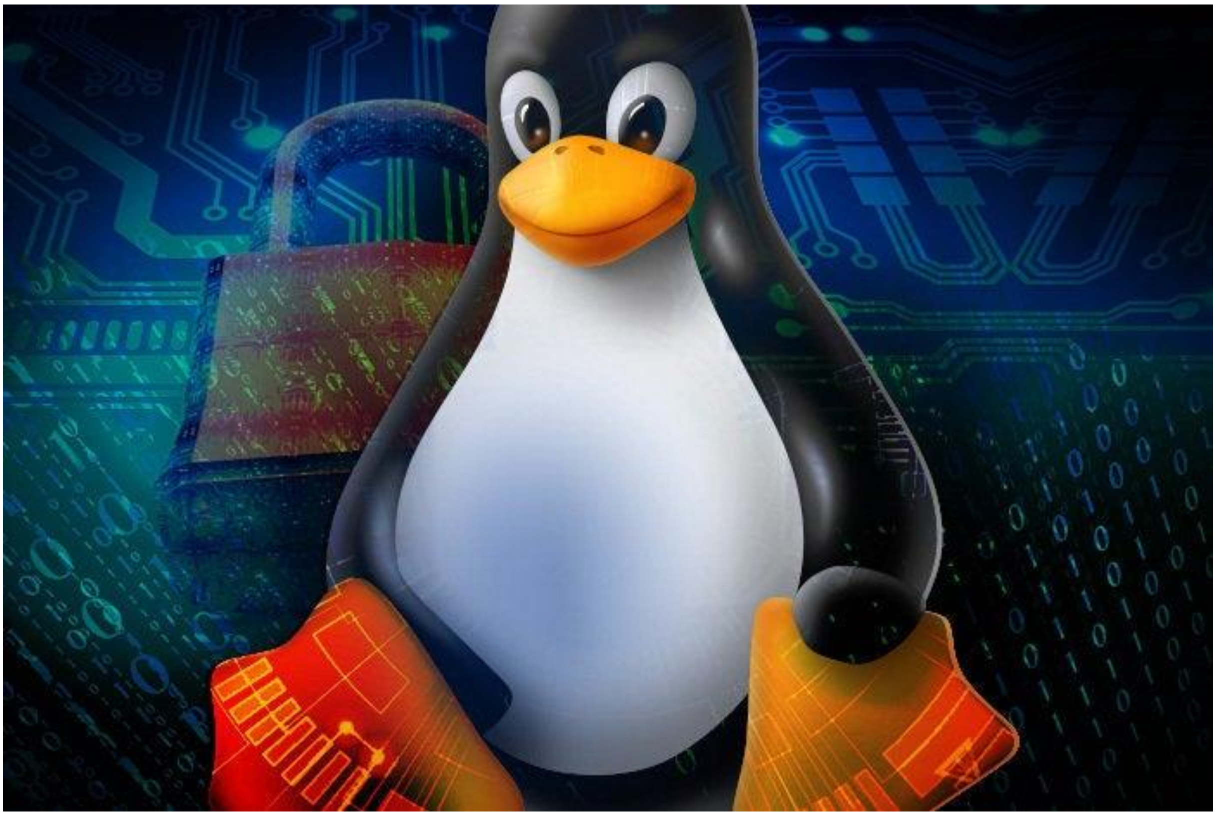
Prepeared By @ALPARSLAN AKYILDIZ

LINUX ASSEMBLY AND EXPLOIT DEVELOPMENT ARTICLE SERIES PART 3

- **GDB USAGE**
- **EXECVE CUSTOM SHELLCODE CREATION**
- **CREATING FUNCTIONS**
- **STRING OPERATIONS**
- **LOOPS**



Contents

LAB 4: Data Types in Assembly Programming 3
LAB 5: Assembly Variable Assignment And Moving Operations 5
LAB 6: Assembly Character Operations 14
LAB 7: Assembly JMP Command, LOOP Creation And EXECVELAB 7 24
LAB 8: Creating a Linux Assembly Function 34
LAB 9: Shellcode with EXECVE 36

LAB 4: Data Types in Assembly Programming

One of the points that should be known when writing a program using the assembly is the size of the data. 1 byt field is reserved for the variable defined by the .byte. .ascii is used for specifying character strings, and .asciz, terminated characters, and .int, 32-bit integer values, and .short, 16-bit integer values, and .float, fractional numbers. Common memory areas are indicated by .comm, and local memory areas, by .lcomm. For a better understanding of the subject, a simple assembly program will be written as follows and analyzed with the GDB.
Within the code, “assembly hacker” will be inserted into the string variable, and the value 32, in the variable integer, and the elements 1, 2, 3, 4, 5, in the variable array, and the element 12, in the variable short, and the element 11, in the variable byte. A field of 2000 bytes, which is not initilized with initial values, is reserved in the common memory area.

```
.data

        string:

                .asciz "assembly hacker"

        integer:

                .int 32
        array:

                .int 1,2,3,4,5
        short:
                .short 12
        bayt:
                .byte 11

.bss

        .comm buffer, 2000

.text
        .global _start

        _start:

                nop
                movl $1, %eax
                movl $0, %ebx
                int  $0x80
```

Since the codes will be analyzed with GBD, the parameter –gstabs is used while they are being converted into the objects.

<u>The program is linked as follows</u>

```
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ as -gstabs datatypes.s -o datatypes.
o
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ ld datatypes.o -o datatypes
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ gdb datatypes
```

While analyzing with the tool GDB, the following steps are examined:

ri

```
28                      movl $1, %eax
29                      movl $0, %ebx
30                      int  $0x80
(gdb)
31
(gdb) break 28
Breakpoint 1 at 0x8048075: file datatypes.s, line 28.
(gdb) run
Starting program: /home/passwd/Desktop/assembly/AT&T/datatypes

Breakpoint 1, _start () at datatypes.s:28
28                      movl $1, %eax
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x08049084  string
0x08049094  integer
0x08049098  array
0x080490ac  short
0x080490ae  bayt
0x080490b0  buffer
```

The beginnig addresses of the data kept within the variables in the memory are displayed using the command info variables after the breakpoint has been placed at the code in the 28th line and the program has been executed with the command run. The initial address point of the “assembly hacker” array that the string variable holds in itself is the address value that is written opposite the string variable. With the x / examine command, the data in the addresses can be displayed as shown below:

```
(gdb) x/15cb 0x08049084
0x8049084 <string>:     97 'a'  115 's' 115 's' 101 'e' 109 'm' 98 'b'  108 'l' 121 '
y'
0x804908c <string+8>:   32 ' '  104 'h' 97 'a'  99 'c'  107 'k' 101 'e' 114 'r'
(gdb) x/4db 0x08049094
0x8049094 <integer>:    32      0       0       0
(gdb) x/2db 0x080490ac
0x80490ac <short>:      12      0
(gdb) x/1db 0x080490ae
0x80490ae <bayt>:       11
(gdb) x/5db 0x08049098
0x8049098 <array>:      1       0       0       0       2
(gdb) x/55db 0x08049098
0x8049098 <array>:      1       0       0       0       2       0       0       0
0x80490a0 <array+8>:    3       0       0       0       4       0       0       0
0x80490a8 <array+16>:   5       0       0       0       12      0       11      0
0x80490b0 <buffer>:     0       0       0       0       0       0       0       0
0x80490b8 <buffer+8>:   0       0       0       0       0       0       0       0
0x80490c0 <buffer+16>:  0       0       0       0       0       0       0       0
0x80490c8 <buffer+24>:  0       0       0       0       0       0       0
(gdb) x/10db 0x080490b0
0x80490b0 <buffer>:     0       0       0       0       0       0       0       0
0x80490b8 <buffer+8>:   0       0
```

As can be seen in the screenshot, there are the variable values created within the memory region pointed by the addresses.

LAB 5: Assembly Variable Assignment And Moving Operations

In order to understand the assembly coding while reading the assembly codes, the syntax and rules of the transport operations must be known. 32-bit data is moved with the command movl, and 16-bit data, with the command movw, and and 1-byte, that is to say 8-bit, data, with the command mowb. The data movment can be done from one register to another register. For example, with the command % ecx, the movl% eax is moved into the register ecx in the eax. The point to be considered while moving data is that the source and target must be the same size. In addition, data can be moved from a memory region to a register (vice versa).

Let's examine the representative code below:

variable_place:

.int 20

movl %eax, variable_place: The address showing the value 20 is placed in the eax.

In that … case, the (% eax) shows 20, and the % eax moves the address value to …. The immediate values (data such as fixed numbers) can be copied directly into the register. For example, the following code can be used to put the value 20 into the register eax:

movl $20, %eax

The value of the variable in a memory region can be changed as follows:

memory_place:

.byte 10

movb $20, memory_place: while there is the value 10 in the memory address, the value 20 is put into the place indicated by this address. The structure ***array(offset, index, size)*** is used to move the value into the array. A sample movement is shown in the output below:

array:

.int 10, 20, 30, 40, 50

movl %eax, array(0,2,4): The value in the eax will be written instead of 30.

<u>Important Rules for AT&T Syntax:</u>

- When the “$” sign comes before the tag name, the memory address of the variable is taken. For example, when variable_place was written in the above example, the value stored in the address was taken. If $ variable_location was used, the address pointing to the variable would be represented.

- (% edi) indicates the value stored in the region indicated by the address in the edi register. The memory region where the value 9 is kept with the edi is displayed with the command movl $11, (%edi), that is the (%edi) gives the value of 9.

- The meaning of the code ***movl $4, 9(%edi)*** is that 9 will be added to the address indicated by the edi and the number 9 will be placed in the newly displayed memory region: (EDI + 9).

- Likewise, the -2 (% edi) indicates the location of the address (EDI -2).

The GDB analysis will be performed on the program by writing additional codes to the codes written in LAB 4 for a better understanding of the subject. The following codes are added to the section _start: on the program codes written in LAB 4:

```
        .global _start

        _start:

                #tamsayı değerini direk register içerisine taşıma register içeri
                sinde adres yerine rakam olacak
                movl $20, %eax

                # değeri değişken içine atama integer değişkeni bir hafıza bölge
                si içerisine 6 değeri atılacak
                movl $6, integer

                # değişken içerisindeki değeri registera atma
                movl bayt, %eax

                # yazmactan hafızaya dolaylı atama yapma
                movb $2, %al
                movb %al, bayt

                #  hafıza bölge adresinin yazmaca atanması ve yazmacın işaret et
                tiği bölge içerisindeki değerin değiştirilmesi
                movl $bayt, %edi
                movl $12, (%edi)

                # dizinin 3. elemanının değiştirilmesi
                movl $0, %ecx
                movl $2, %edi
                movl $13, array(%ecx, %edi, 4)

                nop
                movl $1, %eax
                movl $0, %ebx
                int  $0x80
```

After the program written above has been assembled and linked, it is analyzed with the GDB as follows.

The program is linked using the the commands below:

as –gstabs program.s –o program.o

ld program.o –o program

The analysis starts with the command gdb./program. The breakpoint is set at the point _start. In this way, the program flow will be monitored.

"gdb ./program" ?

```
(gdb) break _start
Breakpoint 1 at 0x8048074: file datatypes.s, line 28.
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x080490bc  string
0x080490cc  integer
0x080490d0  array
0x080490e4  short
0x080490e6  bayt
0x080490f0  buffer
(gdb) run
Starting program: /home/passwd/Desktop/assembly/AT&T/datatypes

Breakpoint 1, _start () at datatypes.s:28
28                      movl $20, %eax
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbffff3b0       0xbffff3b0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048074        0x8048074 <_start>
eflags         0x202    [ IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
(gdb) █
```

When the program which the breakpoint is set on is started with the command run, the instant values of the addresses where the variables are in memory and of the registers are shown in the output as above. When the code is run with the command s, the value 20 is put into the eax register as shown below:

“s” **?**

```
31                      movl $6, integer
(gdb) info registers
eax            0x14     20
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbffff3b0       0xbffff3b0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048079        0x8048079 <_start+5>
eflags         0x202    [ IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

16.1+1.4 =20
Veri direk eax içerisine yazıldı

When looking inside the memory region, it is seen that the integer is the first assigned value and the value 6 is assigned instead of it after the code piece has worked.

```
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x080490bc  string
0x080490cc  integer
0x080490d0  array
0x080490e4  short
0x080490e6  bayt
0x080490f0  buffer
(gdb) x/1bd 0x080490cc
0x80490cc <integer>:    32
(gdb) s
34                      movl bayt, %eax
(gdb) x/1bd 0x080490cc
0x80490cc <integer>:    6
(gdb) print integer
$2 = 6
```

When using the commands print and x (examine), these two commands can be mixed. The following experiment can be done to elucidate this confusion:

```
(gdb) info registers eax
eax            0x14     20
(gdb) print $eax
$4 = 20
```

```
(gdb) x/1db $eax
0x14:   Cannot access memory at address 0x14
(gdb)
```

The print command prints the value in the eax onto the screen, and the x command, the data kept in the memory area indicated by the address value in the eax onto the screen. The GDB has printed an error since the value 20 does not have any address.

```
(gdb) s
37                      movb $2, %al
(gdb) info registers eax
eax            0xb      11
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x080490bc  string
0x080490cc  integer
0x080490d0  array
0x080490e4  short
0x080490e6  bayt
0x080490f0  buffer
(gdb) print $eax
$5 = 11
(gdb) x/1bd $eax
0xb:    Cannot access memory at address 0xb
(gdb) x/1bd 0x080490e6
0x80490e6 <bayt>:       11
(gdb) 
```

As can be seen, the value 11 in the memory area addressed by byte is assigned to the eax register. In the next command, the value 2 will be assigned to the al, this value will be written instead of the value 11 in the memory region indicated by byte as shown below:

```
(gdb) s
38                        movb %al, bayt
(gdb) info registers al
al             0x2      2
(gdb) s
41                        movl $bayt, %edi
(gdb) print bayt
$6 = 2
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x080490bc  string
0x080490cc  integer
0x080490d0  array
0x080490e4  short
0x080490e6  bayt
0x080490f0  buffer
(gdb) x/1db 0x080490e6
0x80490e6 <bayt>:         2
(gdb) print &bayt
$8 = (<data variable, no debug info> *) 0x80490e6
(gdb) 
```

The address of the byte variable is shown above: &bayt. The value 2 is assigned to the variable byte. In the code line movl $ byte,% edi, the address of the variable byte is put into the register edi.

The value in the register edi becomes the value of the variable byte. When looking at the place pointed by the register edi with the command x, the value 2 is seen. When the value kept in the $ edi is displayed with the command print, the decimal address of the variable byte is also printed on the screen as shown below:

```
(gdb) s
42                        movl $12, (%edi)
(gdb) info registers edi
edi            0x80490e6        134516966
(gdb) info variables bayt
All variables matching regular expression "bayt":

Non-debugging symbols:
0x080490e6  bayt
(gdb) x/1db $edi
0x80490e6 <bayt>:        2
(gdb) print $edi
$10 = 134516966
(gdb) print &bayt
$11 = (<data variable, no debug info> *) 0x80490e6
```

After this step, the value 12 will be written into the memory region indicated by the (%edi) edi. Instead of this value, which was 2 previously, the value 12 is assigned as follows:

```
(gdb) s
45                      movl $0, %ecx
(gdb) info registers
eax            0x2      2
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbffff3b0       0xbffff3b0
ebp            0x0      0x0
esi            0x0      0
edi            0x80490e6        134516966
eip            0x804809a        0x804809a <_start+38>
eflags         0x202    [ IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
(gdb) info variables bayt
All variables matching regular expression "bayt":

Non-debugging symbols:
0x080490e6  bayt
(gdb) x/1db 0x080490e6
0x80490e6 <bayt>:       12
(gdb) print $edi
$13 = 134516966
(gdb) print bayt
$14 = 12
(gdb) █
```

In the last step, the value in the array is changed as follows:

```
(gdb) s
46                      movl $2, %edi
(gdb) s
47                      movl $13, array(%ecx, %edi, 4)
(gdb) s
49                      nop
(gdb) info variables array
All variables matching regular expression "array":

Non-debugging symbols:
0x080490d0  array
(gdb) x/20db 0x080490d0
0x80490d0 <array>:      1       0       0       0       2       0       0       0
0x80490d8 <array+8>:    13      0       0       0       4       0       0       0
0x80490e0 <array+16>:   5       0       0       0
(gdb) info registers ecx
ecx            0x0      0
(gdb) info registers edi
edi            0x2      2
(gdb) print $edi
$15 = 2
(gdb) print $ecx
$16 = 0
(gdb)
```

The value 13 is assigned instead of the value 3. In the application above, the assignments made with the command mov in Linux 32-bit assembly are exemplified.

LAB 6: Assembly Character Operations

In this example application, the character operations will be shown and their analysis will be performed on GDB. In the first step, the sample program is written as follows. In the program written, ESI shows the source character, and EDI, the target. As will be remembered from the first chapter, the register esi is used for reading, and the register edi, for writing. The written sample program is shown below:

```
.data
        string1:

                .asciz "Linux Assembly 32 bit"
        string2:
                .asciz "reverse enginnering"
.bss

        .lcomm var1, 100
        .lcomm var2, 100
        .lcomm var3, 100
.text
        .global _start

        _start:
                nop

                #string 1 adresi esi içerisine taşınır
                movl $string1, %esi
                #string 2 adresi edi içerisine taşınır
                movl $string2, %edi
                # tek byte kopyalamak için:
                movsb
                # 16 bitlik veri kopyalamak için:
                movsw
                # 32 bitlik veri kopyalamak için:
                movsl

                # Direction Flag değerlerini temizleme ayar yapma

                std # set direction flag
                cld # clear direction flag

                # Rep kullanarak string kopyalama

                movl $string1, %esi
                movl $var3, %edi
                movl $21, %ecx # kopyalanacak karakter uzunlugu
```

```
movl $21, %ecx # kopyalanacak karakter uzunlugu
cld # direction flag sıfırlanır
rep movsb
std

# hafizadan eax yazmacina karakter kopyalama

cld
# load effective address ( leal ):
leal string1, %esi #leal direk kopyalama yapar
lodsb # 1 byte kopyalama
movb $0, %al
dec %esi # esi değeri 1 azaltilir
lodsw # 16 bit kopyalama
movw $0, %ax # ax içerisine 0 yazildi

subl $1, %esi # esi değeri orjinal karakteri gosteriyor
lodsl

# eax yazmacindan hafizaya karakter depolama

leal var2, %edi
stosb
stosw

# karakter karsilaştirma

cld
leal string1, %esi
leal string2, %edi
cmpsb # compare string byte iki karakter dizisini karsilastirir

# program sonlandirici

movl $1, %eax
movl $0, %ebx
int $0x80
```

```
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ as -gstabs string.s -o string.o
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ ld string.o -o string
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ ./string
passwd@vulnubuntu:~/Desktop/assembly/AT&T$
```

The program will be analyzed with GDB by setting a breakpoint in the _start function. After the variables are defined on .data and .bss, the operations performed in _start will be explained step by step as follows:

As the program will read from the comment lines, after the $ sign is put at the beginning of the variables and their addresses are loaded into the esi and edi, 1 byte data is copied from the source to the destination with the movsb command, and 16 bit data, from the source to the destination with the movsl command, and 32 bit data, from the source to the destination with the movsw command. The GDB analysis of the procedure described so far has beens carried out as follows:

```
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ ./string
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ gdb ./string
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
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 "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/passwd/Desktop/assembly/AT&T/string...done.
(gdb) break _start
Breakpoint 1 at 0x8048074: file string.s, line 16.
(gdb) run
Starting program: /home/passwd/Desktop/assembly/AT&T/string

Breakpoint 1, _start () at string.s:16
16                      nop
(gdb)
```

```
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ gdb ./string
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
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 "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/passwd/Desktop/assembly/AT&T/string...done.
(gdb) break _start
Breakpoint 1 at 0x8048074: file string.s, line 16.
(gdb) run
Starting program: /home/passwd/Desktop/assembly/AT&T/string

Breakpoint 1, _start () at string.s:16
16                      nop
(gdb) info variables
All defined variables:

Non-debugging symbols:
0x080490d0  string1
0x080490e6  string2
0x08049100  var1
0x08049168  var2
0x080491d0  var3
(gdb) info registers $esi
esi            0x0      0
(gdb) info registers $edi
edi            0x0      0
(gdb)
```

The variables are displayed and their addresses are printed on the screen. The insides of the registers esi and edi are empty as seen. The first command to run is the nop command. When the program flow is continued after this command is executed, the initial addresses where the string1 characters are located in memory will be written into the esi, and the initial addresses where the string2 characters are located in memory, into the edi. In the last step, the copying will be done with the mov commands:

```
(gdb) s
19                      movl $string1, %esi
(gdb) s
21                      movl $string2, %edi
(gdb) info registers $esi
esi            0x80490d0        134516944
(gdb) s
23                      movsb
(gdb) info registers $edi
edi            0x80490e6        134516966
(gdb) print /x &string1
$3 = 0x80490d0
(gdb) print /x &string2
$4 = 0x80490e6
(gdb) x/21sb &string1
0x80490d0 <string1>:     "Linux Assembly 32 bit"
0x80490e6 <string2>:     "reverse enginnering"
0x80490fa:        ""
```

string1 değerinin adresi esi
string2 değerinin adresi ediiçerisinde & işareti ile adres elde edilir.

When the mov commands are executed, the string1 characters are copied onto the string2 as follows:

```
(gdb) s
25                      movsw
(gdb) x/1sb &string2
0x80490e6 <string2>:     "Leverse enginnering"
(gdb) s
27                      movsl
(gdb) x/2sb &string2
0x80490e6 <string2>:     "Linerse enginnering"
0x80490fa:        ""
(gdb) s
31                      std # set direction flag
(gdb) x/4sb &string2
0x80490e6 <string2>:     "Linux A enginnering"
0x80490fa:        ""
0x80490fb:        ""
0x80490fc:        ""
```

First 1 byte, then 16 bit (2 byte) and then 32 bit (4 byte) data are copied onto the string2. The edi value changes with each copy operation. The edi register indicates where the last copied character is written. The output below shows the final value of the edi register:

```
(gdb) x/4sb &string2
0x80490e6 <string2>:     "Linux A enginnering"
0x80490fa:      ""
0x80490fb:      ""
0x80490fc:      ""
(gdb) print /x $edi
$7 = 0x80490ed
(gdb) x/1cb 0x80490ed
0x80490ed <string2+7>:  32 ' '
(gdb) x/1cb 0x80490ec
0x80490ec <string2+6>:  65 'A'
(gdb) x/1cb 0x80490eb
0x80490eb <string2+5>:  32 ' '
(gdb) x/1cb 0x80490ea
0x80490ea <string2+4>:  120 'x'
(gdb) x/1cb 0x80490e9
0x80490e9 <string2+3>:  117 'u'
(gdb) print /x $edi-1
$8 = 0x80490ec
```

DF is the flag that decides whether to increase or decrease the esi or edi values after each movs command. The gdb only shows the set flag values. For example, it is seen that the only if (interrupt flag) is set in the program flow in the output below:

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbffff3b0       0xbffff3b0
ebp            0x0      0x0
esi            0x80490d7        134516951
edi            0x80490ed        134516973
eip            0x8048083        0x8048083 <_start+15>
eflags         0x202    [ IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
(gdb) █
```

```
(gdb) s
36                      movl $string1, %esi
(gdb) print /x $esi
$9 = 0x80490d7
(gdb) print /x &string1
$10 = 0x80490d0
(gdb) s
37                      movl $var3, %edi
(gdb) print /x $esi
$11 = 0x80490d0
(gdb) print /x &var3
$12 = 0x80491d0
(gdb) s
38                      movl $21, %ecx # kopyalanacak karakter uzunlugu
(gdb) print /x $edi
$13 = 0x80491d0
(gdb) s
39                      cld # direction flag sıfırlanır
(gdb) info registers $eflags
eflags         0x202    [ IF ]
(gdb) s
40                      rep movsb
(gdb) s
41                      std
(gdb) x/1cb &var3
0x80491d0 <var3>:       76 'L'
(gdb) info registers $eflags
eflags         0x202    [ IF ]
(gdb) s
45                      cld
(gdb) info registers $eflags
eflags         0x602    [ IF DF ]
(gdb) s
47                      leal string1, %esi #leal direk kopyalama yapar
(gdb) info registers $eflags
eflags         0x202    [ IF ]
(gdb)
```

The command rep is a command that repeats the character operations as long as the ecx value is greater than 0. In the first block of codes, the esi and edi values spontaneously increased with every mov operation. Therefore, after the first byte is copied, the movement process is carried out by getting the remaining values to continue from the address reached in the previous process. When the df is set in the operations with the rep, after the operation, the reset process is done with the cld command.

```
(gdb) x/21cb &var3
0x80491d0 <var3>:       76 'L'  105 'i' 110 'n' 117 'u' 120 'x' 32 ' '  65 'A'1
15 's'
0x80491d8 <var3+8>:     115 's' 101 'e' 109 'm' 98 'b'  108 'l' 121 'y' 32 ' '5
1 '3'
0x80491e0 <var3+16>:    50 '2'  32 ' '  98 'b'  105 'i' 116 't'
(gdb) info registers $eflags
eflags         0x202    [ IF ]
```

The GDB analysis of the progressive commands is done as shown below:

```
(gdb) x/21cb $var3
Value can't be converted to integer.
(gdb) x/21cb &var3
0x80491d0 <var3>:       76 'L'  105 'i' 110 'n' 117 'u' 120 'x' 32 ' '  65 'A'1
15 's'
0x80491d8 <var3+8>:     115 's' 101 'e' 109 'm' 98 'b'  108 'l' 121 'y' 32 ' '5
1 '3'
0x80491e0 <var3+16>:    50 '2'  32 ' '  98 'b'  105 'i' 116 't'
(gdb) info registers $eflags
eflags         0x202    [ IF ]
(gdb)
(gdb) s
48                      lodsb # 1 byte kopyalama
(gdb) info registers $eflags
eflags         0x202    [ IF ]
(gdb) s
49                      movb $0, %al
(gdb) s
50                      dec %esi # esi değeri 1 azaltilir
(gdb) info registers $eflags
eflags         0x202    [ IF ]
(gdb) info registers $esi
esi            0x80490d1        134516945
(gdb) s
51                      lodsw # 16 bit kopyalama
(gdb) info registers $esi
esi            0x80490d0        134516944
(gdb) s
52                      movw $0, %ax # ax içerisine 0 yazildi
(gdb) info registers $ax
ax             0x694c   26956
(gdb) s
54                      subl $1, %esi # esi değeri orjinal karakteri gosteriyor
(gdb) info registers $ax
ax             0x0      0
(gdb) info registers $esi
esi            0x80490d2        134516946
(gdb) s
55                      lodsl
```

Using the LODSx command, the loading is done into the eax register. The source character address is indicated by the esi register. The lodsb moves 8-bit data from the memory into the al register, and the lodsw, 16 bit-data from the memoryin to the ax register, and the lodsl, 32-bit data from memory into the eax register. With the command leal (load effective address), the address of the string1 variable is written directly into the esi register. Using the lodsb command, the string1 variable's 1-byte (8-bit) data since the address at which it starts to sit in the memory with the esi is shown. By putting 0 into the al, the esi value is reduced by 1. The reason for this reduction is that the esi value has increased by 1 after the movement, and the code writer wants the register esi to return to its previous value. The 16-bit data is displayed by the esi using the lodsw command, and the 32-bit data, by the esi in order for the movement operation to be done using the lodsl command. With the sub command, the esi register value is reduced by 1 again and is returned to its previous value.

leal var2, %edi

stosb

stosw

In the code line written above, the address var2 is displayed with the edi, and then the relevant value is moved into the location shown in the memory with the edi with the stos command: If the stosb instruction is used, the value is moved with the al, and if the stosw instruction is used the value is moved with the ax, and if the stosl instruction is used the value is moved with the eax.

```
(gdb) s
55                      lodsl
(gdb) info registers $esi
esi            0x80490d1        134516945
(gdb) x/5cb 0x80490d1
0x80490d1 <string1+1>:  105 'i' 110 'n' 117 'u' 120 'x' 32 ' '
(gdb) s
59                      leal var2, %edi
(gdb) x/15cb 0x80490d1
0x80490d1 <string1+1>:  105 'i' 110 'n' 117 'u' 120 'x' 32 ' '  65 'A'  115 's'
115 's'
0x80490d9 <string1+9>:  101 'e' 109 'm' 98 'b'  108 'l' 121 'y' 32 ' '  51 '3'
(gdb) s
60                      stosb
(gdb) info registers $edi
edi            0x8049168        134517096
(gdb) print /x &var2
$14 = 0x8049168
(gdb) x/10sb &var2
0x8049168 <var2>:        ""
0x8049169 <var2+1>:      ""
0x804916a <var2+2>:      ""
0x804916b <var2+3>:      ""
0x804916c <var2+4>:      ""
0x804916d <var2+5>:      ""
0x804916e <var2+6>:      ""
0x804916f <var2+7>:      ""
0x8049170 <var2+8>:      ""
0x8049171 <var2+9>:      ""
(gdb) s
61                      stosw
(gdb) x/10sb &var2
0x8049168 <var2>:        "i"
0x804916a <var2+2>:      ""
0x804916b <var2+3>:      ""
0x804916c <var2+4>:      ""
0x804916d <var2+5>:      ""
```

In the last piece of the code, the first byte values of the two character strings indicated by the esi and edi were compared with the cmpsb command. The comparison is done like subtraction. If the result is zero, the data compared are the same. Therefore, zf (zero flag) will be set as 1.

```
68                      cmpsb # compare string byte iki karakter dizisini karsi
lastirir
(gdb) info registers $edi
edi            0x80490e6        134516966
(gdb) print /x &string2
$15 = 0x80490e6
(gdb) x/20cb &string2
0x80490e6 <string2>:    76 'L'  105 'i' 110 'n' 117 'u' 120 'x' 32 ' '  65 'A'3
2 ' '
0x80490ee <string2+8>:  101 'e' 110 'n' 103 'g' 105 'i' 110 'n' 110 'n' 101 'e'
114 'r'
0x80490f6 <string2+16>: 105 'i' 110 'n' 103 'g' 0 '\000'
(gdb) s
72                      movl $1, %eax
(gdb) info registers
eax            0x78756e69       2020961897
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbffff3b0       0xbffff3b0
ebp            0x0      0x0
esi            0x80490d1        134516945
edi            0x80490e7        134516967
eip            0x80480c4        0x80480c4 <_start+80>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
(gdb) x/20cb &string1
0x80490d0 <string1>:    76 'L'  105 'i' 110 'n' 117 'u' 120 'x' 32 ' '  65 'A'1
15 's'
0x80490d8 <string1+8>:  115 's' 101 'e' 109 'm' 98 'b'  108 'l' 121 'y' 32 ' '5
1 '3'
```

LAB 7: Assembly JMP Command, LOOP Creation And EXECVELAB 7

The JMP command functions the same as the command GOTO in the C programming language. In other words, by showing the place to go in the program, the program flow continues over the shown place with the JMP command. The JMP commands, conditionally and unconditional, can be used within the program. The JMP working logic is examined in the program written below:

```
.section .data
        file_to_run:
                .asciz       "/bin/sh"
        message:
                .asciz       "Simple JMP Program JMP to EXECVE SH3113R"

.section .text

        .globl _start

_start:
        movl $4, %eax
        movl $1, %ebx
        leal message, %ecx
        movl $40, %edx
        int $0x80
        jmp execve

        movl $1, %eax
        movl $0, %ebx
        int $0x80

        execve:
                pushl %ebp
                movl %esp, %ebp
                subl $0x8, %esp        # array of two pointers. array[0] = fil
e_to_run  array[1] = 0
                movl $file_to_run, %edi
                movl %edi, -0x8(%ebp)
                movl $0, -0x4(%ebp)
                movl $11, %eax                      # sys_execve
                movl $file_to_run, %ebx              # file to execute
                leal -8(%ebp), %ecx                 # command line parameters
                movl $0, %edx                       # environment block
                int  $0x80
                leave
                ret
```

When the program is examined, it is seen that global and assigned variables are defined on the .data segment side. The character string starting from the address indicated by the message variable in the _start is printed on the screen. With the JMP command, the outgoing code set was skipped and jumped to the execve tag. Firstly, a prelog operation is performed in the execve and a 8- byte section is reserved in the stack with the SUB command. This is because two 4 bytes of data must be put in the stack. The representation of the Execve program written in the C programming language is as follows:

```
char *data[2];

data[0] = "/bin/sh";

data[1] = NULL;

execve(data[0], data, NULL);
```

In the function, it is necessary to give the values "/bin/sh" and NULL, ie 0 through the array. Therefore, a 8-byte space is opened in the stack. System call number for execve is specified as 11 in the table in which the system call numbers are indicated:

<u>11. sys_execve</u>

Syntax: int sys_execve(struct pt_regs regs)

Source: arch/i386/kernel/process.c

Action: execute program

After the string "/bin/sh" in the variable file_to_run is displayed with the regester edi, this value is placed 8 bytes below the address indicated by the EBP. The value 0 is also placed 4 bytes below the address indicated by the EBP. After 11 is assigned to the eax as the system call number, by putting the start address of the address where the “/bin/sh” characters are located in the memory in the ebx, the command line parameters are taken with the ecx. The program was run by being passed through the operations of being assembled and linked as below:

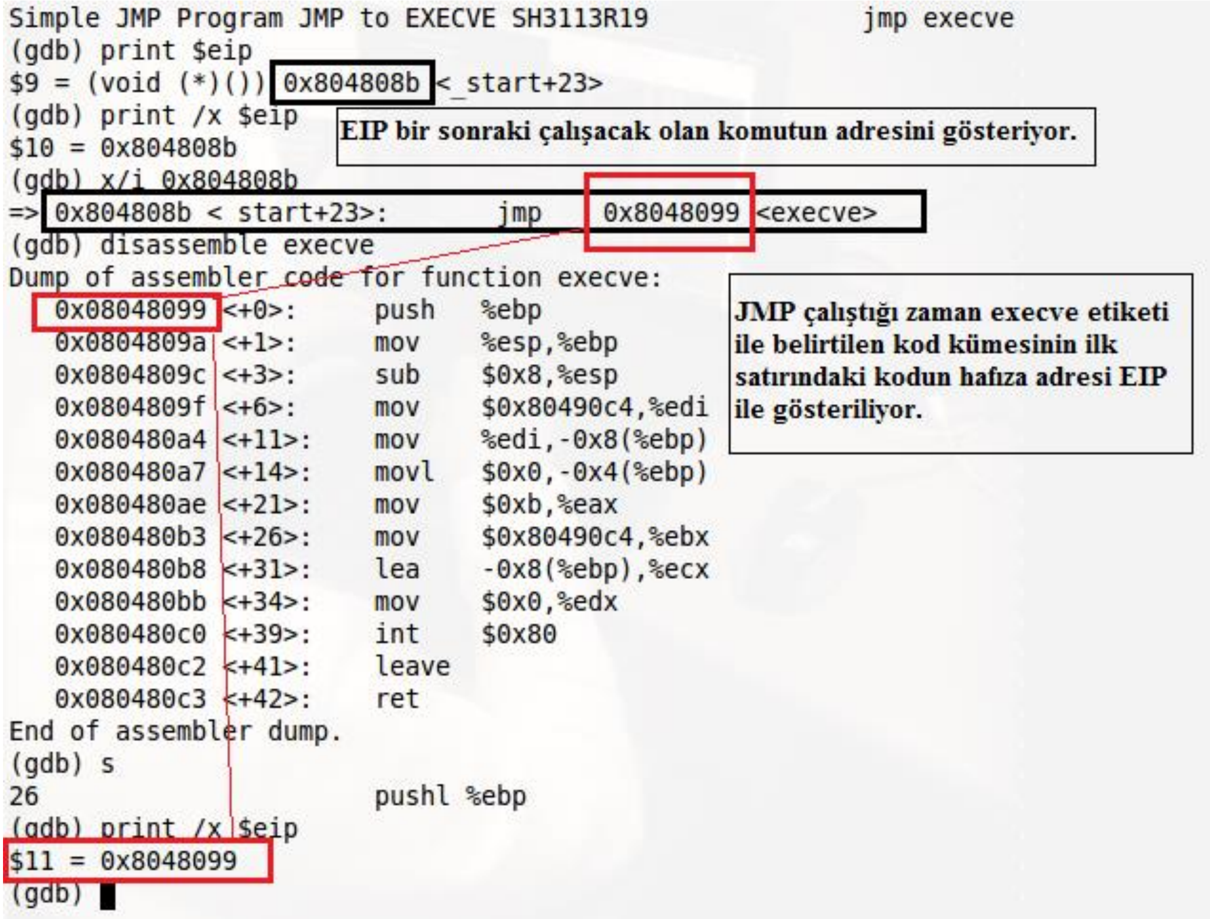
```
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ as -ggstabs execve.s -o execve.o
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ as -ggstabs execve.s -o execve.o
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ ld execve.o -o execve
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ ./execve
Simple JMP Program JMP to EXECVE SH3113R$ ls
datatypes  datatypes.o  datatypes.s  execve  execve.o  execve.s  hello.exe  hel
$ pwd
```

The gdb analysis of the program is carried out as follows for a better understanding of the subject. Any function can be disassembled using the disassemble command. Below, the execve has been disassembled. Register and variable analyses have been performed in the _start designated progress as a breakpoint.

```
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ gdb ./execve
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
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 "i686-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/passwd/Desktop/assembly/AT&T/execve...done.
(gdb) break _start
Breakpoint 1 at 0x8048074: file execve.s, line 14.
(gdb) run
Starting program: /home/passwd/Desktop/assembly/AT&T/execve

Breakpoint 1, _start () at execve.s:14
14              movl $4, %eax
(gdb) disassemble execve
Dump of assembler code for function execve:
   0x08048099 <+0>:     push   %ebp
   0x0804809a <+1>:     mov    %esp,%ebp
   0x0804809c <+3>:     sub    $0x8,%esp
   0x0804809f <+6>:     mov    $0x80490c4,%edi
   0x080480a4 <+11>:    mov    %edi,-0x8(%ebp)
   0x080480a7 <+14>:    movl   $0x0,-0x4(%ebp)
   0x080480ae <+21>:    mov    $0xb,%eax
   0x080480b3 <+26>:    mov    $0x80490c4,%ebx
   0x080480b8 <+31>:    lea    -0x8(%ebp),%ecx
   0x080480bb <+34>:    mov    $0x0,%edx
   0x080480c0 <+39>:    int    $0x80
```

In the first steps, the message is printed on the screen. When the jmp command will run, the eip status is examined as follows. The memory address of the command shown with jmp was loaded into the eip when the jmp command is executed and the program flow continues under the execve tag. The exit operation was skipped not working due to the jmp jumping.



When the execve starts, the variable addresses are placed in the registers as below and the characters to be used in the stack and the value 0 are assigned to the relevant places for the execve function to work. As can be seen in the analysis, after putting the address of the file_to_run variable in the edi, this address value is placed in the memory region indicated by the ebp-8 address. Then, the value 0 is placed in the memory region indicated by the address of ebp-4. Calling the function execve with the system call number 11, the address indicating /bin/sh is placed in the ebx, then the command line parameters are taken by the ecx, and the program gave the shell as follows:

```
Breakpoint 1 at 0x804809f: file execve.s, line 29.
(gdb) run
Starting program: /home/passwd/Desktop/assembly/AT&T/execve
Simple JMP Program JMP to EXECVE SH3113R
Breakpoint 1, execve () at execve.s:29
29                      movl $file_to_run, %edi
(gdb) print $edi
$1 = 0     EDI içerisindeki değer 0
(gdb) print /x &file_to_run
$2 = 0x80490c4   file_to_run değişkeninin adresi
(gdb) x/1sb  0x80490c4
0x80490c4 <file_to_run>:         "/bin/sh"  değişken adresi ile gösterilen hafıza
                                            bölgesi içerisindeki string
(gdb) s
30                      movl %edi, -0x8(%ebp)
(gdb) print /x $edi
$4 = 0x80490c4   EDI içerisine değişken adresi yüklendi
(gdb) s
31                      movl $0, -0x4(%ebp)
(gdb) s
32                      movl $11, %eax                     # sys_execve
(gdb) print /x $ebp-8
$5 = 0xbffff3a4   EBP adresinin 8 byte altındaki adres değeri
(gdb) print /x $ebp-4
$6 = 0xbffff3a8   EBP adresinin 4 byte altındaki adres değeri
(gdb) x/4db  0xbffff3a8   EBP-4 ile gösterilen adres içindeki değerler =>0
0xbffff3a8:     0       0       0       0
(gdb) x/1xw  0xbffff3a4   EBP-8 ile gösterilen adres içinde file_to_run değişkeninin
0xbffff3a4:     0x080490c4   adresi konuldu
(gdb) s
33                      movl $file_to_run, %ebx            # file to execute

(gdb) print /x $ebx
$7 = 0x1
(gdb) s
34                      leal -8(%ebp), %ecx                # command line para
meters
(gdb) print /x $ebx
$8 = 0x80490c4
(gdb) s
35                      movl $0, %edx                      # environment block
(gdb) print /x $ecx
$9 = 0xbffff3a4
(gdb) x/1xb 0xbffff3a4
0xbffff3a4:     0xc4
(gdb) x/4xb 0xbffff3a4
0xbffff3a4:     0xc4    0x90    0x04    0x08
(gdb) x/1xw 0xbffff3a4
0xbffff3a4:     0x080490c4
(gdb) x/1sb 0x080490c4
0x80490c4 <file_to_run>:         "/bin/sh"
(gdb) s
36                      int  $0x80
(gdb) print /x $edx
$10 = 0x0
(gdb) s
process 5117 is executing new program: /bin/dash
```

The command call is a command used to call functions. With the command ret, the function outgoing address is displayed and, the program flow continues from where it left off via the eip. The command ret is similar to the command return in the C programming language. Each time the command call is run, the command ret must also be run within the next set of commands. The address of the first command to run is pushed into the stack in order for the program flow to be able to continue after a function which was called with the command call has completed to work and returned to the main program. After the function has finished working, this address is pointed out from the inside of the eip with the command ret and poped from the inside of the stack, and the program flow continues. In order to demonstrate the use of the command call, the program which has been produced by a small change made in the previous program will be analyzed with the gdb as shown below:

```
_start:

        movl $4, %eax
        movl $1, %ebx
        leal message, %ecx
        movl $40, %edx
        int $0x80
        call sayyou
        nop
        nop
        exit:
                movl $1, %eax
                movl $0, %ebx
                int $0x80

        sayyou:
                loop1:
                        movl $3, %eax
                        movl $0, %ebx
                        leal buffer, %ecx
                        movl $1, %edx
                        int $0x80
                        cmp $0, %eax
                        jle exit

                        movl %eax, %edx
                        movl $buffer, %ecx
                        movl $4, %eax
                        movl $1, %ebx
                        int $0x80


                        jmp loop1
```

The code written was examined on the gdb as follows:

```
Breakpoint 1, _start () at call.s:20
20              int $0x80
(gdb) info registers $eip
eip            0x8048089        0x8048089 <_start+21>
(gdb) x/i  0x8048089
=> 0x8048089 <_start+21>:       int    $0x80
(gdb) x/16xw $esp
0xbffff3b0:     0x00000001      0xbffff52c      0x00000000      0xbffff554
0xbffff3c0:     0xbffff567      0xbffff592      0xbffff5a2      0xbffff5ad
0xbffff3d0:     0xbffff5fe      0xbffff610      0xbffff63a      0xbffff646
0xbffff3e0:     0xbffffb67      0xbffffba1      0xbffffbd5      0xbffffbfb
(gdb) s
Simple JMP Program JMP to EXECVE SH3113R21             call sayyou
(gdb) s
31                              movl $3, %eax
(gdb) x/16xw $esp
0xbffff3ac:     0x08048090      0x00000001      0xbffff52c      0x00000000
0xbffff3bc:     0xbffff554      0xbffff567      0xbffff592      0xbffff5a2
0xbffff3cc:     0xbffff5ad      0xbffff5fe      0xbffff610      0xbffff63a
0xbffff3dc:     0xbffff646      0xbffffb67      0xbffffba1      0xbffffbd5
(gdb) disassemble sayyou
Dump of assembler code for function sayyou:
=> 0x0804809e <+0>:     mov    $0x3,%eax
   0x080480a3 <+5>:     mov    $0x0,%ebx
   0x080480a8 <+10>:    lea    0x8049110,%ecx
   0x080480ae <+16>:    mov    $0x1,%edx
   0x080480b3 <+21>:    int    $0x80
   0x080480b5 <+23>:    cmp    $0x0,%eax
   0x080480b8 <+26>:    jle    0x8048092 <exit>
   0x080480ba <+28>:    mov    %eax,%edx
   0x080480bc <+30>:    mov    $0x8049110,%ecx
   0x080480c1 <+35>:    mov    $0x4,%eax
   0x080480c6 <+40>:    mov    $0x1,%ebx
   0x080480cb <+45>:    int    $0x80
   0x080480cd <+47>:    jmp    0x804809e <sayyou>
   0x080480cf <+49>:    leave
   0x080480d0 <+50>:    ret
End of assembler dump.
```

The eip shows the address of the command interrupt to run next. The values in the stack are checked as above. After running the command call, the program flow is directed into the sayyou. When the stack is checked after this point, it is seen that the address 0x0804890 has been pushed into the stack. As explained in the theoretical section, the address of the command nop after the code line where the function is called is pushed into the stack as the return address so that the program flow can continue after the function codes have completed their functions. In the output below, the address of the nop command is seen with the disassembler. As seen in the output below, the address of the nop command is gained with the disassembler:

```
End of assembler dump.
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x08048074 <+0>:     mov    $0x4,%eax
   0x08048079 <+5>:     mov    $0x1,%ebx
   0x0804807e <+10>:    lea    0x80490dc,%ecx
   0x08048084 <+16>:    mov    $0x28,%edx
   0x08048089 <+21>:    int    $0x80
   0x0804808b <+23>:    call   0x804809e <sayyou>
   0x08048090 <+28>:    nop
   0x08048091 <+29>:    nop
End of assembler dump.
(gdb)
```

The command loop decreases the value of the register ecx by 1 each time the loop rotates. The command jz jumps if it is zero-flag-set. The command jnz jumps if it is not zero-flag-set. Similarly, if the command loopz is zero-flag-set, the loop returns while the loop continues to rotate as long as the command loopnz is not zero-flag-set. The use of the command loop was shown in the example above. Its format is as follows:

Command set

movl $20, %ecx

Lupla:

  Command set

  LOOP lupla

The loop will return 20 times as 20 is loaded to the ecx in the above format. In the program shown below, the character string defined in the data section is printed on the screen 12 times. The change of values in the register ecx is shown with the gdb as follows. The program output is as shown:

```
as -ggstabs loop.s -o loop.o
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ ld loop.o -o loop
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ ./loop
101010101neo1000010ssh101010101neo1000010ssh101010101neo1000010ssh101010101neo10
00010ssh101010101neo1000010ssh101010101neo1000010ssh101010101neo1000010ssh101010
101neo1000010ssh101010101neo1000010ssh101010101neo1000010ssh101010101neo1000010s
sh101010101neo1000010sshpasswd@vulnubuntu:~/Desktop/assembly/AT&T$
```

Program codes:

```
.data
        karakter:
                .asciz "101010101neo1000010ssh"
.text
        .global _start

        _start:

                call loopy
                exit:

                        movl $1, %eax
                        movl $0, %ebx
                        int $0x80

                loopy:
                        movl $12, %ecx

                        loopx:
                                pushl %ecx
                                movl $4, %eax
                                movl $0, %ebx
                                leal karakter, %ecx
                                movl $23, %edx
                                int  $0x80
                                popl %ecx
                        loop loopx
                        cmp %eax, %eax
                        jz exit
```

For the analysis with the gdb, a break point has been placed at the point where the command pushl% ecx is located. Each time the loop rotates, the ecx value is reduced by 1.

```
Breakpoint 1, loopx () at loop.s:20
20                                      pushl %ecx
(gdb) info registers ecx
ecx            0xc      12
(gdb) s
21                                      movl $4, %eax
(gdb) s
22                                      movl $0, %ebx
(gdb) s
23                                      leal karakter, %ecx
(gdb) s
24                                      movl $23, %edx
(gdb) s
25                                      int  $0x80
(gdb) s
101010101neo1000010ssh26                                        popl %ecx
(gdb) s
27                              loop loopx
(gdb) info registers ecx
ecx            0xc      12
(gdb) s

Breakpoint 1, loopx () at loop.s:20
20                                      pushl %ecx
(gdb) info registers ecx
ecx            0xb      11
```

As can be seen in the output, the eip value is decreased by 1 each time the loop rotates.

LAB 8: Creating a Linux Assembly Function

In this lab work, creating a function in Linux assembly software will be mentioned. The syntax is shown below:

```
.type fonksiyon ismi, @fonksiyon

fonksiyon ismi:

        Komut kümesi

        ret
```

The written function is called with the command call. Based on the above structure, the sample function code is written as follows:

```
.data

     warning:

          .asciz "Attack detected execve coming"

     file_to_run:

          .asciz      "/bin/sh"

.text

     .global _start

     _start:

          .type shell, @function

          .type print, @function

   print:

               movl $4, %eax

               movl $1, %ebx

               movl $30, %edx

               leal warning, %ecx
```

```
        int $0x80
        call shell
shell:
        pushl %ebp
        movl %esp, %ebp
        subl $0x8, %esp      # array of two pointers. array[0] = file_to_run  array[1]
        movl $file_to_run, %edi
        movl %edi, -0x8(%ebp)
        movl $0, -0x4(%ebp)
          movl $11, %eax              # sys_execve
        movl $file_to_run, %ebx     # file to execute
        leal -8(%ebp), %ecx         # command line parameters
        movl $0, %edx               # environment block
        int  $0x80
        leave
        ret
    call print
```

LAB 9: Shellcode with EXECVE

In the previous lab studies, the execve was used. In this lab study, the execve will be written using the assembly and C programming languages. The code written will be converted into the machine codes with the tool objdump, and the shellcode will be obtained. In the exploitation process, the code to be injected into the memory must be understood by the RAM. In order for the code to be understood and interpreted by RAM, the code in the format being converted into machine code must be sent to the RAM. For this reason, the shellcode is used. Shellcode is a piece of code used to run the commands in the target shell consisting of the hex characters written in machine language. The execve structure is displayed as follows with the man execve.

```
       execve - execute program

SYNOPSIS
       #include <unistd.h>
                      çalıştırılacak dosya       dosya isminin adresi
       int execve(const char *filename, char *const argv[],
                  char *const envp[]);    Karakterler NULL ile bitirilmeli.

DESCRIPTION
       execve() executes the program pointed to by filename.  filename must be
       either a binary executable, or a script starting with  a  line  of  the
       form:

           #! interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv  is  an  array  of argument strings passed to the new program.  By
       convention, the first of these  strings  should  contain  the  filename
       associated  with the file being executed.  envp is an array of strings,
       conventionally of the form key=value, which are passed  as  environment
       to  the  new  program.  Both argv and envp must be terminated by a NULL
       pointer.  The argument vector and environment can be  accessed  by  the
       called program's main function, when it is defined as:

           int main(int argc, char *argv[], char *envp[])
```

Based on the spelling format on the manual page, the code is written as follows:

```c
#include <stdio.h>
#include <stdlib.h>
main()
{
        char *args[2];
        args[0]="/bin/sh";
        args[1]=NULL;
        execve(args[0], args, NULL);
        exit(0);
}
```

The array args is a pointer array that holds the memory address of the other arrays and has two elements. The first element is the /bin/sh, which is the file that allows the commands to be run on the Linux operating systems. Args [1] is shown as NULL due to the rule that each string must end with NULL. The program is written according to the format by putting file (/bin/sh), file address and NULL in execve. The program was compiled with the gcc using the following command:

gcc -ggdb shell.c -mpreferred-stack-boundary=2 -static -o shell.exe

With the parameter -m**preferred-stack-boundary=2**, it is provided that the sections each of which includes 2 bytes instead of 4 bytes in the values placed in the STACK are reserved.

Static libraries are linked with the parameter –static. Connection can not established with the shared libraries.

```
passwd@vulnubuntu:~/Desktop/assembly/c$ ./shell.exe
$ tty
/dev/pts/3
$
```

When the written code is disassembled with the gdb, the main function is broken down as follows:

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08048ee0 <+0>:     push   %ebp
   0x08048ee1 <+1>:     mov    %esp,%ebp
   0x08048ee3 <+3>:     sub    $0x14,%esp
   0x08048ee6 <+6>:     movl   $0x80c5868,-0x8(%ebp)
   0x08048eed <+13>:    movl   $0x0,-0x4(%ebp)
   0x08048ef4 <+20>:    mov    -0x8(%ebp),%eax
   0x08048ef7 <+23>:    movl   $0x0,0x8(%esp)
   0x08048eff <+31>:    lea    -0x8(%ebp),%edx
   0x08048f02 <+34>:    mov    %edx,0x4(%esp)
   0x08048f06 <+38>:    mov    %eax,(%esp)
   0x08048f09 <+41>:    call   0x8053b90 <execve>
   0x08048f0e <+46>:    movl   $0x0,(%esp)
   0x08048f15 <+53>:    call   0x8049790 <exit>
End of assembler dump.
(gdb) x/1sb 0x80c5868
0x80c5868:       "/bin/sh"
```

Looking at the disassambed code, it is seen that the code starts with the prelog process. A 20-byte (0x14-> 16 + 4 = 20) space is allocated in the stack region. The address showing the character "/bin/sh" is placed under 8 bytes below the address indicated by the ebp. In the output above, it is shown with the command x that this address shows the /bin/sh character string. The value 0 is placed 4 bytes below the EBP address. The address inside the address indicated by ebp-8 is loaded into the eax. The eax is currently shows the character "/bin/sh". The value 0 is placed in the address indicated by ESP + 8. The address in the ebp-8 is loaded into the edx. The value in the edx is placed in the address indicated by the esp + 4. The function execve is called after the value in the eax is placed within the address indicated by the esp. After this point, the shell operation is performed using the value 0 and the "/bin/sh" character string and address by the execve.

The code written in the C programming language was disassembled. So, we got an idea about its working mechanism. The same code was written in the assembly language in the previous LAB study. The sample code was as follows:

11. sys_execve

Syntax: int sys_execve(struct pt_regs regs)

Source: arch/i386/kernel/process.c

Action: execute program

```
.data
        file_to_run:
                .asciz "/bin/sh"
.text
        global _start
        _start:

                pushl %ebp
                movl %esp, %ebp
                subl $0x8, %esp         # array of two pointers. array[0] = file
_to_run  array[1] = 0
                movl $file_to_run, %edi
                movl %edi, -0x8(%ebp)
                movl $0, -0x4(%ebp)
                movl $11, %eax                      # sys_execve
                movl $file_to_run, %ebx              # file to execute
                leal -8(%ebp), %ecx                 # command line parameters
                movl $0, %edx                       # environment block
                int  $0x80
                leave
                ret
```

Below, with the command objdump, the machine-code-converted equivalent of the code to create a shellcode is displayed:

```
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ as shell.s -o shell.o
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ ld shell.o -o shell_exec
passwd@vulnubuntu:~/Desktop/assembly/AT&T$ objdump -d shell_exec

shell_exec:     file format elf32-i386


Disassembly of section .text:
                SH3LLC0D3
08048074 <_start>:
 8048074:       55                      push   %ebp
 8048075:       89 e5                   mov    %esp,%ebp
 8048077:       83 ec 08                sub    $0x8,%esp
 804807a:       bf a0 90 04 08          mov    $0x80490a0,%edi
 804807f:       89 7d f8                mov    %edi,-0x8(%ebp)
 8048082:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%ebp)
 8048089:       b8 0b 00 00 00          mov    $0xb,%eax
 804808e:       bb a0 90 04 08          mov    $0x80490a0,%ebx
 8048093:       8d 4d f8                lea    -0x8(%ebp),%ecx
 8048096:       ba 00 00 00 00          mov    $0x0,%edx
 804809b:       cd 80                   int    $0x80
 804809d:       c9                      leave
 804809e:       c3                      ret
```

In the output above, the characters 00 are seen. The \x00 as a NULL character stops the shellcode from running in memory. In addition, the characters such as carriage return, line feed, \x0a, and \x0d prevent the shellcode from opening as they perform carriage return and jump to the next line. These characters need to be cleared from the shellcode. Automatically clearing bad characters will be shown in the following topics. Instead of the mov commands that generate the null characters, the commands such as push and jmp are used to remove the NULL characters. Below is shown the sample shellcode (Source: http://shell-storm.org/shellcode/files/shellcode-827.php):

Assembly kod:

xor   %eax,%eax

push   %eax

push $0x68732f2f

push $0x6e69622f

mov   %esp,%ebxchine

push   %eax

push   %ebx

mov   %esp,%ecx

mov $0xb,%al

```
int    $0x80
```

<u>ShellCode C</u>

```
#include <stdio.h>

#include <string.h>


char *shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"

                  "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";


int main(void)

{

fprintf(stdout,"Length: %d\n",strlen(shellcode));

(*(void(*)()) shellcode)();

return 0;

}
```

```
root@vulnubuntu:/home/passwd/Desktop/assembly/c# gcc -mpreferred-stack-boundary=
2 -static shellcode.c -o shellcode
root@vulnubuntu:/home/passwd/Desktop/assembly/c# ./shellcode
Length: 23
#
```