TRUSTLOOK

January 10, 2019

Smart Contract Guardian – an online EVM decompiler

Since I started working in the Ethereum ecosystem and auditing Ethereum smart contract in bytecode format. I have evaluated many well-known projects which claimed they can decompile EVM (Ethereum Virtual Machine) bytecode. However, none of them really show good result for real world examples. So reading the EVM opcodes from the smart contracts is a really frustrating job and you can be lost anywhere among the JUMPs and POPs. So an idea popped into my mind that why not make a really working one to speed up the audit of raw EVM bytecode.

Fortunately, there are already several good articles talking about EVM bytecode structures. However, those resources still won't make you fully understand how to make a real EVM decompiler. There are a lot of details lack from either official documents or other research. I have written a series of articles about the things I have learned from the development of the tool. If you are interested in them, please feel free to have a look:

Understand EVM bytecode – Part 1

Understand EVM bytecode – Part 2

Understand EVM bytecode – Part 3

Understand EVM bytecode – Part 4

Besides all the information I have mentioned in those articles, there are still more things need to pay attention to if you want to development your own decompiler or automation tool based on EVM bytecode. The most difficult part I have encountered during the development is how to retrieve back the Control Flow Graph (CFG) for the code. If you are familiar with EVM opcodes, you might have noted there is no function call instruction and return. Instead there is only conditional and unconditional JUMPs. So in order to extract back the original function logic, you need to analyze the pushed return address of the instructions in the stack to decide the range of a function. However, if you think you can skip this step and take the whole code as one function, you will have even more troublesome when dealing with loops. Because without defining functions any re-entrancy of code blocks might be judged as a loop mistakenly.

Another thing worth mentioning is that EVM itself is under development too. Also there are several high level languages you can choose to compile a smart contract. So the compiled EVM bytecode is very compiler dependent, even version dependent on by same compiler. For my research work, I used Remix online Solidity compiler. If you choose different versions to compile on a same piece of code, you will be surprised by the results. Since Solidity is under development too, it is no way you can expect consistent compiled results. So these facts will make the developer life harder, especially when more readable content is expected from the decompiler.

We have talked the problems you might encountered during the development. Let's look at a demo of the online decompiler we have published. You can easily use it by following link:

https://www.trustlook.com/products/smartcontractguardian

To analyze a piece of EVM bytecode, you can either specify the smart contract address which has been deployed on the main network of Ethereum, or simply copy and paste your bytecode into the text box. After clicking on the "**Decompile Now**" button, the result will be shown on the page. Let's take some simple Solidity source code for a testing:

```
pragma solidity ^0.4.18;

contract Bank {
    // balances, indexed by addresses
    mapping(address => uint256) public balanceOf;

    function deposit(uint256 amount) public payable {
        require(msg.value == amount);

        // adjust the account's balance
```

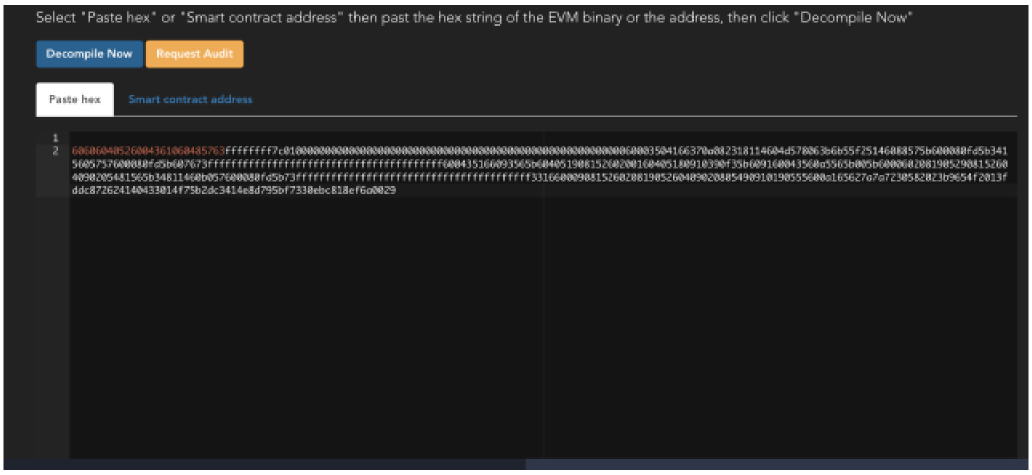
```
            balanceOf[msg.sender] += amount;
        }

    }
```

After compiler the code using Solidity 0.4.18, you will get following EVM runtime bytecode:

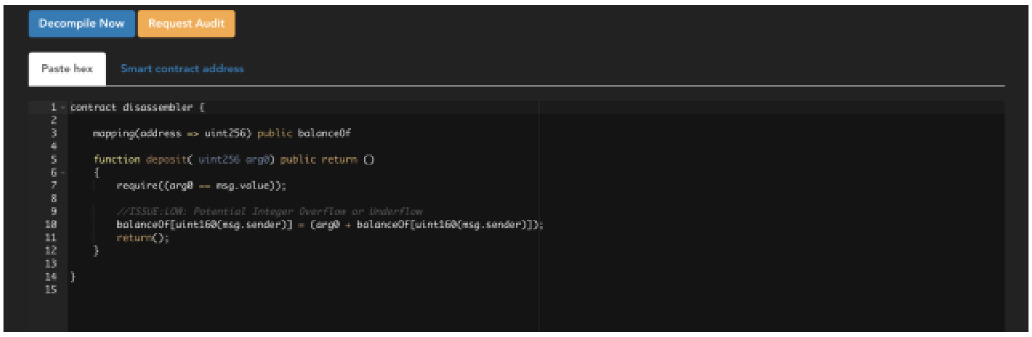
```
60606040526004361060485763ffffffff7c0100000000000000000000
00000000000000000000000000000000006000350416637 0a0823181146
8063b6b55f25146088575b600080fd5b3415605757600080fd5b60767
fffffffffffffffffffffffffffffffff600435166093565b60405
52602001604051809103 90f35b609160043560a5565b005b600060208
9081526040902054815 65b34811460b057600080fd5b73ffffffffff
fffffffffffffffffffffff33166000908152602081905260409020 8
910190555600a165627a7a7230582023b9654f2013fddc872624140 43
75b2dc3414e8d795bf7330ebc818ef6a0029
```

To understand why we call it "runtime" bytecode, you can refer to my write-ups about "Understanding EVM bytecode". You can simply copy above bytecode and submit to the online decompiler like this:


Select "Paste hex" or "Smart contract address" then past the hex string of the EVM binary or the address, then click "Decompile Now"

```
60606040526004361060485763ffffffff7c01000000000000000000000000000000000000000000000000000000006000350416637 0a08231811460 4d578063b6b55f251460885 75b600080fd5b341
5605757600080fd5b607673ffffffffffffffffffffffffffffffffffffffff600435166093565b60405190815260200160405180910390f35b609160043560a5565b005b600060208190529081526 0
4090205481565b34811460b057600080fd5b73ffffffffffffffffffffffffffffffffffffffff331660009081526020819052604090208054909101905 5600a165627a7a7230582023b9654f2013f
ddc87262414043 3014f75b2dc3414e8d795bf7330ebc818ef6a0029
```

Then you can click on the "**Decompile Now**" button. After a few seconds, you could have the decompiled result shown in the text box:



```
contract disassembler {

    mapping(address => uint256) public balanceOf

    function deposit( uint256 arg0) public return ()
    {
        require((arg0 == msg.value));

        //ISSUE:LOW: Potential Integer Overflow or Underflow
        balanceOf[uint160(msg.sender)] = (arg0 + balanceOf[uint160(msg.sender)]);
        return();
    }

}
```

From the decompiled result, we can clearly see the original variable declarations and functions. Besides that, it also inspect the bytecode and look for potential known vulnerabilities. Here, an integer overflow was discovered in function deposit. If you do not trust the automation inspection result, you can always click on the "**Request Audit**" button for a manual review, our team will be happily to assistant you with our expertise.

Of course this sample is pretty simple, the real world example can be much more complicated. You can also check them by using the tool. The tool currently have following features:

- Public and private functions recognition
- Storage variables recognition
- Memory related operations optimization
- External calls commentating
- Embed Solidity functions recognitions (*ecrecover, sha256, ripemd160, sha3*)
- Well-known vulnerabilities inspection

Besides the above features, there are still more things which can be improved in future. Loops recognition can be done in better way other than giving goto statements for now. Also dynamical size variables like bytes and string can be optimized in better way for readability. There might also be bugs on showing local variables since there is no RET instruction in EVM. So the stack alignment is always an issue when function returns. Since the tool is still on experimental, you are very welcome to contact us for any issue you had found.

Hope you enjoy using it, let’s us know your comments and advise!

#PRODUCTS

AUTHOR

c0zzy

Read more posts by this author.

ALSO ON **TRUSTLOOK BLOG**

安全应用程序审核 -- Lionmobi

3 years ago • 1 comment

VirusTotal APK 病毒检测统计 2021-08

4 months ago • 1 comment

VirusTotal (简称 VT), 是谷歌旗下一家免费提供可疑文件扫描服务的网站. VT …

VirusT Malwa

6 montl

At Trus
feed frc
a daily

What do you think?

4 Responses

Upvote Funny Love Surprised Angry Sad

0 Comments **Trustlook blog** 1 Login

Favorite Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.