Rareskills hack solutions

Ethernaut

3

```
The exploit lies in the fact that the block.number is not random

If you write a contract which has the exact same logic to generate the side
and call the main contract flip with this generated side it will work

This is because the whole thing runs in one transaction and hence
the block.number will be the same

The contract doesn't allow 10 flip guesses within the same block though

So you do need to call the exploit contract 10 times each submitted
after the last block is done
```

4

```
The exploit lies in the fact that the contract allows anyone to change the owner
as long as tx.origin != msg.sender

If A calls contract B which calls contract C, tx.origin is A while msg.sender is B

So if we deploy a contract which can call the ethernaut contract's
changeOwner method, tx.origin and msg.sender will differ and we get to
gain ownership
```

5

```
Ethernaut deploys the contract with an _initialSupply of 20

So us as the msg.sender now has a balance of 20

This contract uses solidity ^0.6.0 which means there is no arithmetic overflow
protection

So see what happens if you call the transfer function with _to as some random
```

```
address and _value being 21

balances[msg.sender] - _value becomes 20 - 21 which causes an underflow
and results in 2^256-1 as the value

This means that the require check in line 14 passes as
2^256-1 is greater than or equal to 0

And it sets our balance to 2^256-1 in line 15 which is the
largest possible uint256 value in solidity. Exploited :(

Also the _to address in the transfer call shouldn't be our address as that would cause the
underflow caused in line 15 to overflow again in line 16 back to value 20
```

10

```
The exploit here is reentrancy

The contract does a msg.sender.call before updating the balances object

This means we can write an exploiter contract with a exploit method which calls the withdr
aw method and has a receive callback which again triggers the exploit method

This cycle will keep repeating till the entire balance of the contract is
drained and then the exploit method terminates

Note: this approach would also have failed if the balance reduction logic used
SafeMath or had overflow protection
```

11

```
The solution lies in the fact that we get to:
  1. implement the isLastFloor() method for the building
  2. the elevator has a goTo() method which calls isLastFloor() twice:
      a. once to check if the floor is the top floor of the building
      b. setting value of top

See how the elevator sets the floor state variable between the above 2 events

So all we need to do is implement the isLastFloor() view method such that
it returns two different values depending on the value of floor in the elevator

Return true if the floor state variable is equal to the top floor for our
building (we just take that as 101 here) and false otherwise
```

15

```
`transfer` is guarder by the lockTokens modifier but `transferFrom` isn't!

All we need to do is approve ourselves as a spender for the entire supply
and then use the transferFrom method to move the tokens
```

17

```
There are two aspects to this problem:
  1. Figuring out the token address as its supposed to be lost
  2. How to drain the ether stored in the token

Solving 1 is not hard. The address of a new contract is determenistic and only
depends on the deployer address(aka the factory) and the nonce

Solving 2 is easy too. Notice the token has a unprotected destroy function
which anyone can call. This function calls selfdestruct and hence anyone can drain the tok
en of its ether provided they calculate the address from the step
above
```

20

```
This is a very interesting exploit

We need to make sure that the owner cant call the withdraw method
But the only thing we can control is who the partner is
The withdraw function does send ETH to the partner
But it uses a call method so us writing a contract to act like a partner
and reverting on ETH received wont work. The contract doesn't check the
success bool of this call method

The ethernaut problem saying: whilst the contract still has funds, and
the transaction is of 1M gas or less gives us two hints:

Reentrancy is not the answer here though we can use it to drain the funds in the contract

But gas limit of upto 1M shows us the solution

When a call method happens, the person making the transaction, i.e. the owner
has to spend gas to execute the logic executed by the partner contract
on receiving ETH

So if we write a contract which on receiving ETH executes logic which spends more than 1M
```

```
 gas and make this contract the partner, the owners withdraw call will fail as it is calle
d with a gaslimit < 1M
```

21

```
The solution lies in the fact that we get to:
  1. implement the price() method for the buyer
  2. the shop has a buy() method which calls price() twice:
       a. once to check if the price is acceptable for the shop
       b. setting the actual price paid

See how the shop sets the isSold state variable between the above 2 events

So all we need to do is implement the price() view method such that
it returns two different values depending on the value of isSold in the store

Return a large amount for the min-price check and then return 0 for the final
call
```

Capture the Ether

Guess the new number

```
All you need is a contract to call the guess function
with the guess being the same logic as whats in the above contract

The block.number and now will be the same in the same transaction
so the answer will be the same
```

Guess the number

```
The answer is 42

Its hardcoded in the contract :P
```

Guess the random number

```
Notice how the answer is calculated and stored in the constructor

It just uses blockhash and block.timestamp both of which can be easily queried

i.e. after this contract is deployed, you can easily get all the data required
to calculate this answer by just looking at the deployed and prev block

So calculate it and call the guess function with that value
```

Guess the secret number

```
I thought this was impossible till I saw that the guess was supposed to
be a uint8

So the possible guesses are just 0 to 255

Just calculate keccak256 for all off them and see which one matches the hash in the contra
ct
```

Predict the block hash

```
The trick here lies in the fact that blockhash will return 0 for
block numbers which are more than 256 blocks older than the current block

So lock in a guess of 0 as the guess, wait for 256 blocks to be mined and then
call the settle method

The blockhash will return 0 as the settlementBlockNumber is now
more than 256 blocks old and your guess of 0 will match
```

Predict the future

```
Here you are supposed to guess a future value

But that guess can only be one of 0-9

This future value is represented by a fixed logic for each block

We write an exploiter contract and make it guess a value 0 initially

Then for every block we make it calculate the correct guess for that block
```

```
If it matches the initially guessed value, we make the exploiter contract call
the settle function

We repeat this logic for every block till a match eventually happens
```

Token Bank

```
The bank contract has a reentrancy bug

Notice how in the withdraw method, the token transfer method is called before
the balanceOf object is updated

If we can find a way to call the withdraw method again through the transfer
method before the balanceOf object is updated we can effectively drain the bank of its tok
en backing

But the token is a ERC223 Token

Which means all we need to do is to make the player a contract which implements a tokenFal
lback method which calls the withdraw method on the bank again till
its fully drained
```

Token Sale

```
The max value of a uint256 is 2^256-1

2^256 overflows and becomes 0 because this contract is still on pragma ^0.4.21

The buy logic in line 16 does numTokens * PRICE_PER_TOKEN

Price per token is 1 ether i.e. 10^18 wei

So if the user is able to make numTokens * PRICE_PER_TOKEN equal 0
when calling the buy function with 0 wei, the require check will pass

So call the function with numTokens as 2^238

Multiplying with 1 ether the require statement will calculate 2^256 which becomes 0

The require check passes and the user is alloted 2^238 tokens

Now withdrawing the 1 ether present in the contract is as simple as calling
sell with the numTokens as 1
```

Token Whale

```
The max value of a uint256 is 2^256-1

2^256 overflows and becomes 0 because this contract is still on pragma ^0.4.21

Similarly 0 underflows and becomes 2^256-1 too

The exploit is with how transferFrom calls the _transfer method

Notice how _transfer reduces the balance from msg.sender not the
from address which should be the case when called by transferFrom

So the exploit works as follows:
  1. Assume there are 2 users: user1 and user2. user1 starts with 1000 tokens
     and is the player
  2. First user1 calls transfer(user2, 1000) and transfers all their tokens
     to user2
  3. Then user2 calls approve(user1, 1) and gives user1 approval
     to spend 1 token they hold
  4. So user1 has 0 tokens, user2 has 1000 tokens and user1 has approval
     to spend 1 token of user2
  5. Now all user1 has to call is transferFrom(user2, address(0), 1)
  6. All the require checks pass
     1. balance[user2] == 1000 which is > 1
     2. balanceOf[to] + value >= balanceOf[to] works fine because
        we used address(0)
       Any address here works for value 1 as long as the to address
       doesnt have a balance of 2^256-1
     3. allowance[from][msg.sender] == 1 which is >= 1
     4. line: 27 now does balanceOf[msg.sender] -= value
        but balanceOf[msg.sender] is 0 as msg.sender is user1
        value is 1
        So 0-1 happens which underflows to 2^256-1 and balance[user1]
        gets set to this
```

Damn Vulnerable Defi

Free rider

```
There are two bugs in the marketplace that makes this exploit possible:
  1. The buyMany method calls the _buyOne method which checks msg.value,
     but this means you can buy multiple nft's and it will only check if the
     sent ETH is enough to buy 1 nft
     i.e. if you try to buy 10 nft's worth 10 ETH each
     you can call the buyMany method to buy all of them
```

```
     and just need to send 10 ETH instead of the required 100 ETH
  2. The pay seller part of the _buyOne method is actually
     paying the buyer the ETH not the seller
     This is because it assumes token.ownerOf(tokenId) is the seller which is
     infact the buyer because the previous line already transfered the nft
     from the seller to the buyer

Using the above 2 bugs, we can get a balance of the following in our attacker
address:

0.5 ETH(initial balance) +
90 ETH(total ETH in marketplace) +
45 ETH(buyer reward) -
15 ETH(used to buy all the nft's from the marketplace) -
uniswapFees
to get a total of ~120.45 ETH

The steps to do this is as follows:
  1. Write a exploit contract which implements IUniswapV2Callee and
     IERC721Receiver and has a receive() method to accept ETH
  2. Use uniswap to get a flashloan of 15 WETH
  3. In the uniswap callback method, withdraw the 15 WETH for 15 ETH
  4. Use the 15 ETH to buy all 6 nft's exploiting bug #1
     Now you have ownership of all 6 nft's
     Bug #2 also means the marketplace sent you 90 ETH
  5. Transfer all 6 nft's to the buyer hence getting 45 ETH from the buyer
     as the reward
  6. Calculate the amount to pay back to uniswap to settle the flash loan
     This will be 15 WETH + 0.3% fees
  7. We only posses ETH right now
     Deposit ETH equal to the amount calculated in step #6 to the
     WETH contract to gain the same amount of WETH
  8. Transfer the above WETH to the uniswap pair to settle the flash loan
  9. Send all remaining ETH to the attacker addres
     For the above tests this amount should be ~120.45 ETH
```

Naive receiver

```
The exploit lies in the fact that the flashLoan method in the pool
has borrower as an argument

This means that you can borrow loans on behalf of anyone

And every loan causes a 1 ETH fee to be paid back to the pool

So all we need is to call the flashLoan method with
the borrower as the receiver address to drain ether from the receiver

As the receiver has 10 ETH and the fixed fee for the pool is 1 ETH
we need to call the flashLoan method with the receiver as the borrower
```

```
10 times

This can be done in 1 transaction if you do all 10 calls using
one method in an exploit contract
```

Side entrance

```
Another fun exploit

The flashLoan method only checks if the total ETH in the pool is same at the
end

It doesn't check if the balances value of the borrower is the same

The pool also allows ETH you deposited into the pool to be withdrawn

So the exploit works like this:
  1. write an exploit contract which implements IFlashLoanEtherReceiver
     and can receive ETH
  2. in the execute method called by the pool, deposit all ETH borrowed
     back into the pool
  3. call the flashLoan method borrowing all the ETH in the pool
     followed by calling the withdraw method followed by sending
     the withdrawn ETH to the attacker address we want the final ETH to reach
  4. this causes the exploit contract to first borrow all the ETH
     and then deposit it back into the pool because the flashLoan triggers
     the execute method
     The execute method deposits the borrowed ETH and
     updates the balances of the exploit contract
     Now all the ETH in the pool is withdrawable by the exploit contract
     Hence the following method calls like withdraw work as expected
     draining the pool
```

Truster

```
This is a fun exploit. The exploit lies in the fact that the pool
does a target.functionCall(data) where both target and data is something
you can pass in

But the pool does a balance check after that

So what we want to do is do some sort of call which grants you the ability to
drain the tokens from the pool after calling flashLoan

Wait the token is a ERC20 contract and hence has an approve call
```

```
So the exploit works like this:
  1. write an exploit contract
     we want to do all this in 1 transaction
  2. call the flashLoan method with
     borrowAmount=0,
     borrower=attacker(any address is fine in this case),
     target=erc20TokenAddress and
     data=approve(address(exploitContract),fundsToDrain)

     Essentially we make the pool call the approve method on the token
     to grant our exploit contract approval
     to move all the funds in the pool later via transferMany

     As this is only an approval and not an actual transfer,
     the balance checks towards the end of the flashLoan
     method passes just fine
  3. call the transferFrom method on the token to transfer all the pools funds
     to the attacker
```

Unstoppable

```
The exploit lies in the fact that the flashLoan method in the lender
contains a check saying assert(poolBalance == balanceBefore)

It's assuming that the lender will only receive tokens via
its depositTokens method which updates poolBalance too

But the token is a normal ERC20 contract

Anyone can send tokens to the lenders balance and increase it thus
causing poolBalance to not match the token balance of the lender

This causes assert to revert the flashLoan transaction and the lender breaks!
```