Lab 6
A20453991

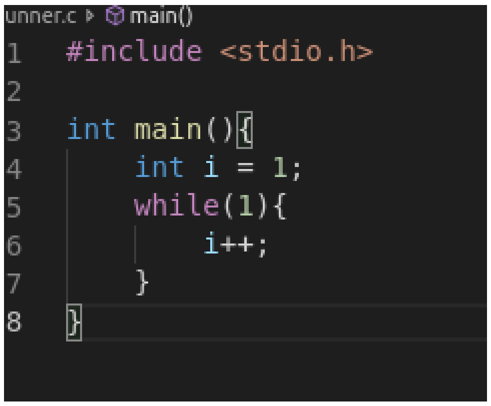
Task 1:

Q. 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.
ptrace is a system call in *nix operating systems. In *nix, parent processes ptrace child processes. Here gdb is the parent process of the debuggee process (the debuggee process is either forked by gdb or gdb). Whenever a process isn't runnning on a CPU core, its register values are saved in memory. *nix's ptrace is the API that you can use for reading/writing the saved register state, and memory. Gdb uses this api to print the register contents at a particular point of execution.

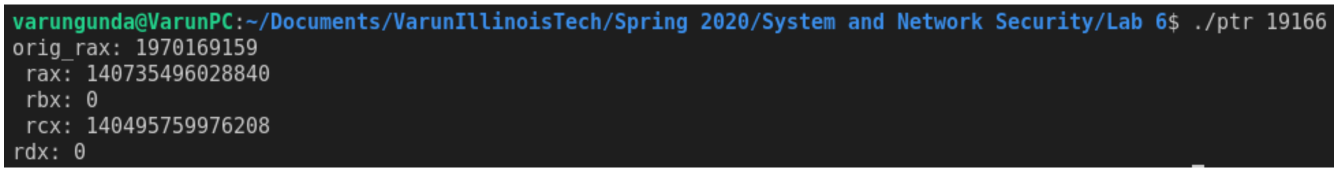
Q. Explain how you could build gdb's memory dump (e.g. x/32x) command.
We can use ptrace API's PTRACE_PEEKTEXT for this purpose. We can give this request along with pid of debuggee process and the addr that we want to dump to ptrace() system call. This will give us the data we need.

Q. 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.

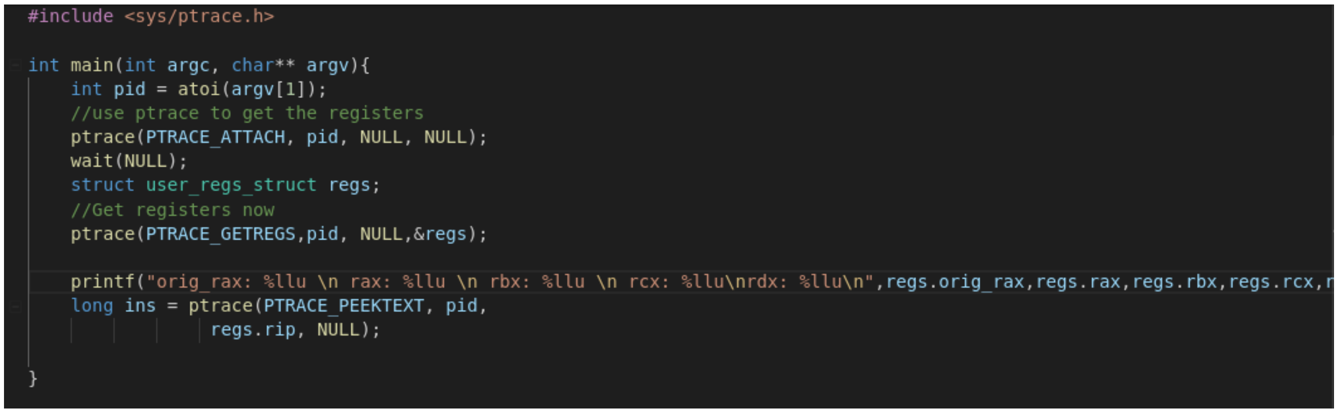
A simple runner program is created whose register contents we want to print. The program takes pid as input argument and attaches that process using ptrace and now we can print the register values as shown in the code.



```c
#include <stdio.h>

int main(){
    int i = 1;
    while(1){
        i++;
    }
}
```



```
varungunda@VarunPC:~/Documents/VarunIllinoisTech/Spring 2020/System and Network Security/Lab 6$ ./ptr 19166
orig_rax: 1970169159
 rax: 140735496028840
 rbx: 0
 rcx: 140495759976208
rdx: 0
```



```c
#include <sys/ptrace.h>

int main(int argc, char** argv){
    int pid = atoi(argv[1]);
    //use ptrace to get the registers
    ptrace(PTRACE_ATTACH, pid, NULL, NULL);
    wait(NULL);
    struct user_regs_struct regs;
    //Get registers now
    ptrace(PTRACE_GETREGS,pid, NULL,&regs);

    printf("orig_rax: %llu \n rax: %llu \n rbx: %llu \n rcx: %llu\nrdx: %llu\n",regs.orig_rax,regs.rax,regs.rbx,regs.rcx,r
    long ins = ptrace(PTRACE_PEEKTEXT, pid,
                      regs.rip, NULL);

}
```

Q. Why should ptrace only be available to a privileged process (that is, one more privileged than the one being traced)?
When a tool attaches to another process using ptrace call, the tool has extensive control over the target process. The tool can then manipulate the target process' file descriptors, memory and registers. Tool can also observe and intercept system calls and their results and can manipulate target's signal handlers and both receive and send signals on its behalf. The ability to write into the target's memory allows not only its data store to be changed, but also the application's own code segment, allowing the controller to install breakpoints and patch the running code of the target. SSH session hijacking and arbtrary code injection is fully possible if ptrace is allowed normally. As the ability to inspect and alter another process is very powerful, ptrace can attach only to processes that the owner can send signals to (typically only their own processes).

Task 2:

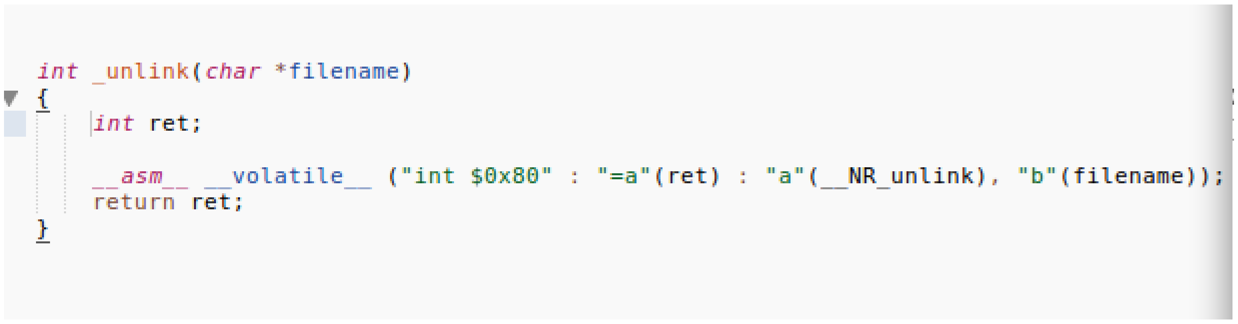
Q. What is the purpose of the PLT?
PLT, procedure linkage table is used to call external procedures/functions dynamically at run time. This is generally used for shared library functions. Procedure Linkage Table redirects position-independent function calls to absolute locations using the offset in the GOT. The linkage editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another, so instead it arranges for the program to transfer control to entries in the PLT. The dynamic linker determines the absolute addresses of the destinations and stores them in the GOT, from which they are loaded by the PLT entry. The dynamic linker can thus redirect the entries without compromising the position-independence and sharability of the program text.

Q. What is the purpose of the GOT?
Position-independent code cannot contain absolute virtual addresses. Global Offset Tables holds absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its GOT using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

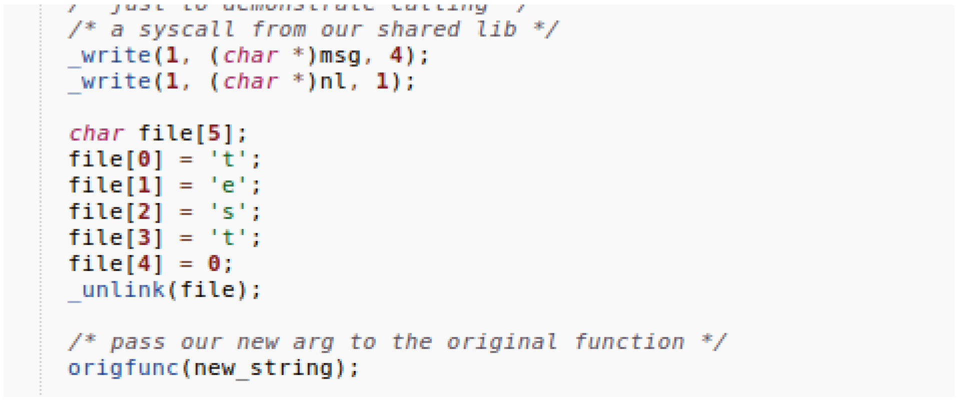
Q. Suppose neither existed in the ELF format. How would a dynamic linker then resolve symbols at runtime?
The dynamic linker resolves all symbols at program startup instead of deferring function call resolution to the point where they are first referenced. (The concept of LD_BIND_NOW)

Task 3:

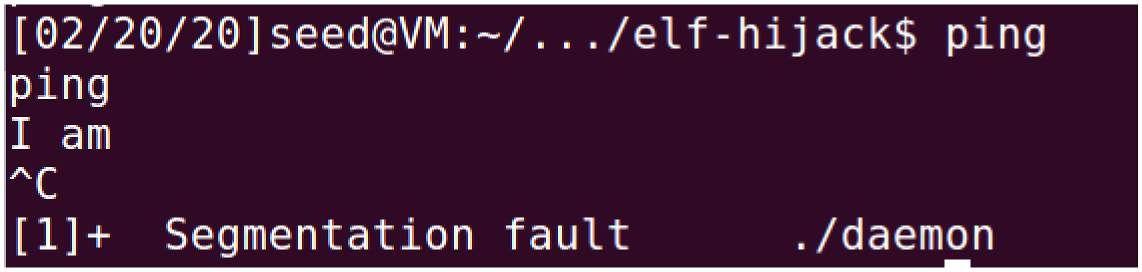
For this parasite.c has to be modified to inject the code that checks the date and removes the file at a certain date. However, I am getting segmentation errors when I tried even simple code as shown below



```
int _unlink(char *filename)
{
    int ret;

    __asm__ __volatile__ ("int $0x80" : "=a"(ret) : "a"(__NR_unlink), "b"(filename));
    return ret;
}
```



```
    /* just to demonstrate calling */
    /* a syscall from our shared lib */
    _write(1, (char *)msg, 4);
    _write(1, (char *)nl, 1);

    char file[5];
    file[0] = 't';
    file[1] = 'e';
    file[2] = 's';
    file[3] = 't';
    file[4] = 0;
    _unlink(file);

    /* pass our new arg to the original function */
    origfunc(new_string);
```



```
[02/20/20]seed@VM:~/.../elf-hijack$ ping
ping
I am
^C
[1]+  Segmentation fault      ./daemon
```