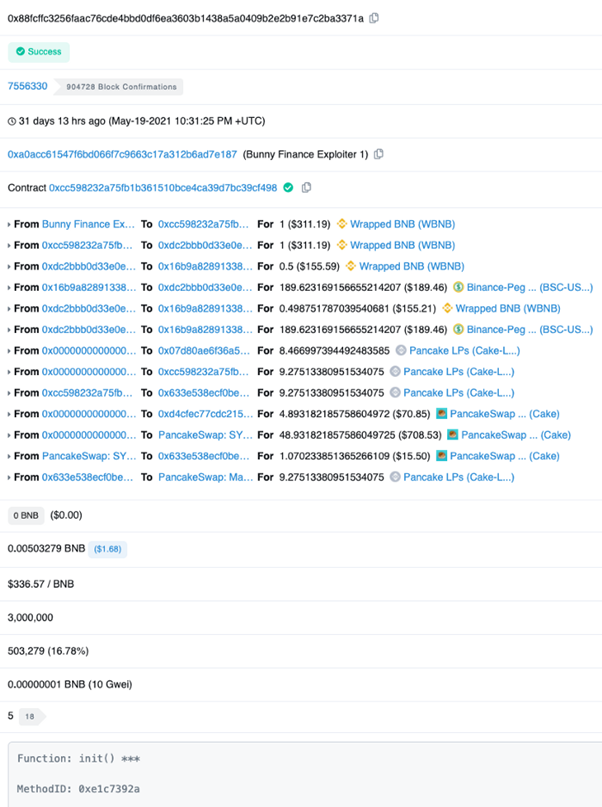
However, **priceCalculator.valueOfAsset() references the amount of BNB and BUNNY (reserve0, reserve1) in the BUNNY\_BNB PancakePair as the price feed,**which enables the bad actor to use flash loans to manipulate the amount of BUNNY tokens minted.



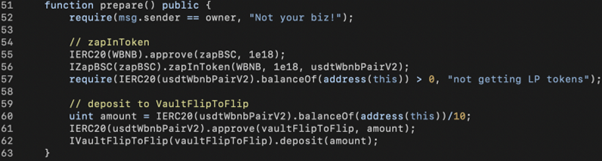
The amountBunnyToMint() function is a simple math calculation. The input contribution is multiplied by five (bunnyPerProfitBNB = 5e18), which itself has no attack surface, but the amplification magnifies the manipulation mentioned above.

**Preparing the Attack**

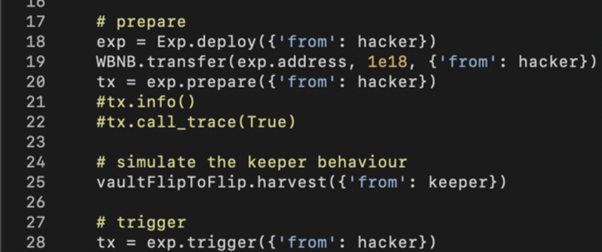
Since the attack is triggered by **getReward()**, we need to be qualified for rewards first.



As shown in the Etherscan screenshot above, the PancakeBunny hacker invoked the init() function of the exploit contract to exchange 1 WBNB to WBNB-USDT-LP tokens and deposit() them into the VaultFlipToFlip contract, such that he would get some rewards by invoking getReward().



As shown above, using the Exp.prepare() function we reproduced the vaultFlipToFlip.deposit() call (line 62). We also used the ZapBSC contract to simplify obtaining LP tokens (lines 54–57). However, one isn’t able to get rewards until the PancakeBunny keeper triggers the next harvest() call. For this reason, the PancakeBunny hacker didn’t trigger [the attack](https://bscscan.com/tx/0x897c2de73dd55d7701e1b69ffb3a17b0f4801ced88b0c75fe1551c5fcce6a979) until the first harvest() [transaction](https://bscscan.com/tx/0x9c48fd13d65f5f951882282444a45a7b84c4f673891bbdcc48af68ed305950bb) following the init() [transaction](https://bscscan.com/tx/0x88fcffc3256faac76cde4bbd0df6ea3603b1438a5a0409b2e2b91e7c2ba3371a).

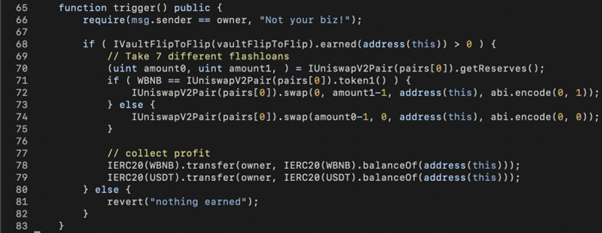


In our simulation, no keeper can trigger the harvest(). Therefore, we leverage the feature of eth-brownie to impersonate the keeper and manually initiate the harvest() transaction (line 25).

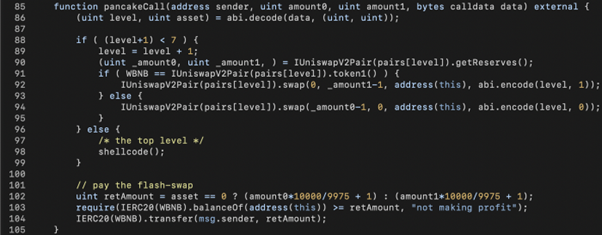
**Recursive Flash Loans**

To leverage funds, the PancakeBunny exploiter utilized eight different fund pools including seven PancakePair contracts and the ForTube Bank. Here, Amber’s blockchain security team team only used the following seven PancakePair contracts’ flash-swap feature to loan 2.3M WBNB:

address[7] pairs = [  
address(0x0eD7e52944161450477ee417DE9Cd3a859b14fD0),  
address(0x58F876857a02D6762E0101bb5C46A8c1ED44Dc16),  
address(0x74E4716E431f45807DCF19f284c7aA99F18a4fbc),  
address(0x61EB789d75A95CAa3fF50ed7E47b96c132fEc082),  
address(0x9adc6Fb78CEFA07E13E9294F150C1E8C1Dd566c0),  
address(0xF3Bc6FC080ffCC30d93dF48BFA2aA14b869554bb),  
address(0xDd5bAd8f8b360d76d12FdA230F8BAF42fe0022CF)  
];

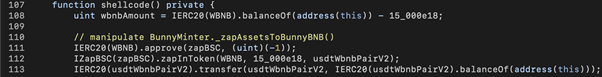


To simplify the flash-swap calls, we packed two parameters into the fourth input argument of the PancakePair.swap() calls (line 72 or line 74): level and asset. The level variable indicates which level of swap() call we’re in; the asset variable is 0 or 1, meaning we need to borrow token0 or token1.

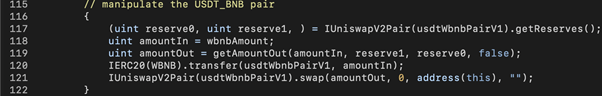


Using the callback function pancakeCall(), we recursively call PancakePair.swap() with level+1 until we reach the seventh level. At the top level, we invoke shellcode() to perform the real action in line 98. When shellcode() returns, the asset variable returns the borrowed asset in each corresponding level (lines 102–104).

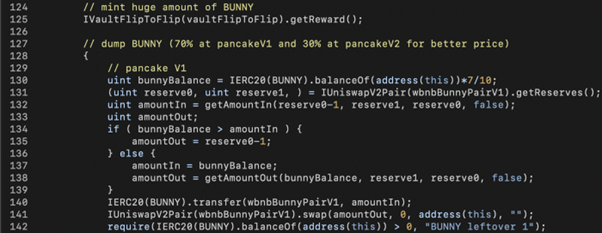
**Pulling the Trigger**



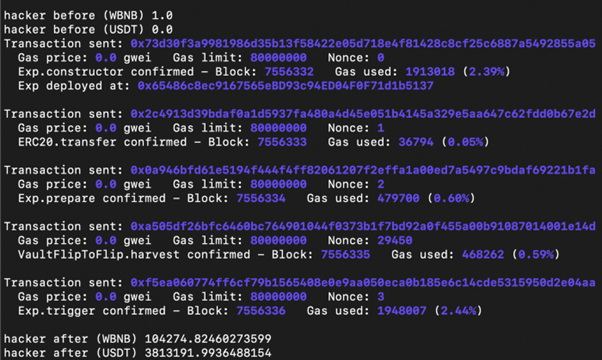
The shellcode() function invoked by the seventh level of pancakeCall() is the actual exploit code. First, we keep the current balance of WBNB in wbnbAmount (line 108), swap 15,000 WBNB into WBNB-USDT-LP tokens (line 112), and send them to the contract which minted those LP tokens (i.e., the PancakePair contract) in line 113. This step aims to manipulate the removeLiquidity() call inside the \_zapAssetsToBunnyBNB() function as analyzed above, enabling us to receive more WBNB+USDT than expected.



The second step is to manipulate the USDT price referenced by \_zapAssetsToBunnyBNB() to swap USDT into WBNB. Since \_zapAssetsToBunnyBNB() uses WBNB-USDT PancakePair to swap USDT to WBNB, **we could swap the rest of the flash loaned WBNB to USDT on PancakeSwap.** Doing so would make WBNB extremely cheap, and \_zapAssetsToBunnyBNB() would receive a disproportionately large amount of WBNB when swapped from USDT. Note that the price manipulation here occurs on the Pancake V1 pool, not Pancake V2’s PancakePair as in the previous step.



**The final step is the getReward() call.** The simple contract call could mint 6.9M BUNNY tokens (line 125). The BUNNY tokens could then be swapped for WBNB on PancakeSwap to pay back the flash loan.



In our simulation, the bad actor pays 1 WBNB and walks away with 104k WBNB + 3.8M USDT (equivalent to ~$45M).